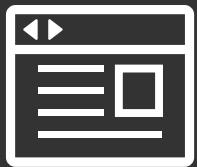


**Nick Abbott, Richard Jones,
Matt Glaman, Chaz Chumley**

Drupal 8: Enterprise Web Development

Learning Path

Harness the power of Drupal 8 to create enterprise-grade,
highly scalable websites



Packt

Drupal 8: Enterprise Web Development

Harness the power of Drupal 8 to create enterprise-grade, highly scalable websites

A course in three modules

Packt

BIRMINGHAM - MUMBAI

Drupal 8: Enterprise Web Development

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: December 2016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78728-319-0

www.packtpub.com

Credits

Authors

Nick Abbott
Richard Jones
Matt Glaman
Chaz Chumley

Content Development Editor

Amedh Pohad

Graphics

Kirk D'Penha

Reviewers

James Roughton
Tracy Charles Smith
Michelle Williamson
Todd Zebert
Vincent Lark

Production Coordinator

Arvindkumar Gupta

Preface

Drupal is a content management system used to build websites for small businesses, e commerce, enterprise systems, and many more. Created by over 4,500 contributors, Drupal 8 provides many new features for Drupal. Whether you are new to Drupal, or an experienced Drupalist, this Learning Path course takes you through the journey of building, extending, and customizing websites to build highly scalable and enterprise-ready websites.

What this learning path covers

Module 1, Learning Drupal 8, takes you step by step through building a Drupal 8 website. You will start with the basics, such as setting up a local "stack" development environment and installing your first Drupal 8 site, then we move on to image and media handling and extending Drupal modules. This section will help you get to grips with the modular nature of Drupal, and you'll learn how to extend it by adding new functionalities to create your new modules. After learning to develop and manage a modern and responsive website using Drupal 8, you'll start exploring different techniques to take advantage of the new Drupal 8 features.

Module 2, Drupal 8 Development Cookbook, is your go-to guide to experimenting with all of Drupal 8's features through helpful recipes. You will explore techniques to customize and configure the Drupal environment, create blocks and custom modules, as well as make your web apps responsive by harnessing the mobile-first feature of Drupal 8. This module will also show you how to incorporate multilingual facilities in your sites, use web services and third-party plugins with your applications from inside Drupal 8, and test and deploy your apps.

Module 3, Drupal 8 Theming with Twig, you will master Drupal 8's new Twig templating engine to customize the look and feel of your website. This section will walk you through a real-world project to create a Twig theme from concept to completion while adopting best practices to implement CSS frameworks and JavaScript libraries. You will see just how quick and easy it is to create beautiful, responsive Drupal 8 websites while avoiding the common mistakes that many front-end developers make.

By the end, you will have learned how to develop, manage, extend, and customize an enterprise-level website.

What you need for this learning path

For Module 1:

In order to follow along with this module you will need to install Drupal on your own development environment. The steps to do this are given in *Chapter 2, Installation*. A laptop or desktop machine should be sufficient, you will not need a commercial web server to complete the exercises detailed in the chapters to follow. Drupal works in any modern browser. You will not need to edit any PHP code to complete the chapters in this module.

For Module 2:

In order to work with Drupal 8 and to run the code examples found in this module, the following software will be required:

Web server software stack:

- Web server: Apache (recommended), Nginx, or Microsoft IIS
- Database: MySQL 5.5 or MariaDB 5.5.20 or higher
- PHP: PHP 5.5.9 or higher

Chapter 1, Up and Running with Drupal 8, details all of these requirements and includes a recipe that highlights an out of the box development server setup.

You will also need a text editor. Here is a list of suggested popular editors and IDEs:

- Atom.io editor: <https://atom.io/>
- PHPStorm (specific Drupal integration): <https://www.jetbrains.com/phpstorm/>
- Vim with Drupal configuration: <https://www.drupal.org/project/vimrc>
- Your operating system's default text editor or command-line file editors

For Module 3:

To follow along with this module, you need an installation of Drupal 8, preferably in a local development environment located on a Windows, Mac, or Linux-based computer. Documentation regarding setting up a local development environment can be found at <https://www.drupal.org/setting-up-development-environment>.

Specific system requirements are listed at <https://www.drupal.org/requirements>. An introduction to MAMP for Windows and Mac is also covered in *Chapter 1, Setting Up Our Development Environment*.

To follow along with each lesson, you will need a text editor or IDE. To see a list of software to consider when developing in Drupal 8 you can refer to <https://www.drupal.org/node/147789>.

Who this learning path is for

This course is suitable for web developers, designers, as well as web administrators who are keen on building modern, scalable websites using Drupal 8 and its wide range of new features.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a course, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Drupal-8-Enterprise-Web-Development>. We also have other code bundles from our rich catalog of books, courses and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Course Module 1: Learning Drupal 8

Chapter 1: Introduction	3
What is Drupal?	3
Some Drupal history	10
Summary	13
Chapter 2: Installation	15
System requirements	15
Setting up a development environment	16
Free cloud hosting	17
Installing Drupal 8	20
Troubleshooting your installation	24
Trusted host patterns	25
Summary	26
Chapter 3: Basic Concepts	27
Modules	27
Entities, nodes, and fields	31
Taxonomy	36
Blocks	37
Views	38
Users, roles, and permissions	39
Themes	40
Hooks	42
Summary	42

Table of Contents

Chapter 4: Getting Started with the UI	43
The Drupal 8 user interface	43
Responsiveness	47
Quick content creation	49
Listing content	51
Structure	52
Configuration	63
Appearance	64
People	65
Reports	65
Extend	66
Summary	67
Chapter 5: Basic Content	69
Introducing your site-building scenario	69
Basic pages	71
Articles	77
Summary	88
Chapter 6: Structure	89
Managing menus	89
Managing taxonomy	92
Working with the Views module	101
Working with Views blocks	109
Summary	117
Chapter 7: Advanced Content	119
Field types	119
Extending content types	120
Creating new content types	125
Listing testimonials with a view	151
Summary	165
Chapter 8: Configuration	167
People – Account settings	167
System	174
Content authoring	176
User interface	181
Development	183
Configuration synchronization	187
Media	187
Search and metadata	195
Regional and language	202

Table of Contents

Web services	205
Summary	207
Chapter 9: Users and Access Control	209
Users and roles	209
Permissions	210
Summary	221
Chapter 10: Optional Features	223
Activity Tracker	224
Aggregator	224
Ban	229
Book	231
Forum	234
Responsive image	241
Statistics	244
Syslog	246
Core (experimental), Multilingual, and Web services	246
Summary	247
Chapter 11: Reports	249
Accessing reports	249
Available updates	250
Recent log messages	251
Field list	255
Status report	256
Top 'access denied' errors	256
Top 'page not found' errors	257
Top search phrases	257
Views plugins	258
Summary	258
Chapter 12: Extending Drupal	259
Installing a module	259
Improving FAQs	260
Pathauto and Token	266
Summary	271
Chapter 13: Theming Drupal	273
What is a theme?	273
Terminology	273
Themes included in Drupal 8	274
Setting the active theme	275
Common settings	276

Table of Contents

Theme regions	279
Color scheme	280
Setting the admin theme	281
Advanced themes from the Drupal community	282
Custom themes	285
Summary	286
Chapter 14: Getting Support	287
What is open source?	287
The Drupal community	288
Drupal.org	288
Issue queues	289
The Drupal security team	291
api.drupal.org	291
IRC chat	291
The Drupal Association	292
DrupalCon	292
DrupalCamps	292
Specialist Drupal companies	293
Training	293
Certification	293
Summary	294

Course Module 2: Drupal 8 Development Cookbook

Chapter 1: Up and Running with Drupal 8	297
Introduction	297
Installing Drupal	298
Using a distribution	304
Installing modules and themes	306
Using multisites in Drupal 8	310
Tools for setting up an environment	312
Running Simpletest and PHPUnit	314
Chapter 2: The Content Authoring Experience	319
Introduction	319
Configuring the WYSIWYG editor	320
Adding and editing content	323
Creating a menu and linking content	326
Providing inline editing	328
Creating a custom content type	330

Table of Contents

Applying new Drupal 8 core field types	332
Customizing the form display of a node	336
Customizing the display output of a node	339
Chapter 3: Displaying Content through Views	343
Introduction	343
Listing content	344
Editing the default admin interfaces	348
Creating a block from a View	351
Utilizing dynamic arguments	354
Adding a relationship in a View	357
Providing an Entity Reference result View	362
Chapter 4: Extending Drupal	365
Introduction	365
Creating a module	366
Defining a custom page	370
Defining permissions	378
Providing the configuration on installation or update	382
Using Features 2.x	386
Chapter 5: Frontend for the Win	393
Introduction	393
Creating a custom theme based on Classy	394
Using the new asset management system	398
Twig templating	405
Using the Breakpoint module	409
Using the Responsive Image module	411
Chapter 6: Creating Forms with the Form API	417
Introduction	417
Creating a form	418
Using new HTML5 elements	424
Validating form data	428
Processing submitted form data	432
Altering other forms	436
Chapter 7: Plug and Play with Plugins	441
Introduction	441
Creating blocks using plugins	442
Creating a custom field type	450
Creating a custom field widget	455
Creating a custom field formatter	460
Creating a custom plugin type	465

Table of Contents

Chapter 8: Multilingual and Internationalization	477
Introduction	477
Translating administrative interfaces	478
Translating configuration	484
Translating content	487
Creating multilingual views	491
Chapter 9: Configuration Management – Deploying in Drupal 8	497
Introduction	497
Importing and exporting configurations	498
Synchronizing site configurations	505
Using command-line workflow processes	509
Using the filesystem for configuration storage	515
Chapter 10: The Entity API	521
Introduction	521
Creating a configuration entity type	522
Creating a content entity type	533
Creating a bundle for a content entity type	543
Implementing custom access control for an entity	553
Providing a custom storage handler	560
Creating a route provider	563
Chapter 11: Off the Druplicon Island	569
Introduction	569
Implementing and using a third-party JavaScript library	570
Implementing and using a third-party CSS library	575
Implementing and using a third-party PHP library	580
Using Composer manager	583
Chapter 12: Web Services	589
Introduction	589
Enabling RESTful interfaces	590
Using GET to retrieve data	595
Using POST to create data	598
Using PATCH to update data	603
Using Views to provide custom data sources	606
Authentication	609
Chapter 13: The Drupal CLI	615
Introduction	615
Rebuilding cache in Console or Drush	616
Using Drush to interact with the database	617
Using Drush to manage users	621

Table of Contents

Scaffolding code through Console	623
Making a Drush command	626
Making a Console command	631

Course Module 3: Drupal 8 Theming with Twig

Chapter 1: Setting Up Our Development Environment	639
Installing an AMP (Apache, MySQL, PHP) stack	640
A quick tour of MAMP PRO	642
Installing Drupal 8	645
Reviewing the new admin interface	658
Using the project files	669
Summary	673
Chapter 2: Theme Administration	675
What is a theme?	676
Exploring the Appearance interface	676
Drupal's core themes	677
Theme states	678
Installing and uninstalling themes	679
Theme settings	681
Theme-specific settings	684
Using prebuilt themes	686
Manually installing a theme	692
Cleaning up our themes folder	694
Managing content with blocks	694
Managing custom blocks	704
Summary	710
Chapter 3: Dissecting a Theme	711
Setting up a local development environment	712
Default themes versus custom themes	714
Folder structure and naming conventions	714
Managing configuration in Drupal 8	715
Reviewing the new info.yml file	715
The role of templates in Drupal	721
Creating our first basic theme	725
Introducing Twig	728
Template variables	736
The role of the theme file in Drupal	737
Summary	739

Chapter 4: Getting Started – Creating Themes	741
Starter themes	742
Creating a Bootstrap starter	743
Creating a Jumbotron	750
Rethinking our layout	753
Using Twig templates	756
Subthemes	764
Touring Classy	766
Summary	769
Chapter 5: Prepping Our Project	771
Walking through the design mockup	772
Restoring our database snapshot	783
Creating a custom theme	784
Summary	793
Chapter 6: Theming Our Homepage	795
Creating our HTML wrapper	796
Creating our homepage	798
Implementing our Header Top region	800
Implementing our Header region	803
Implementing our Headline Region	815
Implementing our Before Content region	824
Implementing the footer	827
Summary	832
Chapter 7: Theming Our Interior Page	833
Reviewing the About Us mockup	834
Creating our interior page template	836
Adding our Global Header	836
Implementing our page title	838
Implementing our main page structure	841
Implementing our Team members section	845
Adding our global footer	861
Fixing JavaScript errors	862
Summary	864
Chapter 8: Theming Our Blog Listing Page	865
Reviewing the Blog Listing mockup	866
Creating our blog listing	867
Creating a Post Listing view	872
Managing our Post Listing block	874
Implementing our Node template	875

Creating a theme file	886
Creating a Categories block	889
Managing our Categories block	890
Implementing responsive sidebars	891
Theming a Block template	893
Drupal Views and Twig templates	894
Managing popular versus recent content	895
Adding the About Us block	904
Summary	907
Chapter 9: Theming Our Blog Detail Page	909
Reviewing the Blog detail mockup	910
Previewing our Blog detail page	911
Creating a Post Full template	912
Working with comments	914
Implementing social sharing capabilities	927
Summary	931
Chapter 10: Theming Our Contact Page	933
Reviewing the contact page mockup	934
Introducing contact forms	935
Contact page layout	939
Adding a Callout block	939
Integrating Google Maps into our contact page	942
Summary	946
Chapter 11: Theming Our Search Results	947
Reviewing the Search Results mockup	947
Looking at default Search results	948
Introducing core search	950
Working with Search Results templates	953
Search alternatives	956
Summary	970
Chapter 12: Tips, Tricks, and Where to Go from Here	971
Working with Local Tasks	972
Reusing Twig templates	976
Where do we go from here?	979
Summary	980
Bibliography	981

Module 1

Learning Drupal 8

Create complex websites quickly and easily using the building blocks of Drupal 8, the most powerful version of Drupal yet

1

Introduction

What is Drupal?

Back in the old days (pre-1995ish), we used to have to download special software to our computers in order to buy things, look up things, and build things.

"Madness", I hear you say.

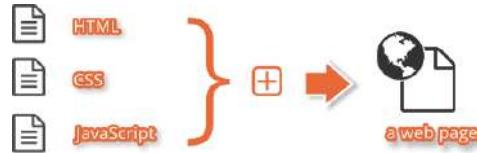
Of course those days are long gone. Nowadays, we all simply expect to be able do everything we need using a web browser. To put it another way, we all expect everything presented to us in some form of "web technology". But, what does it really mean in simple terms?

You probably already know that all the web pages that are a part of our everyday lives are written in the language of **HTML – Hyper Text Markup Language**.

If you've ever dug a little deeper, you might also know that the styling of web pages—the colors, typography, layout, and so on are controlled by **CSS – Cascading Style Sheets**.

Add some **JavaScript** into the mix, and web pages become a bit more interactive with things popping up and dropping down all over the place to make the experience a bit richer.

So, there it is. As good today, as it's always been...



Traditionally, piecing this all together involved a pretty detailed understanding of each of the three parts—it was all a bit too technical for many; it meant becoming fluent in these new "languages"; it meant you had to be a "coder".

So, what is **Drupal**? Where does it fit into all this?

It can be difficult to put a label on what Drupal actually is, since it is many different things to different people. We could start talking about terms such as "PHP-based social publishing software" and "web application framework", but let's not get into all that.

All you really need to understand right now is that Drupal is your LEGO-like toolkit for piecing together HTML, CSS, and JavaScript to build great websites.

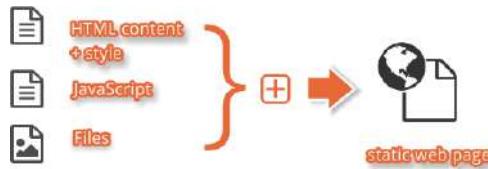
Drupal is a tool that equips anyone, regardless of their level of experience with web technologies, to build a state-of-the-art website. True to the founder's original vision of providing a website-building framework that can be used to spectacular ends without having to learn to code. You no longer need to know HTML, CSS, and JavaScript to create a real state-of-the-art website. Now, that sounds good.

Having set the scene, let's go back in time and discover how we got to Drupal 8.

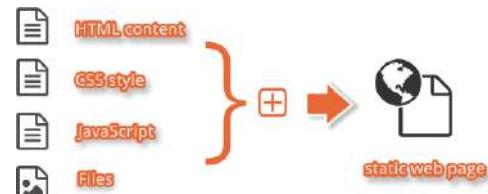
Dynamic web pages – a brief history

We used to piece together our web pages using HTML tags and styles all mixed together with JavaScript, images, and other file assets such as PDFs, Flash animations, and finally, the content itself.

The design, functionality, and the actual content were all mixed together to create the pages. These pages were also "static", in the sense that they could never change. The only way to update the actual content of the pages and/or their layout, cosmetic style, and fancy interactive moving parts was to be a "coder" and do some technical editing. Nasty!



Originally proposed in 1994, it wasn't until the early 21st century (2001) that we managed to separate out web pages' style from their content with the introduction of CSS. Likewise, the fashion moved to splitting out the JavaScript code into separate, more manageable files too.



Even with this useful advance, the pages were still only static. What we were really seeking was a mechanism for having the actual content itself come from other dynamic sources and for the web pages to be generated on the fly. We needed to make our pages dynamic.

Enter the database

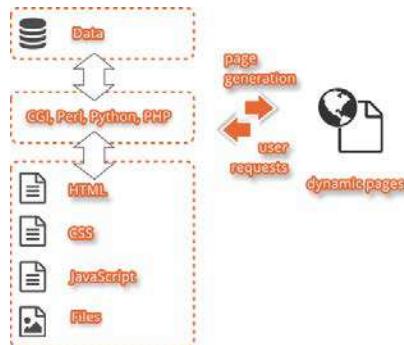
Then came a whole host of what we shall loosely refer to as code engines in various programming languages designed to achieve the dynamic approach: CGI, Perl, Python, PHP, Cold Fusion, ASP, JSP, Ruby, and numerous others.

Let's not worry about the technical differences between all these. What really matters here is that they all strived to meet the challenge of separating out the page structure, style, and content so that these elements could be organized in a more manageable fashion.

In all of the preceding approaches, instead of the content being embedded in the web page, it was now being stored in and retrieved from a database, and the HTML pages were being assembled on the fly from that data and a collection of site-themed elements. See the following for an illustration.

Introduction

This means that our pages can rebuild themselves in response to users' input; communication became two-way between the users and the website. This was the birth of "Web 2.0".

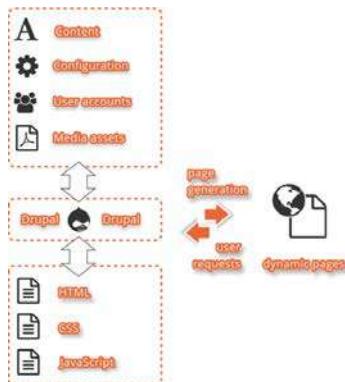


Drupal employs one of the aforementioned languages, namely PHP, to build data-driven pages and so provide us with a neat, manageable split between content, configuration, user accounts, and media assets such as images, documents, and video.

PHP is a very widely used open source scripting language that is especially suited to web development because it can be embedded into HTML pages. PHP "pages" are essentially HTML pages with embedded programming code that reshapes the contents of the HTML page dynamically before you get to see it in your browser.

The PHP code is executed on the server, and it generates a complete HTML page, which is then sent to your browser. As far as you are concerned, the end result is a fully formed HTML page, and you have no evidence that it was constructed dynamically by PHP.

Drupal is the clever PHP that brings together the various assets to form the actual visible pages, which are then made available to us across the web using a web server.



Drupal is completely free and always will be. It is open source. It is a software that is not owned by anyone but is instead developed collectively by a community of people interested in continually improving it as a platform.

Let's dwell a moment on the term "open source" so as to be clear about what it really means.

The word "source" here refers to the actual original program code written by the author. Programmers who have access to a computer program's source code can improve the program by adding features to it or fixing parts that don't always work correctly. The open source license actively promotes collaboration and sharing. Anyone can make modifications to source code and incorporate these changes into their own projects. Thus, open source projects benefit from a potentially infinite number of "authors". However good we might think we are at developing software, the community is better!

The *open source* label itself was created at a strategy meeting held in early 1998 in Palo Alto, California.

A worldwide community

It is important to understand that Drupal really is more than just the actual software. It is also a worldwide community of developers, designers, project managers, business innovators, technology specialists, and user-experience professionals. Community members all pull together to continually make Drupal ever more flexible, extensible, and standard compliant, so as to take advantage of emerging technologies. For a long time, the unofficial strapline of Drupal has been "come for the software, stay for the community," and this is certainly true in our experience.

The success of Drupal

So why does Drupal prosper and why is it steadily gaining momentum as the platform of choice for organizations large and small the world over? This question is answered in the following sections.

Multiple systems integration

We've become used to a diet of multiple software platforms and technologies each with its own cost, interface, storage, and security issues. You may be all too familiar with statements, such as:

"We have multiple systems working here, only some of which seem to talk to each other"

"You want a blog? Use WordPress for that".

Attempting to integrate a range of technologies is usually an expensive and never-ending business, and the management of the middleware (yet more layers of software) required to glue them together is a sizeable debt to be repaid, often over and over.



With Drupal, you will find that you can do it all in one place and in a consistent, coordinated fashion.



Technical debt

"I had to start from scratch"

Anyone in the IT project business knows all too well that the underlying code can quickly become the reserve of the individual developer who actually wrote it. Often it is only when the particular individual leaves the company that the technical debt is realized. The developer may not have documented their approach, let alone the actual code that someone else inevitably has to take on. The legacy may contain all manner of unjustified assumptions, poor coding practices, "hidden features" (that is, bugs), and quick but irreversible fixes that close the doors to integration and further extension. All too often, in trying to deal with their inheritance, the new developer ends up re-inventing it all over again often with a new set of assumptions and with a potential new set of bugs.

With Drupal, you can at least be assured that the code has gone through a clearly defined community peer-review process, and opting to use the Drupal framework as the basis for building your solutions will go a long way to addressing concerns about the code quality and therefore technical debt.

Developer knowledge

"We can't seem to find quality Drupal developers"

However, we should not be naïve and pretend to ourselves that Drupal is the silver bullet we've all been waiting for. It comes with its own significant technical debt not least of which is the absolute necessity for having proper Drupal-savvy developers on your team. Drupal code is very framework specific, so one should not expect a competent PHP developer to be able to be truly effective with Drupal without an investment of time learning to understand how to properly work with the framework.

Drupal 8 initiatives to stay relevant such as its use of the PHP framework known as Symfony 2 – essentially a collection of well-respected state-of-the-art PHP components – have gone a long way to make the underlying code more familiar to modern PHP developers who are used to working with modern object-oriented PHP. However, investment in learning the Drupal way is still crucial.

With its growing library of free extension modules, themes (pre-built skins), and sizable developer community, Drupal is still probably the wisest choice and the most future-proof content management framework around.

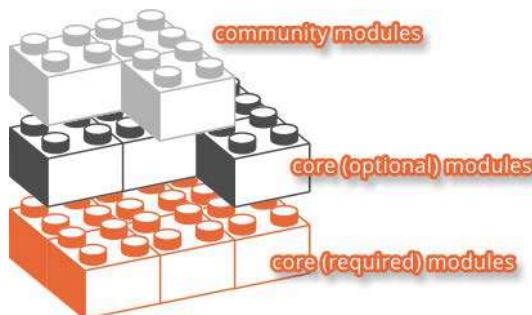


If you are considering adopting Drupal as your platform, then we strongly advise you to make certain that your developer team is professionally trained.



Modularity

Think of Drupal as a gigantic LEGO construction kit for would-be site builders. Drupal site builders develop websites by piecing together Drupal modules. Each module is designed to solve a particular problem but in a Drupal-compliant way, so as to keep all the doors open to integrating with the enormous repository of other community-written modules that are out there.



There's a module for that

When we plan to add a new feature to our website, the first step is to look around and see what's already been done by community members to see if "there's a module for that". For example, imagine that you want to create a community site where members can:

1. Login in and post-up articles and invite other members' comments.
2. Share posts with a variety of common social networking sites such as Twitter, Facebook, and the like.
3. Automate actions such as redirecting some users to key pages on log in.
4. Optionally promote their posts in animated highlighting carousels.

This can be done by simply performing the following steps:

1. A straight out of the box—a freshly installed—Drupal site will enable posting and commenting using only the core modules.
2. This is provided by *Easy Social* module—at the time of writing, v8.x-3.0-alpha3 (drupal.org/project/easy_social).
3. This is provided by the *Rules* module—at the time of writing, v8.x-3.x-unstable4 (drupal.org/project/rules).
4. This is probably best provided by *Nivo Slider* module—at the time of writing, v 8.x-1.4 (drupal.org/project/nivo_slider).

For pretty much anything you might want to add to your website, there's a very high chance that among the many modules out there, the very thing you are after has already been coded and posted up into the Drupal community site for public consumption at:

www.drupal.org.

While it is still early days with regard to many of the most commonly used community modules being ready as stable Drupal 8 releases, the number is growing every day.

Some Drupal history

It all started in 1999 when the founder, Dries Buytaert, began working on a simple website that incorporated a message board software application while he was an undergraduate at the University of Antwerp. For the next 2 years, he and a small group of friends used and developed the as yet unnamed application. In January 2001, they named their creation Drupal, open sourced it, and started the Drupal.org community site.

The rest, as they say, is history. The community continues to grow at an ever-increasing rate and the attendance figures at the official **Drupalcon** conferences worldwide are testament to the developer community's interest.

Language choice

Drupal is written in the open source scripting language PHP.

Because of PHP's relative simplicity and its rather forgiving nature, it's the perfect language choice for the newcomer because it can be learned fairly quickly and can be "assimilated" gradually by dissecting existing Drupal modules. This is certainly true at least for the Drupal 7 core. Drupal 8 code being object oriented, on the other hand, is a somewhat different story because object orientation is a concept that can be initially difficult to grasp for those without prior training.

Is Drupal a framework or platform?

Drupal is not, as is commonly quoted, just a **content management system (CMS)**. It's actually much more than that and we use the terms "framework" and "platform" quite deliberately in the opening paragraph to ignite the debate as to quite which of these Drupal is.

Once you've got to grips with the Drupal approach, you'll find that you can quickly and easily configure it as a content-management system. However, given its versatile and highly-bendable nature, you can in fact use it to build anything from a simple brochure-ware site to a fully-fledged web application with a huge community membership and which interfaces with a myriad enterprise services: Facebook, Twitter, Google, YouTube, Flickr, and whatever else is around the next corner.

The question still remains the same; is Drupal a framework – something with which we can build things, or is it a platform – something from which we launch other web products?

Let's say that we use Drupal to build a website that we intend to re-use over and over again as the basis for a collection of other sites. In this respect, what we have built is indeed a platform in its own right; a platform from which we can spawn all those other sites.

Let's say that we use Drupal to build a web application that integrates with a whole range of services in the wider world, again we have built a platform.

In both of the preceding examples, we created platforms, but in both cases, they are built on the Drupal framework.

Embracing other communities' frameworks

Drupal 8 is also a framework that openly embraces other open source frameworks: Symfony 2 (PHP), jQuery, Backbone, Modernizr, and Underscore (JavaScript) to name a few. The Drupal community does not seek to reinvent the wheel, but rather to integrate and build upon others' efforts and achievements.

How an open source community works

Software development based on the sharing and collaborative improvement of source code has a long history. In the late '90s, interest and participation in collaborative working increased markedly with two initiatives: the mainstream recognition of the Linux operating system and the release of the Netscape browser's source code.

Drupal likes to think of itself as a meritocracy, that is, those who are most influential in the community are those providing the best input, be it code, user experience, documentation, or otherwise. Neither individuals nor businesses can buy influence in the community, although they can of course achieve this by paying their staff to work on specific areas of interest.

The majority of people contributing to Drupal are doing so voluntarily in their own time. Some are sponsored by their employer, while some are just trying to solve a specific problem that interests them personally.

When contributing a new module to Drupal, the module's developer (also referred to as the **maintainer**) is entering into an informal agreement with the community that they will continue to maintain and update the module.

The mindset of the community is always to give back.

Those new to open source often struggle with this concept. There is a strange conflict that says "I don't want to give away my work"—when in fact your work is itself based on the unpaid efforts of thousands of others.

An appreciation of others' efforts is also key to the Drupal community. The Drupal issue queues are the place where bugs and feature requests are placed for both core and contributed modules. When reporting bugs, other community members are generally grateful for the efforts of the maintainer and offer constructive feedback or fixes.

People seem to understand that when you are not paying for something you don't have the right to be rude or disrespectful—although it's fair to say this should never be an option. That said, there can be heated discussions from time to time on contentious issues.

Summary

In summary, Drupal is what you make of it. You can simply download and build a site with Drupal 8 using just the core functionality, or you can extend the functionality using modules freely available in the community. Going further, you can develop your own modules and themes and make Drupal 8 the base for a complex e-commerce delivery system, if that's what you want to do. Like the LEGO analogy, the limitation is only your imagination and, maybe to some extent, your individual coding skills.

The remaining chapters in this module will take you step by step through the modules and functionality you get with a stock Drupal 8 installation without any community-contributed modules. This will show just how much you can already achieve before you have to think about extending Drupal, let alone coding.

2

Installation

This chapter takes you through the steps for installing **Drupal 8** using a local development environment. At the time of writing, the most up-to-date version of Drupal 8 is the 8.01 release, which can be downloaded from:

<https://www.drupal.org/project/drupal>

System requirements

Drupal 8 is a PHP-based software application and as such requires the following:

PHP

The following PHP version is required for the installation of Drupal 8:

- PHP 5.5.9 or higher

Web server options

One of the following:

- Apache
- Nginx
- Microsoft IIS

Database options

One of the following:

- MySQL 5.5.3 or above with PDO
- PostgreSQL 9.1.2 or higher with PDO
- SQLite 3.6.8 or above
- MariaDB 5.5.20 (or greater)
- Percona Server 5.2.8 (or greater)

Browser options

One of the following:

- Internet Explorer 9.x and above
- Firefox 5.x and above
- Opera 12 and above
- Safari 5.x and above
- Google Chrome

Up-to-date system requirements can be found at <https://drupal.org/requirements>.

Setting up a development environment

When you are getting started with Drupal, you may not have access to a web server in order to install it.

Often it is much easier at first to work on your own computer rather than have to worry about setting up an internet-hosted server environment by setting up a local stack of Apache, MySQL, PHP, which are often referred to generically as **AMP** stacks and as **LAMP**, **WAMP**, or **MAMP** stacks on Linux, Windows, or Mac, respectively.

Full instructions for installing Drupal manually into local stack can be found at: <https://www.drupal.org/documentation/install> but if you are not super confident at installing things like this manually, we recommend that you go for the Acquia Dev desktop method described below.

Free cloud hosting

Acquia is a company founded by the creator of Drupal, Dries Buytaert, which specializes in Drupal. **Pantheon** is another company which specializes in Drupal. Both of these offer a free sandbox hosting for Drupal.

<https://www.acquia.com/free> and <https://www.getpantheon.com> are cloud offerings which allow you to host and develop a Drupal 8 site on a hosted Internet-based web server. Both services offer a one-click install of Drupal. If you choose to use one of these, you can skip directly now to the section entitled *Installing Drupal 8*.

Acquia Dev Desktop

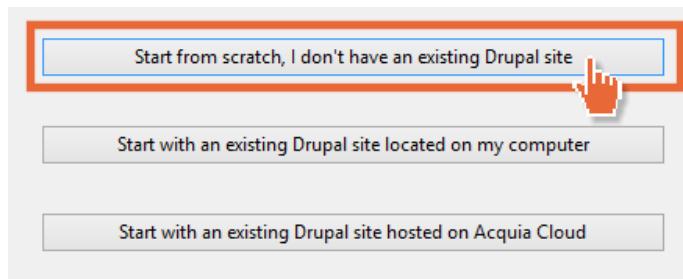
Acquia offers a free local development environment for Windows and Mac called **Acquia Dev Desktop** that you can download and install on your own local machine in order to run Drupal 8 in the shortest possible time.

Download either the Windows or Mac version of the Acquia Dev Desktop from <https://www.acquia.com/downloads> and install it selecting the defaults for the various settings.

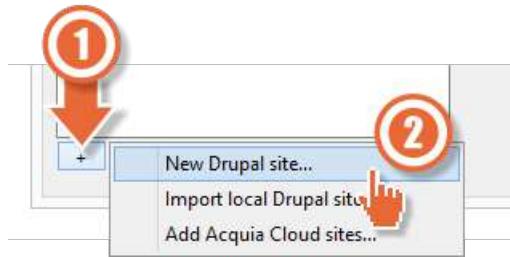
Once you have completed the installation, you will be offered the opportunity to run it.

When you do, you will see the stack start up in the background, and in the foreground you will see a welcome window offering some choices.

The first option is what we are after right now, so click on it:



If for any reason you cannot access the preceding pop-up choice window, you can always get back there by clicking on the + sign in the bottom left-hand corner with choices including **New Drupal site...** that you should select:



At the time of going to click on it, there are several choices available to you, some notable ones being:

- **Drupal:** This is the official Drupal 7 core. No extras added.
- **Drupal 8:** This is the official Drupal 8 core. These official releases come bundled with a variety of modules and themes to give you a good starting point to help build your site.

Also included are a number of other distributions – various ready-to-go packages built around Drupal 7, most notably the following:

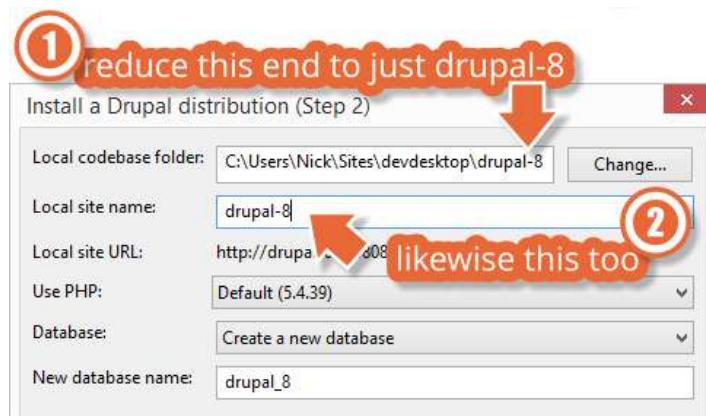
- **Drupal Commons:** This lets you create content-rich community websites built on Drupal 7. You can easily add wikis, calendars, groups, and other social web capabilities.
- **OpenAtrium:** This is an intranet in a box¹ that has group spaces to allow different teams to have their own conversations and collaboration.
- **OpenPublic:** This is designed for open government requirements, such as improving citizen services, providing public access to data and a public forum for two-way communication with agencies, without compromising accessibility, security, or usability.
- **OpenScholar:** This helps the management of educational institutions by providing Drupal-based professor pages, class catalogs, sandboxes, and extra tools for administration.

Locate the Drupal 8 offering and select install to begin the installation process:

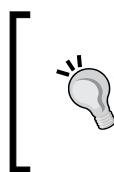


Optionally, in the following dialog and in order to keep your installation precisely in line with the given screenshots, change the following:

1. Local codebase folder.
2. Local site name to Drupal 8 as shown in the following:



Now, select **Finish**. Since you are setting up Drupal from scratch and have no pre-existing database, all the remaining default options will be fine, and a database named `drupal_8` will be created automatically. Apache will be appropriately configured, as will your local host file.



Dev Desktop sets up Apache on port 8083 rather than the standard (default) port of 80 so that it can be run on a local system without administrator privileges and without interfering with other local services. This means that when talking to sites governed by the Dev Desktop, you should always suffix your URLs with :8083.

Installing Drupal 8

Now that you have your server environment configured, whether it be a cloud sandbox, your own web server, or the Dev Desktop, the steps to install Drupal 8 are the same, only the URL will be different. In the case of the Dev Desktop, go to <http://drupal-8.dd:8083/install.php> in your web browser.



Note that the Dev Desktop adds on the '.dd' suffix to the domain.



The Dev Desktop makes launching your new site very easy because it provides a shortcut link from within the control panel window:



You will see a number of option screens during the installation process, each of which we will walk through now.

Select the language you want to install and click on **Save and continue**. This is the language the user interface will be presented in:



Next, you will be asked which of two built-in installation profiles you wish to use.



[ Note that if you are using the Dev Desktop then the Standard profile will be chosen automatically and so the whole step will be omitted.]

If you have installed manually or you are on one of the cloud offerings, opt for the Standard one for this tutorial then click Save and continue.

Next, you will need your database login details.

If you are using the Acquia Dev Desktop to develop your site locally, then you can just leave the defaults as follows, because it creates the database and user itself (you do not need to, and should not, set a database password). Please note that on any hosted environment you would of course always add a secure password.

If you are using your own web server or an alternative local AMP stack, you will need to complete the appropriate database credentials for a database that you have already set up in advance:



The screenshot shows the 'Set up database' step of the Acquia Drupal setup. On the left, there's a vertical navigation bar with links: Choose language, Choose profile, Verify requirements, Set up database (which is highlighted), Install site, and Configure site. On the right, under 'Database configuration', there are fields for 'Database type*' (radio buttons for MySQL, MariaDB, Percona Server, or equivalent, and SQLite, with MySQL selected), 'Database name*', which contains 'drupal_8', 'Database username*', which contains 'drupaluser', and a 'Database password' field which is empty.

Select **Save and continue**. Next, you will see a progress screen entitled **Installing Drupal**, which details the progress of installing each of the standard core modules that are active in a base install:



This may take a few minutes, depending on the speed of your local machine.

Configuring your Drupal 8 site

Once all the modules have been installed and the progress reaches 100%, you will be presented with the **Configure site** dialog as shown here:

Configure site

SITE INFORMATION

Site name *

Site email address *

This step contains quite a few fields, so the different sections have been broken up into individual screenshots.

The value you enter in the **Site name** field will be used in the header of the site (depending on your theme settings) and also in the browser title bar. For now, leave it set to *drupal-8*.

Note that we changed the site name down from `My site` to simply `Drupal 8` purely for neatness:

SITE MAINTENANCE ACCOUNT

Username *

Password *

Password strength: Weak

Confirm password *

Passwords match: yes

Email address *

The e-mail address entered is the main contact address of the site and will be used as the sender address by default whenever an e-mail is sent out by the site. The site maintenance account that is automatically created, now is the master account for the Drupal site – often referred to as the `admin`, `user`, `user 1`, or `uid:1`. This account has special privileges and is not subject to any kind of permission-based controls.

Due to the extremely high powered nature of this user, it is especially important in any real site to use a strong password rather than something like `admin`, as we are using in this tutorial.

The e-mail address will be copied from the site e-mail address by default, but it does not have to be the same.

For security in a real-life Drupal site, it is strongly recommended you use a non-standard username for the site maintenance account (and not something common or easily guessable).

However, for the sake of simplicity, we will just use `admin` for the username and `admin` for the password.

Next, you can set the default country and time zone for your site. The values chosen here will affect the various other regional settings in the site—for example, date formats:

REGIONAL SETTINGS

Default country
United Kingdom

Select the default country for the site.

Default time zone
Europe / London

The final installation step asks if you would like to check for updates automatically. It is highly recommended that you leave this switched on as you will then be notified when there are new versions of Drupal core or any contributed modules you have added.

If you do not select **Receive email notifications**, you will be notified of available updates via a notice on screen when you are logged onto the site with the site maintenance account active (as indeed you are now):

UPDATE NOTIFICATIONS

Update notifications

Check for updates automatically
 Receive email notifications

Save and continue

Select **Save and continue**.

Your installation of Drupal 8 is now complete, and you will see a message that says **Congratulations, you installed Drupal!**

You will also see the "Toolbar" module providing access to: **Manage**, **Shortcuts**, and your (**admin**) account page, and beneath that, the "Tray" as shown in the following screenshot:



Congratulations, your Drupal site is installed and ready to go!

Troubleshooting your installation

If your site does not install correctly, there are some configuration tips that you might want to check.

[ Note that if you are using the Dev Desktop, you need not do any of this.]

Memory settings

Sometimes, memory settings may need to be adjusted to get your site running correctly. Here are some suggestions on things to adjust.

If you are getting out of memory errors during installation, you can try increasing the memory available to Drupal.

When you imported the Drupal 8 site, the Dev Desktop will have created a site folder, which contains a settings file specifically. The site-specific settings file is called `settings.php` and can be found in the `default` folder.

Note that the `default` folder is also linked to the `drupal-8.dd` symlink.

Locate and edit the file:

```
/sites/default/settings.php
```

Add a new line:

```
ini_set('memory_limit', '128M');
```

This increases the memory allocated to PHP when running your site. `128M` should be sufficient, but if you still have problems, try increasing this. Note that by default, the file should have been set to read only on your system and therefore you may need to take some action before you can save your changes.

PHP timeouts

If you have a slower computer, sometimes the PHP timeout settings needs to be increased to allow more time for the installation steps to be completed.

In the same settings file as before:

```
docroot/sites/default/settings.php
```

Add a new line:

```
ini_set('max_execution_time', '240');
```

The `240` here refers to the number of seconds and this should be sufficient, but if you still have problems, try increasing this again.

Trusted host patterns

If you visit the page `/admin/reports/status` then you will see a warning informing you that **Trusted Host Settings** are not enabled.

Drupal 8 use a trusted host mechanism, where site administrators can whitelist hostnames. The mechanism now can be configured in the `settings.php` file.

Setting the pattern as show here will inform Drupal that all sites hosted locally using the Dev Desktop - this with a URL ending in '.dd' - are trusted.

```
$settings['trusted_host_patterns'] = array(  
  '^.*\.dd$',  
);
```

Summary

By following the steps in this chapter, you should now have a full clean install of the Drupal 8. Next, we will look at fundamental Drupal concepts and the terminology that you will come across while learning to build your first Drupal 8 site.

3

Basic Concepts

When you first start learning Drupal, there are some key terms that you will come across, which are used to define the components of the system. These are:

- Modules
- Entities
- Nodes
- Fields
- Taxonomy
- Blocks
- Views

Understanding these terms and how they relate to one another will ease your journey into Drupal. Using the correct terminology when reporting issues will mean you are more likely to be understood by the community and therefore are more likely to get help as and when you need it.

In this chapter, we will look at each of the components to provide a better foundation for learning Drupal.

Modules

As we discussed in *Chapter 1, Introduction*, Drupal is highly modular in its design; you can switch on or off various bits of functionality by enabling/disabling modules, and you can also extend the system by adding new ones.

Other systems may describe modules as **plugins** – the two are synonymous.

Core and contrib modules

The term Drupal core refers to the set of modules that are present in the main Drupal download that you have just installed. You can achieve a great deal using just these, but developers around the world have created their own modules for specific areas of functionality, which you can also use free of charge.

Collectively, these community-contributed modules are referred to as **contrib** modules.

All modules can be downloaded from the Drupal website (<https://www.drupal.org>), and each one has its own individual project page. The project page for each module contains the downloadable code, releases, documentation, and links to the issue queue that we will discuss later in this chapter.

Where the core modules live

In Drupal 8, the core modules are located in the `/core/modules` folder.



Prior to Drupal 8, core modules were located in the `/modules` folder.



Where your extension modules should live

You should never put your own modules or any contributed modules in the `core` folder.

It is recommended that you place your downloaded and custom module extensions in the `/modules` folder located in the Drupal root.

Following this rule will allow you to apply upgrades to the Drupal core system.

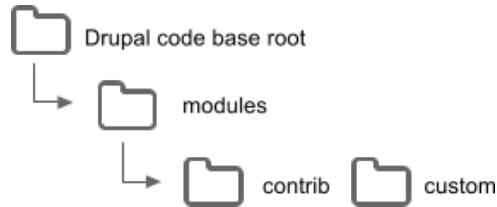


The `/sites/all/modules` structure, which was recommended in previous versions of Drupal, is still supported, but should not be used by default.



You can structure your modules folder however you like, but the standard convention is to have a structure in which you clearly segregate community (contrib) modules from any custom modules that you may have written yourself.

Thus, from the top level, the module folder structure is:



An example community module

As an example, let's look at a typical community module, in this case, the Google Analytics module. The Google Analytics module enables you to track visitors to your site using the popular Google Analytics service and can be found at:

https://www.drupal.org/project/google_analytics

The project page will contain a description of the module and one or more links to download it. Modules may be available to support multiple versions of Drupal. You should be downloading modules marked as version 8.x.

The following screenshot shows a typical module project page, in this case, the Google Analytics module:



Evaluating new modules

You should try to evaluate modules from the perspective of the requirements of the site you are building before you go ahead and download them. There are thousands of modules out there in the community; some are great, some are good, and some don't have a lot of traction, so be sure to choose the ones that are right for your use case.

Before you start downloading modules, first consider the following:

- How many downloads does the module have?

The more popular a module is, the more the code will have been analyzed and bugs reported.

- How many installs does the module have?

This is only a guideline as not all sites report back their usage.

Project Information

Maintenance status: [Actively maintained](#)
Development status: [Under active development](#)
Module categories: Statistics , Third-party Integration
Reported installs: **394,197** sites currently report using this module.
[View statistics](#).
Downloads: 3,039,566
Last modified: October 1, 2015



- Who is the maintainer?

Some maintainers have many modules which they look after. Maintainer reputation is also an indicator of quality: Is the module in active development? How responsive is the maintainer to the issue queue? When was the most recently committed update? and so on.

- Open bugs and issues?

Every module has its own issue queue (discussed in *Chapter 14, Getting Support*). The bug count should not necessarily scare you off. Popular modules have plenty of people reporting bugs; some may not be bugs at all, and some will be duplicates or support requests.

Maintainers for Google Analytics

[hass - 1141 commits](#)
last: 2 weeks ago, first: 8 years ago
[budda - 50 commits](#)
last: 3 years ago, first: 9 years ago
[View all committers](#)
[View commits](#)

Documentation

Some modules have it, while some rely on a `README` file. You can see all of this information in the bar on the right-hand side of the module page.

Module versions

Some modules will have a `-dev` version listed. This is the most up-to-date code, but may be unstable. Running a `-dev` module on a production site is *not* recommended; although pragmatically speaking, since Drupal 8 has only recently been released, it may well be something that you are forced to do. Most modules will have a recommended version that is currently supported by the maintainer. If in doubt, always use the recommended version.

Downloads		
Recommended releases		
Version	Download	Date
8.x-2.0-rc1	tar.gz (40.67 KB) zip (59.39 KB)	2015-Nov-22
7.x-2.1	tar.gz (38.72 KB) zip (45.83 KB)	2014-Nov-29
6.x-4.1	tar.gz (37.5 KB) zip (43.08 KB)	2014-Nov-29
Development releases		
Version	Download	Date
8.x-2.x-dev	tar.gz (40.91 KB) zip (59.88 KB)	2015-Nov-29
7.x-2.x-dev	tar.gz (39.07 KB) zip (46.29 KB)	2015-Nov-15
6.x-4.x-dev	tar.gz (37.87 KB) zip (43.6 KB)	2015-Nov-04

Entities, nodes, and fields

When working with Drupal, one of the most important concepts to understand is that you are not just building pages, you are building page containers that consist of components. These components are made up of things called *entities*.

Entities

Everything you create in Drupal is referred to as an **entity**.

Nodes

Most viewable content you create will be of a particular type of entity known as a **node entity**.

Node types

You may often hear the terms "node entity", "node type", and "content type" interchanged routinely, so it's a good idea to think of these terms as synonymous.

There can be many different types of node entities in any given site, but in the standard installation, there are only two content types (node types) defined and they are: **Basic page** and **Article**.

Fields

All node entities contain a **Title** property and one or more fields, an example of which is the **Body** field. When creating an Article node, we see the following properties, fields, and their types exposed to us ready to populate:

The screenshot shows the Drupal node creation interface. At the top is a 'Title' field with a red asterisk indicating it is required. Below it is a 'Body' field with an 'Edit summary' link, featuring a rich text editor toolbar with buttons for bold, italic, underline, etc. A note below the toolbar states 'Body field, type: Text (formatted, long, with summary)'. The body content area has a 'body' class and a 'p' tag. Below the body is a 'Text format' dropdown set to 'Basic HTML' and a link to 'About text formats'. The next section is 'Tags', with a 'Tags' field type 'Taxonomy term' and a note: 'Enter a comma-separated list. For example: Amsterdam, Mexico City, "Cleveland, Ohio"'. The final section shown is 'Image', with a 'Choose file' button and a note: 'Image field, type: Image'. An orange arrow points from the text 'Image field, type: Image' to the 'Image' section.

Adding new fields

One of the most useful things about Drupal is the ease with which you can add new fields to existing content types. You might, for example, wish to add a date to an article if that article relates to an event. This can be done in literally moments as you will see shortly.

Field types

In the standard profile we installed earlier, the available field types are as follows:

General

Boolean	A yes or no field or checkbox.
Comments	Allow visitors to the site to add comments.
Date	A date and time.

Number

List (float or integer)	A predefined list of values that the user can pick from when creating new content.
Number (decimal, float, or integer)	To hold a number in a specified format.

Reference

Content	A preconfigured reference field for setting up a relationship between one entity and another. Defaults to the node entity type.
File	A preconfigured reference field for setting up a relationship between one entity and binary file entity, such as a PDF.
Image	A preconfigured reference field, similar to a File field, but intended for images that will be displayed visually.
Taxonomy term	A preconfigured reference field useful for creating an association with one or more taxonomy (classification) terms and a specified Taxonomy vocabulary.
User	A preconfigured reference field for setting up a relationship to one or User entities.
Other	A reference field useful for referencing any type of other entity, for example, Commerce Product(s).

Text

Text (formatted, long)	For use where there will be longer descriptive text possibly with html markup.
Text (formatted, long, with summary)	As for Text (formatted, long) but with an additional short version of the text that can be used in a teaser for good SEO.
Text	For short text, generally fewer than 256 characters without markup.

As we saw earlier, the Article content type that is defined by the standard install profile without any additional configuration contains the following fields:

- Body: Text (formatted, long, with summary)
- Comment: (Comments)
- Image: (Image)
- Tags: (Taxonomy term)

LABEL	MACHINE NAME	FIELD TYPE	OPERATIONS
Body	body	Text (formatted, long, with summary)	Edit ▾
Comments	comment	Comments	Edit ▾
Image	field_image	Image	Edit ▾
Tags	field_tags	Taxonomy term	Edit ▾

You can customize any content type, delete any of the existing fields, or add new ones to suit your design. For example, if we want to create a field that allows us to add PDF files as attachments to Articles, we would simply add a new **Reference | File type** field.

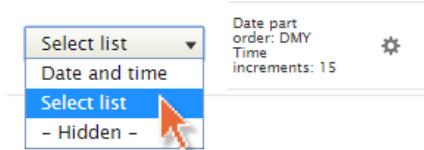
This has been no more than a brief glance at some isolated screenshots for now, because the topic of field editing will be discussed in detail in *Chapter 7, Advanced Content*.

Field settings

Fields can be set to have a single value, a fixed maximum number of values, or an unlimited number of values. For example, we can set the Article content type to allow a single PDF file to be uploaded, or we can set up the field to accept any number of file attachments.

Field widgets

Widgets provide configuration options for how the field is presented to the user when creating and editing content. For example, a Text (long) field may be presented with a **WYSIWYG (What You See Is What You Get)** editor window or as a simple plain text field. If, for example, we were using a Date-type field, then we might prefer to present the content editing team with a drop-down menu for each element: Day, Month, and Year in a standard format such as yyyy/mm/dd in place of the default simple text entry of a date.



Form display

Whenever you are editing an item of content through the Drupal user interface (UI), we refer to the UI as the "Edit form". Without any doubt, you will want to arrange the edit form with the fields in the best possible order to help your content editors, so Drupal allows a high degree of control over this for, via the following:

Managing the form display

Strictly speaking, the Title is not a field, it is a property of the node entity but along with all the other fields it is still configurable here:

FIELD	WIDGET
⊕ Title	Textfield
	Textfield size: 60
	⚙

Managing field display

Similarly, all the fields rendered in the final content view have display-related settings too. In the example below we are looking at the settings for the Image field:

FIELD	LABEL	FORMAT
⊕ Image	- Hidden -	Image style: Large (480x480)
	⚙	

Drupal 8 core entity types

We established earlier that all of the actual items of "content" on a Drupal site (Articles, pages, and so on) are collectively referred to as **nodes**. Put another way, they are Node entities.

Others types of entity in the standard Drupal installation are: Comment, Custom block, Taxonomy term, and User.

Each of these other four entity types has its own set of specialized properties and fields. For example, the User entity, unlike the node entity, does not have the "published" property, but it does have a property relating to user registration settings. Similarly, it also has a specialized field entitled Picture for holding the user's image.

You can extend existing entities by adding more fields any time you like.

You can also define completely new entity types. For example, the Drupal commerce solution provided by the Commerce suite of modules defines, among others, entities to represent Products and Orders. These custom commerce entity types have properties and fields, which are custom-built precisely for selling online; Product entities, for example, have a SKU property and an intelligent Price field.

This approach allows you to create a structured data model to suit the needs of your particular website.

Taxonomy

One of the most powerful features of Drupal is the taxonomy system.

The word **taxonomy** comes from the ancient Greek word meaning 'the practice and science of the classification of things'.

Consider a real-world example where Drupal might be used to manage, search, and display your music collection. It would be very useful to be able to categorize tracks within, for example, genre, because that way you could easily filter the entire collection down to say "soul" or "disco". Moreover, if you categorize the music tempo to say "slow", "medium", and "upbeat", then preparing a playlist of slow soul tracks would be simple just as you probably routinely do when using a dedicated organizer such as iTunes.

Thus, Drupal's taxonomy system is simply a means of enabling you to classify your content in many different ways.

The core taxonomy system allows you to define one or more vocabularies, each of which is a list of terms. For example, you may want to define an Article category vocabulary and associate it with your Article content type. The Article category vocabulary could contain terms such as "blog" and "news", thus enabling you to classify your Articles. The articles classified as blog appear on the blog page, and those categorized as news appearing on the news page.

A more abstract example might be to use taxonomy to categorize things by color. The vocabulary might be entitled `Color` and then you would have the actual colors as the terms.

You could then use the `Color` taxonomy to classify products such as t-shirts, socks, footballs, and so on, on a commerce site.

Fixed terms or tags

Taxonomies can be structured to have a fixed set of terms as discussed earlier, or they can be *tag based* in that the terms are defined as required when the content is being created just as you do when you are tagging content on say Facebook.

The Standard installation already has a single tag-based vocabulary called **Tags** associated with the Article content type.



Careful planning of your taxonomy is essential to the ease of use and management of your website.



Blocks

If you have a layout that contains sidebars, headers, or footers, you will most likely have content that is repeated between pages. If you edit this content in one place, you would expect it to be updated on every page.

This is where blocks come in.

In Drupal 8, there is another entity type called a **block**.

A block is a piece of content that can be placed in a specific region of a page, and you can set rules that determine when (on which actual pages) the block appears, depending on various conditions such as the page URL or the currently logged in user's role(s).

An example of a block is the `Main menu`.

A typical simple website page template will often have a header, footer, and perhaps, left and right sidebar regions as shown in the following diagram:



Within these page template regions, you can add repeating content such as the site logo, a menu, shopping cart, terms and conditions, or other items of content.

Because Drupal is a modular framework, modules can define a block that you can place on your site. For example, the Commerce module provides a "Shopping cart" block.

You can also create custom blocks, which are essentially individual blocks of hand-coded HTML.

The important point to remember here is that much of the content on your Drupal site appears one way or another, as a series of blocks and blocks are placed within regions on the page.

Views

Once you have started to create content, it won't be very long before you find yourself in a situation where you need to create lists of it, as follows:

- A list of article titles with each title linked to the article detail
- A list of article "teasers" (shortened versions) linked to the full detail
- A list of articles associated with a particular taxonomy term

The core module `Views` is a powerful module that enables you to do all this and much more.

`Views` is a powerful and flexible query-building and content-display tool that can be used to build complex content listing pages to present your content the way you want.

You can specify the number of items to display in a list and publish your view to a specific URL (a page) or as a block to be included on one or more pages.

There are a number of built-in views included in the standard profile, which we will explore later. For example, Drupal enables you to quickly and easily mark content to be "Promoted to the front page" and the following screenshot shows the default Drupal front-page view, which lists all that content:

Article no. 1

Submitted by [admin](#) on Tue, 15/12/2015 - 14:37

Ut wisi enim ad minim veniam, quis nostrud exerci tation
ullamcorper suscipit lobortis nisl ut aliquip.

Article no. 2

Submitted by [admin](#) on Tue, 15/12/2015 - 14:40

Lorem ipsum dolor sit amet, consetetur sadipscing elitr,
sed diam nonumy eirmod tempor invidunt ut labore et
dolore magna aliquyam erat, sed diam voluptua.

The Views module has been a staple of the Drupal site builders' toolkit for many years, but Drupal 8 is the first version of Drupal to include Views in the core module set.

In short, Views is a flexible query-building and content-display tool that can be used to build and present complex sets of filtered content. Views is so popular that almost every Drupal site uses it in some form or another. For this reason, the decision was made to move it into the Drupal 8 core.

You will see how to create views in detail later in *Chapter 6, Structure*.

Users, roles, and permissions

In order to log in to your Drupal 8 site, you will need a user account.

A site maintenance account was created automatically when you installed Drupal, and the user can perform all actions on this site. You may hear this user referred to as user 1, which is a reference to the user ID in the database.

A user is another form of entity, and like all other entity types, this means that you can add fields to the user definition in order to include more information in the user account such as forename, surname, and telephone number.

Each user is assigned one or more roles, and roles have permissions attached to determine exactly what the user is permitted to do when logged into the site. Standard roles are:

- **Anonymous user:** Assigned to anyone not logged into the site—visitors
- **Authenticated user:** This is assigned to anyone logged in to the site
- **Administrator:** This is for site owners, site maintainers, and site builders

You can, and most likely will, create additional roles for use on your site.

Drupal modules can add their own set of permissions. There is a permissions page where you can view and configure permissions for each role. You will find this page by navigating to **Manage | People | Permissions**.

The following example is an excerpt of the permissions page showing the custom permissions defined by the **Comment** module.

PERMISSION	ANONYMOUS USER	AUTHENTICATED USER	ADMINISTRATOR
Skip comment approval	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
View comments	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Each role is assigned any number of these permissions, and then each user is assigned any number of roles. This design allows you to create a very fine-grained permission system in your Drupal site.

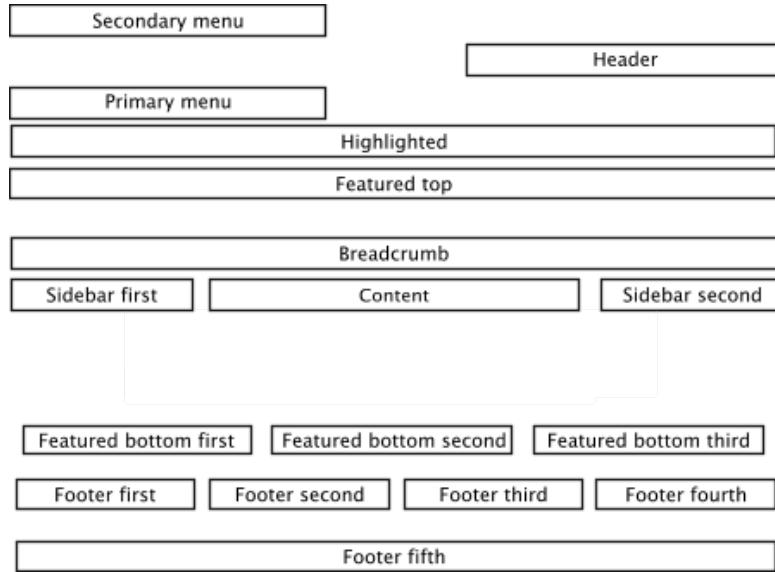
We will be covering roles and permissions in detail in *Chapter 9, Users and Access Control*.

Themes

A Drupal **theme** is the engine that provides the framework for the visual layout and design of your site, including bringing together the CSS, JavaScript, images, and colors. A Drupal theme itself determines the visual design of the website you are building.

In the standard install, there are a number of basic themes provided, and you can switch between them at any time.

The Standard profile installation ships with the Bartik theme. This theme is comprised of 17 page regions into which you can place one or more blocks. The 17 regions are shown in the following diagram:



As you can see from the preceding diagram, the list of default regions for the Bartik theme and their relative positions on the page is enough to fill the layout of many sites.

Page and theme template regions will be covered in detail in *Chapter 13, Theming Drupal*.

Just as with modules, you can download themes from <https://drupal.org> or you can create your own.

All themes have a settings page allowing you to change certain elements of the display, for example the color scheme used or whether to display the site logo and name. More complex themes have a greater range of settings within them.

Administration themes

There are also themes that are designed for use when visiting administrative pages, such as when managing users or create actual content. Themes such as these are known as **administration themes** or **admin themes** for short.

Drupal 8 ships with **Seven** as its admin theme.

A lot of effort has been made in Drupal 8 to ensure that admin themes work well on mobile devices allowing you to edit your content while on the move.

Base themes and subthemes

Some themes are not designed as finished products, but instead as starting points for your own theme development. These are commonly referred to as **base themes**. You can, and should, extend and override a base theme by creating a **subtheme**.

Themes will be covered in more detail in *Chapter 13, Theming Drupal*.

Hooks

An important concept in Drupal is the concept of module **hooks**. A module can expose one or more hooks empowering other modules to modify its behavior.

This means that if a module doesn't do exactly what you want, you can use a hook to "hook into" the process and change that module's behavior without having to change the original module code. Moreover, it means that when the original module whose behavior you have overridden gets updates, you'll be able to reap the benefits of the update while still keeping your modification(s) in place.

An example is `hook_entity_update` — a hook that is called after anything (any entity) is saved. Other modules may want to perform an action of their own in response to an entity update, and so we will also implement this hook.

Implementing hooks in your own modules is beyond the scope of this module. However, we'll see more on hooks in *Module 2, Drupal 8 Development Cookbook* and *Module 3, Drupal 8 Theming with Twig* of this course, but it's important to understand the terminology that you will no doubt come across as you continue working with Drupal.

For much more detail about the available hooks in Drupal, visit <https://api.drupal.org>.

Summary

This chapter introduced some key terminology used within Drupal 8, so you will know how to describe your system and understand online documentation and issues. Before continuing be sure that you have understood each of the terms such as modules, entities, nodes, fields, taxonomy, blocks, and views.

In the next chapter, we will start to experiment with your new Drupal 8 installation and work through the different functionality provided by the standard profile.

4

Getting Started with the UI

The Drupal 8 user interface

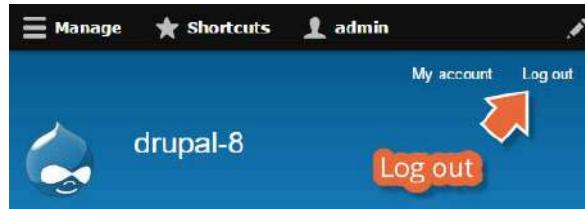
In this chapter, we will walk you through the key parts of the standard Drupal 8 User Interface (UI) focusing our attention on the main visible components like the Toolbar, Shortcuts and Drupal's main administrative menu, the Administration menu.

As you work through this chapter, please keep in mind that the intention here is merely to provide you with a quick overview and that much more detail and guided tutorial assignments will follow in later chapters.

Logging out

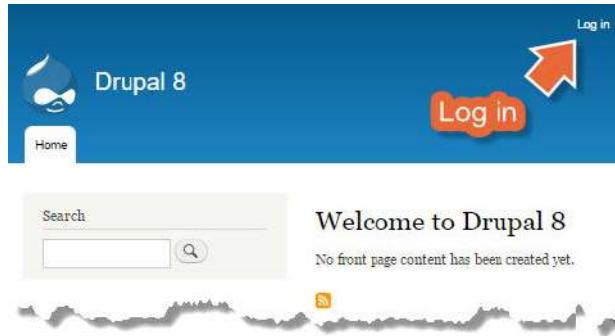
When you first visit your new Drupal 8 site immediately after installation at: <http://drupal-8.dd:8083> you will do so, from the point of view of the admin user, that is, you are logged in as the special user with user ID (uid) set to 1.

To see the site from an anonymous site visitor's perspective, click on the **Log out** link at the top right of the page:



Logging in

You will now see only the login link at the top right corner of the page:



Log back in using the credentials you set up in *Chapter 2, Installation*.

Username *

Enter your Drupal 8 username.

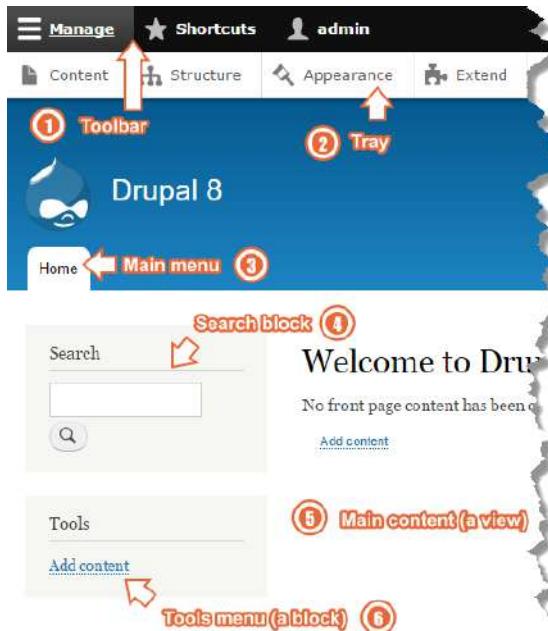
Password *

Enter the password that accompanies your username.

After logging in again, click on the **Home** tab or Drupal logo to go back to the front page.

Front page

We will now walk you through the page in order to illustrate what we were talking about back in *Chapter 3, Basic Concepts*, when we referred to pages being made up of **blocks** and **views**:



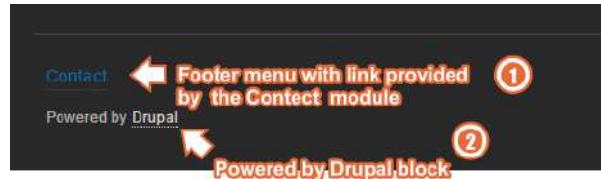
If your screen does not look exactly like the previous screenshot, try changing the width of your browser. The page layout will change, depending on the available size of the window. Matching the window size we used for the screenshots will help you get acquainted more quickly.

The default front page comprises the following elements:

1. The **Toolbar**, which in turn contains four icons: **Manage**, **Shortcuts**, the currently logged in user (**admin**), and at the right-hand end, the **Edit** toggle icon that reveals / hides the contextual links (the encircled pencil icons that appear as you hover over editable content, see later).
2. The **Tray**, which is immediately underneath the Toolbar, is a set of illustrated links that relates to the top level of the Administration menu.
3. The **Main menu**, which is also hard coded in the default theme page template.
4. The **Search block** is provided by the Search module, which is active by default.
5. The **Main content (a view)** of the page in this case is provided by a View, which lists all of the content of any type that has been promoted to the front page. On a brand new site, there isn't any content, so don't expect to see much here.
6. The **Tools menu**, like most things, exposed as a block.

In addition, at the bottom of the page you will see the Footer, which contains the following:

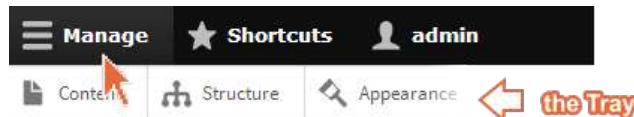
1. The **Footer menu (a block)**, which again can be contributed to by any module. In a fresh installation, the only module which is contributing a menu item is the **Contact** module. This is itself switched on by default.
2. The **Powered by Drupal block** is provided by the core **System** module.



By default, the **Toolbar** module displays links to top-level items from the core Administration menu of which, in a standard install, there is only one: **Administration** which itself comprises: **Content, Structure, Appearance, Extend, Configuration, People, Reports, and Help**.

Other modules have the opportunity to add extra links at any level with associated icons.

Let's experiment a little with it now, to illustrate some key navigation points. Click on the **Manage** link to show/hide the default set in the **Tray** which houses the Administration menu as follows:



Repositioning the tray

You can easily toggle the position of the **Tray** between horizontal and vertical modes by clicking on the position toggle icon at the far right-hand side:



We'll now step through each item in the tray (although not in quite the order that you see them presented on the screen), and cover the basic features provided by each. Before we do that, let's first introduce three key ideas:

Responsiveness

The Bartik theme that we talked about in *Chapter 3, Basic Concepts*, like all modern Drupal 8 themes, is designed to be responsive and so reacts to the size of the browser viewing the page using **breakpoints**. The breakpoints are designed for the desktop (wide), tablet (standard), and mobile (narrow) browser widths.

Let's test that out now.

With the tray still visible, gradually resize your browser and you will see two things happen. At the first breakpoint, you will see the **Main menu** change shape. Try it now.

At the second breakpoint, you'll see two more significant changes.

The **Search** and **Tools** blocks both clear so that they fit nicely within a portrait smartphone screen and the items within the **Tray** itself also clear and reposition themselves on the left-hand side of the screen.



Administration theme

As we mentioned earlier in *Chapter 3, Basic Concepts*, when you visit any administration type page on your site (any path beginning with /admin/), Drupal switches to the standard profile admin theme, **Seven**.

From the home screen, click on **Manage** and then click on the **Content** link.

You will be taken to the content page, which displays in the **Seven** admin theme. To get back to the main site, click on the Back to site link in the top left-hand corner of the screen.



Now that we have those two key ideas covered, we can take a quick tour of the main administration pages of a standard Drupal 8 install.

Contextual links

Drupal's **contextual links** provide you with immediate access to edit content and configuration without having to go and visit a backend admin screen.

The **Contextual link trigger** icons appear whenever you hover your mouse pointer over an item of content to which you have the rights to perform any kind of action such as configuring a block or editing content.

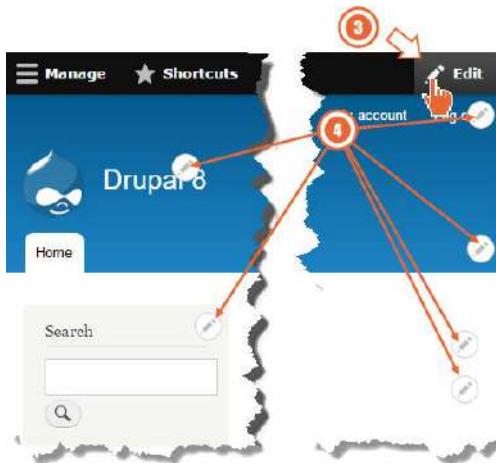
Try this:

1. Hover your mouse pointer over the **Search** block, and you'll see a contextual trigger.
2. Click on the trigger (pencil) icon, and you'll see the only relevant contextual link which is shown is one to configure that block.



You can place the whole session into edit mode by clicking on the pencil (3) at the right-hand end of the toolbar as shown in the following screenshot.

This will reveal all the contextual link triggers (4) across the whole page showing exactly which parts you have in-line control over:



Quick content creation

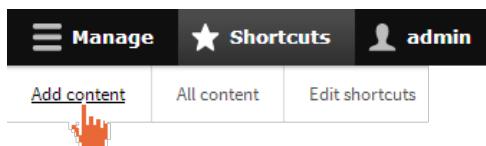
An appreciation of this section will be much more effective if you have at least a small amount of content created so to that end let's create a couple of items of content with some *lorem ipsum* filler text and tag them with some makeshift keywords.



There are a variety of free *Lorem Ipsum* filler text browser plugins around which can help you to speed up the process of inserting filler text. For even more fun, we recommend the *Corporate ipsum* Chrome plugin, which provides paragraphs of corporate marketing nonsense.

Don't concern yourself with the fine detail here, just follow the steps as follows:

Click on the **Shortcuts** button in the Toolbar and choose **Add content**.



Next, click on **Article** to create a new article and enter a **Title** for the article, some filler text for the **Body**, and add a couple of **tags**: 'tag1' and 'tag2'.

The screenshot shows the 'Article' creation form. At the top, there's a title field containing 'Article no. 1'. Below it is a rich text editor toolbar with various formatting options like bold, italic, and lists. The main body text area contains placeholder text: 'Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip.'. Underneath the body is a 'Tags' section with a dropdown menu showing 'tag1, tag2'.

Finally, click on **Save and publish** at the bottom of the screen to publish the article.

Save and publish

You'll be taken to the default view of article with a comment form directly underneath.

The screenshot shows the published article 'Article no. 1'. It includes basic metadata ('Submitted by admin on Tue, 04/14/2015 - 09:23') and the article body text. Below the article, a red box highlights the 'article (default view)' link. Below the article, a red box highlights the 'attached comment form' link. The comment form itself has fields for 'Your name' (set to 'admin'), 'Subject', and a large 'Comment' area with a rich text editor toolbar.

Repeat the same steps again for Article no. 2 and this time, tag the article with tag1 and tag2 as before but also add a couple of more tags such as tag3 and tag4:

The screenshot shows the 'Article' creation form for Article no. 2. The 'Tags' section now includes 'tag1 (1), tag2 (2), tag3, tag4'.

You should end up with at least two Articles of content on your new site, two of which share at least two tags.

Try clicking on the tag links for Article no. 2, to see the effect of this basic tagging.

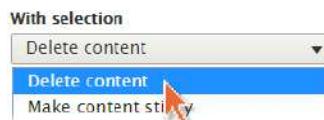
Listing content

Click on the **Content** link under **Manage** again, and you will see a list of all content that you have created so far.

You can filter content according to its **Published status** and by content **Type** and **Title**. Thus, you can easily reduce the list down to, for example, only show Articles.

The screenshot shows a form for filtering content. At the top are three dropdown menus: 'Published status' (set to 'Any'), 'Type' (set to 'Any'), and 'Title'. Below them is a 'Filter' button with a red circle containing the number 1. Underneath is a 'With selection' section with a dropdown menu. The menu has three options: 'Delete content', 'Delete content' (which is highlighted with a blue selection bar and has a red circle with 2 over it), and 'Make content sticky'. Below the menu is an 'Apply' button with a red circle containing the number 3.

You can then select one or more items using the checkboxes and apply various actions using the **With selection** menu.



At this stage we are only aiming that you visit this page to witness the kind of bulk operations that you can do to your content nodes.

Don't actually select any of the nodes and apply any of the **With selection** options just yet and be sure to leave the articles promoted to the front page.

You may also note that you can create new content from this page too, and you'll be covering this in full detail later in *Chapter 5, Basic Content*.

Revisiting the home page

Return to your home page by clicking on the **Back to site** link followed by the Drupal logo or the **Home** menu item.



Alternatively, you can get there by typing `http://drupal-8.dd:8083` directly in the browser address bar.

Note that you can also jump straight to the home page by clicking on the site name.

You may recall from the illustration earlier, in the section entitled **Front Page**, that the **Front page Main content** region is actually a **view**. When you first visited your home page at the beginning of this chapter, you hadn't created any content, but now that you have, the dedicated Front page view lists out all the Articles that have been **Promoted to the front page** in reverse order of their creation date. By default, all the newly created Article nodes are Promoted to the front page, so you should see the following listing of teasers to your recently created articles:



Structure

This section provides access to the all the key administrative screens for site building:

A screenshot of the Drupal Structure administrative menu. The top navigation bar includes "Back to site", "Manage", "Shortcuts", and "admin". Below this, a secondary navigation bar has tabs for "Content", "Structure" (which is highlighted), "Appearance", "Extend", and "Configuration". The main content area is titled "Structure" with a star icon. It shows a list of administrative tasks: "Block layout", "Comment types", "Contact forms", "Content types", and "Display modes". Each task has a brief description and a link. A decorative wavy line graphic is at the bottom.

You'll be covering each section of this page more comprehensively in later chapters so we won't worry about fine detail here.

As a Drupal site builder, you'll be spending a lot of time in here and so it is well worth a quick look around the various pages to get an idea of what this section contains.

Block layout

You'll recall from *Chapter 3, Basic Concepts*, that most Drupal pages are actually made up of collections of blocks with each block being placed in a one or more theme regions.

The administration page **Manage | Structure | Block layout** provides you with a single screen to manage all the available blocks on your site. The list in the left-hand side shows you which blocks are currently active and into which regions they are placed.

You will see a list of region names and their current block content. For example, you'll see the **Site branding** block placed in the **Header** region.

You can click the **Place blocks** button to add a new block into a region and then, *when* (on which pages) they should appear according to visibility rules.

Note that the Place blocks pane is filterable and that the blocks are grouped according to whether they are: forms, lists (views), menus, system blocks, or user related:



 You can place a block as many times as you like so that you could, for example, have your search block appearing in the header and on a sidebar.

Note that the secondary tabs at the top left of the window provides access to the separate block placement pages for each of the currently enabled themes: Bartik, and Seven.



It's worth just a quick browse around the various tabs (primary and secondary) to see what is where, but as always, rest assured that guided exercises will follow later in *Chapter 6, Structure*.

Comment types

The ability for your site visitors to attach their own comments to content is a key social-publishing feature, and so in Drupal 8, you are afforded a high degree of control over when, where, and how such comments appear.

By default, in a standard installation, only the Article content type is comment enabled.

Use this structure page to dictate exactly how comment entry forms should be customized on a per-content-type basis and also how posted comments appear when browsing that content.

Comment forms can be customized in that you can add extra fields to capture different data and opt to show all or just some of these fields when the comments are later viewed attached to some actual content.

We will be covering the idea of customizing fields in detail in *Chapter 5, Basic Content*, and how to add new ones in detail in *Chapter 7, Advanced Content*, but for now, it's worth clicking on the **Manage fields** button so as to glance at the field-editing options here:

COMMENT TYPE	DESCRIPTION	OPERATIONS
Default comments	Allows commenting on content	Manage fields

Contact forms

The Contact module provides a very useful site-wide contact form and can be configured and added to.

The **Manage | Structure | Contact Forms** link takes you to the configuration page for the Contact module.

Try editing the existing **Website feedback** category by adding in a **Recipient** address:

FORM	RECIPIENTS	SELECTED	OPERATIONS
Personal contact form	Selected user	No	Manage fields
Website feedback	YourEmail@example.com	Yes	Edit 
Label *			
Website feedback			
Machine name: feedback			
Example: 'website feedback' or 'product information'.			
Recipients *			
YourEmail@example.com			

Save your changes and then visit <http://drupal-8.dd:8083/contact> (or click **Back to site** and then click on the existing **Contact** link in the footer menu).

Click the **Add contact form** button to add another new contact form, called **Bug report**, with its own unique URL:



Personal contact form

You can also use this contact forms configuration page to add new fields to the **Personal contact form** which, role-permitting (see later), enables site visitors to send emails directly to any other registered site users without needing to know their actual email address.

Content types

As we have seen, Drupal 8 ships with just two very basic content types: **Article** and **Basic page**. These are reasonably simple out of the box with only a few fields present but you can use the Content type administrative page at `/admin/structure/types` to significantly customize them by adding and configuring fields.

Customizing a content type

Visit `/admin/structure/types` by clicking on the **Content types** link, then use the drop-down menu to choose **Edit**.

NAME	DESCRIPTION	OPERATIONS
Article	Use <i>articles</i> for time-sensitive content like news, press releases or blog posts.	Manage fields Manage form display Manage display Edit Delete
Basic page	Use <i>basic pages</i> for your static content, such as an 'About us' page.	Manage fields Manage form display Manage display Edit Delete

For each content types, you can:

- Provide a human-readable name and description and also the underlying machine name.
- Provide a description of the purpose of the content type that appears on the **Manage | Structure | Content types** page.
- Set the label for Title field; the default is 'Title' but if, for example, it was a News item content type then you might want to set this to 'Headline'.
- Disable previewing of content before submission, make it mandatory to do so or make it optional.
- Specify whether content of this type, published by default, is automatically promoted to the front page and whether or not it should be marked as sticky at the top of any listings.
- Opt to display author and date information on every post of this type – ideal for news.
- Specify which, if any, Drupal menus are available for adding links to content of this type.

Customizing fields

As a Drupal site builder, you'll spend a lot of time in this administrative screen customizing your content types' fields and display settings.

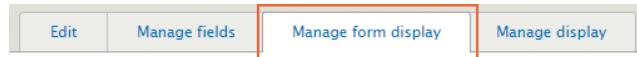
We will be covering this in fine detail later in *Chapter 7, Advanced Content* but for now, it's worth a quick look at the contents of this section as a brief preview of what's to come:

1. Click on **Content types** to visit the `/admin/structure/types` page
2. Click on **Manage fields** button on the **Article** content type
3. Click on the **Edit** button for **Image** field in the column label **Operations**.

NAME	DESCRIPTION	OPERATIONS	
Article	Use <i>articles</i> for time-sensitive content like news, press releases or blog posts.	Manage fields	
LABEL	MACHINE NAME	FIELD TYPE	OPERATIONS
Body	body	Text (formatted, long, with summary)	Edit
Comments	comment	Comments	Edit
Image	field_image	Image	Edit
Tags	field_tags	Entity reference	Edit

Customizing content entry screens

Access this page directly at `/admin/structure/types/manage/article/form-display` or via the primary tabs across the top of your screen:



You can use this admin area to edit order and behavior of the various fields when your site user is in the content-editing screen. This is referred to in Drupal as the **form display**:



Probably, the main point of interest to you right now is the choice of widgets via which content editors can interact with the fields. As a quick exercise as to what is here, try clicking on the gear icons to the right of each field to see the configuration options that are available to you.

Full details and guided exercises are in *Chapter 7, Advanced Content*, but for now, have a quick look to see what's here.

Customizing the display

By default, the Article and Basic page content types ship with two view modes configured to control the display of those types. The **Default** is used when viewed as a whole and a shortened **Teaser** mode when the content is viewed in a listing.

In this administrative section, you are granted a lot of control over how content is displayed within these two view modes.

Click on **Manage display** and look at the **Default** view mode settings.

Note also that when you view an Article, the setting on the right-hand side controlled by the cog-wheel icon is such that any included image is automatically scaled to a predefined **Large** (480 x 480 pixel) image style:

Now, click on the **Teaser** secondary tab and you should probably be able to make good sense of the settings here too.

Note that when you view a **Teaser** of an Article, any included image is automatically scaled to a predefined **Medium** (220 x 220 pixels) image style and that clicking on the image is set to take you to the full piece of the content.

By default, the main **Body** copy of the article is trimmed down to 600 characters when viewed as a teaser. A default length, that in our experience, you will almost always want to reduce.

To reiterate, this section of the module only aims at giving you a basic feel for what's available here. You'll be covering the topics of customizing fields and their displays in detail in *Chapter 7, Advanced Content*.

Display modes

The term **display modes** refers to a wider grouping of how form elements are displayed in different scenarios.

There are two types of display modes: view modes (already mentioned) and form modes.

View modes

In the previous section, *Customizing the display*, you learned that we have two pre-defined view modes: Default and Teaser available for use.

We looked briefly at the differences comparing field settings and field visibility between these two, but there are actually several other inactive view modes available for content types making the full list:

- Full content
- RSS
- Search index
- Search result highlighting input
- Teaser

Likewise, there are also two view modes for users as follows:

- Compact
- User account

There are others too for different entity types such as comment, but more about these later. In *Chapter 7, Advanced Content*, you'll see how you can activate/customize these and how you can set up your own custom view modes.

Form modes

Form modes are aptly named because they refer to collections of layout and configuration settings of the various relevant fields when users are filling in forms.

For example, the standard user profile edit page uses the default form display for users, but you can customize the existing **Register Form** mode so that registering a user sees a different layout when registering on the site or routinely updating their own profile.

Likewise, you can set any number of other form modes for users and employ them in different contexts throughout the site. In summary, you have a high degree of control over the default presentation of content when it is being viewed or edited. You can create any number of new **view modes** and **form modes** to suit your site's needs.

Once again, we'll cover Display modes in more detail and with practical exercises in *Chapter 7, Advanced Content*.

Menus

Drupal 8 ships with five standard menus:

- Administration
- Footer
- Main navigation
- Tools
- User account menu

Access this page directly at `admin/structure/menu` or click on the **Menus** link from the structure page:



This will take you to a page where you can edit any of the existing menus and component links, as well as create new custom menus:

TITLE	DESCRIPTION	OPERATIONS
Administration	Contains links to administrative tasks.	<button>Edit menu</button>
Footer	Use this for linking to site information.	<button>Edit menu</button>
Main navigation	Use this for linking to the main site sections. 	<button>Edit menu</button>
Tools	Contains links for site visitors. Some modules add their links here.	<button>Edit menu</button>
User account menu	Links related to the user account.	<button>Edit menu</button>

Try clicking on the **Edit menu** button next for the **Main navigation** menu and then edit the existing **Home** item.

Note that this is a special item and you can actually edit it—it simply points to the default Drupal front page which, you may remember, is a list of Teasers of all items that have been Promoted to the front page.

Try the same exercise for the **Contact** entry in the **Footer** menu. Again, you will find that this item is similarly protected.

Detailed exercises with menus are in *Chapter 6, Structure*.

Taxonomy

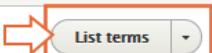
You may recall from *Chapter 3, Basic Concepts*, that Drupal ships with a single taxonomy in place, the **Tags** vocabulary.

You'll also recall that in the earlier section, *Quick content creation*, you created two Article nodes and tagged the first with two tags: **tag1** and **tag2**, and the second with two additional ones: **tag3** and **tag4**.

These four taxonomy terms are written into the vocabulary as you entered them.

Thus, if you click on the **list terms** button, you should see those terms in place.

The administrative page at **Manage | Structure | Taxonomy** lists out all the available vocabularies and provides links for you to edit terms associated with them:

VOCABULARY NAME	OPERATIONS
Tags	 List terms

Clicking on any of the terms listed here will take you through to a page that lists out all the content that is tagged with that term:

NAME	OPERATIONS
+ tag1	 Edit
+ tag2	 Edit
-	

Views

You'll recall from *Chapter 3, Basic Concepts*, that we pointed out that almost every Drupal site uses the **Views** module to provide filtered, grouped, and sorted lists.

The administrative page at **Manage | Structure | Views** lists out all the currently defined Views split into two distinct groups: those that are enabled and active and those are not:

VIEW NAME	DESCRIPTION	TAG	PATH	OPERATIONS
Content Displays: Page Machine name: content	Find and manage content.		/admin/content	

You'll be dealing with Views in detail in *Chapter 6, Structure*, so no need for you to dig too deep now, but what is interesting though is that one of the listed Views is entitled **Content** (see the preceding screenshot).

This View actually defines what you saw on the Content administration page back in the Listing Content section. Try clicking on the link.

Configuration

This takes you to an administrative page where you'll find configuration settings for the whole site broken down into nine categories.

We will not attempt to step though each of these here, now, but instead visit them in the context of other chapters most notably *Chapter 8, Configuration*

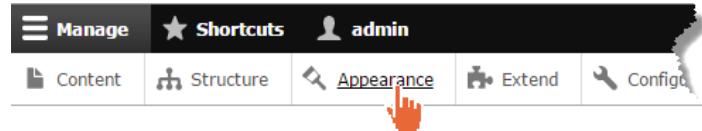
Click on the **Site information** link in the **SYSTEM** category:



Use this link now to change the name of your site to **My Drupal Agency** or something similar. While you're there, add a slogan too. Then, save the settings and return to the site to see the effect of your configuration changes.

Appearance

Next, we'll go back into the Main Administration menu and visit the **Appearance** page:



This link takes to an administrative page where you can browse all themes and instantly switch your site into any of those available.

When you visit the page, you'll note that the currently active (default theme) is **Bartik**.

The other currently enabled theme is **Seven** and that is in use as the current administration theme and when creating and editing content.

If you scroll down to the bottom of the page, you will also see an area labeled **Uninstalled themes** which currently only contains **Stark**.

The **Stark** theme is designed to show you what the raw Drupal HTML markup will look like without any kind of style applied.

Try clicking on the **Install and set as default** link to see it in action:



Having set **Stark** as the default, you'll need to click on the **Back to site** link in Toolbar to see it in action because the screen that you are currently on is an administrative page and is therefore using the **Seven** theme.

Having witnessed **Stark** in action, click on **Appearance** again and switch back to **Bartik** by clicking its **Set as default** link.

Finally, at the very bottom of the page you'll note that the Administration theme is currently set to **Seven**. There are no other administration themes supplied with the standard profile installation.

People

This is the default user-management page; it enables you to add new users, lists all users according to filtered criteria, and edit existing user account details:

The screenshot shows the 'People' page with various filtering options and user actions.

- Filtering:**
 - 'Name or email contains' field with a dropdown for 'Role' set to '- Any -'.
 - 'Permission' dropdown set to '- Any -'.
 - 'Status' dropdown set to '- Any -'.
 - A red arrow points from the 'filter criteria' label to the filter fields.
 - A red arrow points from the 'apply filtering criterion' label to the 'Filter' button.
- User Actions:**
 - 'With selection' section with a button to 'Add the Administrator role to the selected users'.
 - A red arrow points from the 'user actions' label to the selection button.
 - An 'Apply' button below the selection section.
 - A 'Show all columns' link.
- User List:**

	USERNAME	ROLES	OPERATIONS
	admin	• Administrator	Edit
- Buttons:**
 - A red arrow points from the 'users' label to the 'Edit' button.
 - An 'Apply' button at the bottom of the list.

You can also apply a number of actions to the currently selected user(s).

Like any administrative screens, the **People** page is actually powered by a **View** and so, as you will see later in *Chapter 6, Structure*, you can customize the screen just as you like.

We will also be covering user management again in depth in *Chapter 9, Users and Access Control*.

Reports

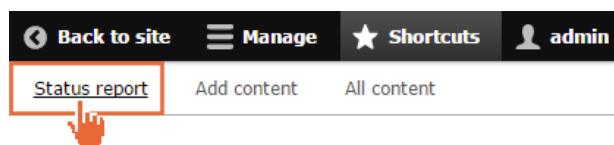
The **Reports** page lists out links to a useful set of reports:



Note that when you are on any page (not just **Report** pages), you'll see a small star symbol at the end of the page's title. If you click this link, then the current page will be added to your **Shortcuts**. For example, visit the **Status report** page now and click on the star symbol sign:



From now on, access to that page will be instantly accessible from your list of shortcuts:



Extend

Last but not least in our quick tour of the Drupal 8 UI, let's visit the **Extend** page which lists out all the modules that are currently available to your site: `core`, `contrib`, and `custom`, their versions and whether or not they are currently active.

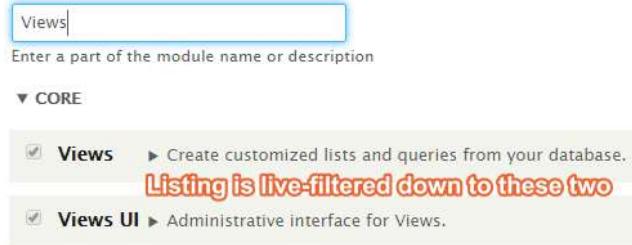
By default, when you first visit the page, all the module packages are expanded but if you click on any package heading, then the group will collapse.

Try this out now by collapsing all the packages:



The Drupal 8 modules page has a live, interactive search feature.

To try this out, type **Views** into the **Search** box:



Summary

This chapter introduced some key UI terminology and introduced you to the numerous ways that you can quickly customize the UI. If you find this quite a bit to take in at once, don't worry as each of the topics is covered in greater depth in forthcoming chapters.

In the next chapter, we will introduce a site-building scenario that will help you understand the different elements of Drupal 8 using real-life examples.

5

Basic Content

In this chapter, we will introduce the site-building scenario that we will use as a vehicle for learning throughout the rest of this module. Think of this as a project brief from a new client (except without a budget).

We will be building on this scenario in each of the future chapters, improving on your Drupal 8 knowledge as we do.

Introducing your site-building scenario

Your site-building scenario is to set up a simple website for your own small business which specializes in offering a number of Drupal-specific digital services.

Types of content

The simple website will include the following types of content:

Pages

The website comprises pages which contain rich text, embedded images, and links to other areas of the site.

Pages need to be directly linked from within the Main navigation menu.

Articles

The Article content type will be used to implement the general articles, blog, and news features on the site.

Whether or not a particular article node is deemed to be a general article, blog post, or a news item will be specified by using a Category Taxonomy of either 'Articles', 'Blog posts', or 'News'.

Additionally, articles can be free tagged with one or more words or phrases from an organic collection of categorizing terms or 'Tags'.

Articles can be illustrated with images.

Articles should be comment-enabled so that users of the site can contribute to threaded conversations much as one would expect in a forum. All comments by anonymous site visitors should be directed into a moderation queue before being published.

There is no requirement for site visitors to preview comments before submitting them.

News articles should all be accessible from a news listing page.

Likewise, blogs should be listed on the blog page and general articles on an articles page.

All three lists should be sorted with the most recently created at the top of the list down to the oldest at the bottom.

Clients

Client profiles should provide a summary description of each of your clients, a logo and some client contact details in the form of an e-mail address and a telephone number.

Additionally, you should be able to categorize each client within a work area.

The website should provide a dedicated page which lists all of your clients, grouped according to their work area classification.

Services

The site should provide a brief summary of each of the services you offer and you should be able to categorize each one in one or more work areas.

You should be able to optionally link each Service to one or more Clients.

Testimonials

The website should include a list of testimonials and each testimonial should be linked to an existing Client. There should be a dedicated page which lists testimonials grouped according to their work area classification and then sorted from newest to oldest with each group.

The testimonials should include the writer's name, job description, and company.

FAQs

The website should provide a page that lists Frequently Asked Questions (FAQs) grouped into work areas.

Contact information

The website should include a dedicated contact page where you can include details of your location as an address and a simple contact through which the public can submit enquiries.

The contact page should direct site visitors to e-mail addresses and telephone numbers for dedicated customer advisers.

We'll use the above scenario website as a vehicle for introducing new ideas and concepts from here on in this module and we'll illustrate how you might go about using Drupal 8 to achieve your ends step-by-step.

SEO considerations

The site should use Search Engine Optimized (SEO) URLs.

For example, when visiting a news item, the site should use a pattern like this:



Basic pages

In this chapter, we will focus on just the core content types: **Article** and **Basic page**.

We'll discuss step by step how to create content of these types and how to adjust their settings.

Let's imagine for argument's sake that you want to create the following pages:

- Home
- Our Services
- Our Clients
- Contact us

- About us
- News
- Blog

With this in mind, let's go about creating these pages and linking them into the Main Navigation menu.

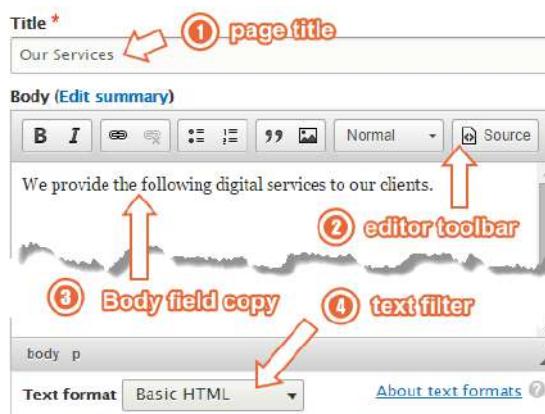
Right now, you are going to see how to create basic 'container' pages that come with nothing more than a single rich text area in which you can add your content. Later, in *Chapter 6, Structure*, you will see how you can include blocks within these pages too. For example, the news page should list out all of the news items.

We already have a home page link so let's start with the **Our services** page.

Creating a new page

To create a new Basic page, navigate to **Shortcuts** | **Add content** | **Basic page**.

On the left-hand side, we have the main area, which is where you get to populate the various fields of the basic page content type:



Title

Enter Our Services in the **Title** field.

Body

Enter some introductory text as shown earlier, but don't actually save the content yet.

Body field summary

You may have noticed the **Edit summary** link just to the right of the **Body** field's label.

Understanding the significance of this with respect to good **Search Engine Optimization (SEO)** is quite important, so let's look for a moment at how the summary gets used when the content is presented in a summarized (teaser) form.

When Drupal provides information in a summarized format, it applies the rules:

- If the Summary of the Body field is populated, then Drupal will use the content intelligently, truncating it at a predetermined length if it is too long
- In the absence of a populated Summary, Drupal will fall back to intelligently truncating the Body content itself.

To illustrate this in action, edit the **Our Services** page and add some summary text:



Then, even though the Body text is populated, the teaser would appear:



You won't be able to see this action for yourself unless you have covered either manually promoting content to the front page in *Chapter 4, Getting Started with the UI* or done the Views work in *Chapter 6, Structure*, but for now, take my word for it that it's always worth taking time out to populate these Summary elements with concise descriptions.

Rich text toolbar

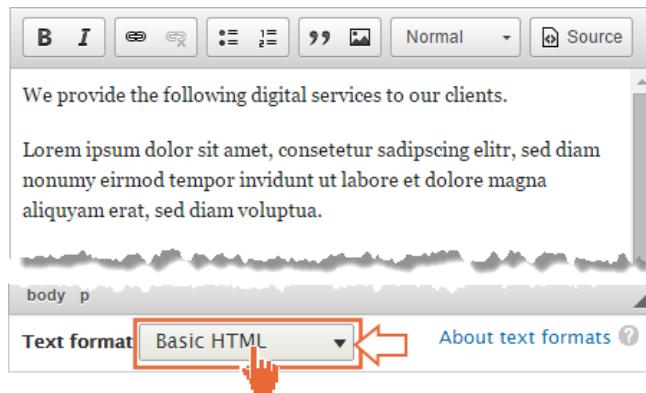
Drupal 8 employs a WYSIWYG (What You See Is What You Get) JavaScript editor to provide for a rich text editing toolbar.

Default HTML restrictions

Note that by default, Drupal 8 ships with the Basic **Body** field configured to filter the rich text according to a set of rules limiting the HTML tags. The allowed tags are those defined in the **Basic HTML** Text format.

Note how the WYSIWYG editor toolbar for this particular text format is also equipped with only the right set of icons to match the restricted set of HTML tags.

The **Basic HTML** text format provides a basic toolbar:



One of the default rules restricts the possible tags to those in the following list:

```
<a> <em> <strong> <cite> <blockquote> <code> <ul> <ol> <li> <dl> <dt>
<dd> <h4> <h5> <h6> <p> <span> <img>
```

Full HTML

By contrast, this text format provides the full set of HTML tags to be used and therefore comes with a much busier toolbar.

Restricted HTML

This restricts the user to the same list as for Basic HTML, but excludes the `` and `` tags. It also removes the rich toolbar.

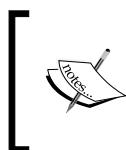


Try changing the Text format between the available options, both with and without text in place to see various messages and their effects on the **Body** field.

In either Basic HTML or Full HTML, you'll also note that one of these toolbar icons is the **Source** switcher, which allows you to switch out of the rich editing mode and instead manipulate the HTML directly:



One should not be misled here into thinking that if you switch to the **Source** editing mode, you are no longer restricted in terms of the HTML tags that you can use. This is not so; Drupal actually performs the filtering on output not the input so that you can actually enter any old HTML into the **Body** field while in Source mode, but only the allowed tags list (above) will actually be output.



Configuring the application of different text formats to different content types is a key feature for avoiding the situation where clients may have too much freedom to enter custom HTML, which may well break the website's layout.



Adding a page to the main navigation menu

Next, cast your eye over to the advanced editing area on the right-hand side where you will find an expandable field set entitled **MENU SETTINGS**.

Expanding this area will enable you to enter a title for the menu entry:

The screenshot shows the 'MENU SETTINGS' configuration form. It includes the following fields and their values:

- Provide a menu link**: An input field with a checked checkbox, indicated by a red arrow and the text "① tick this".
- Menu link title**: A text input field containing "Our Services", indicated by a red arrow and the text "② Title is used by default".
- Description**: A large empty text area.
- Parent item**: A dropdown menu set to "<Main navigation>", indicated by a red arrow and the text "③ Main navigation by default".

Ensure the **Provide a menu link** checkbox is selected, and then enter a title—Our Services. Make a mental note that the default **Parent item** is automatically set to <Main navigation> and then **Save and publish** the page:



You should now be viewing your finished page and see that it is conveniently displayed in the Main navigation menu.



Nodes and unique IDs

All content on a Drupal website is stored and treated as "nodes". The node is a terminology that has persisted throughout Drupal's long history. It is less important in Drupal 8 where you will hear discussion of "entities" more often than nodes. Suffice to say that a node is any posting of content such as a Basic page or Article.

By saving the new Basic page, you have just created your first node. In fact, assuming that you are following this tutorial guide to the letter, you have just created a node with a unique ID of 3 because you created nodes 1 and 2 back in *Chapter 4, Getting Started with the UI*.

Hover over the **Edit** tab (2) and glance at your browser's status line where you should be able to see the **Edit** link `drupal-8.dd:8083/node/3/edit`.

Note that the pattern of this URL ends with `/node/ [nid] /edit` where `[nid]` is the unique **node ID** of the node you have just created.



The node ID is often shortened to `nid` in Drupal parlance. You will see references to `nid` repeatedly during your Drupal journey.

Adding more pages

Next, using the same techniques, add the following further Basic pages to your site, each with a small amount of filler text and a Main navigation menu item:

- Our Clients
- Contact us
- About us
- News



Note that Drupal 8 automatically orders the menu items in alphabetical order, but also note that you can easily rearrange these as you will see in *Chapter 6, Structure*.

So, you now have a working website with five pages; not overly impressive, but a good start.

Let's look at Articles again to see how they differ from Basic pages.

Articles

The only other Content type that is available out of the box with Drupal 8 is the Article.

We touched on this content type earlier in *Chapter 4, Getting Started with the UI*, but that was only in a simple example. This time you'll look at them in more detail.

Creating a new article

To create a new Article, navigate to **Shortcuts | Add content | Article**.

Let's look at the extra fields you get with an Article that didn't appear when we created Basic pages earlier. You'll create an article about the company pet as a vehicle for investigating what's different about Articles nodes.

Basic Content

In addition to the **Title** and **Body** fields, the Article caters for categorizing the article content using one or more "tags", and it also provides an image field so that you can attach an image to the article.

Add a **Title** and some **Body** text:

Title *

The Company pet  ① **Title**

Body (Edit summary)

Meet Yukí, our company pet.
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

 ② **Body field**

Tag the article with a couple of keywords:

Tags

pet, dog  ③ **two taxonomy 'tags' added**

Enter a comma-separated list. For example: Amsterdam, Mexico City, "Cleveland, Ohio"

Finally, upload a suitable picture (the exact one here is available in the module resource ZIP file), and add an image with accompanying Alt-text for accessibility.

Note the alt text is compulsory forcing you to follow accessibility best-practices.

Image



Alternative text *

Yuki, the company pet

This text will be used by screen readers, search engines, or when the image cannot be loaded.

 chapter_5_01.png (210.22 KB)  Remove

Click on the **Save and publish** button to publish the article, and it should appear:



Note some key points about Articles compared with Basic pages:

- The author and date information is displayed at the top of the page
- The image is automatically resized from the original – more about how in *Chapter 8, Configuration*
- Tags are clickable, and will take you to a page of similarly tagged content

Now create a second article with some similar content, and tag that with the word **dog**.

Having saved the second article, try clicking on the **dog** tag at the bottom of the page:



You should see that Drupal takes you to a listing of **teaser** (shortened) versions of all content that is tagged with the same tag, with the most recently created at the top:

The screenshot shows a search results page for the tag "dog". The first result is an article titled "Dog fish(er)" submitted by "admin" on "Wed, 12/16/2015 - 16:48". The article thumbnail shows a dog. To the right of the thumbnail, a red box highlights the text "a 'teaser of the most recent article with its 'Read more'" and a "Read more" link. Below the article, under "Tags", are "dog" and "fish". At the bottom, there are "Read more" and "Add new comment" links. The second result is an article titled "The Company pet" submitted by "admin" on "Wed, 12/16/2015 - 16:20". Its thumbnail is partially visible.

Note that the **Edit** tab located at the top of the page is an edit shortcut to the dog tag itself and not one for the first article in this. You might like to click on the **Edit** link just to see this for yourself:



In this simple example, you can already see the enormous power inherent in being able to display categorized content in this way.

Thus, you can see that Drupal is already proving its worth by providing a useful summary of all content throughout the site, which is similarly tagged. Imagine if, instead of just these two articles about pets, you had a large range of clothing products tagged with "shoes".

Front page promotion

Visit your site home page now by clicking on either the **Home** link in the Main navigation menu or the site logo:



You may be forgiven for thinking that you are looking at the same view as before, that is, a teaser listing of everything tagged with **dog**. However, what you are actually looking at is a teaser listing of all articles, at least all articles that have been **Promoted to front page**.

So why are we seeing all these articles? The answer is that, by default, the Article content type is **Promoted to front page** and the default Drupal front page displays up to ten of them in reverse order of creation date and time.

Demote your **Dog fish(er)** article. Remember that in addition to the **Edit** tab, you can use the contextual links for quick editing:



Locate the **Promotional options** in the advanced editing area and un-tick **Promote to front page** (2) before resaving the Article node (3).



Your **Dog fish(er)** article will no longer appear as a teaser on the front page.



I'll remind you that you can also demote multiple items of content from the front page at `admin/content` using bulk actions as you saw back in *Chapter 4, Getting Started with the UI*.



Adjusting the settings for a content type

Drupal gives you complete control over the default settings for each content type, and we'll now illustrate how to make site-wide adjustments to the behavior of any of them quickly and easily through the administrative interface.

Disabling front page promotion

While it is a useful default feature that Drupal automatically promotes Article nodes to the front page, let's imagine that we have a plan to provide a list of article teasers another way (almost certainly using Views) and so wish to turn off the automatic facility.



Note that once we have this setting adjustment in place, it will only affect the behavior of all new articles.



Basic Content

To adjust the settings for the content type as a whole, locate the **Content types** link from the **Structure** page.



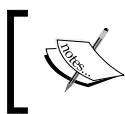
Next, choose **Edit** to edit the **Article** content type as a whole by dropping down the **Operations** menu:

The screenshot shows a table listing the 'Article' content type. The 'NAME' column shows 'Article', the 'DESCRIPTION' column says 'Use articles for time-sensitive content like news, press releases or blog posts.', and the 'OPERATIONS' column contains a dropdown menu. A red circle labeled '③' points to the 'Edit content type the settings' link in the operations menu. A red arrow points from this link to a red circle labeled '④' with the text 'Edit content type the settings'. The dropdown menu is open, showing options: 'Manage fields', 'Manage form display', 'Manage display', 'Edit' (which is highlighted with a red hand cursor), and 'Delete'. A red circle labeled '⑤' points to the 'Edit' option in the menu.

In the following dialog, locate the **Publishing options** section and uncheck the **Promote to front page** option and save the settings:

The screenshot shows the 'Submission form settings' and 'Publishing options' sections. In the 'Publishing options' section, the 'Published' and 'Promoted to front page' checkboxes are checked. A red circle labeled '④ un-tick "Promoted to front page"' points to the 'Promoted to front page' checkbox, which has a red hand cursor icon. Below the checkboxes, there are other options: 'Sticky at top of lists' (unchecked) and 'Create new revision' (unchecked). At the bottom, there are two buttons: 'Save content type' and 'Save the settings'. A red circle labeled '⑤ Save the settings' points to the 'Save the settings' button.

You have now configured your site such that all the newly created articles are no longer automatically promoted to the front page.



Note that the same set of default content type settings are available for adjustment for all content types such as core, contributed, and your own custom types, as you will see later in *Chapter 7, Advanced Content*.

Adjusting comment settings

Let's now adjust the settings for the Article content type so that articles do not, by default, have comment forms attached to them. This will involve your first visit to the field management page for the article content type – a key area to get to grips with when configuring the various types of content on your Drupal site.

We have two distinct possibilities:

- Keep any existing comments, but disable all comments on the newly created content
- Remove the ability to leave comments for all Articles and remove all comment data

The first involves adjusting the settings for the **Comment** field, and the second involves the complete removal of the **Comment** field from the Article content type.

Either way, to adjust the settings for the content type, again locate the **Content types** link from the **Structure** page by navigating to **Manage | Structure | Content types**.

Next, choose **Manage fields** for the Article type:

NAME	DESCRIPTION	OPERATIONS
Article	Manage the fields Use <i>articles</i> for time-sensitive content like news, press releases or blog posts.	Manage fields

You will be taken to a page that shows the different fields that make up the Article content type.

Disabling future comments

In the first example, you'll adjust the settings of the **Comment** field so that no further comments can be added to the article nodes; this will have the effect of removing the comment form from all newly created articles, but will leave it on any pre-existing nodes. Comments that already exist will be retained.

Basic Content

Locate the menu on the right-hand side of **Comments** and click on the **Edit** option:

LABEL	MACHINE NAME	FIELD TYPE	OPERATIONS
Body	body	Text (formatted, long)	
Comments	comment	Comments	(1)
Image	field_image	Image	
Tags	field_tags	Entity reference	

You will be taken to the **Comment** field settings page.

Locate the **Default Value** field settings about half way down the page and click on the radio button labeled **Hidden**.

▼ **DEFAULT VALUE**

The default value for this field, used when creating new content.

Comments

Open
Users with the "Post comments" permission can post comments.

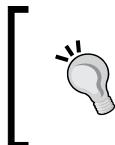
Closed
Users can't post comments, but existing comments will be displayed.

Hidden
Comments are hidden from view.

Save the settings.

No retrospective action

It's important to be clear that having hidden comments in the default field settings for the article content type, you will no longer see the comment form on *new* articles.



Existing articles, comment settings are unaffected by changes to the default field settings. So, the existing comments and more importantly, the comment entry form will still show any existing articles enabling user to continue adding comments and replying to existing ones.

Retrospective action

By contrast, the other settings found in the **COMMENT FORM SETTINGS** area do apply retrospectively; these are:

▼ COMMENT FORM SETTINGS

Threading
Show comment replies in a threaded list.

Comments per page *

Show reply form on the same page as comments

Preview comment
 Disabled
 Optional
 Required

Threading

Comment **threading** means that replies to comments and indeed replies to replies will all be indented so as to show visually how the individual threads of conversation have progressed:

Comments

With comment threading ON

admin
Wed, 12/16/2015 - 19:14
[Permalink](#)

Here is a top-of-thread comment

Lorem ipsum dolor sit amet, consetetur sadipscing elitr.

[Delete](#) [Edit](#) [Reply](#)

indented (threaded)

Wed, 12/16/2015 - 19:15
[Permalink](#)

Reply to a top-of-thread comment

Lorem ipsum dolor sit amet, consetetur sadipscing elitr.

[Delete](#) [Edit](#) [Reply](#)

Thus, if you uncheck the **Threading** option, then replies to comments will no longer be indented so as to show the various conversation threads.

Show reply form on the same page as comments

This option controls whether the comment entry form is automatically attached to the bottom of every comment-equipped article or whether it instead presents a simple text link:



Once un-ticked and the settings saved, the full comment form will no longer show and will be replaced by a much neater **Add new comment** link:



Use the preceding idea now to configure Articles to display the simple link as shown in the preceding screenshot.

Preview comments before posting

Similarly, you can opt to allow your site users to preview their comment(s) before posting them, and you can even force them to preview before they can post:



Removing all comments and the ability to comment

We are not actually going to do this now, but if we have no intention of using comments on articles on the site and we don't mind removing any existing comment data, then another approach is to remove the **Comments** field altogether.

You can remove the field from the field management page by clicking on the down arrow next to the **Edit** operation and selecting **Delete**.

LABEL	FIELD TYPE	OPERATIONS
Body	Text (formatted, long, with summary)	Edit ▾
Comments	Comments	Edit Storage settings Delete 
Image	Image	

Note that the action of removing the field is permanent and cannot be undone; you will lose all the comment data associated with articles, so use with care in real projects.

Are you sure you want to delete the field **Comments**? 

[Home](#) » [Administration](#) » [Structure](#) » [Content types](#) » [Article](#) » [Manage fields](#) » [Comments](#)

This action cannot be undone.

Delete

Cancel

It's fine to do so; you can always add the field back in later if you change your mind about allowing comments on the Article content type.

Summary

In this chapter, we introduced the basic plan for your new site and we started creating real pages and articles. We looked in some detail at the key features of the fields available within the Basic page and Article content types, and in particular, how to adjust the settings on the comment field.

Equipped with these new skills, you should be ready to start creating new content types and customizing them exactly as you need to implement Clients, Services, Testimonials, and FAQs on your new site in *Chapter 7, Advanced Content*. First, let's look into some more of the key structural elements of Drupal: Menus, Taxonomy, Views, and Blocks in the next chapter.

6 Structure

In this chapter, we will be looking at organizing the structure of content on your Drupal website. To start with, we'll look at menus before investigating taxonomy in more depth than in our earlier introduction. Finally, we will take a look at creating lists of data as pages and blocks using the `Views` module.

Managing menus

The representation of the structure of your site to a user will often be via menus. Here we will show you how to manipulate these menus so that you can help a user navigate your site.

Reorganizing menu items

As it stands, our **Main navigation** menu items are listed in alphabetical order:



However, we would like the items in the following order:

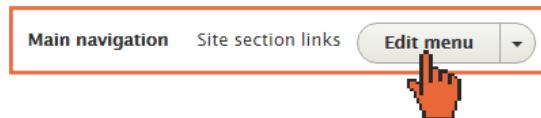
Home | Our Services | Our Clients | About us | News | Contact us

To edit the menu, visit the Menu management at **Manage | Structure | Menus**:



Structure

You will see a list of all the currently defined menus; try editing the Main navigation menu:



Use the 'drag handles' to reorder the menu items:

A screenshot of the 'Edit menu' interface. It shows a table with three rows of menu items. The first row has a 'drag & drop' handle highlighted with a red oval. An orange curved arrow points upwards from the bottom of this handle towards the second row. The second row also has a 'drag & drop' handle. The third row contains a button labeled 'Drag to re-order'. At the bottom left is a blue 'Save' button, which has a hand cursor pointing at it.

MENU LINK	ENABLED	OPERATIONS
Home*	<input checked="" type="checkbox"/>	Edit
About us	<input checked="" type="checkbox"/>	Edit
Drag to re-order	<input checked="" type="checkbox"/>	Edit

Rearrange the order and save so that your **Main navigation** menu appears like this:



Editing menu items

Earlier, we made a dedicated **Contact us** Basic page and we linked it into the **Main navigation** menu at the same time. To illustrate the idea of editing menu items manually, let's point this menu item to the existing Contact form that was automatically provided to us by the Contact module instead of the Basic page.

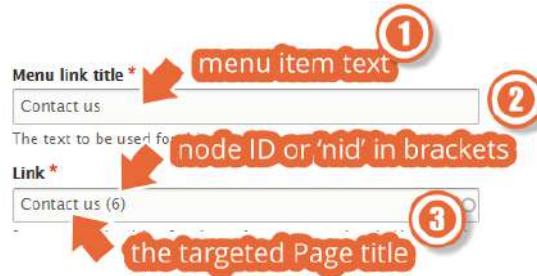
Even though the menu item was created at the time of creating the page, we can easily overwrite the location that it points to. You may remember from *Chapter 4, Getting Started with the UI*, that the Contact module provides a contact form at <http://drupal-8.dd:8083/contact>.

Visit the Menu management page again by navigating to **Manage | Structure | Menus | Main navigation**.

In order to have the Contact us menu item point to the form instead of the Basic page, all you need to do is manually edit the menu item by clicking on the **Edit** button in the **OPERATIONS** column.



The **Menu link title** is simply the text attached to the menu item (1) but the **Link**, in our case the **Contact us** Basic page, is neatly disguised as the title of that targeted page with the `nid` in brackets afterward:



Edit the item's **Link** replacing the existing reference to your Contact page (the Basic page with a node ID of 6) with the replacement path of `/contact`, which is provided by the Contact module.

To manually enter a relative path such as this, precede the path with a forward slash (/).

Save your changes.



Since you are no longer linking the Main navigation menu to the Basic page you created in *Chapter 4, Getting Started with the UI*, we could remove that page but in fact, we'll keep it and use it later to demonstrate some other features.

Managing taxonomy

In *Chapter 3, Basic Concepts*, we briefly discussed Drupal's Taxonomy feature.

In *Chapter 4, Getting Started with the UI*, you created Article nodes tagged with the terms: `tag1`, `tag2`, `tag3`, and `tag4`, and in *Chapter 5, Basic Content*, you created more Article nodes tagged with the terms `pet` and `dog`.

The **Tags** vocabulary is a **tag-based vocabulary**, which means that content editors can make up as many terms (words or phrases) as they like and will be prompted to re-use any existing terms as they type. Tag-based vocabularies are great when you need your content-editing team to use their expert subject knowledge to build up a collection of terms, but the downside of them is that the collections can fill up with duplication owing to misspellings.



Taxonomy vocabularies based on a fixed set of terms are often more useful. What we will do next is set up a fixed vocabulary to help us to categorize all the content (pages, services, clients, and news) across the site.



Creating a new Taxonomy vocabulary

We'll create the following collection of seven terms that we can use to categorize all content:

- Charities
- Consultancy
- Documentation
- E-Commerce
- Government
- Fun
- Training

We'll refer to this as the *Category* vocabulary.

To create a brand new taxonomy vocabulary for our new list of terms, visit **Structure** | **Taxonomy** | **Add vocabulary**.



Enter a name and description for the new vocabulary and save it.

Name *

Machine name: category

Description

Save

Click on the **Add term** link and start entering all the terms in the list.

Note that whilst we have done so in the screenshot below to illustrate the feature, you do not need to enter an administrative **Description** for each and every term.

+ Add term

1

Name *

The term name.

Description

2

3

This term should be applied to any and all items of content that focus in any way on working with registered charities.

Enter the full list of terms as follows:

- Charities
- Consultancy
- Documentation
- E-Commerce
- Fun
- Government
- Training

Re-ordering Taxonomy terms

Since you have only just populated the vocabulary from scratch, you will find that the terms in it are already arranged in alphabetical order.

It is a common requirement to list out taxonomy terms in a precise curated order, and so you can simply drag and drop the various terms until you have what you want.

Visit the **Manage | Structure | Taxonomy** page and click on the **List** tab.

NAME	drag & drop	OPERATIONS
+ Charities		Edit
+ Consultancy		Edit
+ Documentation		Edit
Drag to re-order		Edit
+ Fun		Edit

Save **Reset to alphabetical**

Apply a taxonomy vocabulary to content types

Now that you have defined the new list of terms with which to classify the website content, the next step is to associate the vocabulary to your existing content types: Basic page and Article. To do this, we need to add a field to each of those content types. Let's first attach the vocabulary to the Article content type.

Go to **Manage | Structure | Content types** then click on the **Manage Fields** button in the operations dropdown for the Article content type:



Add a new **Reference | Taxonomy term** field called **Category** and **Save and continue**:



You will then be taken to the first of two field configuration pages.

Let's say that we want to allow the content editors to have the option to apply multiple terms, so select **Unlimited** as the first option, and then make sure that the field is linked up to the correct (Category) taxonomy vocabulary:



Finally, you'll be taken to the second field configuration screen where you can optionally make the field mandatory, set any default value(s), and finalize the settings such as whether to limit the vocabularies you wish to use or to set any default terms.

At the bottom of the configuration screen, use the **Default** reference method and make sure that only the **Category** vocabulary is selected:

The screenshot shows the configuration screen for a 'Category' field. Step 8 highlights the 'Label' input field with the value 'Category'. Step 9 highlights the 'Required field' checkbox. Step 10 highlights the 'DEFAULT VALUE' section with the 'CATEGORY' vocabulary selected. Step 11 highlights the 'Reference method' dropdown set to 'Default' and the 'Vocabularies' section with the 'Category' vocabulary checked. Step 12 highlights the 'Save settings' button.

Label *
Category
Help text

8 Optionally adjust the label

9 Required field

10 Set one or more default terms

11 Limit the selection to only this vocabulary

12 Save the field settings

Leave **Create references entities if they don't already exist** unticked, otherwise new terms will be created in this vocabulary and it will essentially become a tag type vocabulary when this is selected.

When creating and editing articles, your content-editing team will now be able to categorize their content.

Before we test this out, let's first use the opportunity to visit another field configuration page again – that of **Manage form display**.

Adjusting the order of fields when editing

We touched on this in *Chapter 4, Getting Started with the UI*; now let's look at it in a bit more detail. As a reminder, when we use the word **Form** in Drupal in relation to editing content, we are referring to the **Edit form**, that is, the actual screen which the content editors use to edit content.

Visit the **Manage form display** screen for the Article content type. Click on **Manage form display** from the operations dropdown:



The most intuitive widget that we can use would be a **Select list**, so let's configure the field to use that when in the edit form.



You should leave the **Tags** field widget set to the default **Autocomplete** since that is the only really sensible open for a tags vocabulary.

It is often a good idea to bring Taxonomy term fields to the top of the list just underneath the Title to encourage content editors to think about classification early on when they are creating new content.

Thus, re-order the fields, as shown in the following, so that the two Taxonomy fields are positioned just below the **Title**.



Categorizing content

Now that we have set up a link between the Article content type and the Category taxonomy, we can set about categorizing Articles.

To categorize your four existing articles, simply edit them and choose one or more terms using *Ctrl* within Windows or *Command* on OS X, and then save.



Categorize all four existing articles now for example, The company pet and Dog fish(er) articles might reasonably be categorized as Fun and so on.

Create five new articles now in a variety of categories.

You may find it useful to copy the list of titles that we have used in this tutorial guide:

- An article classified with Charities
- An article classified with Consultancy
- An article classified with Charities and Documentation
- An article classified with Consultancy and E-Commerce
- An article classified with Consultancy, Government, and Training

Note that we have not added any images to the example articles created in this guide, but you, of course, can.

Viewing categorized content

Visit one of your articles now in full and you will see that any **Category** terms that you have applied appear as links at the bottom of the page:

An article classified with Consultancy, Government, and Training

[View](#) [Edit](#) [Delete](#)

Submitted by [admin](#) on Thu, 12/17/2015 - 11:05

Category
[Consultancy](#)
[Government](#)
[Training](#)   **Click one of the taxonomy terms**

Clicking on one of these links will again take you to a listing of all content with that term applied. Not bad at all, but now let's see how we can use taxonomy to provide further improved listing.

Segregating article types using taxonomy

So far we have used taxonomy to group together articles whose actual content is concerned with key terms such as Charities, government, and Training. Now let's investigate how we can use exactly the same methods to distinguish between three different *types* of Article: Articles, News, and Blog posts.

Creating another taxonomy vocabulary

We are now going to subdivide Article nodes into three distinct types: Articles, Blog posts, and News. To do this, we first need to create a new Taxonomy vocabulary.

Visit [Manage | Structure | Taxonomy](#).

Create a new vocabulary called `Article type` containing the following terms:
Articles, Blog posts, and News:

NAME	OPERATIONS
 Articles	Edit 
 Blog posts	Edit 
 News	Edit 

Structure

Next, in order to enable you to classify Article nodes with these terms, add another **Taxonomy term** field entitled **Article type** to your Article content type so as to attach this new vocabulary. As you do, set the **DEFAULT VALUE** of the field to **Articles** so as to classify all newly created articles as Articles by default:

Home > Administration > Structure > Content types > Article > Manage fields

Add a new field

Type: Taxonomy term  **①**

Label * Article type  Machine name: field_article_type [Edit] 

Leave the Limit set to 1 since an Article should only be of a single type:

Type of item to reference * Taxonomy term 

Allowed number of values Limited 1  **③**

Adjust the default value to be Articles:

▼ DEFAULT VALUE
The default value for this field, used when creating new content.

Article type Articles (15) 

Lastly, limit the choice of terms to the **Article type** vocabulary and save the settings a final time:

▼ REFERENCE TYPE

Reference method * Default 

Vocabularies *    

Create referenced entities if they don't already exist

Save settings  **⑥**

Set the widget to **Select list** and adjust the order of the fields in the **Manage form display** page to bring the new Taxonomy field close to the top of the page.



Now go through each of your existing articles further classifying with an Article type.

You may find it useful to copy the classification that we have used in this tutorial guide:

- An article classified with Charities [News]
- An article classified with Consultancy [News]
- An article classified with Charities and Documentation [News]
- An article classified with Consultancy and E-Commerce [Articles]
- An article classified with Consultancy, Government, and Training [Articles]
- The company pet, Dog fish(er), Article no. 1, and Article no. 2 [Blog posts]

Test your new classification by clicking on one of the three terms to see a nicely segregated listing showing only Articles, News, or Blog posts.

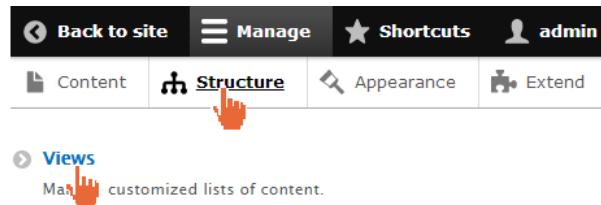
In the next section, you will see how you can improve significantly on this listing by controlling the order, grouping, and presentation of the articles using the **Views** module.

Working with the Views module

You are now in a good position to start investigating how to use the **Views** module to create carefully filtered and organized listings of content that are organized just as you want.

Creating a Views-powered News page

To create a new View, visit the **Manage | Structure | Views** page:



You will see a list of all the existing views, some of which are active and some disabled:

A screenshot of the 'Views' administration page. At the top, there are tabs for 'List' and 'Settings'. Below that is a breadcrumb trail: 'Home > Administration > Structure'. A blue button labeled '+ Add new view' is visible. There is also a search bar with the placeholder 'Filter by view name or description'. The main content area is titled 'Enabled' and contains three rows of view details:

VIEW NAME	DESCRIPTION
Content	Displays: <i>Page</i> Find and manage content. Machine name: content
Custom block library	Displays: <i>Page</i> Find and manage custom blocks. Machine name: block_content
Files	Find and manage files.

View wizard

The view wizard will help you create new views more quickly by guiding you through the common setup steps.

Creating a new view

Next, click on the **+Add new view** button and you will be taken into the Views wizard. We'll keep this explanation simple to start with by concentrating only on the elements of this screen that are essential to build the News page view.

Later in this chapter, we will go into more detail about the finer points of the Views wizard. Name the new View **News** and filter the content to the **Article** type.

VIEW BASIC INFORMATION

View name *
News ② Machine name: news [Edit]

Description

VIEW SETTINGS

Show: Content of type: All ③
All
Article

PAGE SETTINGS

[ At this point, you may be forgiven for thinking that you can also set the Tagged with field to narrow down the News page to only those Articles that are really News items because they are marked with the News term from the Article type taxonomy list and you'd be quite right. However, while the wizard is a simple starting point within this wizard, the only taxonomy vocabulary that Views is aware of at this stage is the core Tags vocabulary, and that won't do in our case.]

We want to create a full page that is accessible via a unique URL path, so check the **Create a page** checkbox and check and adjust the **Page title** and a **Path** if you need to:

PAGE SETTINGS

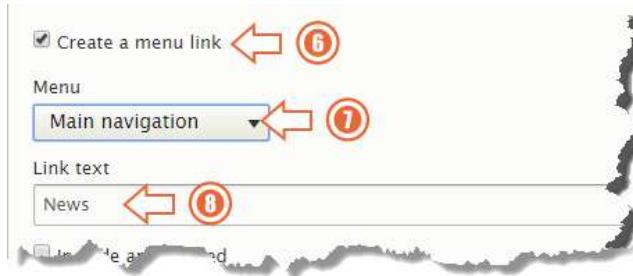
Create a page

Page title
News ④

Path
news ⑤

Structure

We want the News page to be accessible from the **Main navigation** menu, so also check the **Create a menu link** item and direct the link to the **Main navigation** menu:



Click on the **Save and edit** button:



You will be taken into the Views editing page showing the view with one display entitled **Page**.

Renaming a display

It's worth noting at this point that Views can actually contain any number of different types of **Displays**, but more about that later in this chapter when we add a block display.

For now, though let's specifically rename the display that we have just created so that it is more descriptive. Click on the current **Display name** and rename it to **News list as a Page** so that it's very clear what it does.



The newly revised display name will replace the original one in the list:



Live preview

Scroll down the editing page and you'll find a live preview currently listing out all the published Article nodes. We have not yet filtered the list down to only those marked as News items, so this is correct.

Content settings

Before adding the finishing touches, let's take a look at some key **Format**, **Filtering**, and **Sort** settings that have been set up by the View wizard.

1. Firstly, scroll back up and look at the Format settings and note that we are displaying items in their currently defined Teaser view mode.
2. There are two Filter criteria currently applied limiting the listed items to only **Articles** that are **published**.
3. The listed items are sorted in reverse order of **Authored on** (creation) date.

Page settings

It's worth noting at this point that if you made an error or an omission when in the wizard, you can re-adjust the path settings and, therefore, change the actual URL through which your site visitors will view the News page.

You can also adjust the **PAGE SETTINGS** for the menu item used to provide access to page and to change the actual Drupal menu in which that item is contained.



There are other useful and interesting settings here too such as the ability to control precisely who can see the results of the View based on permission(s) or role(s), and we'll cover those later.

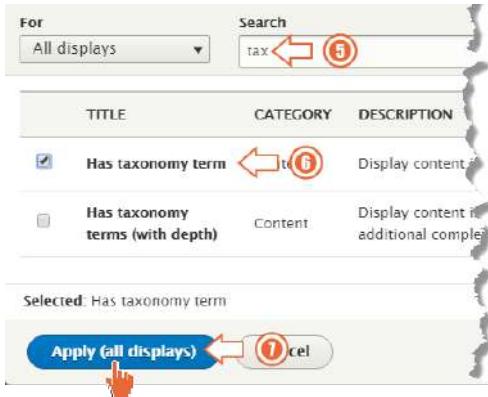
Filtering to News only

The only thing we need to add now is to further filter down the list of Articles to only those that have their **Article type** field (a Taxonomy reference) to **News**. The following are the steps to achieve this.

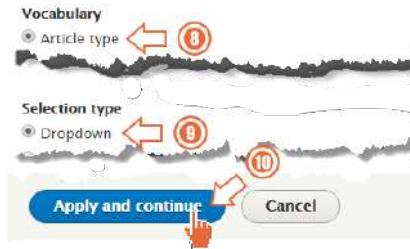
1. Add a new **Filter criterion** to the view:



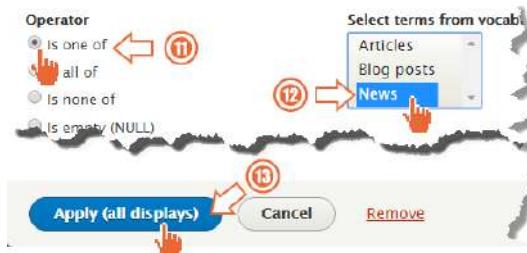
2. Enter tax in the **Search** input in order to filter down the list of possible criteria to only those relating to taxonomy.
3. Select **Has taxonomy term**.
4. Click on **Apply (all displays)**.



5. Next, in order to target the specific vocabulary that contains the News term, choose the **Article type** vocabulary, opting for a Dropdown widget for convenience. Click on **Apply and continue**.



6. From the list of terms within the Article type vocabulary, choose **News** in order to filter out only those Article nodes that are marked as news items. Click on **Apply to all displays** once more and then click on **Save** (between the main editing area and the preview) to finalize the whole view.



7. Finally, after you have been returned back to the underlying Views edit page, hit the Save button at the bottom left.



8. Return to the Home page and you should see two **News** pages accessible from the **Main navigation** menu: the first being the original **Basic page** that you added back in *Chapter 5, Basic Content*, and the new Views-powered **News** page that you have just created.



9. Click on the second News tab to see the Views-powered page in action. You should see a list of all Article nodes that have been categorized as "News".

Don't be tempted to remove the original Basic page from the menu right now because you are going to use it later to help illustrate the usefulness of creating Blocks using the **Views** module.

Creating a blog page

In this exercise you will create a page that lists blog posts out as teasers.



The steps to achieve this are straightforward and exactly as per the *Creating a Views-powered News page* section, but let's just recap on them as revision.

We already have some posts categorized as Blog posts.

Create a new page view entitled **BLOG** that lists only Articles with the list filtered down those which are marked as **Blog post**.

If you are unsure at this point, then please look back at the **Creating a Views-powered News page** section for a step-by-step recap on building the News page.

Working with Views blocks

In the previous section, we created two entirely separate Views-powered pages to list all the News items and Blog posts, that is, those Article nodes that had been marked as News or Blog posts. As an illustration of the ease with which you can create separate blocks for content lists using the Views module, let's now create a block version of the news listing and place that block on the Basic page entitled News that you created in *Chapter 5, Basic Content*.

Creating blocks using Views

Locate the newly created Views-powered News page and hover your mouse pointer near the top of the view's output. This should reveal the contextual link for the view as a whole, allowing you to edit the view quickly and easily without even knowing its name.



Create a block version of the existing display by clicking on the **+Add** button and choosing to add a new **Block** Display.



Click on the new **Display name:** label entitled 'Block'



Rename the **Block** to **News list as a Block** as you did earlier:



Since this is a Block display and not a page that can be visited via a URL path, it does not have any Path settings but instead has some block-specific settings in the central region of the dialog. In order for the block to be visible to the block placement system, you must name the block specifically, so click on the word **None** (next to block name) now.



Set the name of the block to **News listing block** and click on **Apply**:



Save the view before continuing.

You have now created a new block and it will be available for placement on the Block layout page that you first saw back in *Chapter 4, Getting Started with the UI*.

Let's look again at how to do just that.

Placing the News blocks

The overall goal here is to place the new News block onto the existing News page immediately underneath the Body field.

Go to **Manage | Structure | Block layout**.

Locate the **Content** region and click the **Place block** button.



Scroll down the page to locate the newly-created block entitled **News listing block**.



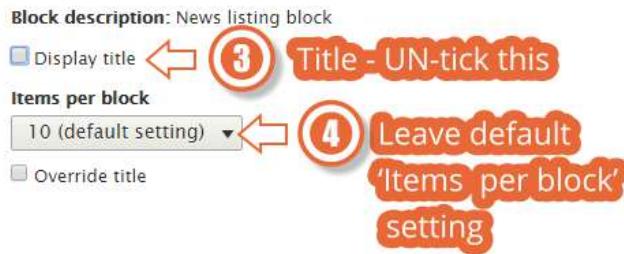
Block position

Let's place the block on the News page.

You'll concentrate only on the key points for now, which are essentially where, how, and when we want the News listing block to appear in simple steps.

We don't want the Title showing because we already have a title on the news page, so unclick **Display title**.

The default option to view 10 news items at a time seems sensible so leave that alone.



Block visibility

So that's the how and the where dealt with, we now need only to tell Drupal when we want that block to be displayed, and the criterion for that is that we are viewing the News page which, if you've been following along, will be node/7.

So, the two visibility settings that need adjusting within the collapsed **Pages** field group so as to make the block is only visible on the News page are:

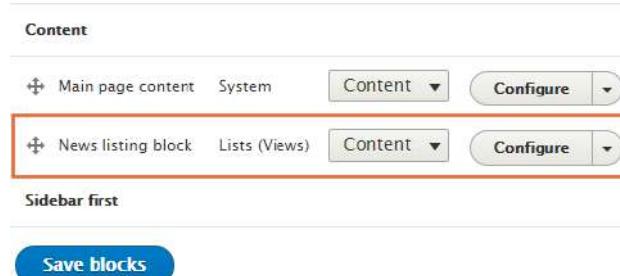
Click on the left-hand side **Pages** section.

Enter /node/7.

Select the **Show for the listed pages** option:



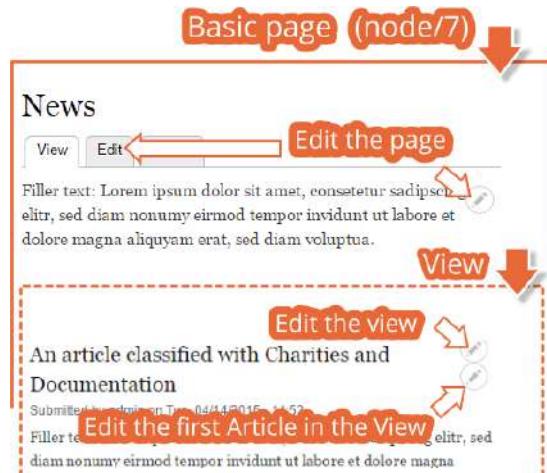
You can now re-position the **News** block directly underneath the existing Main content block and then click on **Save blocks**:



You should only see the **News** listing block when you are actually viewing the original News page (node/7) that you created back in *Chapter 5, Basic Content*.



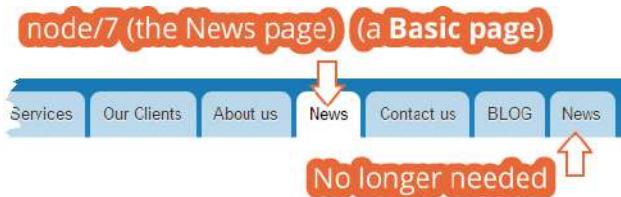
The visible "page" now comprises the original **Basic page** and the **View** block.



Note that you can see multiple contextual links on the overall visible page: one for the Basic page, one for the View, and one for each News article.



Now that you have placed the **News** listing block on the **News** page, we no longer need the second menu item that was provided by the **News** Page display within the **News** View.



Structure

This is a good opportunity to re-visit the menu management screen and to see how, rather than deleting the item, you can just temporarily disable it instead. Likewise, we could actually remove the Page display from the News View but, again, leaving it in place will prove very useful later in *Chapter 7, Advanced Content*, when we touch on some of the more advanced features of the Views module.

For now, just disable the second **News** menu item – the one provided by your News view - in the Main navigation menu.

Go to **Manage | Structure | Menus**.

Edit the **Main navigation** menu, then disable the item, and save the settings.



Let's finish by taking a look at what Views offers in terms of options for the HTML markup of its output.

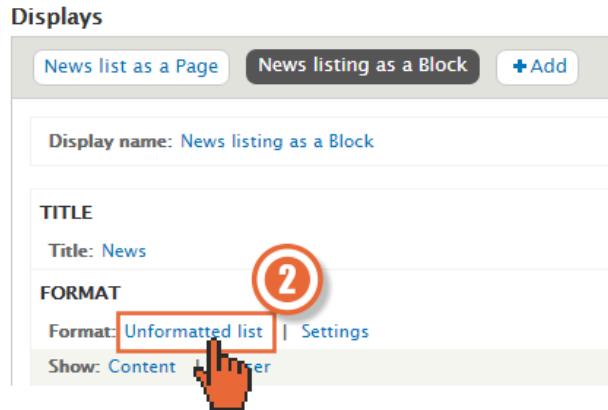
Out of the box, the Views module offers four formats of output:

- Grid
- HTML List
- Table
- Unformatted list

Visit the **News** page (`node/7`) again and edit the news view directly using the contextual link, as shown in the following screenshot:



This should take you to the View edit screen centered on the News list as a Block display. Locate the **FORMAT** section on the left-hand side and click on the **Unformatted list** default.



As an exercise, we are going to change the output markup to be in a grid style, but we are only going to do this for the News **block** and not the News **page**.

So, in the next screen, dropdown the **For:** menu and choose **This block (override)**, which means that any changes we are about to make will only apply to the block display and not to the original Page display.



Structure

Change the style setting from **Unformatted list** to **Grid** and below and set the **Number of columns** to **2**. Apply the new settings:



Finally, save the View:



Save the view and go and revisit the **News** page.

The news article listing should now be output as a grid of two column widths like the following:

An article classified with Documentation	An article classified with Consultancy and Government
Submitted by admin on Wed, 12/31/2014 - 14:43	Submitted by admin on Wed, 12/31/2014 - 14:42
Lore ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.	Lore ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

[Read more](#) [Add new comment](#)

[Read more](#) [Add new comment](#)

Summary

In this chapter, we introduced some of the key ideas for managing the structure of a Drupal site. We briefly covered menu management to help provide a basic top level site navigation. Then we concentrated on employing Drupal's powerful Taxonomy system to help provide grouped lists of all of the site content according to how the content might be tagged with keywords or more specifically categorized with one or more words or phrases from fixed lists of categorizing terms. Then we looked at how you can employ taxonomy again to provide three forms of article—Article, Blog, and News—based around the basic Drupal Article content type.

We spent some time employing the powerful Views module to query the site database and building content lists based on filtering criteria and we saw how the module could be used to create both pages and blocks output. Finally, we saw some examples of the built-in formats that the Views module provides to control the markup style of its output.

In the next chapter, we will be returning to content editing and exploring some of the more advanced editing and configuration Drupal 8 has to offer.

7

Advanced Content

In this chapter, we will look more at the various field types that are available and we will present some practical exercises creating new content types that you will need to help realize the remaining functionality laid out in the site-building scenario introduced in *Chapter 5, Basic Content*.

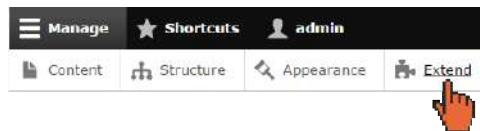
You will experience scenarios where you might use the different field types in real site builds and cover some more tips and tricks with the Views module along the way.

Field types

Before we describe each of the core field types, let's first make sure that they are all available because by default, not all the available field type modules are active.

We'll need the **Link** and **Telephone** type fields to be active. The Link module should be enabled already, but we'll need to activate the Telephone field module.

Visit the modules listing page at **Admin | Manage | Extend**.



Scroll down to **FIELD TYPES** and enable the **Telephone** field, as shown in the following screenshot:



Extending content types

In the next section, we'll look at some common adjustments made to fields—their content-editing interfaces, prompt text, default values, and their display settings—so as to get used to the options available to you.

Adjusting field settings

We'll start by making some adjustments to the core **Article** and **Basic page** content types and then move on to creating some new content types and employing some new field types to meet the requirements as laid out in *Chapter 5, Basic Content*.

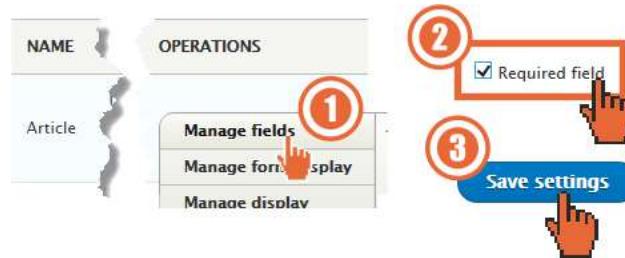
Forcing the Article type field to be mandatory

It is often a good idea to insist that certain fields are populated by making them required.

You have actually already done this in *Chapter 6, Structure*, when you set the relationship to the *Category* taxonomy field, but let's go through this again now.

Another typical candidate field for this might be the **Article type** field in the **Article** content type. You'll recall that its purpose is to enable us to classify Articles as either Blog posts, News, or Articles, and as such it's essential that the field is populated in order to power the various listing pages.

To set the field as mandatory, visit the **Manage fields** page for the **Article content type**, choose to edit the **Article type** field and tick the **Required field** checkbox. Then **Save settings** as shown in the following screenshot:



Adjusting edit form settings

Next we will look at some additional settings available for the various field types.

Placeholder text

New in the Drupal 8 core is the facility for adding placeholder text to help prompt and guide content editors. This is default text that will appear in the field before the user has typed anything in.

Let's add placeholder text to the **Tags** field to help prompt editors to fill it in properly.

Go to **Manage | Structure | Content types | Article**.

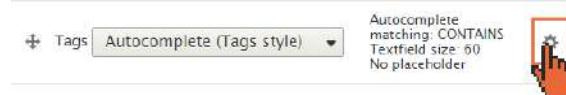
Choose **Manage form display** from the dropdown menu options, as shown in the following screenshot:



This is where you manage the various field widget settings that are active in the content-editing screens, that is, the form elements through which you interact with the field when editing content.

You are going to add some placeholder text to the **Tags** field to prompt users that they can select suggestions for words or phrases (terms) that have been used before.

Click on the cog wheel icon on the right-hand side of the **Tags** field:



You should see that the field settings area expands down to give you access to editing the placeholder text.

Enter a short prompt then click on the **Update** button:



Note that after clicking on the **Update** button, you must also scroll to the bottom of the screen and click Save to make your placeholder a permanent fixture.



Once you have saved your settings, you should see your placeholder text when creating a new **Article**.

Tags

As you type, pick off 'suggestions' if they appear.

Enter a comma-separated list. For example: Amsterdam, Mexico City, "Cleveland, Ohio"

Customizing view modes

We looked at the concept of view modes in *Chapter 4, Getting Started with the UI*, but now let's look at them in a bit more detail; specifically, at how to create new ones to meet particular site requirements.

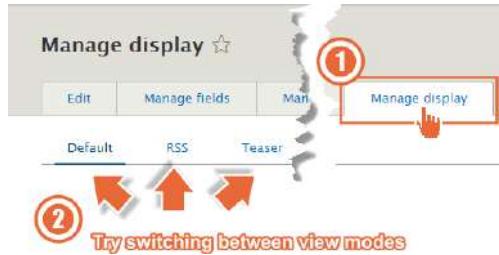
In the standard Drupal 8 install, you have two view modes active for all content types: **Default** and **Teaser**. The Teaser defines the collection of fields; their visibility and their order of display for all content when viewed as a teaser and the Default defines the presentation otherwise.

Visit the **Manage display** page again for Articles:

NAME	DESCRIPTION	OPERATIONS
Article	Use <i>articles</i> for time-sensitive content such as news, press releases or blog post	Manage fields Manage form display Manage display  Edit Delete
Basic page	Use <i>basic pages</i> for static content such as an 'About us' page.	

In a standard Drupal 8 installation, only RSS and Teaser view modes are customized, resulting in three secondary tabs on the **Manage display** page.

Try switching between the three tabs: Default, RSS, Teaser:



Return to the **DEFAULT** view mode and open up the **CUSTOM DISPLAY SETTINGS** region at the bottom of the page to reveal some other view modes:

CUSTOM DISPLAY SETTINGS

Use custom display settings for the following modes

- Full content
- RSS
- Search index
- Search result highlighting input
- Teaser

You will see a list of all view modes available for all nodes.

As you might expect, the **Teaser** view mode is already ticked since it is active in a standard Drupal 8 installation.

Ticking any one of these additional view modes tells Drupal that you wish to customize it for the particular content type—in this case Articles.

Tick the **Full content** view mode now and save the settings.

Use custom display settings for the following modes

- Full content
- RSS

After saving the settings, you will now see an additional secondary tab entitled **Full content**. Now we can go and reshape that in terms of the fields' visibility and order.



Removing the labels and fields from the display

When visiting an Article node in **Full content**, you wouldn't want to see the label on the **Category** field or to see the **Article type** field at all since the latter only exists for filtering purposes.

An article classified with Consultancy, Government, and Training

[View](#) [Edit](#) [Delete](#)

Submitted by [admin](#) on Thu, 12/17/2015 - 11:05

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.



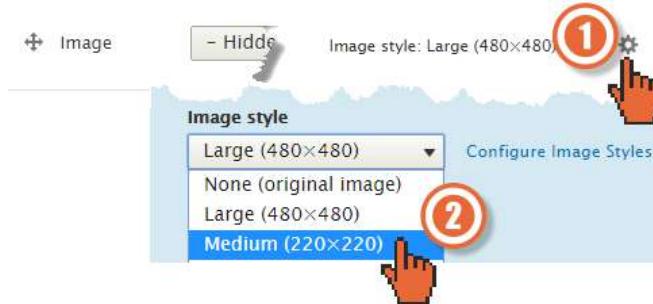
To effect these changes from the **Manage Display** page, and while this time editing the **Full content** view mode, hide the label on the **Category** field and move the entire **Article type** field into the **Disabled** area then **Save** your settings.



Controlling image size using styles

Drupal uses image styles that allow you to crop, resize, rotate and desaturate images without affecting the originally uploaded image. The default style for Articles nodes is the pre-defined **Large** which automatically scales the image to 480 x 480 pixels.

Adjust the image style from **Large** to **Medium**, and then click on **Update** and **Save**.



Go back and look at a full view of any Article node to see the effect which should be a re-scaled image:

The Company pet

Submitted by [admin](#) on Wed, 12/16/2015 - 16:20



Meet Yuki, our company pet.
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Re-scaled to
220 by 200 pixels

Creating new content types

When we were going through the site-building scenario in *Chapter 5, Basic Content*, we defined four more content types that we need on the website: Services, Clients, Testimonials, and Frequently Asked Questions (FAQs).

We'll now work through building each of these content types and add extra fields of various types as we go.

Creating the Client content type

To create the Client content type first visit: **Manage | Structure**.

Click on **Content types** and then **+Add content type**.



Name your new content type and optionally include a brief HTML-formatted administrative description which will appear on the Add content page.

A screenshot of the 'Add content type' form for 'Client'. The 'Name' field is highlighted with a red box and circled with a red number 1. The 'Description' field contains an HTML-formatted description of the client, which is also highlighted with a red box and circled with a red number 2. A red arrow points from the 'Name' field to the 'Description' field.

Name *	<input type="text" value="Client"/> Machine name [Edit]
The human-readable name of this content type, e.g. 'friendly name' played as part	
Description	<p>Clients company's logo, email and telephone contact details.
 Clients can be categorized via the 'Category' taxonomy. </p></p>

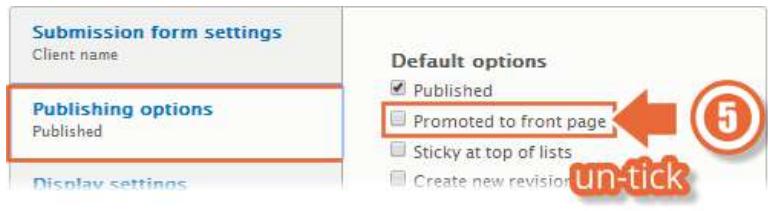
In the additional settings area at the bottom of the screen, under **Submission form settings**, change the label for the **Title** field to **Client name** so as to prompt your content editors to supply just that.

Note that you can provide an optional set of further guidelines and disable the **Preview before submitting** setting if you so wish.

A screenshot of the 'Submission form settings' section. The 'Title field label' is set to 'Client name' and is highlighted with a red box and circled with a red number 3. The 'Preview before submitting' setting is set to 'Disabled' and is highlighted with a red box and circled with a red number 4. A red arrow points from the 'Title field label' to the 'Preview before submitting' setting.

Submission form settings
Client name
Publishing options
Published , Promoted to front page
Display settings
Display author and date information.
Title field label*
<input type="text" value="Client name"/>
Preview before submitting
<input checked="" type="radio"/> Disabled
<input type="radio"/> Optional
<input type="radio"/> Required

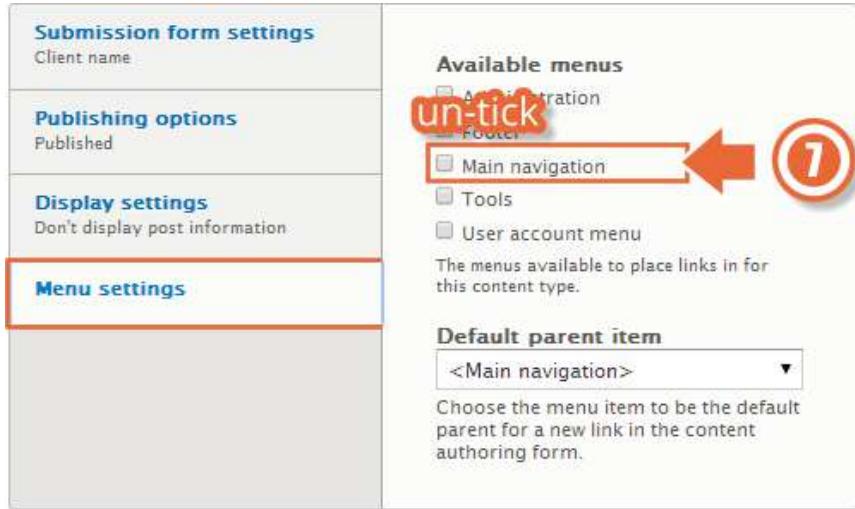
In the **Publishing options** section, untick the **Promoted to front page** option as you probably won't want Clients promoted.



Likewise, we are not interested in displaying who created Client content items so un-tick **Display author and date information** in the Display settings section.



Lastly, you probably wouldn't want content editors putting individual clients into a menu and certainly not into the Main navigation menu, so un-tick all menus in the **Menu settings** section.



You have now created and configured the Client content type as a whole; the next step is to add some new fields, so do so now by clicking on the **Save and manage fields** button.



Inherited fields

When you are taken to the **Manage fields** page, you might be surprised to find that you have inherited an instance of the **Body** field.

Adding a logo field

Add a new **Image** type field to hold the client logo.

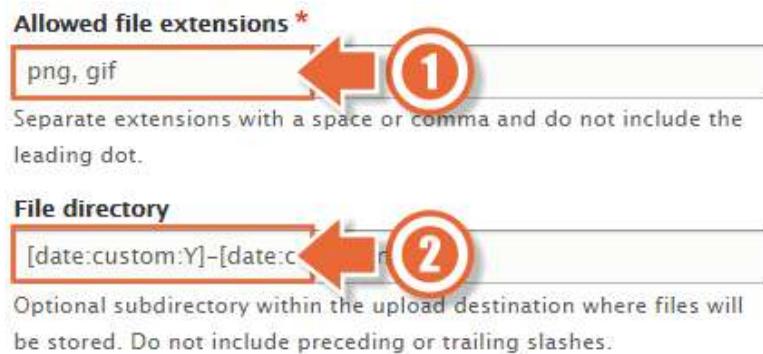
You may also notice a section here entitled **Re-use an existing field**. More about that will be given later. Click on the **Save and continue** button.

On the first page of options, you can leave all the default settings as they are and simply click on the **Save field settings** button.



Since the field is designated to hold logos, it might be a good idea to limit the types of images to only PNG and GIF, thereby eliminating the potential for someone uploading a low-quality JPEG.

In the following settings page, adjust the **Allowed file extensions** to be png, gif, as shown in (1) in the following screenshot:



Tokens

Drupal stores images and other asset-based fields' contents as actual physical files, then references them from within the database. The **Files directory** input in the previous screenshot enables you to be specific about exactly where you want these stored on a field-by-field basis.

Notice the default use of [date:custom:Y] and [date:custom:m] in this input.

These are examples of *tokens* that get replaced in real time when used throughout Drupal. Thus, if you uploaded files on a particular day for example, 18 December 2015, these tokens will result in the uploaded logos be stored physically in the path:

/sites/default/files/2015-12

Image accessibility

For good accessibility, notice that the **Alt field** is enabled and required by default, thereby enabling content editors to attach Alt text to their uploaded logos so that, for example, users with a screen reader can interpret what an image is displaying.

Click on **Save settings** to complete this field.



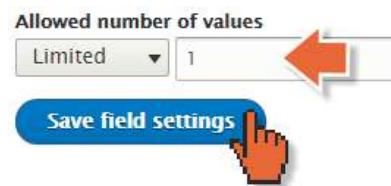
Linking clients to their web sites

It will be useful to be able to reference a client's website URL when viewing their details; you can use the core **Link** field type to achieve this. The Link field (provided by the Link module) allows you to create fields that contain internal or external URLs and optional link text.

Add a new field labeled **Website** and set the field type to **Link**, and then click on **Save and continue**.



In the next screenshot, you get the opportunity to say just how many URLs you might want to add to each Client. The default of a single value seems the right choice here.



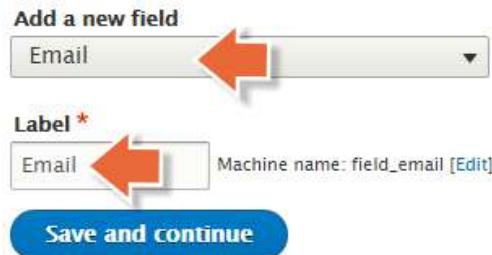
For this example, we'll set the **Allowed link type** to **External links only**, and we'll set the **Allow link text** to **Required**. Finally, we'll make the default title be `Client website`.



Configure your Website field as shown in the preceding screenshot and click on **Save settings**.

Providing an e-mail address for a Client

Using exactly the same procedure as in the Website field, now add a new **Email** field so that you can also provide a contact (`mailto:`) link for the client. In this case, the optional help text aside, you can leave all the field settings at their defaults.



Continue with all the default settings and save the field.

Providing a telephone number for a Client

Again, using the same procedure as before, now add a new Telephone field with a field label of Telephone so that you can also provide a contact number.

Again, you can leave all the field settings at their defaults.

The screenshot shows a form titled 'Add a new field'. In the 'Type' dropdown, 'Telephone' is selected. In the 'Label' field, 'Telephone' is entered. A red arrow points to the 'Label' field. Below the form is a blue 'Save and continue' button.

Marking a Client as high profile

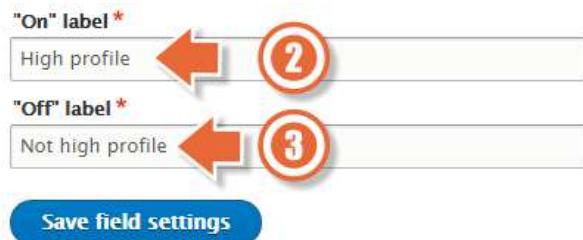
Finally, in order to illustrate the use of the Boolean field type, add a High profile (yes/no) field to your Client content type, which you can later use to filter your list of clients down to those you wish to promote most prominently.

The screenshot shows a form titled 'Add a new field'. In the 'Type' dropdown, 'Boolean' is selected. In the 'Label' field, 'High profile' is entered. A red arrow points to the 'Label' field. Below the form is a blue 'Save and continue' button. A red circle with the number '1' is overlaid on the 'Save and continue' button.

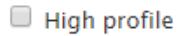
Leave the **Allowed number of values** to the default of 1.

The screenshot shows a form titled 'Allowed number of values'. A dropdown menu shows 'Limited' with a value of '1' selected. Below the form is a blue 'Save field settings' button. A red hand icon is pointing to the 'Save field settings' button.

Initialize with the following settings: "On" label as High profile and "Off" label as Not high profile.

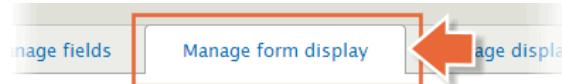


By default, the Boolean field only shows the "On" label value as a single checkbox.

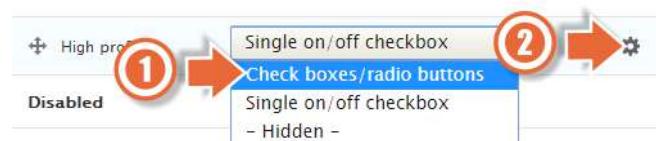


However, it's worth noting here that you have other options as to how the field is represented within the input form. You can, for example, show it as two checkboxes, which, while not really right here, might be useful in other scenarios.

To see the options, visit the **Manage form display** page again for the **High profile** field.



Adjust the Widget to be **Check boxes/radio buttons**.



Now when you create a Client node, you will see that the input for the **High profile** is a little more verbose.



Note that the N/A entry is there because we did not insist on the field being mandatory.

Having tried this out, it's probably best to put the setting back to the original **Single on/off checkbox** version.

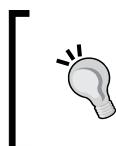
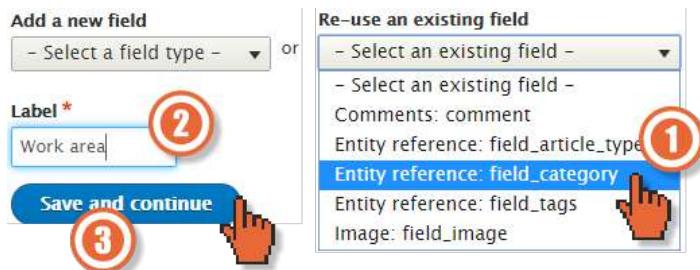
Attaching taxonomy

In *Chapter 5, Basic Content*, we said that we want to be able to categorize Clients into one or more Work areas.

The easiest way to achieve this is to enable content editors to apply terms from within the **Category** taxonomy vocabulary by reusing the field that we created earlier.

Reusing fields

Visit the **Manage fields** page again and choose to **Re-use** the existing **Category** field but relabeled as **Work area** as follows:



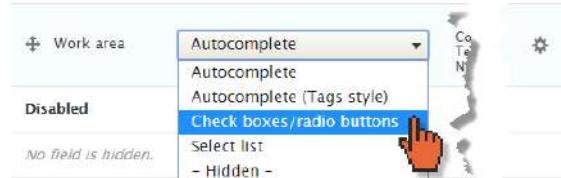
Detailed discussion of the idea of reusing fields is beyond the scope of this module save to say that reusing existing fields is a good recommendation when you become a more advanced site builder. See drupal.org/node/1577260 for more.

Once you've added the field, you can optionally make it mandatory and/or choose a default value but neither of these are required for now.

Also, make sure you select only the **Category** vocabulary in the **REFERENCE TYPE** section.



Adjust the widget on the **Manage form display** as you wish. In the following example, we have opted for Check boxes/radio buttons because there are only currently six terms in the Category vocabulary.

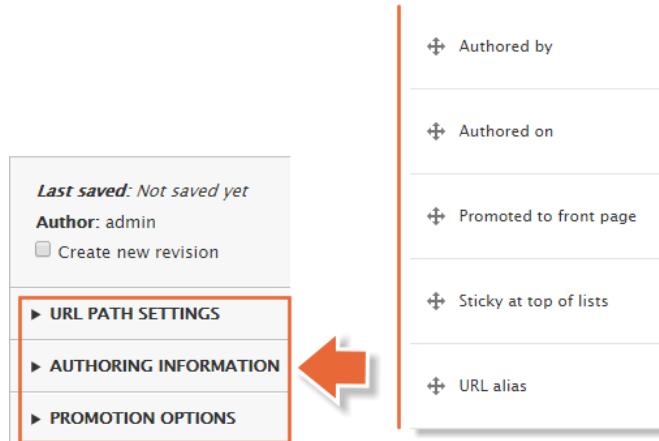


Finally, in this field-adding exercise, adjust the fields order on the **Manage form display** page so that you have the following in this order with these widget settings:

Client name	Textfield
Work area	Check boxes/radio buttons
Body	Text area with a summary
Logo	Image
Client website	Link
Email	Email
Telephone	Telephone number
High profile	Single on/off checkbox

Advanced Content

Do not concern yourself with the presence and/or relative positions of the other fields right now as these all get placed neatly into the **Secondary editing region** anyway.



Create at least three Clients now and classify them with a mixture of Work areas.

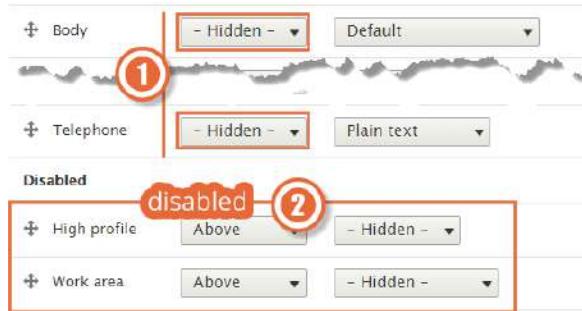
	TITLE	CONTENT TYPE	AUTHOR	STATUS	UPDATED
<input type="checkbox"/>	Acquia	Client	admin	Published	04/15/2015 – 15:0
<input type="checkbox"/>	Inviqa	Client	admin	Published	04/15/2015 – 14:3
<input type="checkbox"/>	iKOS	Client	admin	Published	04/15/2015 – 14:3

You'll notice that the logo and contact details display at the bottom of the Full content view clearly need attention!



Adjusting field display settings

The contact details collection is easily cleaned up on the Full content view by visiting the **Manage display** tab and editing the **Default** fields and labels' visibilities as follows:



Note which fields we have chosen to disable here so that we only see the fields that we should and without unnecessary labeling:



Similarly, when viewed as a **Teaser**:

1. Trim the Body right down to a 140-character Twitter-size taster.
2. Move the **Links** so as to position the **Read more**.



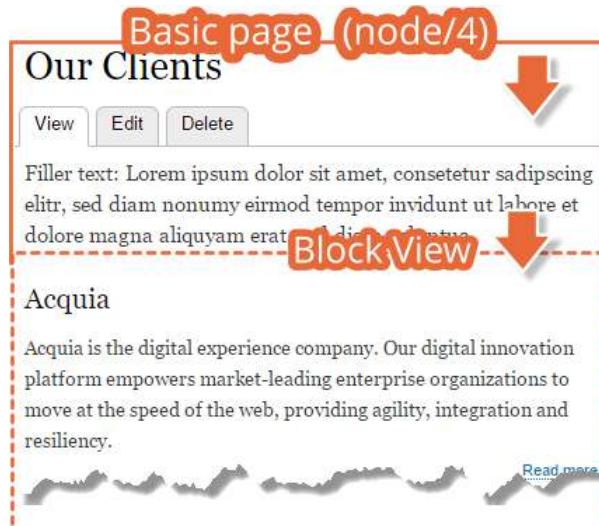
Adding a sorted client list to the Clients page

This is further revision of what you covered in *Chapter 6, Structure*, when you created a **block** using the **Views** module, this time without quite so many step-by-step illustrations.

Start by visiting **Manage | Structure | Views** then...

1. Create a new View called Clients that provides a block listing.
2. Set the **Show** setting set to **Content** and the **of type:** set to **Client**.
3. Under **BLOCK DISPLAY SETTINGS**, set **Display format** to **unformatted list of teasers**.
4. Visit **Manage | Structure | Block layout** and place the block, untitled, in the **Content** region but only visible on the Our Clients page (in our case, that's `/node/4`).

You should end up with the Our Clients page looking something like this:



Views sort criteria

This time we'll add a little more functionality to the View as to list the clients with the high profile setting at the top.

You can edit the view directly from within the Our Clients page by using the contextual links. You'll see three sets of contextual links as you hover: one for the editing the page, another for editing the first Client, and below that, a third one for editing the View.

Our Clients

Filler text: Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Acquia

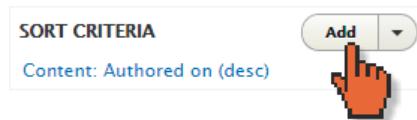
Acquia is the digital experience company. Our digital platform empowers market-leading enterprise organ move at the speed of the web, providing agility, integrati resiliency.

[Edit view](#)

[Read more](#)

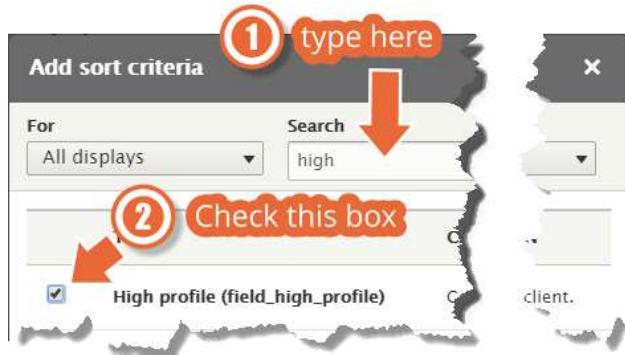
Edit the view and locate the SORT CRITERIA section at the bottom left of the screen.

Click on **Add** to add another criterion:



At first sight, the field popup dialog might seem a little overwhelming, but it is in fact a highly usable field-section/filtering tool. The idea here is to make the primary sort criteria work off the value of the **High profile** and thereby promote those clients to the top of the list.

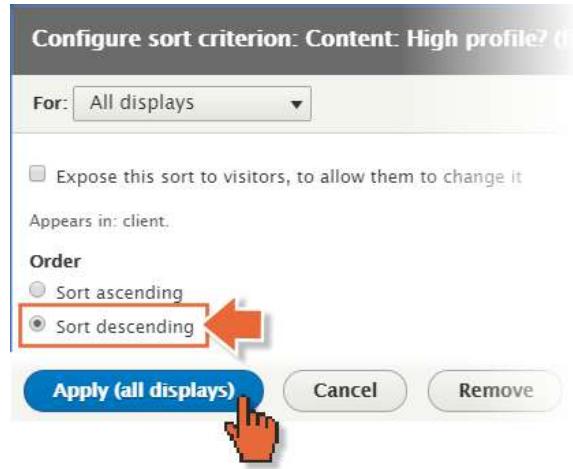
Click in the **Search** input and type a few letters from the field name and you will see the list of fields, and so on, automatically narrow down so that you can locate the **High profile** field.



Click on the **Apply (all displays)** button to add the field as a new sort criterion.



In the following dialog, choose **Sort descending** because we want to ensure that those clients with the High profile field checked are listed first, and so sorting on that field in descending order will work.



You will see the new field added as a second sort criterion, but we need to rearrange the criteria to make the newly added one the first in the list. Click on the menu icon to the right of the list and drag to rearrange before applying the changes to all displays.



In the modal popup window, reorder the fields so that the High profile field is the first sort criteria and hit **Apply (all displays)**.



If all is well, the two sort criteria are now in the right order to do the job.



Save the view; the high profile clients should be listed first.

Adding a pager to your view

Assuming that you have quite a few clients, you won't want all of them listed all at once on the Our Clients page since that may have a performance effect in terms of page load time. Fortunately, the Views module is equipped with a **pager** facility that enables you break up your lists into sensibly sized chunks.

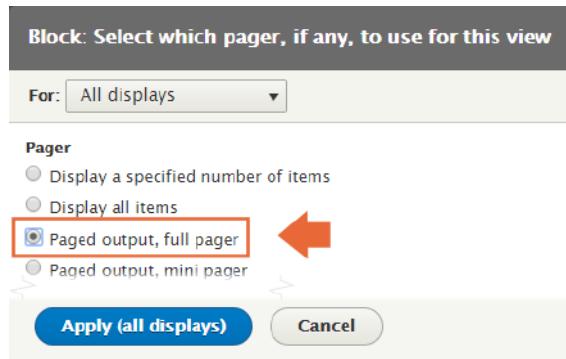
We'll create a pager for Our Clients view now.

Advanced Content

Locate the central region in the View edit dialog, specifically the **PAGER** section. Note that your list is currently limited to only five items because the current setting is to **Display a specified number of items**.

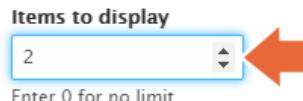


In the following dialog, pick the option to use a **Page output, full pager** and apply this to all displays.



You can change the number of items on each page, but the default value of five items per page is probably a good choice. Since we only have three Client nodes right now, let's set it to just two.

Don't concern yourself with the other two options here for now.



Further down the same popup window, you can also override any of the default text labels on the controls for moving forward and backward through the various pages.

▼ PAGER LINK LABELS

First page link text
« first

Previous page link text
< previous

Next page link text
next >

Last page link text
last »

Apply the settings to all displays, save the View and you should see the pager in action.

iKOS

iKOS provides the complete website delivery life cycle:
consulting, training, developing and supporting.

[Read more](#)



weo, providing agility, integration and resiliency.

pager

[Read more](#)



1 2 [Next >](#) [Last »](#)

Creating the Service content type

Creating the basic Service content type and adding in the Taxonomy reference field is exactly the same as for the Client covered step by step earlier in this chapter.

To create the Service content type first visit: **Manage | Structure**, click on **Content types** and then **+Add content type**.

Repeat what you did before to create yourself a Service content type with the following basic settings:

- **Title field label:** Service
- **Preview before submitting:** Disabled
- **Published:** YES
- **Promoted to front page:** NO
- **Display author and date information:** NO
- **Available menus:** NONE

Add/adjust fields:

- Re-label the **Body** field to **Description**.
- Add a `work_area` field linking up the terms from the Category taxonomy vocab—hint: you can again reuse the existing Category field.

Click the **Save and manage** fields button.

When you're done, the **Manage fields** page should look like the following:

LABEL	MACHINE NAME	FIELD TYPE	OPERATIONS
Description	body	Text (formatted, long, with summary)	Edit ▼
Work area	field_category	Entity reference	Edit ▼

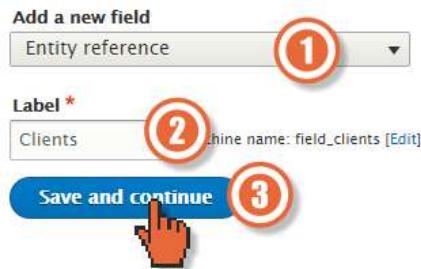
On the **Manage form display** page, set the widget to **Checkboxes / radio buttons** just as you did last time when you re-used this field. Adjust the order of the fields in the **Manage display** screen so that, in the Teaser view mode, the **Body** field is truncated to 140 characters and the Link field is moved to the bottom.

Enabling the linking of Services to Clients

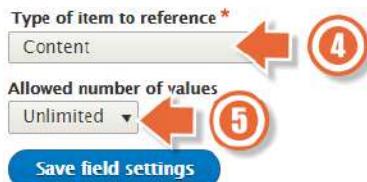
It may be very useful to be able to make references to one or more Clients from within a Service so as to show the capacity in which you have worked with them.

In order to make one or more direct references from a Service node entity to a Client node entity, you'll need to use a **Reference | Content** type field.

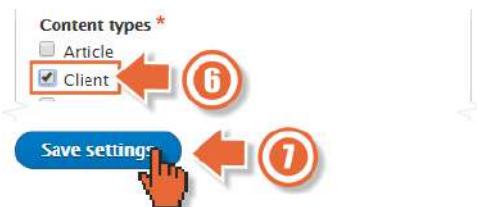
Add a **Reference** field now labeled 'Clients'.



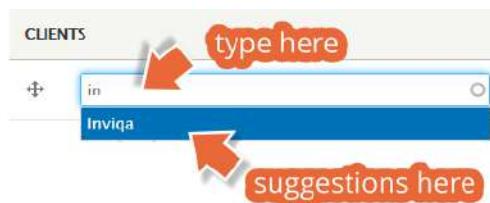
In the field settings page, set the **Allowed number of values** to Unlimited and make sure that the type of item (actually the type of entity) is set to **Content**.



Finally, in the **REFERENCE TYPE** section, set the content type so that editors can only make references to Client entities.



Note that when creating a new Service from now on, as you type a character or so into the **Clients** field, Drupal will search through the database and offer suggestions.



Also, since you made the field a multi-value (unlimited) one, you can add as many other clients as you like.

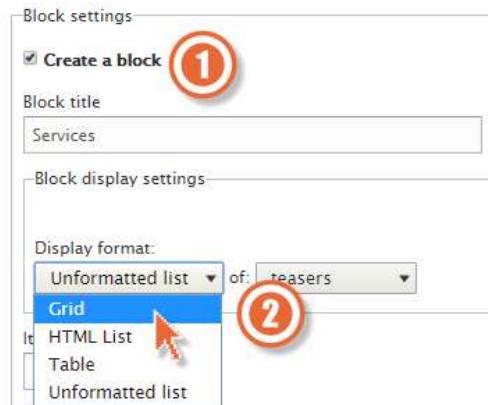


Create at least four Service nodes, each linked to at least one company.

	TITLE	CONTENT TYPE	AUTHOR	STATUS	UPDATED	
<input type="checkbox"/>	Drupal training courses	Service	admin	Published	04/16/2015 – 12:30	Edit
<input type="checkbox"/>	Supporting your websites	Service	admin	Published	04/16/2015 – 12:29	Edit
<input type="checkbox"/>	Drupal development	Service	admin	Published	04/16/2015 – 12:27	Edit
<input type="checkbox"/>	Drupal consulting	Service	admin	Published	04/16/2015 – 12:24	Edit

Displaying services listing using Views

As yet another revision of what you covered in *Chapter 6, Structure*, create another block view now that provides a block listing of teasers of all services in a two-column grid format. As a reminder, the quickest and easiest way to set the grid format is to choose it from within the basic Views wizard when you first opt to create the block.



When you click on **Save and edit** to go into the Views editing screen, you can then adjust the grid setting to two columns.



Place that block on the Services page (in our case /node/3).

Once you have adjusted the various fields display settings teaser, the finished Services page should look something like the following (shown here in a mobile view):



Creating the Testimonial content type

Next, create another content type to hold client testimonials, each of which holds the details (name, job role etc.) of the person providing the reference.

To create the Testimonial content type first visit: **Manage | Structure**, click on **Content types** and then **+Add content type**.

Set up the content type with the following basic settings:

- **Title field label:** Testimonial title
- **Preview before submitting:** Disabled
- **Published:** YES
- **Promoted to front page:** NO
- **Display author and date information:** NO
- **Available menus:** NONE

Add/adjust fields:

- Re-label the **Body** field to **Testimonial statement**
- Add an **Individual** plain text field to hold the testimonial writer's name.
- Add a **Role** field (**Text plain**) to describe the writer's role.
- Add a Work area field linking up the terms to a single term from the Category taxonomy vocab' – hint: you can again reuse the Category field.

When you're done, visit the **Manage fields** page, which should look like the following:

LABEL	MACHINE NAME	FIELD TYPE
Individual	field_individual	Text (plain)
Role	field_role	Text (plain)
Testimonial statement	body	Text (formatted, long, with summary)
Work area	field_category	Entity reference

Based on all the exercises you have already done with regard to form management, the suggested field order for the (editing) form page with the Work area (Category taxonomy) widget set to Checkboxes/radio buttons might be as follows:

FIELD	WIDGET
⊕ Testimonial title	Textfield ▾
⊕ Individual	Textfield ▾
⊕ Role	Textfield ▾
⊕ Testimonial statement	Text area with a summary ▾
⊕ Work area	Check boxes/radio buttons ▾

Adjust the **Manage display** (Default) settings so as to hide the unnecessary labels on the Individual and Job description fields and disable the visibility of the Work area field.

FIELD	LABEL
⊕ Testimonial statement	- Hidden - ▾
⊕ Individual	- Hidden - ▾
⊕ Role	- Hidden - ▾

For the **Teaser** display, let's just have the Title (will always show anyway), the Individual, and the Role fields on display plus the Links for the Read more:

FIELD	LABEL	FORMAT	
⊕ Testimonial statement	- Hidden - ▾	Summary ▾	≈ 140 characters ⚙
⊕ Individual	- Hidden - ▾	Plain text ▾	⚙
⊕ Role	Trimmed to Twitter size again	Visible ▾	⚙
⊕ Links			

Advanced Content

Once you have set up the new content type and added the various fields, create several testimonials, all marked as relevant to different Work areas (Category terms) and with an individual's name, role, and company name.

Create Testimonial

Home » Add content

Title *

A comprehensive and high quality course

Individual

Anonymous (identity hidden)

Writer's name

Work area

Charities
 Consultancy
 Documentation
 E-Commerce

Taxonomy

If all is well, the **Default** and **Teaser** views of a Testimonial should look like those shown below. Note you can view the teaser version on the front page if you selected **Promoted to front page** in the publishing options:

A comprehensive and high quality course

View **Edit** **Outline** **Delete**

After providing a broad set of requirements it was clear right from the beginning of **Full content** pg/consultancy that the instructor had spent a significant amount of time researching the topics and preparing the content to deliver.

This prep^a flexible col^b requested^c teaching w^d A really enjoyable, relaxed and flexible course.
Rob Roadie
Civil Servant, UK Government

Teaser

Read more 1 view

Create at least three more Testimonials now so that you have something to work within the next section.

Listing testimonials with a view

You are now going to create another view, and this time to list all your testimonials.

In the next section, you will also see just one example of the type of customizations that you effect on a view – in this case, how to group testimonials according to the Work area(s) that they relate to.

As you have already seen, the Views module is a very accessible tool for querying and presenting site content, and even the core version provides you with a range of options for providing both pre-filtered, pre-sorted, and user-filterable lists in a variety of formats.

Creating a grouped view

This view will list a number of testimonials nodes together under a heading of **E-Commerce**, another under **Consultancy** and then some more under **Training** as illustrated in the following screenshot:

The screenshot shows a grouped testimonial view with three main sections:

- e-Commerce**: Contains two testimonials from Holly Howell and Natalia Radcliffe-Brine. A red arrow points to the "Work area (Category term)" label above the first testimonial.
- Consultancy**: Contains two testimonials from Jae Rance and Adam Waters. A red arrow points to the "Work area (Category term)" label above the first testimonial.
- Training**: Contains two testimonials from Chris Holden and Stephannie Hay. A red arrow points to the "Work area (Category term)" label above the first testimonial.

Each section has a red "nodes" button on the right side.

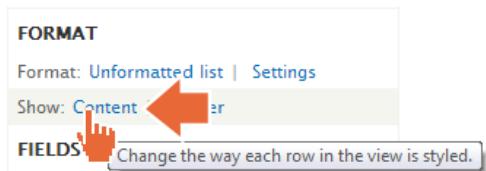
This time around, we'll return to using a Page view again rather than a block because we don't have an existing Testimonials page into which to embed a block anyway.

Starting with a page view is also a good vehicle for learning how to switch a View from being a content view (based in view modes: default, teaser, and so on) into a field-based view.

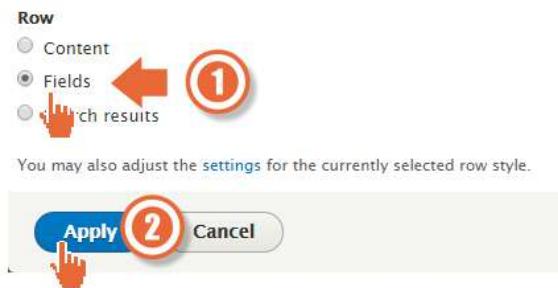
Start by creating a new view in the usual way via the opening wizard screen that filters content down to just **Testimonials**.

Once you have clicked on **Save and edit** and are in the main edit screen, locate the **Content** link inside the **FORMAT** section on the left-hand side of the View edit page. This controls the style of each item in the output and defaults to **Content** with a **Display mode of Teaser** which should make good sense to you.

Click on the link:



Change to **Fields**, then **Apply (all displays)**, and then, finally, **Apply** again.



Go with the defaults settings on the next popup window and then Click on **Apply** one final time.



Opting for the Fields style means that instead of rendering each Testimonial using either the **Default** or the **Teaser View modes**, you have the opportunity to specify exactly which fields are included in addition to the Title.

You are going to add the following extra fields: Individual, Role, and Work area.

Click on the **Add** button in the **FIELDS** section.



Typing **ind** in the Search input should be enough to locate the **Individual** field.

Check the field once you find it.

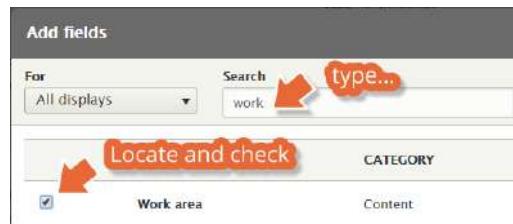


Note that you do not need to add each field one by one, but you can enter another Search, and locate and check the next field.

Check the **Role** field.



Also, check the **Work area** field.



Then, finally, click on **Apply (all displays)** again, and you will be taken through a series of dialogs asking you further questions about how you would like each field rendered in the output of the View.

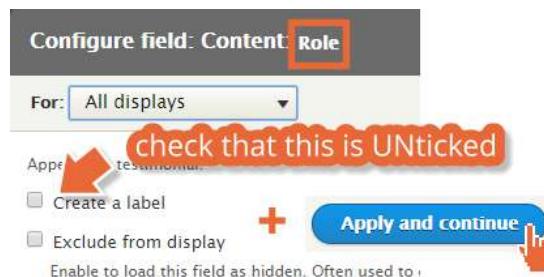


If the **Individual** field was the one you added first, then that will be the first you are asked more about.

Now that you are adding fields individually, the various labels' visibility settings that you set up within the Display views modes (in Manage Display) should all be respected, but if you omitted to hide the labels, then you can do so now by simply un-checking the **Create label** field and then clicking on **Apply**.

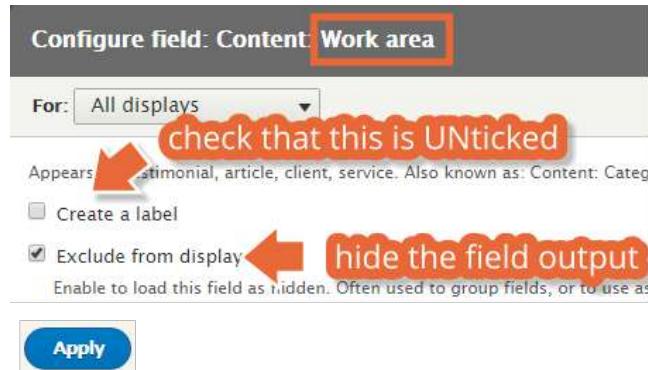


You can do the same for the **Role** field.



When it comes to the **Work area** field, things are different because while you don't actually want this field to be visible in the views output, the data from the field needs to be included in the view in order for the Views module to be able to group the testimonials by that field.

Thus, in this case, you should to tick **Exclude from display**.



In the next window, choose to forego the link to the Taxonomy term by un-checking the **Link label to the referenced entity** checkbox.



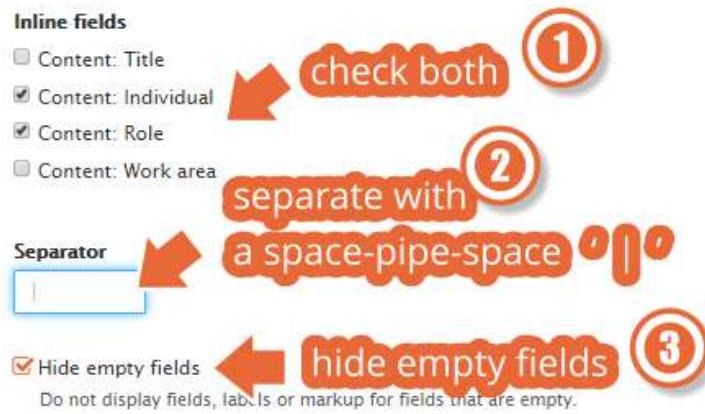
You should then see all four fields listed in the **FIELDS** section on the left-hand side:

The screenshot shows the 'FIELDS' section. It lists four fields: 'Content: Title', 'Content: Individual', 'Content: Role', and 'Content: Work area [hidden]'. Above the list is a 'Show: Fields | Settings' link. To the right is an 'Add' button with a dropdown menu.

To illustrate one of the many fine-tuning options that are available within Views, click on the Fields **Settings** link in the **FORMAT** area:

The screenshot shows the 'FORMAT' area. It includes a 'Format: Unformatted list | Settings' link and a 'Show: Fields | Settings' link, which is highlighted with a red arrow.

Opt to present the Individual and Role fields inline separated with a spaced pipe (that is, space | space):



At this point, the un-grouped view should appear as shown in the following screenshot. Scrolling down the views page to the Preview area shows how the combination of putting the two fields inline and applying the **Hide empty fields** works well for us.

Content

A great requirement gathering workshop

Paula House | Digital Manager, Independengrow

fields in-line and separated

The final piece of the puzzle is to ask the Views module to group the testimonials by Work area so that, for example, all the Training-related ones appear listed together.

You'll remember that we opted not to display the Work area in the View earlier on, but we, nonetheless, needed the field to be present in order to perform the grouping.

Do this by adjusting the **FORMAT** settings again:

FORMAT

Format: [Unformatted list](#) | [Settings](#)

Show: [Fields](#) | [Settings](#)

This time choosing the **Work area** field as the **Grouping field** and **Apply**.



Once you have saved the view, your finished Testimonials page should appear as shown in the following screenshot.

One thing you will notice is that there is still work to be done if one or more Testimonials are tagged with more than one Work area since, with the current grouping in place, Views treats every combination as a separate group:

Testimonials

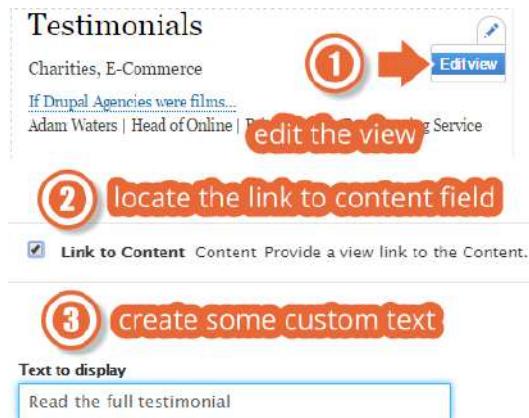
Consultancy, E-Commerce
A great requirement gathering workshop
Paula House Digital Manager, Independengrow
E-Commerce
You have been instrumental in guiding our management
Howard Holly Wholesale Manager, Krispy Donuts
I feel incredibly positive about our website project now!
Chris Holden Head of Information Services and Systems, Higher Education A
Government
A comprehensive and high quality course
Anonymous (identity hidden) Civil Servant, UK Government

grouped by Work area

We could easily devote a whole chapter if not a whole module to various Views recipes using all of the core options for filtering, fine-tuning presentation, content aggregation, caching, and access control to name a few, but we'll keep things simple for now.

Finally, you'll notice that there is no **Read more** link for each testimonial. This is because we have made a field-based view, and although the titles are linked to the full content by default, you might also want to add a specific read more link as well.

To do so, edit the view again and add one of the Views module's special built-in fields, the **Link to content** field.



Save the view and you'll now see a custom **Read more** link on each testimonial.

Testimonials

Charities, E-Commerce

If Drupal Agencies were films...

Adam Waters | Head of Online | British Forces Broadcasting Service

Read the full testimonial ← your custom 'Read more'

The FAQ content type

We'll create the FAQ functionality now using only core functionality, but later in *Chapter 12, Extending Drupal*, we will extend its behavior by extending the Views module so as to improve the presentation, particularly on mobile devices.

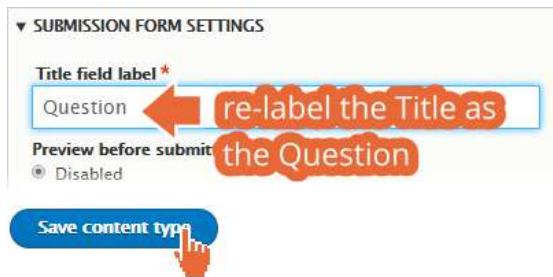
The build for the FAQ content type is very similar to the existing three that you have already created (Client, Service, and Testimonial) with the only real differences being that we will re-label the Body field as the Answer, and we probably don't need to have a summary populated, so we can adjust the settings accordingly.

Lastly, we'll also use the FAQ functionality to demonstrate how you can build live-querying tools that empower your site visitors to interactively filter content according to their own selection criteria.

Content type settings

- **Title field label:** Question
- **Preview before submitting:** Disabled
- **Published:** NO
- **Promoted to front page:** NO
- **Display author and date information:** NO
- **Availability in menus:** NONE

It also makes good sense to re-label the Title as the word **Question** since this way, when presented with a list of standard Teasers, the viewer will see a list of actual Frequently Asked Questions, each of which they can click on to go through to the answer.



Field settings

- Re-label the **Body** as **Answer**
- Make the field mandatory
- Set the text format on the **Answer** field to be Full HTML by default because we might want to add in URLs, tables and so on for a richer answer

- Remove the facility for adding **Summary input** text to the **Body** field

Label *

Answer

Help text

Required field

▼ DEFAULT VALUE

The default value for this field, used when c

Answer

B I S x* x_x I_x

body p

Text format Full HTML

Summary input

This allows authors to include an explicit summary,

Save settings Delete field

Finally, add yet another shared use of the Category field to the FAQ so that you can categorize it accordingly to a Work area: Consultancy, E-Commerce, and so on. As with all previous uses of the shared field, set the *widget* in the **Manage form display** page to **Checkboxes / radio buttons**.

Display settings

As with the last two content types, we will hide the **Links** field and the **Work area** field in the **Default** view mode.

FIELD	LABEL
⊕ Answer	- Hidden -
⊕ Links	No need for these
⊕ Work area	Above

The **Teaser** view mode should only be the **Question** itself, that is, the **Title** and the **Links** (for the **Read more**).

FIELD	LABEL
No field is displayed.	
Disabled	We don't need any fields here because
Links	'Question' is the Title
Work area	and that shows anyway
Answer	and is clickable
	ADD

Create at least three FAQ nodes, each categorized with different Work areas such as Consultancy, e-Commerce, and Training.

Question *
How can we be sure that we get the most from your training courses?

Answer *

Once you have made a decision to take on one of courses we'll begin by having our principal trainer contact you to talk through your precise requirements and to learn more about the personal skills and specialisms of the people you would like us to teach. This way, we will be able to shape the training to best fit the audience.

Work area

- Charities
- Consultancy
- Documentation
- E-Commerce
- Fun
- Government
- Training

Creating the simple FAQ page

The final step is to create the actual FAQs page using the Views module as we have done earlier.

With this new view, though, we are going to take it a step further and equip the view with the ability to live-filter the results to FAQs that are tagged with particular taxonomy term(s). For example, if the user were to choose E-Commerce from a dropdown menu, then the FAQ list will be filtered down to only those marked with that Category term.

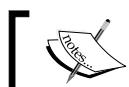
Set up a new page view that provides access to a page at /faq is also accessible from the **Main navigation** as with all the other page views that you have created.

The output from the basic view should be like the following:

Frequently Asked Questions

How can we ensure that we get the most from a
training course? 

If you forget to include the page view in the Main navigation menu or for any reason you need to adjust the path, it's useful to know that you can adjust these settings within the main Views editing screen in the **PAGE SETTINGS** section in the center of the UI.



Take care when adjusting the menu settings since Views does not necessarily default to the Main navigation menu.



Adding interactive querying to a view

Edit your FAQ view via the contextual link toward the bottom left of the UI just above the **SORT CRITERIA** section where you can add a new filter criterion.



Frequently Asked Questions

How can we ensure that we get the most from a training course?

FIELDS
The selected style or row format does not use fields.

FILTER CRITERIA

Content: Publishing status (Yes)  **Add**

Content: Type (= FAQ)

Add a new filter criterion to enable filtering via applied taxonomy terms.



In the following dialog, choose the **Category** as the taxonomy vocabulary of interest and **Dropdown** as the widget, and then click on **Apply and continue**.



The next step is to **Expose this filter to visitors to allow them to change it** so that they can see the dropdown menu.



Then, directly underneath, you may wish to relabel before applying the settings.

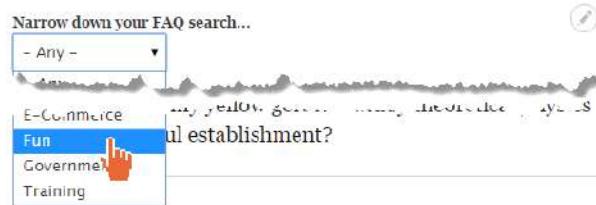


You should now be able to see the newly created and exposed filter in place.

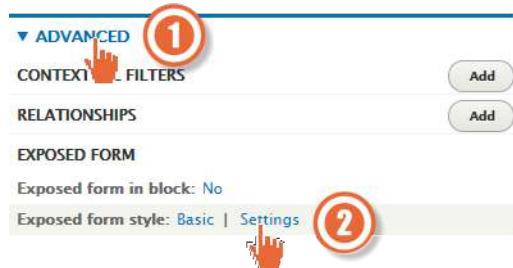


Save your view. You have empowered your visitor to live filter the FAQs page.

Frequently Asked Questions



This works well, but it doesn't come with a Reset button included automatically. To add this, go to the **Advanced** section on the right-hand side of the Views editing window and locate the **EXPOSED FORM** section, and then click on the **Settings** link.



You can then adjust the text for the filter and the reset buttons.

Submit button text *

Apply this filter 3

Include reset button (resets all applied exposed filters)

Reset button label *

Reset and return to all FAQs 4

Text to display in the reset button of the exposed form.

The result will be a useable client querying screen. Note, however, that our FAQs are set to be unpublished by default, so only published nodes will show.

NARROW down your FAQ search...

Consultancy ▾

Apply this filter Reset and return to all FAQs

Summary

In this chapter, we covered examples of extending existing contents in terms of extra purpose-built field types such as e-mail, link, and the entity reference type.

We looked at some alternative data-entry widgets and how the finer control options work for display output, including resizing images for optimal output.

We then looked in detail at creating four new content types and at some extra options within the standard Views UI for presenting the content in a variety of ways.

In the next chapter, we'll look at more of the configuration options that Drupal 8 has to offer, including setting up user accounts.

8

Configuration

Many of Drupal's modules include a settings screen that can be accessed from the **Configuration** menu. In this chapter, we will work our way through these screens to describe the options available and how they apply to the web site.

People – Account settings

The user account settings screen can be reached at **Configuration | People | Account settings** (`admin/config/people/accounts`).

The section is quite extensive and covers a few different areas, so we will look at them individually.

The type of website you are building will dictate the exact choices you make here, but it's important to consider all the options in turn before you allow users to start creating accounts.

Note that the actual management of users and permissions is covered in the *Chapter 9, Users and Access Control*.

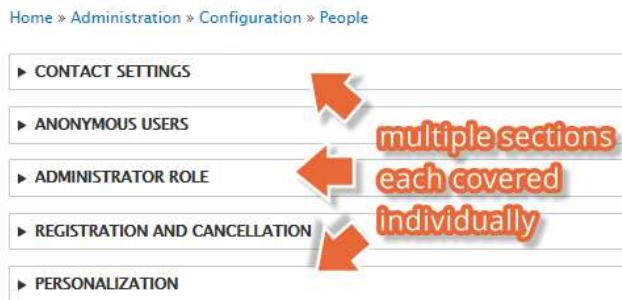
The website you are building has the following requirements:

- You need individual user accounts for different people in the company.
- Visitors (anonymous users) should be able to create an account.
- Visitors should not be able to post comments.
- Site members should be able to post comments without approval.

Keep these requirements in mind as we work through the user configuration pages.

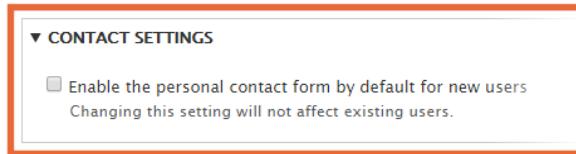
Configuration

The configuration comprises multiple sections shown in the following screenshot, all collapsed to save space:



Contact settings

When a user has a personal contact form, a contact tab appears on that user's account page:



By unchecking the checkbox here, all newly created users will not have a personal contact form on their account page. You can still select it to enable one on a per-user basis; the setting here only configures the default for new users. If the setting is enabled on a particular user, other users with the permission **Use users' personal contact forms** will be able to send e-mails to them using the form.



On a new installation, only authenticated users can use contact forms. This is a good default as it prevents spam e-mails being sent to users by web bots.

Anonymous users

If you allow anonymous users to perform any actions such as comment or create content on the website, then that content will be shown as being submitted by the Drupal username **Anonymous**. The name displayed is easily changed here to something of your choosing – a common alternative is to use the word **Guest**.

▼ ANONYMOUS USERS

Name *

The name used to indicate anonymous users.

Change the value to Guest now.

Administrator role

When modules are enabled for the first time, they may define new permissions. It is generally convenient that your highest permission role is granted these new permissions so that users in that role are able to use and configure the module right away.

The Administrator role is the default role that is assigned the new permissions, but you can use this dropdown menu to change the role to an alternative. For example, you may have a Superuser role to which you want all new module-management permissions to be assigned automatically.

▼ ADMINISTRATOR ROLE

Administrator role

Administrator ▾

In your site, we can leave this setting as it is.

Registration and cancellation

▼ REGISTRATION AND CANCELLATION

Who can register accounts?

Administrators only

Visitors

Visitors, but administrator approval is required

require email verification when a visitor creates an account

This screen allows you to define the behavior for newly created user accounts. Firstly, you can determine who is entitled to create new user accounts. The setting here is often determined by the type of site you are building:

Who can register accounts	Example uses
Administrators only	A private site or intranet. A site that does not have user interaction.
Visitors	A blog site where new users are encouraged to create content, an open forum, or an e-commerce site.
Visitors, but administrator approval is required	A blog where comments are moderated.



If you select the '...administrator approval...' option, don't forget that someone will have to manually enable each newly registered user account. This can be very time consuming for a busy site.

Based on the requirements we mentioned at the beginning of this section, we will need to change the setting to **Visitors**.

We will keep the checkbox for e-mail verification ticked so that people can only create accounts if they have a valid e-mail address. This should reduce the amount of spam accounts.

Enable password strength indicator

Unchecking this box will prevent the password indicator showing on the new user creation page. While this is a useful tool to assist the user in selecting a password, some find it annoying and it might not work well on old browsers.

Account cancelation behavior

A canceled user account in Drupal does not necessarily mean that the account is deleted. The next option allows you to determine what happens to the user account itself and to the user-generated content when the user account is canceled.

For example, if a user has commented on a news article, what should happen to the comment if the user account is deleted?

The options are given in the following table:

Option	Examples of why you might use this option
Disable the account and keep its content.	You want to keep any content the user has created but do not want the user to access the site or create any more content. You may want to allow the user to access the site again at a later date. You want to preserve the threads of conversation in comments.
Disable the account and un-publish its content.	You want to keep the content the user has created but want to prevent other site users seeing it.
Delete the account and make its content belong to the Guest user.	You want to permanently remove the user account but keep the content the user generated. Any content this user generated will now be displayed as "guest" or anonymous content.

Note that it is possible to set a permission for a role to enable the user to make their own decision on what happens to their content when they delete their account.

For our example, to make sure you don't lose any content if a user account is deleted, you should use the third option otherwise when a key member of the content-editing team left, all of the content they created would be deleted.

Select **Delete the account and make its content belong to the Guest user**. Now, when a user account is deleted, the author of the article or comment will be shown as belonging to **Guest**.

Notification e-mail address

There are a number of system e-mails (discussed next) sent in response to certain user-related events.

The `from` address of the e-mails will default to the site e-mail address set in **Configuration | Site information**, unless you set a different value here.

For example, you might want the site e-mail to be `info@mysite.com` but the `from` address for automatic e-mails to be `noreply@mysite.com`.

Emails

These are the emails that are sent out automatically in different user scenarios.

They are triggered based on some of the settings you have just been looking at in the previous section.

Welcome (new user created by administrator)
Welcome (awaiting approval)
Admin (user awaiting approval)
Welcome (no approval required)
Account activation
Account blocked
Account cancellation confirmation
Account canceled
Password recovery

Multiple
scenarios
detailed in the
table below

Welcome (new user created by administrator)	An introductory e-mail is sent to a new user whose account was created manually by a site admin user. This corresponds to having the new account registration setting set to Administrators only. It would also be triggered if a new user is created in the People screen even if another new account registration setting were selected.
Welcome (awaiting approval)	This is a welcome e-mail telling the user their account has been created, but they will not be able to use it until the account has been approved. This corresponds to the Visitors, but administrator approval is required for selection in new account registration settings.
Admin (user awaiting approval)	This e-mail is sent to the Notification Email Address of the site to tell you a new user is waiting for you to approve their account.
Welcome (no approval required)	An introductory e-mail is sent to a new user when there is no other account verification required.
Account activation	A notification e-mail is sent to the user when their account has been approved by a site admin and is ready for use.

Account blocked	A notification e-mail is sent to the user when the account has been blocked by a site admin informing them that their account can no longer be accessed.
Account cancellation confirmation	Confirmation e-mail is sent to the user when they request the cancellation of their account. This will contain a link to confirm the cancellation request so that the user does not cancel their account by mistake.
Account canceled	Confirmation e-mail is sent to the user after they have verified the account cancellation request and the account has been cancelled.
Password recovery	An e-mail is sent to the user in response to filling in the password reminder form. This will contain a one-time link to reset their password.

You can change the contents of any of these e-mails so that you can set the tone of voice and message to match your audience.

When you edit the templates, note the use of *tokens* here again such as `[user:name]`.

Tokens are elements that are replaced live by actual content in the message that gets sent out. For example, the `[user:name]` token gets replaced by the actual username of the currently logged in user, in our case `admin`.

While some tokens are optional, it's important not to remove key information like the one-time login URL (`[user:one-time-login-url]`) on a password reminder e-mail.

Available tokens are as follows:

Token	
<code>[site:name]</code>	The name of the site as set in Configuration System Site information.
<code>[site:url]</code>	The full URL of the site including the prefix <code>http://</code> .
<code>[user:name]</code>	The username of the user the e-mail being sent is related to.
<code>[user:mail]</code>	The e-mail address of the user the e-mail being sent is related to.
<code>[site:login-url]</code>	A link to the login page of the site.
<code>[site:url-brief]</code>	The website URL.
<code>[user:edit-url]</code>	A link to the user profile edit page of the user the e-mail is related to.

Configuration

Token	
[user:one-time-login-url]	A link that allows the user to login to the site once only without their password. This is used for password resets.
[user:cancel-url]	A link to a page allowing the user to confirm cancellation of their account.

In our scenario, the e-mails that will be active are as follows:

- Welcome (no approval required)
- Account blocked
- Account cancellation confirmation
- Account canceled
- Password recovery

If you are using Dev Desktop for your site development, you will be able to test these e-mails assuming that you used real e-mail addresses when setting up the site and user accounts.

System

The next section of the configuration page is concerned with site-wide settings. The section is broken into two subsections.

Site information

The Site information screen can be reached at **Configuration | System | Site Information** ([admin/config/system/site-information](#)).

It allows you to set some important site-wide parameters for your site.

Setting	
Site name	This setting is used throughout the Drupal site to represent your site—from the <title> tag displayed on each page to the welcome e-mails sent out to new users.
Slogan	The Slogan field is used in the default themes and is the strapline or purpose of your site.
Email address	The main e-mail address of the site—this is the address that any contact forms will be sent to by default and the default from address of any messages sent out by the site.

Setting	
Default front page	When a user visits your site domain name with no additional data in the URL, this is the page that will be presented. You can specify node/x where x is the node ID you want the user to see.
Default 403	When a user attempts to visit a page they are not permitted to see (such as an admin page when they are not logged in), they will be presented with the page entered here. You can specify node/x where x is the node ID you want the user to see.
Default 404	When a user attempts to visit a page that does not exist on the site, they will be presented with the page entered here. You can specify node/x where x is the node ID you want the user to see.

Cron

Cron is a technical term referring to timed processes that run on a webserver.

Typically, these are used for background tasks like cleaning old stale cache data and indexing search terms. Many Drupal modules will react to a cron event and perform relevant background tasks.

Depending on the purpose of your site, it may be appropriate to run cron once a day or once a minute. This choice really depends on what background tasks will run and how often they need to be triggered.

You can either run cron manually or set how often it should run automatically. The default value is every 3 hours and this is an appropriate setting if you are not sure.

The latter is a useful setting to have switched on locally when you are developing a website as it will essentially simulate a cron run every three hours. However, it may become annoying to experience the delay cron running every time you go back and visit your site after anything more than a three-hour period away so you may choose to switch it to **Never** and only run cron manually when you need to test its effect.

The screenshot shows the Drupal 'Cron' page. At the top, there's a header bar with the title 'Cron' and a star icon. Below the header, the breadcrumb navigation shows 'Home > Administration > Configuration > System'. The main content area contains the text: 'Cron takes care of running periodic tasks like checking for updates and indexing content for search.' Below this text is a large, rounded rectangular button with the text 'Run cron' and a red hand cursor icon pointing to it. At the bottom of the page, a message says 'Last run: 38 min 2 sec ago.'

For a live site you will need to configure your web server to call the cron task in Drupal periodically, and the setup will vary depending on where you eventually host your site. The setup is, therefore, beyond the scope of this module but is detailed thoroughly at: <https://www.drupal.org/cron>

Also note that there is a link in this screen that allows you to run cron manually from outside of the site. It's deliberately a rather long URL so that people cannot guess the path.

To manually simulate a cron run simply from this screen, click on the **Run cron** button.

Content authoring

The settings within this section control the experience of content editing for your users. It's quite a complex area with many options, so we'll go through one step at a time.

Text formats and editors

Go to **Configuration | Text formatters and editors** (`admin/config/content/formats`).

You will see four text formats listed, the particular JavaScript editor that is assigned to them, and which roles can use them:

NAME	TEXT EDITOR	ROLES
Basic HTML	CKEditor	Authenticated user,Administrator
Restricted HTML	—	Anonymous user,Administrator
Full HTML	CKEditor	Administrator
Plain text	—	<i>This format is shown when no text editor is selected.</i>

CKEditor is the name of the WYSIWYG editor included with Drupal 8.

We first looked at fields back in *Chapters 3, Basic concepts*, and *Chapter 4, Getting Started with the UI*. You also touched on the idea of text formats back in *Chapter 5, Basic Content* when you first looked at creating and editing page content, specifically the **Body** field, which is a text area rather than simply a single line of text.

Text formats apply whenever there is a text area field in use. It is possible to specify a text formatter on a per-field basis.

The idea of text formatters is that different markup effects can be applied in different scenarios. For example, you may not want people adding links in article comments.

In normal use, the text formats are allocated to a site role (see roles and permissions in *Chapter 9, Users and Access Control*). Thus, some users can use advanced formatting while others may only have more restricted options.

Let's explore this by looking at the **Basic HTML** format. Click on the **Configure** button on the **Basic HTML** row.



We can decide which roles are permitted to use this text format. You'll see from the checkboxes that this formatter can be accessed by users when they are logged in to the site.

Name *

Roles

Anonymous user
 Authenticated user
 Administrator



Note that there is also a dropdown menu that suggests that you can choose from a variety of JavaScript editors in addition to the **CKEditor**. However, by default, there are no other editors installed.

You'll recall how, back in *Chapter 5, Basic Content*, we were able to demonstrate how the toolbar changes when you switch between Basic HTML and Full HTML.

Configuration

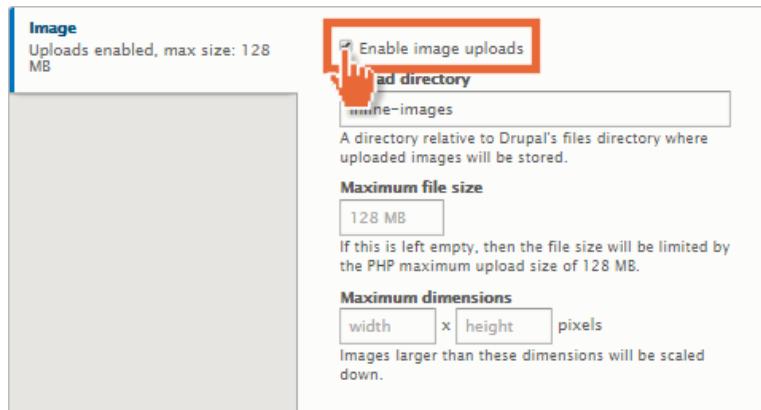
You can configure the toolbar that is available to the user when editing text areas in this particular text format.



Try dragging some buttons from the available list to the active toolbar now. You'd most likely want to do this if the type of content you are creating calls for addition HTML elements, for example, superscript or subscript if you are creating mathematical content.

As a rule, we want to keep the Basic HTML format simple so that it is easy for all content editors to understand.

Further down the page, we have the **CKEditor** plugin settings. It is configured so that you can include images within text areas and those images will be automatically uploaded to Drupal.



[ Not everyone likes this function as it is very easy to break the layout of a page by uploading an image of an inappropriate size directly into an HTML area. We recommend using image fields instead wherever practical, like the one you saw in *Chapter 5, Basic Content* when you created your first Article.]

If you don't want to allow images to be uploaded to text area fields that are using this format, untick the **Enable image uploads** box.

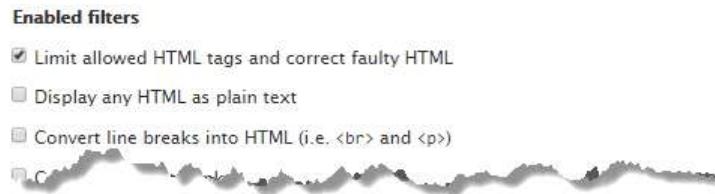
Note that dragging away the image button also removes the plugin:



In general, adding or removing elements from the toolbar will also adjust the corresponding plugins and their respective settings.

The final three sections of the screen (**Enabled filters**, **Filter processing order**, and **Filter settings**) work in combination.

There are a number of filters that can be set to automatically apply. The filters are actually applied at the time of building the page and do not modify the original content. We'll go through in detail what each one of these does, but you'll see here that you can easily enable and disable the filters using the checkboxes.



Underneath the filter list, you'll see that some filters have a settings panel and the order in which filters are applied can be changed by dragging them up and down the list.

Configuration

Depending on what you are looking to achieve, the order in which the enabled filters are applied could be important.

The screenshot shows the CKEditor configuration interface. On the left, there is a vertical list of filters: Limit allowed HTML tags, Align images, Caption images, Restrict images to this site, and Correct faulty and chopped off HTML. Below this list is a section titled "Filter settings". Under "Filter settings", there is a box for "Limit allowed HTML tags". The box has a blue header bar with the title "Limit allowed HTML tags" and the status "Enabled". Inside the box, there is a list of "Allowed HTML tags" which includes <a>, , , <cite>, and <ct>. Below this list is a descriptive text: "A list of HTML tags that can be used. JavaScript event attributes, JavaScript URLs, and CSS are always stripped." At the bottom of the box, there are two checkboxes: "Display basic HTML help in long filter tips" and "Add rel="nofollow" to all links".

Available filters

Name	Description
Limit allowed HTML tags	Strips out any HTML in the content except for those tags included in this list. Note that you can specify which HTML tags you want to allow in the settings for this filter.
Display any HTML as plain text	Do not allow any HTML markup at all—strip these out completely and only display as plain text.
Convert line breaks into HTML	When the user has entered line breaks and paragraph breaks in the editor, convert these to and <p> tags so that the same layout is achieved in a web browser.
Convert URLs into links	If a URL is entered in the text, automatically make this a link so that the user doesn't have to when creating the content.
Align images	This applies an HTML 5 standard attribute to image tags to correctly align them.

Name	Description
Caption images	This applies an HTML 5 standard attribute for image captions.
Restrict images to this site	This ensures that images used are hosted within the same domain so that you cannot reference images on an external source.
Correct faulty and chopped off HTML	If the user has entered html directly, attempt to correct any mistakes or missing tags.
Track images uploaded via a Text Editor	This ensures that images uploaded via the editor are displayed correctly if they are updated elsewhere in Drupal for example, by editing the image in the files tab under: Manage Content Files (admin/content/files)

Don't forget to save your settings after making alterations.

User interface

The user interface section allows you to define some settings for the back end/admin behavior of Drupal.

Shortcuts

Shortcuts appear in the top menu, and when clicked on, show a second level of navigation. If there are a common set of tasks you use to manage your Drupal site, it may be useful to add them here.

You have been using the shortcuts throughout this module so far in the form of the links for **Add content** and **All content**.

You can set up multiple shortcut sets and assign them to different users. This allows you to set up easy navigation for different types of users of your site without exposing them to the full range of settings, which may be confusing.

Configuration

Let's improve the default shortcut set now on our site at:

Configuration | User interface | Shortcuts ([admin/config/user-interface/shortcut](#)).

The screenshot shows the 'Shortcuts' configuration page. At the top, there is a header bar with the title 'Shortcuts' and a star icon. Below the header, a breadcrumb navigation shows 'Home > Administration > Configuration > User interface'. A note says 'Define which shortcut set you are using on the [Shortcuts tab](#) of your account page.' A blue button labeled '+ Add shortcut set' is visible. Below this, a table lists three shortcut sets:

NAME	OPERATIONS
Default	List links
Status report	Edit
Add content	Edit
All content	Edit

Press the **List links** button and you will see the three existing ones; two defaults and the Status report on you added back in *Chapter 4, Getting Started with the UI*.

The screenshot shows the 'List links' view, which displays the three existing shortcut sets from the previous screenshot. Each row contains a plus sign icon, the shortcut name, and an 'Edit' button with a dropdown arrow.

NAME	OPERATIONS
Status report	Edit
Add content	Edit
All content	Edit

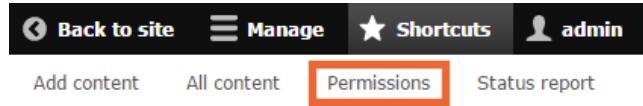
We'll add two new shortcuts now using different techniques. Firstly, we'll add a shortcut to the permissions page, which will be useful in the next chapter.

Click on **Add shortcut**.

All we have to do now is give the link a name (**Permissions**) and then add the path: </admin/people/permissions>.

The screenshot shows the 'Add shortcut' form. It has two main fields: 'Name *' with the value 'Permissions' and 'Path *' with the value 'http://drupal-8.dd:8083/admin/people/permissions'. There is also a small circular icon with a question mark.

Click on **Save** and rearrange the order so that they now appear like this:



We'll add another shortcut second in a different way like you did back in back in *Chapter 4, Getting Started with the UI*.

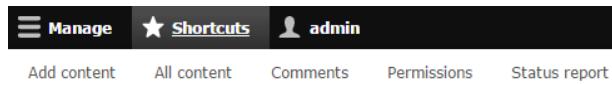
Go to **Content | Comments** and click on the star symbol next to the **Comments** title:



Go to **Configuration | User Interface | Shortcuts** (`admin/config/user-interface/shortcut`).

Your shortcuts set should now contain four links in total.

Note that you can change the order as before by returning to the listing page and dragging and dropping the list items.



Development

The development section has a number of screens and settings that are most useful when you are building your website and less so once it is live. The settings here are often different between development and going into production. Some of these settings will be changed just before a site goes into production.

Performance

The performance settings page has two subsections that perform distinctly different functions.

Caching

For anonymous visitors to your website, it's very common for each visit to a specific page to return the same generated HTML.

Depending on the complexity of the page, Drupal will have to make many calls to the database to collect all of the data it needs to assemble the page view. If the page is always the same, it is wasted effort to reassemble it every time someone requests the page. Therefore, we can specify that the generated page output is cached for a certain period of time.

This means that subsequent visitors to the same page will receive it more quickly and with less effort by the web server.

The downside of caching is that if you change the content, the user may be presented with the cached page and not see the updated content. This is why you can set the maximum age of the cache here to anything from no cache at all to 1 day. A good compromise is about 3 hours, but you must make sure that you have a non-zero value specified here or your site will be very slow.



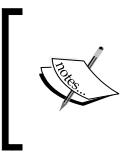
There has been a great deal of work in Drupal 8 to develop internal caching so that it is cleared in an intelligent way when content is updated. This means that if you do change something in an article for example, it will be cleared in the cache even if you have a long cache lifetime value.

Clear cache

Once you have spent any amount of time working with Drupal or looking online for answers to common problems, you will inevitably come across the response: "Have you cleared the caches?"

There are many sets of data Drupal needs to process a page that are called every time. It is not very efficient for these to be loaded from the source database each time. For this reason, these data are cached for varying lengths of time. Often you will find that a change to a setting will not be immediately obvious on the website. This is a common scenario where the cache is delivering the data and the website does not know about the change yet.

Pressing the **Clear all caches** button on this page will reset these caches and force Drupal to reload the latest values, rebuilding the cache as it does so.



Note that when you do click on this, the next page and any other pages will be slower the first time they load after clearing cache as Drupal rebuilds its cache data.



Bandwidth optimization

Drupal modules often introduce their own CSS and JavaScript files to support module functionality. As there are often a large number of modules enabled on a site, this can lead to many small files being delivered per page request. This means that a web browser has to make multiple requests to load the page and therefore increase the total time for the page to load. In addition, some older web browsers will only load a certain number of JS and CSS files before stopping – meaning that pages on your website will not render correctly.

To avoid this problem, Drupal is able to aggregate the JS and CSS files, that is, join all those small files together and deliver them in one request (or at least a significantly smaller number of requests).

Note that both **Aggregate CSS files** and **Aggregate JavaScript files** are enabled by default.

Logging and errors

When working on a site, various modules will log errors and warnings in different scenarios.

Logging and errors

[Home](#) » [Administration](#) » [Configuration](#) » [Development](#)

Error messages to display

- None
- Errors and warnings
- All messages
- All messages, with backtrace information

It is recommended that sites running on production environments do not display any errors.

Database log messages to keep

1000 ▾

The maximum number of messages to keep in the database log.

Requires a [cron maintenance task](#).

Configuration

The first section determines whether error messages are displayed on the screen in the messages region of the page. This is useful when you are building a site but should always be set to **None** when your site is in production mode—otherwise your users could see some rather ugly error messages that they probably cannot do anything about.

In the second section, the **Database log messages to keep** value determines how much data is stored in the watchdog database table; this is the table where logging messages are stored and can be viewed at **Reports | Recent log messages** (admin/reports/dblog).

This database table will keep growing infinitely as the site is used, so it's a good idea to keep just the most recent messages.

We'll be looking in more details at reports later.

Maintenance mode

If you are working on some changes to your site that might cause problems for users, it can be useful to place the site to **Maintenance mode** while you perform the work. This means that only users with the permission **Use the site in maintenance mode** can view anything on the site.

This permission would usually only be applied to administrator level users:

Maintenance mode ☆

[Home](#) » [Administration](#) » [Configuration](#) » [Development](#)

Use maintenance mode when making major updates, particularly if the updates could disrupt visitors or the update process. Examples include upgrading, importing or exporting content, modifying a theme, modifying content types, and making backups.

Put site into maintenance mode
Visitors will only see the maintenance mode message. Only users with

For everyone else, you can customize the message that is displayed to the user:

Message to display when in maintenance mode

@site is currently under maintenance. We should be back shortly. Thank you for your patience.

Note that @site will be replaced with the site name set at **Configuration | Site information** (admin/config/system/site-information).

Configuration synchronization

Detailed coverage of configuration management is beyond the scope of this module (see more on configuration management in *Module 2, Drupal 8 Development Cookbook* of this course) but in short, this section empowers you to export either the entire configuration of your site; content types, taxonomies, views and so on as a single (zipped) archive or to choose to target one particular item such as a view to be exported as pure code.

It also enables you re-import and even provides a neat comparison view of what's changed in your configuration since you last exported it.

This section is enormously useful when it comes to deploying new or updated configuration from a development site to live.

Media

Next we will look at the different settings available for media management in Drupal 8.

Go to the **Configuration** page where you will see a number of entries under the heading **MEDIA**.

The screenshot shows the 'MEDIA' section of the Configuration page. It contains three items:

- File system**: Tell Drupal where to store uploaded files and how they are accessed.
- Image styles**: Configure styles that can be used for resizing or adjusting images on display.
- Image toolkit**: Choose which image toolkit to use if you have installed optional toolkits.

File system

Starting from the top, click on the **File system** option.

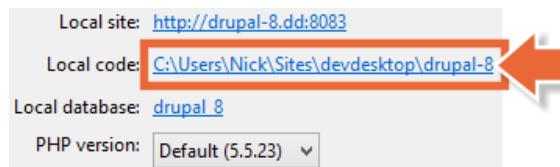
Later we will find that other modules give us more options in this screen, but for now, let's have a look at what this setting screen tells us.

Public file system path
sites/drupal-8.dd/files
A local file system path where public files will be stored. This directory must exist and be writable by Drupal. This directory must be relative to the Drupal installation directory and be accessible over the web. This must be changed in settings.php

The public file system path is the location on the web server where any files uploaded to Drupal via the user interface will be placed.

The public file system folder can be found inside the sites/default/files folder in your Local Drupal codebase.

You can locate the Drupal codebase on your computer now using Dev Desktop; click on the **Local code** link on the control panel:



Then navigate down to public files folder at sites/default/files.

The **Temporary directory** is used when files are uploaded for temporary storage. The value of this depends on the operating system your computer or server is running.

Temporary directory
C:\windows\temp

It is important to note that Drupal understands what files are available based on a database table that is updated when files are uploaded. For this reason, simply copying files to the folder will not allow them to be used within Drupal.

This is also where the **Delete orphaned files after** setting comes in.

Delete orphaned files after
6 hours ▾
Orphaned files are not referenced from any content but remain in the file system and may appear in administrative listings. Warning: If enabled, orphaned files will be permanently deleted and may not be recoverable.

If a file exists in the folder but is not in the Drupal database, it is considered orphaned and will be cleaned up periodically. The length of time between becoming orphaned and the file actually being deleted is set here. You'll not normally need to adjust this.

Image styles

We looked at image styles earlier when we were applying them to content fields in the section entitled *Customizing the display*, back in *Chapter 4, Getting Started with the UI*.

You may recall, by default we have three image styles:

STYLE NAME	OPERATIONS
Large (480×480)	Edit ▾
Medium (220×220)	Edit ▾
Thumbnail (100×100)	Edit ▾

You can modify these from their default settings or we can define completely new ones.

Configuration

To recap, image styles are used to apply to images uploaded as content on your website. This means that even if your users upload huge photo direct from their camera, Drupal can automatically resize the image so that it is appropriate for use on the Web.

Also, this functionality means that we don't need to manually create lots of different size versions of an image for use as thumbnails. This is all completely automated using image styles.

We can also apply more advanced image styling such as color desaturation, as we'll see next. This means you can apply consistent effects for images across your site without having to remember something like the Photoshop filter settings used.

Let's go back to our scenario now and create a new image style.

Click on **Add image style** and create a new style called `Black and white thumbnail`.

The screenshot shows a web interface for adding a new image style. At the top, a grey bar says "Add image style ☆". Below it, a breadcrumb navigation shows "Home > Administration > Configuration > Media > Image styles". A form field labeled "Image style name *" contains the text "Black and white thumbnail". Below the field, a smaller text box shows the "Machine name: black_and_white_thumbnail [Edit]".

Click on **Create new style** and you will be on the image style editing screen, as shown in the following:

The screenshot shows the "Image style editing screen". At the top, there is a "Preview" section with two images: "original (view actual size)" and "Black and white thumbnail (view actual size)". Both images show a colorful hot air balloon scene. Below the preview, the "Image style name *" field contains "Black and white thumbnail" and the "Machine name" is "black_and_white_thumbnail [Edit]". There is a "Show row weights" link. At the bottom, there are "EFFECT" and "OPERATIONS" tabs. Under the "EFFECT" tab, there is a dropdown menu "Select a new effect" and a "Add" button.

Image styles are created by applying a number of image effects one after another. Once all the image effects are applied, Drupal saves a modified copy of the image, leaving the original one untouched. This means that when the styled version of the image is required, it is quickly available.

Click the **Select a new effect** dropdown and you'll see there are a number of different effects.

Each of these effects can have additional settings as well, as described in the following:

Effect	Description
Convert	This converts the original image format to a new format – PNG, JPG, or GIF. This is useful if the original images are uploaded in a format that is not commonly used in web browsers.
Crop	This allows you to crop the image to a preset size. This means that parts of the image outside of the crop size will be lost in the styled version. You can also specify where the center of the crop should be.
Desaturate	This removes all color from an image (converting it to black and white) and offers no further options.
Resize	This allows you to specify the exact size of the styled image. Note however this can cause the image to be distorted if (for example) you enter square dimensions for an image that was originally rectangular.
Rotate	This rotates the source image by a specified angle. You can also specify the background color that is exposed as a result.
Scale	This is similar to resize but the aspect ratio (shape) of the original image is retained. This avoids the distortion effect you can end up with when using resize. When dealing with images, it's always better to scale down rather than up for best quality. For this reason, this option will only allow scaling more than 100 percent of the original size when specifically set to do so using the Allow Upscaling option.
Scale and crop	As you might have guessed, Scale and Crop is a combination of the scale and crop actions where the image is scaled to the smaller dimension specified with the larger one being cropped. This is a way to make a square thumbnail out of a rectangular image without distorting the image.

For our scenario, we are going to replace our image thumbnail style for the Article listing to be a desaturated square.

Configuration

Press **Select new effect** and then select **Scale and crop** and press **Add**, set the width and height to 200 and click on **Add effect**.



You'll now see that the list of effects for your style has been updated:



Press **Select a new effect** again and this time choose **Desaturate**.

You should notice that when you return to the style page, you will see a preview of the image style with the effects applied.



Each time you apply a new effect, you will see the preview update.

Be sure to press **Update style** when you have everything as you want.

Now we have created our new style, which we can apply to the Article content type. To do so, visit **Structure | Content types** (`admin/structure/types`).

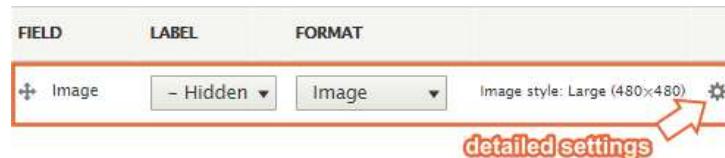
Locate the **Article** content type (1) and click on **Manage display** (2) on the article content type dropdown.



You'll recall that back in *Chapter 7, Advanced Content*, you customized the **Full** content view mode for the Article content type so to see the effect of your new Image style in action, switch to the **Full content** view mode in order to apply it.



Locate the image field and click on the gear icon.



Configuration

Now change the image style from **Large** to **Black and white thumbnail**.



Click on **Update** and then take care to also hit **Save**, and when you go back to the news page that you made on the site in the *Working with the Views module* section in *Chapter 6, Structure*, you should now see that the new image style has taken effect, as shown in the following:

The screenshot shows a news article titled 'The Company pet'. Below the title are buttons for 'View', 'Edit', 'Outline', and 'Delete'. The text 'Submitted by admin on Wed, 16/12/2015 - 16:20' is displayed. The main content area features a black and white photo of a dog wearing glasses. To the right of the image, the text 'Meet Yuki, our company pet.' and a snippet of placeholder text ('Lorem ipsum dolor sit amet, consetetur diam nonumy eirmod tempor invidunt magna aliquyam erat, sed diam volut') are visible. A red arrow points to the image with the text '200 by 200 pixels'.

Image toolkit

A final note about image resizing before we move on.

Go back to the **Configuration** page and navigate down to the **MEDIA** section again.

The screenshot shows the 'Image toolkit' configuration page. At the top, there's a header with the title 'Image toolkit' and a star icon. Below the header, the breadcrumb navigation shows 'Home > Administration > Configuration > Media'. Under the title, there's a section titled 'Select an image processing toolkit' with a radio button next to 'GD2 image manipulation toolkit' which is selected. Below this, there's a section titled '▼ GD2 IMAGE MANIPULATION TOOLKIT SETTINGS' containing a 'JPEG quality' field with a value of '75 %'. A note below the field says 'Define the image quality for JPEG manipulations. Ranges from 0 to 100.' At the bottom of the configuration area, there's a blue 'Save configuration' button.

The setting here determines the quality of JPEG images when they are resized. The default value is 75 percent, which is a reasonable compromise between file size and quality.

However, we'll change the value to 90 percent so as to have slightly better quality resized images.

Before we do, try drastically changing the quality to something like 10 percent so that you can see the effect of JPEG data loss, and then save the configuration settings and go back to the news page to see the effect.

Search and metadata

As we add more content to our demo site, it might be useful for us to add a search function for our users. Drupal 8 has a built-in search that we can configure here.

Go to the Configuration page again and look at the contents of the **SEARCH AND METADATA** section.

Click on **Search pages** and you will see the configuration for the main search of the site.

We'll work our way through these settings now.

Indexing progress

In order to allow users to search for content, Drupal will index the content in its search database. When new content is added to the site, this is not indexed immediately but is instead queued to be processed later. This shows you how many items of content have been indexed and how many are waiting.

The screenshot shows the 'Search pages' configuration page. At the top, there's a header bar with the title 'Search pages' and a star icon. Below it, a breadcrumb navigation shows 'Home > Administration > Configuration > Search and metadata'. A section titled 'INDEXING PROGRESS' is expanded, showing a message: 'Only items in the index will appear in search results. To build and maintain a correctly configured cron maintenance task is required.' It also displays the status: '100% of the site has been indexed. There are 0 items left to index.' A 'Re-index site' button is visible at the bottom of this section.

Note that if you choose to re-index the site then you will also need to run cron again manually to update the index.

Indexing throttle

Drupal will look at the contents of the indexing queue periodically in what's referred to as a Cron task, as discussed earlier in the chapter in the *Cron* section. This basically means the indexing process will happen in the background periodically. We'll discuss how to set this up later, but for now we should assume this will be running once every 3 hours.

The screenshot shows the 'INDEXING THROTTLE' configuration section. Under 'Number of items to index per cron run', a dropdown menu is set to '100'. A descriptive text below explains: 'The maximum number of items indexed in each pass of a cron maintenance task. If necessary, reduce the number of items to prevent timeouts and memory errors while indexing. Some search page types may have their own setting for this.'

If lots of content is created, we can add a throttle so as not to overwhelm the webserver by trying to process all of the content at once.

Instead, we can set a throttle so that (as is the default) only 100 items are processed each time.

This is a sensible default for most sites.

Default indexing settings

The default indexing settings allow you to have more control over what content is added to the search index.

▼ DEFAULT INDEXING SETTINGS

Search pages that use an index may use the default index provided by the Search module, or they may use a different indexing mechanism. These settings are for the default index. *Changing these settings will cause the default search index to be rebuilt to reflect the new settings. Searching will continue to work, based on the existing index, but new content won't be indexed until all existing content has been re-indexed.*

The default settings should be appropriate for the majority of sites.

Minimum word length to index

The number of characters a word has to be to be indexed. A lower setting means better search result ranking, but also a larger database. Each search query must contain at least one keyword that is this size (or longer).

Simple CJK handling

Whether to apply a simple Chinese/Japanese/Korean tokenizer based on overlapping sequences. Turn this off if you want to use an external preprocessor for this instead. Does not affect other languages.

By default, only words of three characters or more are indexed, but you can change this if you want. This default is sensible though as there's not much value in searching for two letter words in most sites.

Logging

By default, searches are not logged.

▼ LOGGING

Log searches

If checked, all searches will be logged. Uncheck to skip logging. Logging may affect performance.

If you change this option, every search carried out will be recorded to the Drupal database. This can have a negative effect on performance, so it is usually sensible to leave it switched off.

Search pages

The last section of the search configuration page is the most important one.

LABEL	STATUS	INDEXING PROGRESS	OPERATIONS
Content	Default	30 of 30 indexed	Edit
Users	Enabled	Does not use index	Edit

Before we go into detail here, click on the **Back to site** link to go back to the site and see how the search works now.

The search box appears on the left-hand side of the screen.

Let's try out an example search now based in the content we added earlier. Try searching for **dog**.



Notice the search term used is highlighted in the results.

Search results

Dog fish(er)

Dog fish(er) ... Tags dog fish ...

admin - 06/26/2015 - 08:02 - 0 comments

At the top of the screen, you should see two tabs. These tabs represent the different search pages as detailed in the configuration screen we were looking at earlier.



If you switch between the tabs, you are searching different types of content for the same search term.

Go back to the search configuration now **Configuration | Search pages** (admin/config/search/pages).

We will look in more detail at the search page configuration for content.

Select **Edit** next to the **Content** search page.

Here we can control a number of things about the search page.

The **Label** corresponds to the value in the tab the user can see.

Let's change this to `Site search`.

The path is the URL of the search page—we can change this as well, but for now let's leave it alone.

Label	Site search
Machine name: node_search	
The label for this search page.	
Path	search/
	node

Finally, we can change the content ranking settings, which determine the order of the search results. If a user searches for a term that is commonly used in the content across the site, it can be useful to change the influence of certain factors so that more relevant content appears earlier in the search results.

Changing the settings as follows would mean that more recent content that is promoted to the front page will appear earlier in the rankings.

Configuration

Don't forget to save the changes.

FACTOR	INFLUENCE
Number of comments	0 ▾
Keyword relevance	0 ▾
Content is sticky at top of lists	0 ▾
Content is promoted to the front page	5 ▾
Recently created	10 ▾

This is easier to see in effect when you have lots of content on the site, but have a play with these settings now and see how they change the searches you perform.



Note that you can also create multiple search pages with different influencing factors if you want to create a more complex search on your site.



URL Aliases

Now go back to **Configuration | URL Aliases** (`admin/config/search/path`) (inside the **Search and metadata** section).

When you create content in Drupal, it is assigned a unique node ID (referred to as a `nid`).

You can use this to view the node on your site by entering the URL as follows:

`/node/nid` for example, to view node 1, the URL is `/node/1`

Rather than having to know the `nid` of all content, you can also specify one or more URL alias entries that will also get you to the page.

For example, `/latest-news` is easier to remember than `node/55`.

You can also use URL aliases for promotional URLs, for example, `/competition` linking to `node/1234`.

The URL alias page shows a list of all aliases that have been created.

A screenshot of a web page titled 'URL ALIASES'. At the top left is a blue button labeled '+ Add alias'. Below it is a section titled 'FILTER ALIASES' with a search input field and a 'Filter' button. A horizontal navigation bar below shows 'ALIAS' (which is highlighted in grey) and 'SYSTEM'. A message at the bottom states 'No URL aliases available. Add URL alias.'

At the moment, this is blank as we have not been setting up aliases for our scenario so far. So let's set some up now.

From the URL alias page, we can create them directly. You may remember from earlier that our **Contact Us** page is node 6.

Click on **Add alias**.

The screenshot shows a form for adding a URL alias. It has two main sections: 'Existing system path *' and 'Path alias *'. In the 'Existing system path *' section, the value '/node/6' is entered into a text input field, circled with a red number 1. In the 'Path alias *' section, the value '/contact-us' is entered into a text input field, circled with a red number 2. Below the fields are two buttons: 'Save' (highlighted with a red circle and a hand cursor icon) and 'Delete'. A red number 3 is circled around the 'Save' button.

Enter node/6 for the existing system path and contact-us for the path alias. Press **Save**.

Now, when you go to the URL `http://drupal-8.dd:8083/contact-us`, you will get exactly the same page as if you had typed `http://drupal-8.dd:8083/node/6`.

Go to the page now and click on **Edit**.

Configuration

If you expand the URL alias section in the right-hand side of the edit form, you should see your alias presented.

The screenshot shows a 'Published' content item with the last save date as 06/10/2015 - 15:07 and author as admin. There is an option to 'Create new revision'. Below this, there are sections for 'MENU SETTINGS' and 'URL PATH SETTINGS'. Under 'URL PATH SETTINGS', the 'URL alias' field contains 'contact-us', which is highlighted with a red box and has a red arrow pointing to it from the left. A tooltip below the field states: 'The alternative URL for this content. Use a relative path. For example, enter "about" for the about page.' At the bottom of the panel, there are sections for 'AUTHORIZING INFORMATION' and 'PROMOTION OPTIONS'.



You can have multiple URL aliases that point to the same page, but be aware that search engines such as Google could interpret this as duplicate content and penalize you as a result.



Regional and language

Drupal 8 is capable of supporting sites in multiple languages. The details of the translation mechanism are discussed later. The regional and language settings allow you to set some defaults for the target market of your site.

Regional settings

Go to Configuration | Regional and language | Regional settings ([admin/config/regional/settings](#)).

Locale

From here you can set the **Default country** and **First day of the week**. These will have been set based on your choices during the install process, but you can change them now if they are incorrect.

▼ LOCALE

Default country
United Kingdom

First day of week
Monday

Time zones

The default time zone for the site will also have been set based on the country you selected during the install process. You can change this now, but also you can decide whether you want to allow visitors to your site to specify their own time zone.

If you allow visitors to determine their own time zone, time-sensitive data will be presented to that user in their own localized time. This would affect, for example, the posting times of articles or the time of an event.

▼ TIME ZONES

Default time zone
Europe/London

Date and time formats

Go to Configuration | Regional and Language | Date and time formats (admin/config/regional/date-time).

NAME	PATTERN
Default long date	Saturday, December 19, 2015 – 10:25
Default medium date	Sat, 12/19/2015 – 10:25
Default short date	12/19/2015 – 10:25
Fallback date format	Sat, 12/19/2015 – 10:25
HTML Date	2015-12-19
HTML Datetime	2015-12-19T10:25:47+0000
HTML Month	2015-12
HTML Time	10:25:47
HTML Week	2015-W51
HTML Year	2015
HTML Yearless date	12-19

Configuration

Here you can set the date formats for your site.

By default, there will be a set of date formats which may not be appropriate for your local users:

Format Name	Presented as
Long	Saturday, December 19, 2015 - 10:23
Medium	Sat, 12/19/2015 - 10:23
Short	12/19/2015 - 10:23

For example, in the UK, dates are presented as dd/mm/yyyy as opposed to mm/dd/yyyy, so let's change this now.

Click on **Edit** on the **medium** date format and edit it to the UK standard.

Name

Machine name: medium

Name of the date format

Format string *

Displayed as *Tue, 30/06/2015 - 17:11*

A user-defined date format. See the [PHP manual](#) for available options.

Entering a valid date format string here will ensure that all dates presented in short form will display in your new format. You can see the effect of this immediately in the preview.

Similarly, edit the **short** format to UK style too.

Name

Machine name: short

Name of the date format

Format string *

Displayed as *30/06/2015 - 17:10*

A user-defined date format. See the [PHP manual](#) for available options.

The current value uses standard PHP abbreviations for dates that can be found in full at <http://php.net/manual/function.date.php>.

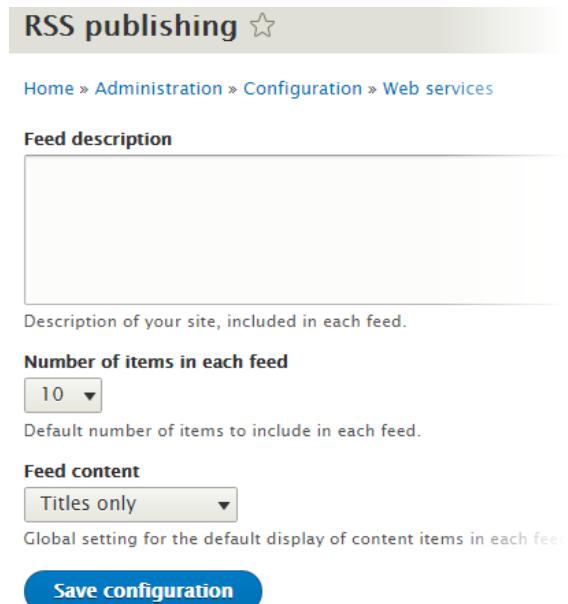
Now when you go back to the article list views we created earlier, the new date formats will be applied.

Web services

In the final section of the configuration screen we will look at is the web services section. In the standard installation, this relates only to RSS publishing of content on the site.

RSS publishing

RSS stands for **Really Simple Syndication** and is a standard way to share your content to people as a channel.



The screenshot shows the 'RSS publishing' configuration page. At the top, there's a title bar with the text 'RSS publishing' and a star icon. Below it, a breadcrumb navigation shows 'Home > Administration > Configuration > Web services'. The main content area has a section titled 'Feed description' with a large text input field. A placeholder text 'Description of your site, included in each feed.' is visible inside. Below this is a 'Number of items in each feed' section with a dropdown menu set to '10'. A note below says 'Default number of items to include in each feed.'. Under 'Feed content', there's a dropdown menu set to 'Titles only'. A note below says 'Global setting for the default display of content items in each feed.' At the bottom is a blue 'Save configuration' button.

Precisely what you see when you enter this URL in your web browser will vary depending on your computer and browser.

In the standard installation, Drupal will expose an RSS feed of content that is currently promoted to the front page on the URL <http://drupal-8.dd:8083/rss.xml>.

Configuration

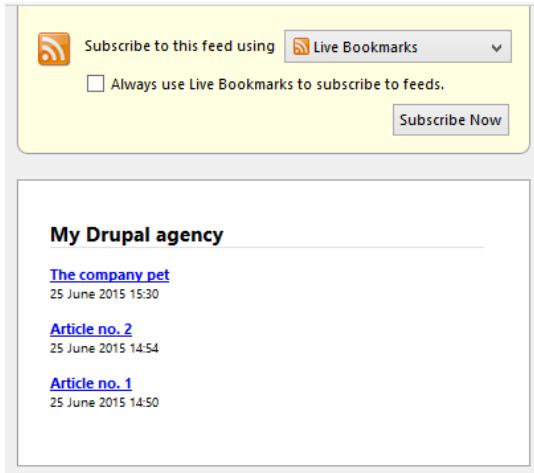
It will be interpreted as a list of links and summaries of the articles on your website.

For example, in Google Chrome without any dedicated plugins, you will see:

```
<?xml version="1.0" encoding="utf-8" ?>
<rss version="2.0" xml:base="http://drupal-8.dd:8083/rss.xml"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:content="http://purl.org/rss/1.0/modules/content/"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:og="http://ogp.me/ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:schema="http://schema.org/"
  xmlns:sioc="http://rdfs.org/sioc/ns#"
  xmlns:siocTypes="http://rdfs.org/sioc/types#"
  xmlns:skos="http://www.w3.org/2004/02/skos/core#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
  <channel>
    <title>My Drupal agency</title>
    <link>http://drupal-8.dd:8083/rss.xml</link>
    <description></description>
    <language>en</language>

    <item>
      <title>The company pet</title>
      <link>http://drupal-8.dd:8083/blog-posts/2015-06/company-pet</link>
      <description><span data-quickedit-field-id="node/8/title/en/rss" class="field field-node--title field-name-title field-type-string field-label-hidden">The company pet</span>
<span data-quickedit-field-id="node/8/uid/en/rss" class="field field-node--uid field-name-uid field-type-entity-reference field-label-hidden"><a href="/users/admin" title="View user profile.">admin</a></span>
<span data-quickedit-field-id="node/8/label/en/rss" class="field field-node--label field-name-label field-label-hidden">about</span></description>
      <content><p>The company pet</p></content>
    </item>
  </channel>
</rss>
```

By contrast, in standard Firefox, you might see:



The configuration page allows us to determine how many articles are in the RSS feed and whether you should include summary content as well as the article title.

Summary

We covered a great deal in this chapter. You saw that many modules in the standard installation profile expose a configuration screen and this controls the behavior and user interaction of the website.

In particular, we looked in detail at the e-mails sent by Drupal and more in depth image styles. As you activate more optional modules, you'll find that more configuration options become available in this section.

Next we'll look into User accounts and the detailed roles and permissions options that Drupal offers.

9

Users and Access Control

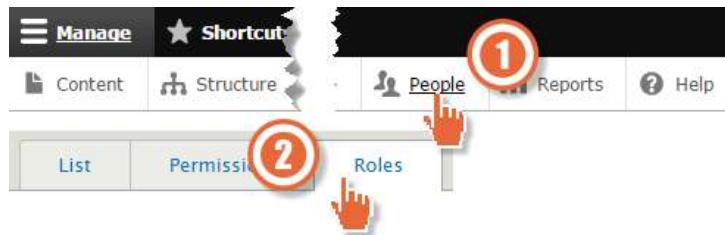
In this chapter, we will look at how Drupal uses the idea of roles as containers to allocate permissions to the actual site users. We'll investigate how to create user accounts in your example site and how you can give those users different capabilities using the roles and permission system. We will end by showing you how to customize the user management screen.

Users and roles

Drupal allows an unlimited number of **user accounts**—site users each with a unique username, e-mail address, and password—and it employs roles to which you can attach permissions. A site user's capabilities are dictated by the role(s) the user is a member of and the permissions that are assigned to those role(s).

Each role, including the special **Anonymous** and **Authenticated user** roles, is granted one or more named permissions allowing them to perform certain tasks.

A list of the currently defined roles is found by navigating to **People | Roles** (`admin/people/roles`):



Here, you'll see a list of the roles available on your site by default:

NAME	OPERATIONS
⊕ Anonymous user	<button>Edit</button>
⊕ Authenticated user	<button>Edit</button>
⊕ Administrator	<button>Edit</button>

Let's see how these roles are defined:

- **Anonymous user:** This is anyone who is not logged in, that is, public visitors to your site
- **Authenticated user:** This is used for logged in users
- **Administrator:** This is a special Drupal role that is granted a high level of permission—often dangerously high for many users and really only meant for senior site administrators

Permissions

Navigate to **People | Permissions** (`admin/people/permissions`).

This page lists out all the permissions organized by a module and provides a column for each role. You can see that it's possible to allocate permissions in a very granular way—a key feature of Drupal:



Navigate your way through the permissions grid and locate the section containing all the permissions related to the Contact module:

PERMISSION	ANONYMOUS USER	AUTHENTICATED USER	ADMINISTRATOR
Contact			
Administer contact forms and contact form settings	(1)	(2)	(3)
Use the site-wide contact form			
Use users' personal contact forms	<input type="checkbox"/>	<input type="checkbox"/>	(4) <input checked="" type="checkbox"/>

If you want both visitors and existing account holders to be able to use the contact form, then you need to allocate the **Use the site-wide contact form** permission to the **Anonymous user** (marked as (1) in the preceding screenshot) and the **Authenticated user** (2).

Note that the **Administrator** role is automatically granted all available permissions whenever a module is enabled that defines permissions, so the check box is automatically ticked (3).

Next, move down to the **Node** module section.

Drupal is frugal with its allocation of permissions. For example, while the **Node** module allows both the **Anonymous user** and the **Authenticated user** to view published content, it is very specific about who can create and edit content:

PERMISSION	ANONYMOUS USER	AUTHENTICATED USER
Node	anyone can view content	
View published content	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Typical roles

You are going to add three new roles, which are somewhere in-between the Authenticated user and the overly powerful Administrator in terms of authority:

- **Contributor:** This role can contribute content to the site, but it has limited content administration rights
- **Editor:** This role can also contribute content to the site, but has the administration rights over all content so that they can moderate others' work
- **User manager:** This role can only create and manage user accounts

Navigate to **People | Roles** (`admin/people/roles`) again, and add the Contributor role now:

The screenshot shows a web form for adding a new role. At the top is a blue button labeled '+ Add role'. Below it is a red arrow pointing down to a text input field labeled 'Role name *'. Inside the field is the word 'Contributor'. To the right of the input field is a note: 'Machine name: contributor [Edit]'. Below the input field is a descriptive text: 'The name for this role. Example: "Moderator", "Editorial board", "Site architect".' At the bottom of the form is a blue 'Save' button.

Repeat the process twice until you have three roles set up: Contributor, Editor, and User manager.



Let's look at an example scenario that we have been given, relating to various roles on our site. We'll then visit the permissions grid again and implement these stories.

A typical scenario

Let's imagine the following three criteria:

- Contributors should be able to create, edit, and delete their own Articles
- The creation and editing of Basic pages is reserved for Editors only
- We also want Editors to be able to edit any content irrespective of what type of content it is or who created it

Putting aside **Revision** permissions, here are the permissions with respect to Articles:

PERMISSION	CONTRIBUTOR	EDITOR	USER MANAGER
<i>Article: Create new content</i>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<i>Article: Delete any content</i>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<i>Article: Delete own content</i>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<i>Article: Edit any content</i>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<i>Article: Edit own content</i>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Note that we omitted permissions concerned with **Revisions** in the preceding screenshot just for clarity.

The permissions are identical for clients, services, and the testimonials sections of the permission grid, so set them up too.

In addition:

- Only the editor can delete any Articles
- Only the editor can edit any Articles
- If a role is granted permission to edit any, then editing their own is implied

Basic pages on the other hand are different, we only want editors to be able to create, edit, and delete (any of) them, so the permissions are set accordingly:

PERMISSION	CONTRIBUTOR	EDITOR	USER MANAGER
<i>Basic page: Create new content</i>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<i>Basic page: Delete any content</i>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<i>Basic page: Delete own content</i>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<i>Basic page: Edit any content</i>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<i>Basic page: Edit own content</i>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<i>Basic page: Revert revisions</i> Role requires permission <i>view revisions</i> and <i>edit rights</i> for nodes in question, or <i>administer nodes</i> .	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<i>Basic page: View revisions</i>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Once again, since editors are granted permission to edit any, then editing their own is implied.

Note that we are not assuming that editors need to be able to view and manipulate revisions – this being out of the scope of our stories.

Finally, we want user managers to manage user accounts but not more than that:

PERMISSION	CONTRIBUTOR	EDITOR	USER MANAGER
<i>Administer permissions</i> <i>Warning: Give to trusted roles only; this permission has security implications.</i>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<i>Administer users</i> <i>Warning: Give to trusted roles only; this permission has security implications.</i> Manage all user accounts. This includes editing all user information, changes of e-mail addresses and passwords, issuing e-mails to users and blocking and deleting user accounts.	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Creating user accounts

You are going to create three new users and assign each of them to one of the new roles, so navigate to **People** and click on the **Add user** button.



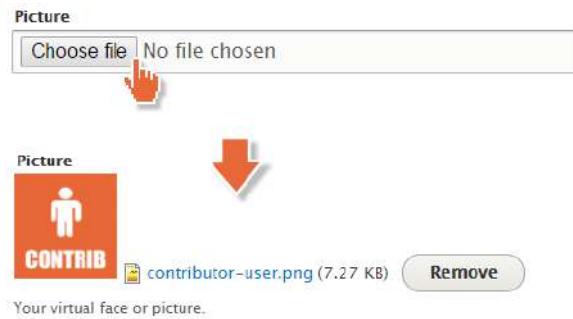
Enter each user account with a unique e-mail address, a username, and a reasonably secure password.

 A screenshot of the 'Add user' form.
 1. The 'Email address' field contains 'YourEmail+1@example.com'.
 2. The 'Username' field contains 'Contributor'.
 3. The 'Password' field shows a masked password and a progress bar labeled 'Password strength: Fair'.
 Below the password field is the 'Confirm password' field, which also shows a masked password and the message 'Passwords match: yes'.

Ensure that the account is active and you add the user to the appropriate role(s):

 A screenshot of the 'User Settings' page for a user named 'Contributor'.
 - **Status**: The 'Active' radio button is selected, with a red arrow pointing to the text 'enable the account login'.
 - **Roles**: Several checkboxes are checked: 'Authenticated user', 'Contributor' (which is highlighted with a red arrow and the text 'add the user to the right role'), 'Editor', and 'User manager'.
 - **Notifications**: The 'Notify user of new account' checkbox is checked, with a red arrow pointing to the text 'optionally email the user an account creation notification'.

Add an optional picture to your user:



Finally, choose whether or not you would like the user to be contactable by other users. Then, go ahead and create the account:

The screenshot displays two sections of a configuration form. The first section, "CONTACT SETTINGS", contains a checkbox labeled "Personal contact form" which is checked. A red annotation with a left-pointing arrow and the text "optionally adjust the default setting" is overlaid on this section. The second section, "LOCALE SETTINGS", includes a dropdown menu set to "Europe/London" and a descriptive text about selecting a local time zone. A red annotation with a left-pointing arrow and the text "optionally adjust the default setting" is overlaid on this section. At the bottom of the form is a blue "Create new account" button, with a red hand cursor icon pointing directly at it.

You will be notified that the account has been created:



You will remain on the same page ready to create another account.

Repeat the same procedure for creating the `Editor` and the `User manager` accounts until you have a total of four accounts.



Drupal does not allow you to use the same email address twice, but you can, for example, add a plus sign (+) into the email address and anything else to the right of that will be ignored. This works for Google mail (Gmail) and some other mail systems, so you can actually create multiple Drupal accounts essentially linked to the same e-mail address such as `YourEmail@example.com`, `YourEmail+1@example.com`, `YourEmail+2@example.com`, `YourEmail+3@example.com`, and so on, while keeping Drupal happy.

When you have created all three new accounts, visit `admin/people` again and you should see them listed. Take a moment to see if your user list matches this:

<input type="checkbox"/>	USERNAME	ROLES	OPERATIONS
<input type="checkbox"/>	User manager	• User manager	Edit
<input type="checkbox"/>	Editor	• Editor	Edit
<input type="checkbox"/>	Contributor	• Contributor	Edit
<input type="checkbox"/>	admin	• Administrator	Edit

Hovering over an account **Edit** link will reveal the full path in your browser status line:



For example, the `User manager` role, as the fourth user to be created has `User ID` (UID) of 4, thus the edit link is:

`http://drupal-8.dd:8083/user/4/edit?destination=/admin/people`.

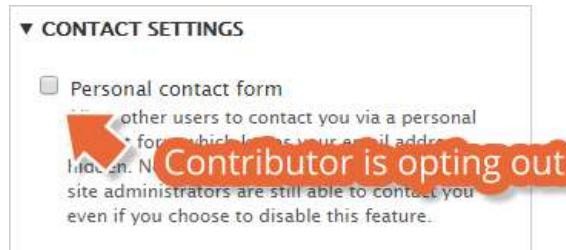
The `?destination=` suffix merely brings you back to this page after an edit. Note that destination suffixes like this are used throughout the whole of Drupal, not just on the user editing pages.

User ID 1

You are currently logged in as the administrative user with a User ID of 1.

The user with ID of 1 is a very special, all-powerful user whose actions are not governed at all by the permissions. This means that you can do absolutely anything you like without being controlled in any way by the permissions system.

Thus, even though earlier in the chapter when setting up the Contributor user account, you may have chosen not to enable the **Personal contact form**:



As the admin user, permissions such as these are irrelevant, so you still see the **Contact** tab:



Once again, a very important point, as the 'admin' user, you are not subject to any form of permission control.



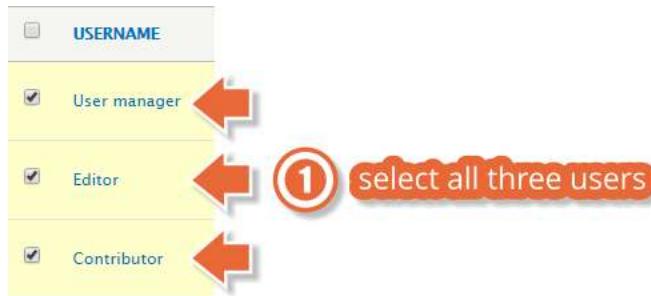
Editing accounts

Visit the account management page again by navigating to **Manage | People** (admin/people) to see the list of users.

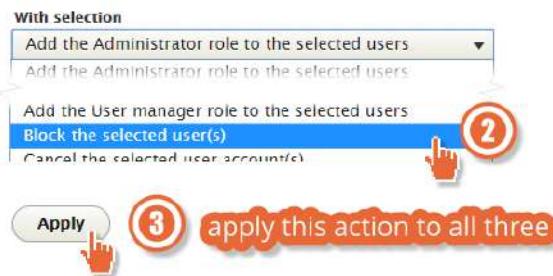
You have already seen that you can click on the **Edit** link to edit the account details but it's worth noting too that you can also perform some account management tasks enmasse.

As an example, let's temporarily block several users from logging in.

First, select the three user accounts that you just made:



Then from the **With selection** menu, choose the **Block the selected user(s)** option and click the **Apply** button:

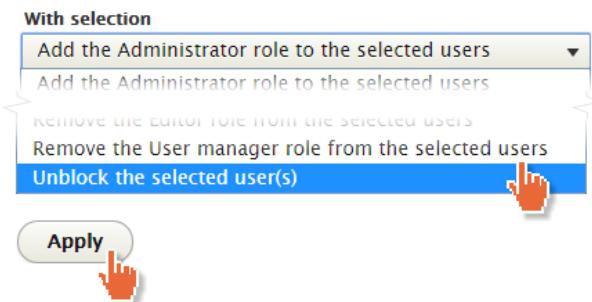


Since this is a quickly and easily undoable action, no confirmation is asked for and you will see a notification of the action and, as you can see in the following screenshot, the three users' accounts will be blocked:

✓ Block the selected user(s) was applied to 3 items.

USERNAME	STATUS	ROLES
User manager	Blocked	• User manager
Editor	Blocked	• Editor
Contributor	Blocked	• Contributor

You can, and should, reactivate the accounts by following the same procedure, but this time choosing unblock:



Taking control of the People page

You should note that the **People** screen, like many administrative screens in Drupal 8 is actually a View.

Despite the fact that you can't see the contextual links in the form of the cog wheel, you can always go the main Views listing page at **Structure | Views** (`admin/structure/views`).

Edit the view just as you would any other:

A screenshot of the "People" administrative page. On the left, there is a sidebar with "Display: Page", "Machine name: user_admin_people", and a "Find and manage with your site." link. On the right, there is a main content area with a "default" tab, the URL path "/admin/people", and an "Edit" link. A red arrow points to the "Edit" link.

Since many of the Drupal administrative pages are also views, you can start to appreciate how easily you can modify the **People** page and many other administrative UI elements within Drupal:

The screenshot shows the 'FIELDS' and 'PAGE SETTINGS' configuration panels for the 'People' administrative interface in Drupal 8.

FIELDS:

- User: Bulk update (Bulk update)
- User: Name (Username)
- User: User status (Status)
- User: Roles (Roles)
- User: Created (Member)
- User: Last (Last visited)
- User: Open (Open user)
- User: Email (Email)

FILTER CRITERIA:

- Global: Combine fields filter (exposed)
- User: Roles (exposed)

PAGE SETTINGS:

- Path: /admin/people/list
- Menu: Tab: List | Parent menu item
- Access: Permission | Administer users

Summary

This chapter introduces just the basics of managing user accounts. We covered creating new roles and how to apply permissions to these roles with fine granularity.

It also introduces you to a fantastically powerful idea that many of the administrative UI screens within Drupal 8 -- for example the People page -- are themselves simply views, thus empowering you to take total control of the backend experience for your content-editing team.

In *Chapter 10, Optional Features*, we'll look at all of the other modules provided in Drupal 8 core that are not enabled in the Standard installation profile.

10

Optional Features

There are many features of Drupal that are switched off in the standard installation. This chapter explains what they are and how you can use them.

We will be looking at the following modules:

- Activity Tracker
- Aggregator
- Ban
- Book
- Forum
- Responsive image
- Statistics
- Syslog
- Telephone

When you visit the `Modules` page of your Drupal installation at `admin/modules` or from the main menu by clicking on `Extend`, you will note that a number of modules are unticked, indicating they are not enabled. Some of these modules are really meant for module developers and are outside the scope of this module, so we won't cover every single one.

Go to `Extend` (`admin/modules`) now.

Activity Tracker

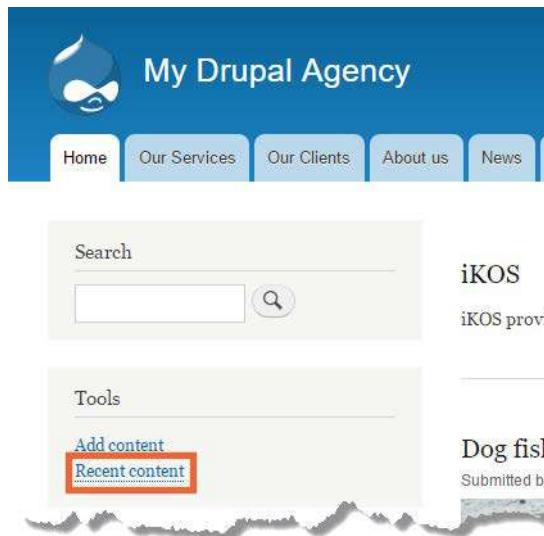
The Activity Tracker module provides a new tab on the user account page.

admin



Clicking on the **Activity** tab gives a list of content that this user has modified recently.

It also adds a Recent content menu item into the Tools menu block:



Aggregator

The Aggregator module allows you to collect information from external sources and publish it on your website. This could include RSS, RDF, or Atom feeds. For example, if you wanted to publish a list of headlines and article summaries from an external news site, you could use the Aggregator module.

Navigate to **Extend** (admin/modules) now and enable the Aggregator module.

Once enabled, the module provides new screens for us to manage external site feeds. You'll find the configuration page listed under the **WEB SERVICES** section of the administrative configuration page at: **Configuration** (`admin/config`):



The screenshot shows the 'WEB SERVICES' configuration page. It lists two options: 'RSS publishing' and 'Feed aggregator'. A hand cursor is hovering over the 'Feed aggregator' link.

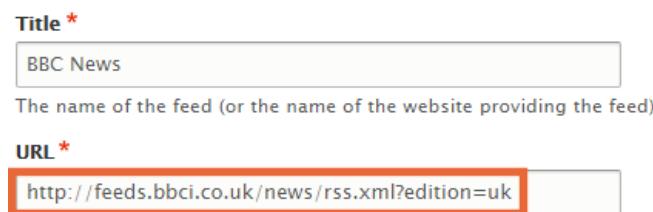
You will be taken to the overall Feed overview page where existing feeds are listed and you can add new ones.



The screenshot shows the 'Feed aggregator' overview page. It features a title bar with 'Feed aggregator' and a star icon, and three buttons: 'List', 'Manage fields', and 'Manage form display'. Below the title bar, there's a breadcrumb trail: 'Home > Administration > Configuration > Web services'. The main content area contains text about feed aggregation and links for 'Add feed' and 'Import OPML'.

As an example, we can set up a headline feed from the BBC news website, which provides an RSS feed at `http://feeds.bbci.co.uk/news/rss.xml?edition=uk`.

Click on **Add feed** and fill in the following details:



The screenshot shows the 'Add feed' configuration form. It has two fields: 'Title' (containing 'BBC News') and 'URL' (containing 'http://feeds.bbci.co.uk/news/rss.xml?edition=uk'). The URL field is highlighted with a red border.

Title *	BBC News
The name of the feed (or the name of the website providing the feed).	
URL *	http://feeds.bbci.co.uk/news/rss.xml?edition=uk

Optional Features

Save the new feed, and you will see a confirmation message that the feed has been created. You will be taken back to the **Add feed** page ready to add another.

If at this point you click on the **Sources** link in the breadcrumb.



You will be taken to a page listing all the current aggregator sources, of which you will find only one:

Sources

BBC News



Click on the **More** link, and you will see that the Aggregator reports that it has never been checked.

BBC News

[View](#) [Configure](#) [Delete](#)

Checked: never



The feed has not been checked and is therefore empty because the Aggregator module only gets triggered to check for new content on a **cron** run.

You have two options here: either run cron as described back in *Chapter 8, Configuration*, or visit the feeds overview page again by navigating to **Configuration | Services | Aggregator** (`admin/config/services/aggregator`).

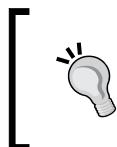
Locate the BBC News feed you just set up.

Open up the menu in the rightmost **OPERATIONS** column and click on the **Update items** button:



You should then see that the number of items is set.

TITLE	ITEMS
BBC News	83 items



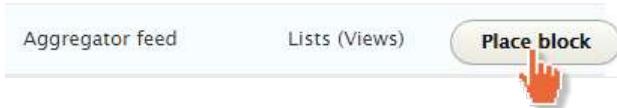
Note that the Aggregator module will also try to update your feed every hour, but that the cron is set by default to run every 3 hours. If you really want to update your feed every hour, then set cron to run once per hour too.

The Aggregator module provides a dedicated block for each feed that you create. We'll add the **BBC News** block to the **Sidebar second** now. Navigate to **Structure | Block layout** (`admin/structure/block`).

Press the **Place block** button in the Sidebar second to place the new feed's output there:



Choose the **Aggregator feed** block:



Optional Features

You can then title the block (1) and choose the precise feed (2) to show from the select list. There is of course only one item present at the moment—**BBC News**—because you have only created one feed:



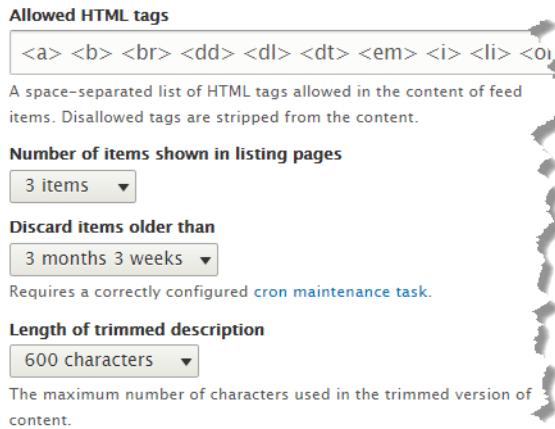
The screenshot shows a configuration form for a block. At the top, there is a field labeled "Title *" with the value "BBC News". Below it is a note: "Machine name: aggregatorfeed [Edit]". Underneath is a checked checkbox labeled "Display title". A large red circle labeled "1" is drawn around the "Title" field. Below this is a section titled "CACHE SETTINGS". Underneath that is a heading "Select the feed that should be displayed". A dropdown menu is open, showing "BBC News" as the selected option. A large red circle labeled "2" is drawn around the dropdown menu. A hand cursor icon is shown pointing at the dropdown menu.

When you save the block, it will appear on all pages on your site in the second sidebar:



There is another screen provided by the Aggregator module by navigating to **Manage | Configuration | Web Services | Feed aggregator** (`admin/config/services/aggregator`).

Click on the **Settings** tab to see the details:



The screenshot shows a configuration form for a feed settings module. It includes fields for allowed HTML tags, number of items shown in listing pages, items discarded older than a certain date, and the length of trimmed descriptions.

Setting	Description
Allowed HTML tags	<a> <dd> <dl> <dt> <i>
A space-separated list of HTML tags allowed in the content of feed items. Disallowed tags are stripped from the content.	
Number of items shown in listing pages	3 items
Discard items older than	3 months 3 weeks
Length of trimmed description	600 characters
The maximum number of characters used in the trimmed version of content.	

Here, you can filter some of the content from the external feed and also decide how long to retain it—bearing in mind if you take a feed from a busy site, like in our example, it won't be long before you have thousands of content items on your site.

Ban

This module allows you to block visitors to your site that originate from a specific IP address. The IP address can uniquely identify a particular user, but more commonly, it will resolve to a geographical or organizational group, such as a particular business or institution. Typically, you would want to do this if you are experiencing trouble from automated "bots", which are attempting to break in to the site or send spam via a web form.

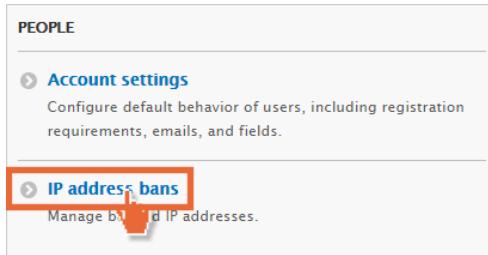
Enable the Ban module now.

Once enabled, the Ban module provides a new configuration screen labeled IP address bans.

Optional Features

Visit that screen now by navigating to **Home | Administration | Configuration | People**

(admin/config/people/ban):



To identify potential IP addresses to ban, go to the recent log messages by navigating to **Reports | Recent log messages** (admin/reports/dblog).

If you see a message that reads something like **Login attempt failed from 100.200.300.400**, you can identify that a failed login has occurred from the IP address 100.200.300.400.

Note that a single failed login may not justify a ban, but if you see repeat offenders in the logs or multiple failed login attempts from the same source, you might want to copy that IP address and add it to the ban list.

Don't worry about banning yourself because the Ban module will not let you ban your own IP address.

A screenshot of the 'IP address bans' management screen. At the top, there's a header bar with the title 'IP address bans' and a star icon. Below the header, the breadcrumb navigation shows 'Home > Administration > Configuration > People'. A red box highlights an error message: 'Enter a valid IP address.' In the main area, there's a list of banned IP addresses with the note 'IP addresses listed here are banned from your site. Banned address:'. An input field labeled 'IP address' contains '127.0.0.1', which is also highlighted with a red box. Below the input field, there's a placeholder text 'Enter a valid IP address.' and a large 'Add' button. At the bottom, there's a section titled 'BANNED IP ADDRESSES' with the message 'No blocked IP addresses available.'

Book

If you want to create a set of structured content rather than unrelated single pages, you can enable the Book module. This is ideal when creating a user guide where each page links to the next and content may be formed into chapters.

Enable the Book module now.

When you enable the Book module, a new content type Book page becomes available on the **Add content** screen.

As an example of how to use this module, let's digress a little from the *My Drupal agency* site build for a moment, and look at how this chapter could be created as a Drupal book.

To start with, we will create a new Book page called Learning Drupal 8. From the **Add content** screen, click on **Book page**.

On the right-hand side, there is a new expandable section, **BOOK OUTLINE**.

When you expand this item, you can add a new page to a specific place in the book structure. As this is the first book page we are creating, we can select **Create a new book**, fill in the **Title** field with Learning Drupal 8, and then **Save and publish** the page as normal:



This will make more sense once you have created a few more items to go in the book.

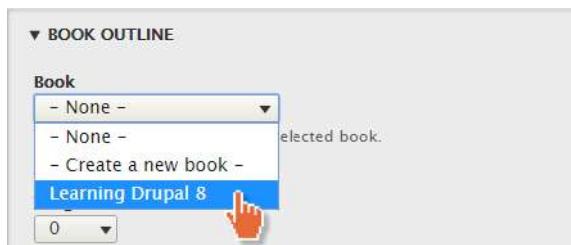
Optional Features

To demonstrate, you'll now add a Chapter 10 book page and a series of child pages as follows:

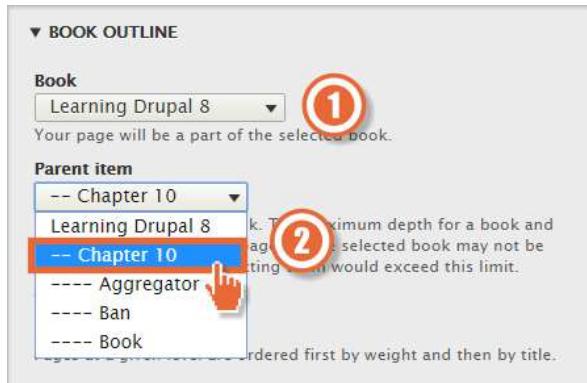
- Chapter 10 (parent)
 - Activity Tracker (child)
 - Aggregator (child)
 - Ban (child)
 - Book (child)

These pages represent the sections of this chapter so far.

Begin adding the Chapter 10 page to the book by selecting the existing **Learning Drupal 8** book as follows. Navigate to **Add content | Book page**:



Now create each of the other pages, this time expanding the **BOOK OUTLINE** menu and setting (1) the **Book** to **Learning Drupal 8** and (2) the **Parent item** as **Chapter 10**:



Note that you can also use the **Weight** option to change the precise order of the pages in the outline, but probably more easily done later using some dragging and dropping once the Book is built.

Once you have created the book pages, you can browse to the Book (Learning Drupal 8) from the **Content overview** page by navigating to **Content** (`admin/content`):

From here, you can now see an automatic link created to **Chapter 10** and all the other book pages:



The screenshot shows the 'Learning Drupal 8' page. At the top, there are four buttons: View, Edit, Outline, and Delete. Below them, it says 'Submitted by admin on Sat, 19/12/2015 - 13:08'. The main content area contains some placeholder text: 'Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.' In the sidebar on the left, there is a list of links: 'Activity Tracker', 'Aggregator', 'Ban', and 'Book'. The 'Book' link is highlighted with a red box and has a red arrow pointing to it from below. At the bottom of the sidebar, there are links for 'Add child page' and 'Printer-friendly version'. To the right of the sidebar, there is a link 'Chapter 10 >'.

When you click through to this, you'll see links to the subsection pages that you created earlier:



The screenshot shows the 'Chapter 10' page. At the top, there are four buttons: View, Edit, Outline, and Delete. Below them, it says 'Submitted by admin on Sat, 19/12/2015 - 13:08'. The main content area is empty. In the sidebar on the left, there is a list of links: 'Activity Tracker', 'Aggregator', 'Ban', and 'Book'. The 'Book' link is highlighted with a red box and has a red arrow pointing to it from below. At the bottom of the sidebar, there are links for 'Add child page' and 'Printer-friendly version'. To the right of the sidebar, there is a link 'Activity Tracker >'.

The Book module creates all of the navigation for you based on the outline choices you make.

In addition, the Book module provides extra links at the footer of each page:

- **Back**
- **Forward**

Optional Features

- An option to **Add child page**
- An option to present a **Printer-friendly version**:



Note that the Book module adds a new tab to every item of content thus enabling you to quickly and easily include that content in any existing book or create a new book on-the-fly.



Forum

The Forum module empowers you to create discussion forums on your website.

Enable the Forum module now, and once you have enabled, navigate to **Structure | Forums** ([admin/structure/forum](#)).

When installed for the first time, the Forum module creates an initial forum called General discussion:

The screenshot shows the 'Forums' administration page. At the top, there are 'List' and 'Settings' tabs. Below them, a breadcrumb trail shows 'Home > Administration > Structure'. A message says 'Forums contain forum topics. Use containers to group related forums.' There is a 'More help' link. Below this, there are two buttons: '+ Add forum' and '+ Add container'. A table follows, with columns 'NAME' and 'OPERATIONS'. The first row in the table is highlighted with a red box and has a red arrow pointing to its 'NAME' column, which contains 'General discussion'. To the right of this row is a blue 'edit forum' button.

Containers

You can set up the hierarchy of your forum by creating one or more containers and then creating a number of forums inside these containers.

Click on the **Add container** button to create a new container:



Then, give the container a name and some optional descriptive text. After doing this, **Save it**:

The screenshot shows the 'Add container' form. It has a 'Name' field with a red asterisk, containing the text 'Example Forum container'. Below it is a description: 'The term name..'. Underneath is a 'Description' field with a rich text editor toolbar above it. The description text area contains 'This is a dummy Forum container.' A red arrow points to the 'Name' field.

What you have actually done here is set up a taxonomy as discussed back in *Chapter 6, Structure*.

Optional Features

If you navigate to **Manage | Structure | Taxonomy** (admin/structure/taxonomy), you will see a vocabulary called **Forum**.

Click to list the terms:

VOCABULARY NAME	OPERATIONS
+ Forums	List terms 

Top level categories in this vocabulary represent the **Containers** and second level categories represent the **Forums**.

You can drag and drop to set up your Forum structure just as you like indenting to represent hierarchy:

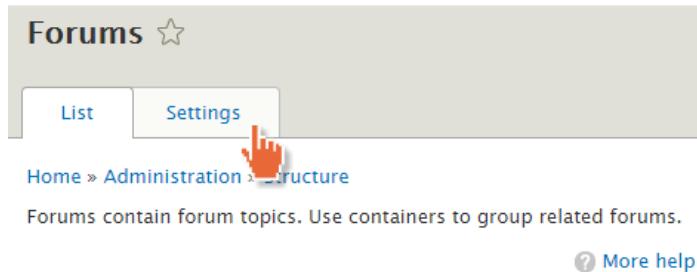
NAME		OPERATIONS
+ Example Forum container		 
→ + General discussion		 

**drag and drop to create
your Forum hierarchy**

Forum settings

Once you have set up the basic structure of your forum, you can go to the **Settings** page and change some of the behavior by visiting **Manage | Structure | Forums** (admin/structure/forum).

Click on the **Settings** tab:



This section comprises of three settings, each one discussed here:

Hot topics

Every time someone posts a reply to a forum topic, a score is added to the topic and the hot topic threshold is used to decide how many posts are required before a topic can be considered popular (or hot):

Hot topic threshold
15 ▾
The number of replies a topic must have to be considered "hot".

Once a topic is marked as hot, it can be sorted to display at the beginning of topic lists.

Topics per page

Where there are multiple replies to a topic, set how many should be visible before the pager is displayed:

Topics per page
25 ▾
Default number of forum topics displayed per page.

Default order

Most forums sort content by date, showing the newest content first. However, you can also take advantage of the hot topic threshold set here using the most active sort order:

- Default order**
- Date – newest first
 - Date – oldest first
 - Posts – most active first
 - Posts – least active first
- Default display order for topics.

Forum permissions

Next, you should decide who has permission to post forum topics or replies on your website. The Forum module exposes a number of new permissions for you to apply to one or more roles. For example, let's assume you only want logged in users to be able to post forum topics and replies.

From the main menu, navigate to **Manage | People | Permissions** (admin/people/permissions).

Scroll down to the Node module section, to locate the settings for the **Forum topic** content type.

As you can see, only the site **Administrator** role will be allowed to create new Forum content initially:



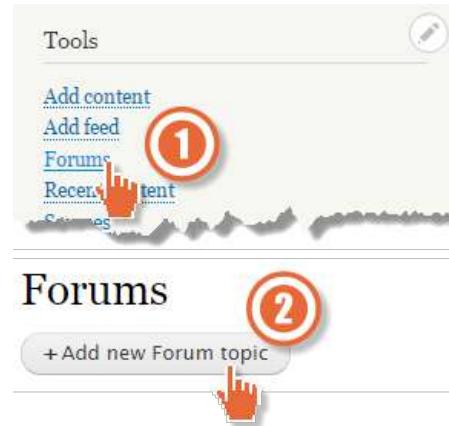
Allow authenticated user to create and delete their own topics only as follows:

PERMISSION	AUTHENTICATED USER
<i>Forum topic.</i> Create new content	<input checked="" type="checkbox"/> ①
<i>Forum topic.</i> Delete any content	<input type="checkbox"/>
<i>Forum topic.</i> Delete <u>own</u> content	<input checked="" type="checkbox"/> ②

Remember to scroll right down to the bottom of the page and press **Save permissions** to save your settings. We'll leave Forum editing and the ability to delete anyone's forum as the reserve of (say) the Editor role.

Contributing Forum topics

To add a new Forum topic, visit the **Forums** page at: /forum and click the **Add a new Forum topic** button:



Optional Features

Add a new topic to the General discussion forum:

Create Forum topic ☆

Home » Add content

Subject * 1
A new topic in the General discussions forum

Forums *
-General discussion 2
- Select a value -
Example Forum container
-General discussion Source



You can set the details of the content created on the right-hand side just like for any other type of content such as the ability to post the Topic into a particular Book—a very useful inclusion. However, in this case, you can leave them all set to their defaults.

Click **Save and publish** to publish your topic within the forum. Now, users with the appropriate permissions will be able to see your forum post and add comments to it:

A new topic in the General discussion forum

1 View Edit Outline Delete

Submitted by [admin](#) on Sat, 19/12/2015 - 13:55

Forums 2
[General discussion](#)

The Forum topic content

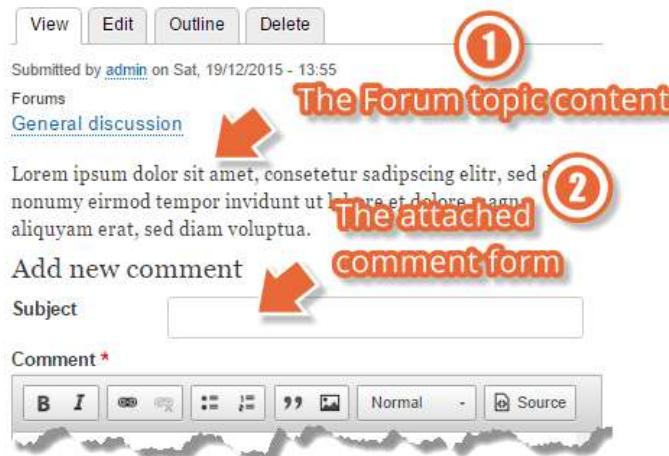
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Add new comment

Subject comment form

Comment *

Source



The Forum module depends on the Comment module—a sensible idea since there is no point in re-inventing the wheel when we already have a perfectly good system for enabling threaded conversations.



Both Forum topics and Comments can have fields added to them just like other content in Drupal. So the Forum functionality can be greatly extended, the default configuration is only a simple implementation to get you started.

Responsive image

Modern web designs need to take into account the device that is being used to view the content that Drupal is presenting. Different devices have different form factors from mobile phones, to tables, to traditional desktop computers. You may have heard of the terms 'Responsive design' or 'Adaptive design', and basically these mean that rather than having a different mobile version of your site for a mobile device, the content display is modified automatically depending on the device being used.

The following are examples of how content may be presented differently:

- Show smaller images on a mobile device to conserve mobile data bandwidth.
- Rearrange template regions to best present the most important information on a small screen size.

Responsive design relies on a concept known as a **breakpoint**. This can be thought of as the width of the screen in pixels where the design changes to a layout more appropriate for that device.

In the default theme Bartik, there are three breakpoints defined (which is a common starting point in responsive design)—mobile, narrow, and wide. You could also think of these as mobile, tablet portrait, and desktop.

The Responsive image module allows you to add some simple mobile optimization to your website.

Enable the Responsive image module now.

When activated, a new configuration screen will be available on the Configuration page.

Optional Features

Locate the **MEDIA** section and click on **Responsive image styles**:



You will see two pre-defined responsive image styles:

A screenshot of the 'Responsive image styles' configuration page. At the top, it says 'Responsive image styles' with a star icon. Below that is a breadcrumb trail: 'Home » Administration » Configuration » Media'. A descriptive text follows: 'A responsive image style associates an image style with each breakpoint defined by your theme.' There is a blue button labeled '+ Add responsive image style'. Below that is a table with three columns: 'LABEL', 'MACHINE NAME', and 'OPERATIONS'. It contains two rows: one for 'Narrow' (machine name 'narrow') and one for 'Wide' (machine name 'wide'). Each row has an 'Edit' button followed by a dropdown menu icon.

We won't go into detail here on how to create and configure new responsive styles because that would involve too lengthy a discussion involving technical responsive design terminology.

You've already got the two pre-defined ones: **Narrow** and **Wide** so instead we'll simply employ those to kick in for your **Article** content type at the existing breakpoints.

Go to the **Manage** display page of the **Article** content type by navigating to:

Structure | Content Types | Article | Manage display (`admin/structure/types/manage/article/display`).

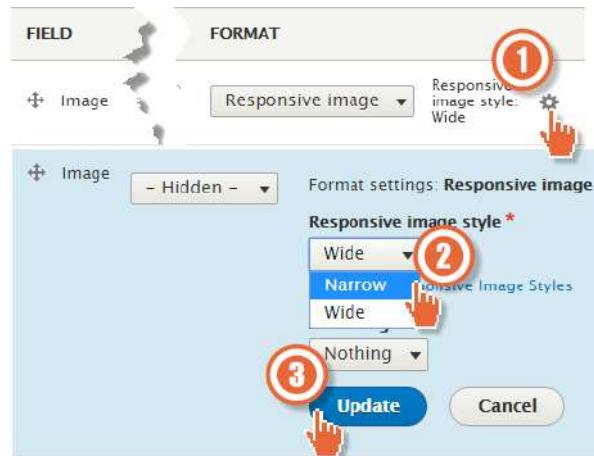
Note again that since we opted to customize the **Full content** view mode:



When you select the **FORMAT** of the **Image** field, you will see there are now two additional ones: **Responsive image** option. Select this option:



Once you have selected this, click on the gear icon to choose the required **Responsive image** for use within this view mode:



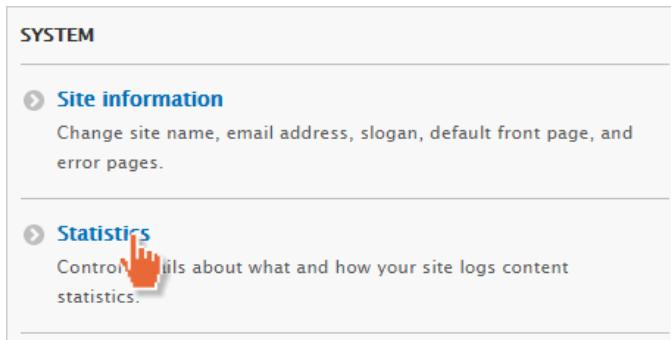
If you use some quite distinct image styles it should be easier to spot the difference between the three breakpoints by resizing your browser window. Just bear in mind that only up-to-date browsers support this feature.



Statistics

The Statistics module allows you to collect data about the visitors to your site. When the module is active, the number of times each piece of content is viewed is counted.

Enable the Statistics module now, and you will see a configuration screen available by navigating to **Configuration | System | Statistics** (`admin/config/system/statistics`):



Enable the collection of stats by ticking the **Count content views** option:



You'll then have access to a new block, which is initially entitled `Popular content`.

Visit **Manage | Structure | Block layout** and place the block.

Click on the **Place block** button now:



Choose to place the **Popular content** block.



This block can be configured to show a list of content that is:

- Most popular for the current day
- Most popular for all time (since the module was enabled)
- Most recently viewed

There are multiple sets of options available in this screen, but we'll only set the **Number of day's top views to display** for now, and we'll place the block in the Sidebar second region:

The configuration screen for the "Popular content" block. It includes fields for "Title" (set to "Most viewed"), "Display title" (checked), "Number of day's top views to display" (set to 5), "Number of all time views to display" (set to "Disabled"), "Number of most recent views to display" (set to "Disabled"), and a "Save block" button. Red circles numbered 1, 2, and 3 highlight the "Title" field, the "Number of day's top views to display" input, and the "Save block" button respectively. A red hand cursor icon points to the "Save block" button.

Don't forget that multiple instances of blocks can be placed within regions each with different settings.



Note that the very nature of the Statistics module requires a record to be added to the database whenever a page is viewed. Therefore, it is not recommended to run this module on a very busy website since it will have a negative effect on caching.

Syslog

The standard Drupal 8 installation enables a module called `Database logging`.

The `Database logging` module records log entries that are created by different modules and can be viewed at **Reports | Recent log entries** (`admin/reports/dblog`).

However, this means that data is written to the database, so often the database logging module is turned off in production websites for performance reasons.

The `Syslog` module is an alternative logging module, which logs the same data to the system log of the operating system Drupal is running in.

This is considered to be a faster operation than a database write and therefore is preferable for production sites.

You can alter the settings of the `Syslog` module by navigating to **Configuration | Development | Logging and errors** (`admin/config/development/logging`).

When this module is activated, Drupal's log messages will appear in the main system log on your server. You can specify log labels and patterns to help you filter Drupal messages from other content in your log files.

Core (experimental), Multilingual, and Web services

There are three sections we haven't discussed yet. These are a set of modules grouped under Core (experimental) and other sets grouped under the heading Multilingual and Web services. These are powerful sets of functionality that warrant in-depth study and there are other *Packt Publishing* titles dedicated to these topics.

Summary

In this chapter, we explored some (but not all) of the functions in Drupal core that are not enabled during the *Standard installation*. These are areas that may not be needed for every site, but are worthy of exploring and understanding.

You now know how to use the built-in `Forum` capabilities and how to aggregate content from another site.

Drupal core is deliberately lean in its functionality – attempting to only include the most common website functionality. The power of Drupal 8 is that you can extend this functionality with modules from the community or those you create yourself. We will explore extending Drupal further in *Chapter 12, Extending Drupal*.

In *Chapter 11, Reports*, we'll complete our journey through the main sections of the Drupal 8 administration interface looking at the `Reports` section.

11

Reports

Drupal 8 has a number of reporting features that you can use to find out more about what's happening behind the scenes. In this chapter, we will look into all of the reporting options available in the Drupal 8 core and how they can help you as you develop your site.

In particular, we will look at the reports enabled in the standard installation profile:

- Recent log messages
- Field list
- Status report
- Top "access denied" errors
- Top "page not found" errors
- Top search phrases
- Views plugins

Accessing reports

Click on **Manage** to expand the drawer as normal, and then select the **Reports** link:



You will then be presented with a list of the available reports on a single screen:

The screenshot shows a list of reports under the heading "Reports". The reports listed are:

- Available updates**: Get a status report about available updates for your installed modules and themes.
- Recent log messages**: View events that have recently been logged.
- Field list**: Overview of fields on all entity types.
- Status report**: Get a status report about your site's operation and any detected problems.
- Top 'access denied' errors**: View 'access denied' errors (403s).
- Top 'page not found' errors**: View 'page not found' errors (404s).
- Top search phrases**: View most popular search phrases.
- Views plugins**: Overview of plugins used in all views.

Available updates

The Update Manager module checks all installed modules against drupal.org to see if there are any updated versions available. You should see that everything is up to date as expected.

Available updates ☆

[List](#) [Update](#) [Settings](#)

Home » Administration » Reports

Here you can find information about available updates for your installed modules and themes. Note that each module or theme is part of a "project", which may or may not have the same name, and might include multiple modules or themes within it.

[+ Install new module or theme](#)

Last checked: 0 seconds ago ([Check manually](#))

Drupal core Everything up to date

Drupal core 8.0.1 Up to date ✓

Includes: Activity Tracker, Aggregator, Automated Cron, Ban, Bartik, Block, Book, Breakpoint, CKEditor, Last Comment, Configuration Manager, ...

Recent log messages

The **Recent log messages** report is probably the most commonly used of the reports available in Drupal core. Any module is able to report its activity here in a standardized way, whether it is for error reporting or simply informational details.

Click on the **Recent log messages** link, and you'll see a report similar to this:

Recent log messages ☆

Home » Administration » Reports

The Database Logging module logs system events in the Drupal database.

[► FILTER LOG MESSAGES](#)

[► CLEAR LOG MESSAGES](#)

TYPE	DATE	MESSAGE
system	19/12/2015 - 14:59	statistics module installed.
system	19/12/2015 - 14:02	responsive_image module installed.

Due to the fact that any module can insert messages here, the display can be a little confusing. Each log entry has the following fields:

- **Type:** This is defined by the module that recorded the log entry and is most often the module name.
- **Date:** The date and precise time (not shown in the preceding screenshot to save space) of the recorded log entry.
- **Message:** This field displays more detailed information associated with the log entry. This can be a truncated summary, and if you click on the message, you can often get more information.
- **User:** If the message was generated from a logged in session, in which the user was logged in, this will show the user's name. If there was no user logged in, this will say anonymous.
- **Operations:** Some log messages may provide additional actions that can be carried out as a consequence of the message – this is determined by the modules, but can include, for example, a link to a settings page to fix a reported error.

Log details

Sometimes, there is more detail to a log entry than the summary report is able to display. Clicking on the message opens up another page with more log details:

Details 	
Home » Administration » Reports » Recent log messages	
Type	system
Date	Saturday, December 19, 2015 – 14:59
User	admin
Location	http://drupal-8.dd:8083/admin/modules
Referrer	http://drupal-8.dd:8083/admin/modules
Message	<i>statistics</i> module installed.

This provides the following additional information that was not on the summary page:

- **Location:** The URL being loaded when the log was generated
- **Referrer:** The URL immediately preceding the location URL
- **Message:** This may be a more detailed message than was presented on the summary screen
- **Severity:** The predefined severity levels are in descending order of urgency – **Emergency, Alert, Critical, Error, Warning, Notice, Info, and Debug**
- **Hostname:** The IP address or domain name of the user originating the request

Filtering log messages

When you have a number of modules operating on your Drupal site, it can be difficult to find the messages in the report that you are looking for. On the reports page, there is a filter system to help with this:

- Go back to the **Recent log messages** page.
- Expand the **FILTER LOG MESSAGES** section:



This will allow you to only show messages for a specific type (for example, just for one module) and specific severity. This is a quick way to reduce the number of entries, so you are only presented with what you are looking for.

For example, to check for login attempts, select **user** from the **Type** menu, **Notice** from the **Severity** list then click the **Filter** button:



The resultant filtered report may be as follows:

TYPE	DATE	MESSAGE
user	19/12/2015 - 12:15	New user: User manager <Your User Manager>

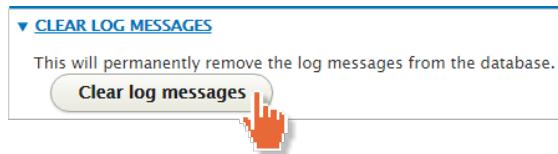
This will show you only the log records of interest. Click on the **Reset** button at any time to clear the filter and display the full list.

Clearing logs

Each log message is stored in the Drupal database, up to a limit defined in the site configuration. It is unwise to store all messages forever, due to the potential amount of data and therefore database size. 1000 messages is a sensible default.

To clear the current list, go back to the main reports screen by navigating to **Manage | Reports | Recent log messages**.

Expand the **CLEAR LOG MESSAGES** section:



This has one simple function – **Clear log messages** – which will permanently delete all log messages from the database. By default, the maximum number of messages is set to 1,000; to change the setting go to:

Manage | Configuration | Development | Logging and errors:

A screenshot of the "Development" configuration page under "Configuration". The "Performance" section is visible above. Below it, a red hand cursor is hovering over the "Logging and errors" link, which is part of a collapsed section indicated by a circular arrow icon.

Log messages and performance

Recording so much data to your database can have an impact on the performance of your site as it gets busier. For this reason, there is an alternative module in Drupal 8 core called `Syslog`. Once enabled, `Syslog` will record exactly the same information that we have been looking at so far into the server log files rather than the Drupal database. This is a significantly faster process, if slightly less convenient for you to access as the site administrator.

Field list

The **Field list** report is a useful check on the content types you created for your site.

Navigate to **Manage | Reports | Field list**.

FIELD NAME	FIELD TYPE
body	Text (formatted, long, with summary) (module: text)
comment_body	Text (formatted, long) (module: text)

In the first tab, **Entities** (shown in the preceding screenshot), you can see all of the fields that have been defined for your entity types so far and which entities are using each field (in the case where a field is shared).

The second tab, **Used in views**, shows a list of fields that have been used in views that are defined on your site.

FIELD NAME	USED IN
field_category	view name testimonials
field_high_profile	clients

Clicking the link on the view name takes you directly to the edit screen for the view.

Status report

The status report page provides information about the settings of your site.

Navigate to **Manage | Reports | Status report**.

The screenshot shows the 'Status report' page with a star icon. It includes a breadcrumb trail: Home > Administration > Reports. A message states: 'Here you can find a short overview of your site's parameters as well as Drupal.org's support forums and project issue queues. Before filing a support ticket, please check this page to see if your issue has already been reported.' Below this, there is a table with the following data:

Drupal	8.0.1
Access to update.php	Protected
Configuration files	Protected
Cron maintenance tasks	Last run 23 hours You can run cron . To run cron from outside the web server, use the command: 0%5Babsolute%5D=1

Drupal core and each community-contributed module, used on your website, can show notifications and warnings about its settings on this page.

When there is an advisory message such as the need for a security update, there will usually be instructions about how to rectify the issue and a link to the appropriate configuration page if applicable.

Top 'access denied' errors

This report shows you the secure pages that visitors have tried to access on the site which they do not have permission to view. Most often, this will be when an anonymous user or unauthenticated site visitor attempts to access the admin areas of the website.

From the main menu, navigate to **Manage | Reports | Top access denied errors**.

The screenshot shows a Drupal administrative report titled "Top 'access denied' errors". The URL is [/admin/reports/log/access-denied](#). The interface includes a navigation bar with "Home" and "Administration" links, followed by a "Reports" link. Below the navigation is a filter bar with "COUNT" and "MESSAGE" tabs, where "COUNT" is selected. A message below the filter bar states "No log messages available."

Top 'page not found' errors

The `page not found` report lists the URLs that users tried to access but did not resolve to any content on the Drupal site. Test this by trying to visit a URL that you know doesn't exist: `http://drupal-8.dd:8083/an_unreachable_url`

The screenshot shows a Drupal administrative report titled "Top 'page not found' errors". The URL is [/admin/reports/log/404](#). The interface includes a navigation bar with "Home" and "Administration" links, followed by a "Reports" link. Below the navigation is a filter bar with "COUNT" and "MESSAGE" tabs, where "COUNT" is selected. A table displays one entry: "1 /an_unreachable_url".

Note that this report is cleared when you clear the logs from the **Recent log messages** report page.

Top search phrases

The top search phrases report shows a list of the most popular words and phrases searched for using the internal **Drupal Search** (the Search module).

If the Search module is disabled, this report will not appear:

The screenshot shows a Drupal administrative report titled "Top search phrases". The URL is [/admin/reports/log/search-phrases](#). The interface includes a navigation bar with "Home" and "Administration" links, followed by a "Reports" link. Below the navigation is a filter bar with "COUNT" and "MESSAGE" tabs, where "COUNT" is selected. A message below the filter bar states "No log messages available."

The report shows a list of phrases and how often they have been searched for—you can use this information to better understand how visitors are using your site. Often a search phrase, that is used often, can guide you toward improving the site navigation.

Views plugins

The final default report for the standard installation is the `views` plugins report. This is a technical report showing all of the available views plugins installed and the views in which they are being used. This is not an everyday use report, but if you are looking to remove a plugin from your site, it's useful to know where it has been utilized so that you can modify the views before removing it. You'll be adding a Views plugin in *Chapter 12, Extending Drupal*.

Summary

We have now had a look at the default reports available in Drupal and how they can benefit you during your site development. We also investigated which other reports become available when optional core modules are enabled.

In the first part of the module, we covered the core of Drupal 8 and what you get "out of the box". In the next section, we will discuss how you can extend Drupal for your own purposes and how you can change the look and feel of your website.

12

Extending Drupal

One of the key strengths of Drupal is the ability to add new modules outside of the core ones that are available to you. There is a huge library of contributed modules available on the drupal.org website that you can download and use for free.

We'll explain how you can install modules you download and also present a few specific modules that are useful across many different types of sites.

Installing a module

Each module has its own project page on drupal.org.

At the bottom of the page will be a list of available versions of the module:

Downloads

Recommended releases

Version	Download	Date
8.x-2.0-rc1	tar.gz (40.67 KB) zip (59.39 KB)	2015-Nov-22
7.x-2.1	tar.gz (38.72 KB) zip (45.83 KB)	2014-Nov-29
6.x-4.1	tar.gz (37.5 KB) zip (43.08 KB)	2014-Nov-29

Development releases

Version	Download	Date
8.x-2.x-dev	tar.gz (40.9 KB) zip (59.86 KB)	2015-Dec-17
7.x-2.x-dev	tar.gz (39.07 KB) zip (46.29 KB)	2015-Nov-15
6.x-4.x-dev	tar.gz (37.87 KB) zip (43.6 KB)	2015-Nov-04

Note that many modules will have supported versions for previous releases of Drupal – you can only use modules with an 8.x version.

Ideally, you will want to use a recommended version of a Drupal 8 release. However, at the time of writing, many modules are still in the development stage, which means they are not recommended for production sites.

Now, we will continue with our example site and try out some different methods of installing new modules, in particular to improve one of the listings – the FAQs.

Improving FAQs

In *Chapter 7, Advanced Content*, you created a page view that represented the list of FAQs as simple clickable questions like this:

Frequently Asked Questions

How can we ensure that we get the most from a training course? 

Now, we are now going to extend Drupal so that we can represent the FAQs page using the commonly used accordions pattern as shown here:



We are first going to download and install a contributed module called `views_accordion`, and then we'll edit the exiting FAQs view to use the new Views Display plugin that the module provides.

Downloading the Views Accordion module

Modules can be downloaded from the Drupal website (<http://www.drupal.org>). The **Views Accordion** project page can be found at:

Installing the module through the UI

We covered the idea of evaluating modules on drupal.org back in *Chapter 3, Basic Concepts*, and at the time of writing the Views Accordion module for Drupal did not have a stable release (currently 8.x-1.0-alpha1). However, we know it works well enough for our purposes here.

Copy the URL of the link to either of the archives:

Recommended releases		
Version	Download	Date
8.x-1.0-alpha1	tar.gz (13.45 KB) zip (17.31 KB)	2015-Dec-03
7.x-1.1	Open link in new tab	5-Jan-29
6.x-1.5	Open link in new window	1-Sep-20

Development releases		
Version	Download	Date
8.x-1.x-dev	Open link in incognito window	2015-Dec-03
7.x-1.x-dev	Save link as...	2015-May-19
6.x-1.x-dev	Copy link address (1) tar.gz (14.45 KB) zip (18.51 KB)	2013-Oct-19

Visit the **Extend** page on your local site and click on **Install new module**:

Back to site Manage Shortcuts admin

Content Structure Appearance Extend

+ Install new module (3)

Install from a URL
http://ftp.drupal.org/files/projects/viewsAccordion-8.x-1.0-alpha1.zip
For example: http://ftp.drupal.org/files/projects/name.tar.gz

Or

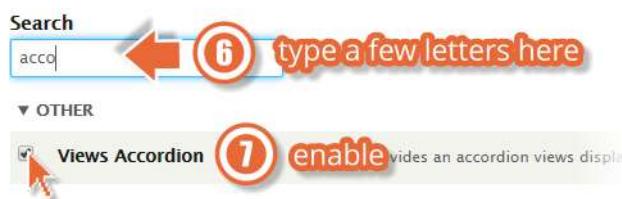
Install (5)

You will see a momentary modal popup window appear as the module code is downloaded and then you will see a second confirmation page which tells you that the installation was completed successfully.

Click the **Enable newly added modules** link to do just that:



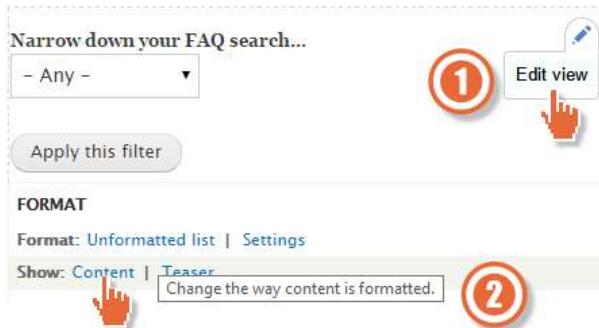
Type a few appropriate letters in the **Search** box to filter down the modules list, then enable the **Views Accordion** checkbox:



In order to see the views_accordion plugin at work, we will need to rework our view as a field-based view rather than a content-teaser view.

Visit your FAQ page at /faq and click on **Edit view** in the contextual links and click the **Content** link in the **Format** section:

Frequently Asked Questions



Then, in the popup window choose **Fields** and click on **Apply (all displays)**:

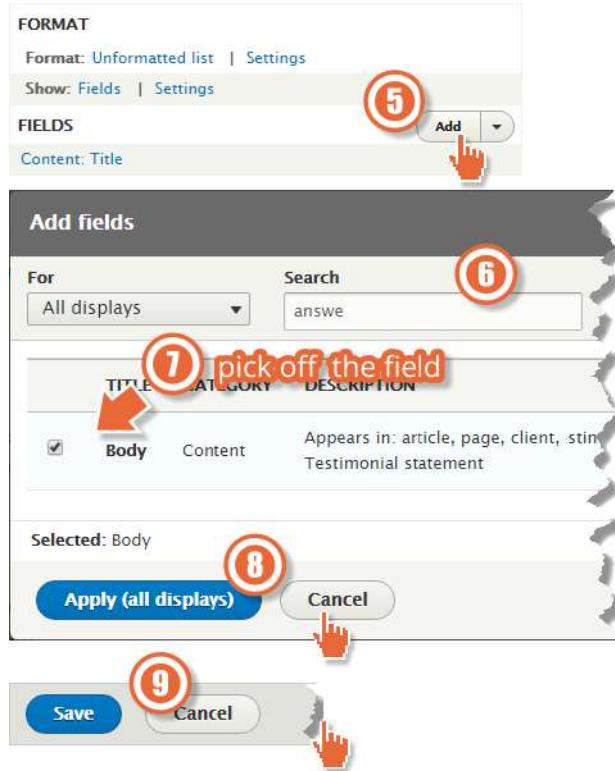


Click on **Apply** one last time and the view will now be field-based.

The view will need two fields active: firstly, the **Title**, which forms the clickable parts of the accordion, and then the **Answer** as the content, that gets revealed – this is, in fact, simply the **Body** field. So, the next step is to add the **Answer** (the **Body** field) into the field list.

To do this, click on the **Add** button (marked as **5** in the following screenshot). Then, type **answer** into the **Search** input (**6**), so as to narrow down the list of available fields on the left-hand side.

Enable the **Body** field (7), and finally hit **Apply (all displays)**:



In the subsequent pop-up window, you can leave all settings at default values and click on **Apply (all displays)** again.

Lastly, choose to use the newly provided plugin. Under **Format**, click on the **Unformatted list** link (1):



In the resulting pop-up window, choose the **jQuery UI accordion** option (2) and then click on **Apply (all displays)** again (3):

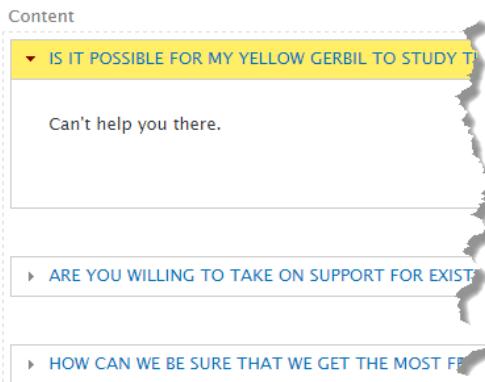


In the subsequent pop-up window, you can leave all the settings at their default values and click on **Apply** one final time.

Lastly, press the **Save** button to make all your changes permanent:



You will see from the Views preview area that the view is now accordion-powered:



Save the view and return to the FAQ page to see the new accordion-powered view:

Frequently Asked Questions

The screenshot shows a search interface with a dropdown menu set to '- Any -' and a 'Apply this filter' button. Below the search is an accordion view of questions. The first question is expanded, showing the title 'Is it possible for my yellow gerbil to study theoretical physics in a university establishment?' and the text 'Can't help you there.' The other questions are collapsed.

Pathauto and Token

Next, we are going to install two modules that work together: Pathauto (drupal.org/project/pathauto) and Token (drupal.org/project/token).

Download, install, and enable both of these modules as earlier. You'll see that Pathauto has a dependency on Token, meaning you cannot use Pathauto until the Token module has also been installed.

Earlier in the module, you'll remember that we specified the URLs of pages as node/4, and so on. This is not particularly user friendly, and we saw that the URL of a page can be added manually.

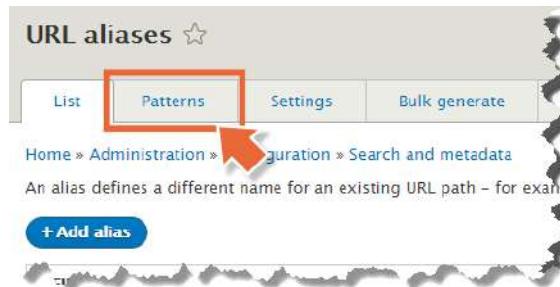
The Pathauto module allows you to create patterns for new content URLs so that a standard pattern is created for the content. This is beneficial to your users and also to search engines.

Install both the Pathauto and the Token module now

Once you have installed both the modules, navigate to:

Configuration | Search and Metadata | URL aliases (`admin/config/search/path`).

You've been here before, but now you'll see additional tabs on this screen, in particular, a **Patterns** tab.



Click on **Patterns** and scroll down to the **CONTENT PATHS** section.

You will see the list of existing content types with any associated patterns. The default pattern is preset:

A screenshot of the 'CONTENT PATHS' section. It shows a list of content types with their associated URL patterns. The first item is 'Default path pattern (applies to all content types with blank patterns below)'. The value is 'content/[node:title]' and has a red arrow pointing to it. Below it are three more items: 'Pattern for all Article paths', 'Pattern for all Client paths', and 'Pattern for all FAQ paths', each with an empty input field.

Immediately below the patterns, you can see a list of possible replacement patterns; this is where the Token module comes in.

There is a huge array of available tokens you can use:

REPLACEMENT PATTERNS		
Click a token to insert it into the field you've last clicked.		
NAME	TOKEN	DESCRIPTION
▶ Current date	[date]	Tokens related to the current date and time.
▶ Current page	[page]	Tokens related to the current page request.
▶ Current user	[user]	Tokens related to the currently logged in user.
▶ Nodes	[node]	Tokens related to individual content items, or entire node types.
▶ Random	[random]	Tokens related to random data.
▶ Site information	[site]	Tokens for site-wide settings and other global information.

Enter the following path pattern for the Article content type made up of three tokens (refer to the following screenshot):

1. [node:field_article_type]
2. [node:created:custom:Y/m]
3. [node:title]



The idea here is to create a URL in the same shape as those that we specified back in the section entitled *Introducing your site-building scenario* in the early part of *Chapter 5, Basic Content*.

What we have now done is created patterns so that each article created will have a URL that includes the creation date and the title of the article in a URL-friendly format:

1. The first token is replaced with the category of the article: article, blog-post, or news.
2. The second token is replaced with the date of creation of that article in a simple year-month format.
3. The third token is replaced by the article title.

All the replacement tokens are also cleaned up, in the sense that they are all converted to lowercase, have spaces, and have some predefined words (*in*, *is*, *that*, *the*, *this*, and *with*) removed.

Try creating a new blog article now entitled `another-blog-article`.

When you view the new article, you should see what would have been the original un-aliased URL (in our case, `node/29`) is now replaced by something that is much more human readable.


`http://drupal-8.dd:8083/node/29` 
`http://drupal-8.dd:8083/blog-posts/2015-06/another-blog-article`

Note that you can still visit `node/29`, since we have only created an alias of it, not replaced it.

Now, go ahead and supply similar patterns for all the content types.

Note the following:

- We removed the `content/` prefix from the **Default path pattern** field.
- We have left the **Pattern for all Basic page paths** field blank, so it will inherit the new default:



CONTENT PATHS	
Default path pattern	[node:title] 
Pattern for all Article paths	[node:field_article_type]/[node:created:custom:Y/m]/[node:titl...
Pattern for all Client paths	clients/[node:title]
Pattern for all FAQ paths	faqs/[node:title]
Pattern for all Basic page paths	 [node:title]
Pattern for all Service paths	services/[node:title]
Pattern for all Testimonial paths	testimonials/[node:title]

Generating paths from patterns

Creating and editing URL patterns has no retrospective effect on the existing content. However, click on the **Generate** tab, and you'll see there's a way to regenerate all content paths based on the patterns you created.



Note that this will only work on content with no path already set. If you want to overwrite an old pattern, you will have to use the **Delete aliases** tab first.



The screenshot shows the 'Bulk generate' page. At the top, there are tabs: 'List', 'Patterns', 'Settings', and 'Bulk generate'. The 'Bulk generate' tab is highlighted with a red box and a circled '1'. Below the tabs, a breadcrumb trail shows: Home » Administration » Configuration » Search and metadata » UI. A note below the trail says: 'Bulk generation will only generate URL aliases for items that currently have existing un-aliased content that needs to be aliased in bulk.' Underneath, it says: 'Select the types of un-aliased paths for which to generate URL aliases'. Three checkboxes are checked: 'Taxonomy term paths', 'User paths', and 'Content paths'. A circled '2' is over the 'Content paths' checkbox. At the bottom is a 'Update' button with a circled '3' over it, and a hand cursor icon pointing at it.

You should now see that all your node paths as seen in your browser address bar are much more friendly, for example, <http://drupal1-8.dd:8083/services/drupal-training-courses>.

Pathauto settings

Finally, the settings tab allows you to fine-tune how the URL patterns are generated. We won't go into depth here as there are many options; hopefully, you will find them self-explanatory. From here, you can decide how to handle spaces and non-alphanumeric characters in your paths.

Summary

In this chapter, we introduced some of the key ideas for extending Drupal using some example community or contrib modules that were published on the drupal.org website.

We covered improving the FAQs page by installing a Views format plugin that enabled us to present the FAQs as "accordions".

We also covered the use of a couple of key real-life build tools, notably the Pathauto and Token modules, to provide us with SEO- and human-friendly URL aliases.

13

Theming Drupal

Now that you know how to build a site using the toolkit provided by Drupal 8, it's time to learn how to make it look the part. In this chapter, we will look at applying a design to your website—a process often referred to as **theming**.

In particular, we will look at the themes included in Drupal Core and the differences between them as well as how you can customize them.

Then, we will look at adding new themes from the Drupal community site and extending them to get the results we are looking for.

What is a theme?

The **theme** used for your Drupal site is the design and layout that is applied to the functional site you have built so far.

In other systems you may have used, this could have been described as a "skin" or "design layer". Some might also refer to this as the "look and feel" of the site.

It is possible for you to use one theme for the users of the site and another for the editors, and this is often the case in practice.

There are many themes available for Drupal, and these can make exactly the same Drupal site look very different with the quick change of a setting.

Terminology

A theme can be broken up into *templates*, each of which consists of a number of regions. You will remember from earlier chapters that you place content (for example, blocks) into different theme regions. Each theme you use will define its own regions.

This fact there is a common pattern of region names in all the core themes means that it is often possible to switch between themes without having to change any other settings.

Themes included in Drupal 8

When you install Drupal 8, there are three themes that are included. These can be classified as:

- Accessible via the UI
- Base themes

Accessible via the UI

These themes are accessible via the user interface.

Bartik

The default Drupal frontend theme, this will be active for all content pages when Drupal is initially installed. It is in fact a subtheme of **Classy**—that is, it is based on or "built on top of" the Classy theme.

Seven

This is the default backend/admin theme. This can also be set as the default frontend theme, but it is really designed to be the common administration theme. This too is a subtheme of Classy.

Stark

This is a very minimal theme that includes the smallest amount of HTML markup and styling. Unlike Bartik and Seven, this theme is not built on top of anything and shows off the raw HTML markup that comes out of modules. As such, it's useful to developers to determine whether module-related CSS and JavaScript are interfering with a more complex theme.

Base themes

These are the base themes.

Classy

Classy is hidden from the UI at **Manage | Appearance** because you are not supposed to use it directly; is designed as a theme on which to base other themes – it is a *base theme*.

Classy is so-named because it presents Drupal's common classes on HTML elements within the mark-up that render on a page.

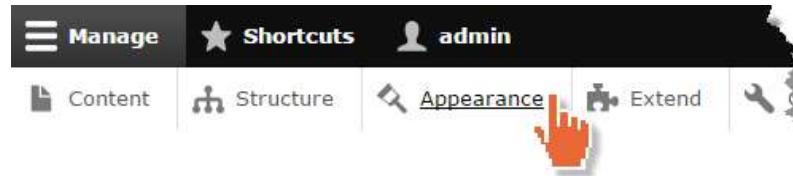
Both the Bartik and Seven themes are based on Classy.

Stable

The Stable is another hidden theme that is used if you only want to see the stable Drupal core mark-up. Creating your own themes is beyond the scope of this module (for more on themes see *Module 3, Drupal 8 Theming with Twig*) but what we will say is that if you make a brand new theme and you don't explicitly tell Drupal that your new theme is based on Classy then the Stable theme is used by default. Themes based on Stable will present much leaner mark-up than those based on Classy.

Setting the active theme

From the **Manage** menu, click on **Appearance**:



You will then see a list of (visible) themes that are available:

The screenshot shows the 'Installed themes' page in the Drupal admin interface. At the top, there's a header with the title 'Installed themes'. Below it, there are two theme preview cards. The first card is for 'Bartik 8.0.1 (default theme)', which is described as a 'flexible, recolorable theme with many features'. It shows a screenshot of a website with a dark header and footer, and a light main content area featuring a small image of Earth. A 'Settings' link is next to the theme name. The second card is for 'Seven 8.0.1 (admin theme)', described as the 'default administration theme for Drupal'. It shows a screenshot of the Drupal admin interface with a light background and various configuration tabs like 'People', 'Content', and 'Development'. A 'Settings' link and a 'Set as default' button are also present.

For each theme, you should see a representative example screenshot. If the theme is not the current active theme, you should see a **Set as default** link. Clicking on this will replace the active theme on your site with the new one from the list.

For now, we will continue with Bartik as our default theme as it is the theme most suitable for customizing your site so far without having to write any code.

Common settings

Each theme can provide its own settings, but all themes have a common configuration. Click on the **Settings** link next to the **Bartik** theme to open the available configuration.

For now, skip over (collapse) the **Color scheme** option (we'll come to that later).

Toggle display

Inside the TOGGLE DISPLAY menu, you will see a list of checkboxes as follows:

▼ TOGGLE DISPLAY

Enable or disable the display of certain page elements.

- User pictures in posts ①
- User pictures in comments ②
- User verification status in comments ③
- Shortcut icon ④

Item	Description
1	If we are opting to show the author information with posted content, if the author has an attached picture, then display it as a small thumbnail icon.
2	If we are opting to show author information with posted comments and the comment author has an attached picture, then display it as a small thumbnail icon.
3	User verification status in comments.
4	Use a shortcut icon also known as the favicon.ico. Defaults to the drop.

By default, if the theme you have chosen to use includes its own `logo.svg` (Scalable Vector Graphic) file then that will be used as the logo.

The default logo is a semi-transparent version of the Drupal 'drop' character:



Thus, when used overlaying the Bartik's theme blue header, it appears:



If you choose to un-tick the **Use the default logo supplied by the theme** option then you are given the chance to point to, or upload, an alternative.

▼ LOGO IMAGE SETTINGS

Use the default logo supplied by the theme
to custom logo  Supply an alternative

Examples: logo.svg (for a file in the public filesystem),
public://logo.svg, or core/themes/seven/logo.svg.

Upload logo image

No file chosen

If you don't have direct file access to the server, use this field to upload your logo.

The default shortcut icon, often referred to as the favicon, is also the Drupal drop icon.



Once again, if you would like to replace it with something else then the option is there.

▼ SHORTCUT ICON SETTINGS

Your shortcut icon, or 'favicon', is displayed in the address bar and bookmarks of most browsers.

Use the default shortcut icon supplied by the theme
to custom icon  Supply an alternative

Examples: favicon.ico (for a file in the public filesystem),
public://favicon.ico, or core/themes/seven/favicon.ico.

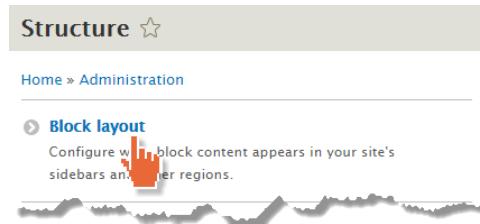
Upload icon image

No file chosen

If you don't have direct file access to the server, use this field to upload your shortcut icon.

Theme regions

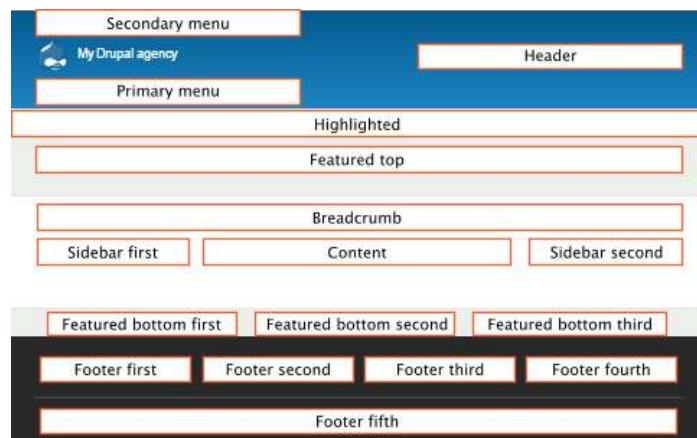
To see an onscreen preview of the Regions within the Bartik theme, visit the **Block layout** page:



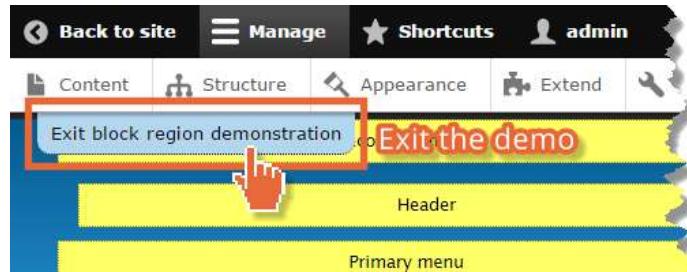
Check that you are on the Bartik page and click **Demonstrate block regions (Bartik)**:



There are 17 named regions in the Bartik theme depicted slightly differently to what you are currently seeing so as to make their names a bit more readable on the printed page:



To exit the preview display, click on the **Exit block region demonstration** link:



Color scheme

If you enable the optional **Color** module (enabled by default in a standard installation), it is possible to change the color scheme of some themes. Note that not all themes support this, but the default Bartik theme does.

Go back to the **Appearance** page and select **Settings** for the Bartik theme.

At the top of the page (collapsed earlier), you will then see the following:

Color set	Custom
Header background top	#055a8e
Header background bottom	#1d84c3
Main background	#ffffff
Sidebar background	#f6f6f2
Sidebar borders	#f9f9f9
Footer background	#292929
Title and slogan	#ffffff
Text color	#3b3b3b
Link color	#0071B3

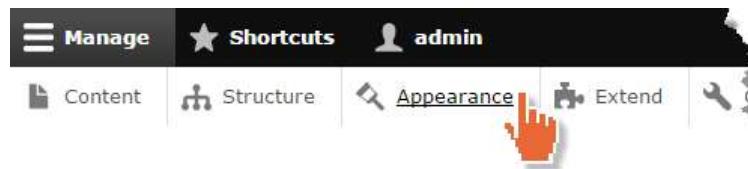
There are a number of preset color sets that work together—you can select these from the **Color set** dropdown. The default theme is called **Blue Lagoon**. If you don't like the defaults or have some specific theme in mind, you can change the colors of individual components manually by selecting **Custom** from the dropdown and then clicking on each of the color fields to set the hexadecimal number values to match any existing designs.

Alternatively, you can click on the color selector wheel to the right when you have focused on a specific field to automatically fill in the value.

Setting the admin theme

The theme you use when editing your site is not necessarily the same as the one users see when they are browsing your site. Drupal 8 includes a specially designed admin theme.

Go back to the **Appearance** page:



At the bottom of the appearance page is a panel allowing you to specify the **Administration theme**:



The checkbox underneath the theme selection allows you to specify which themes (the frontend theme or the admin theme) are used when the content is being edited.

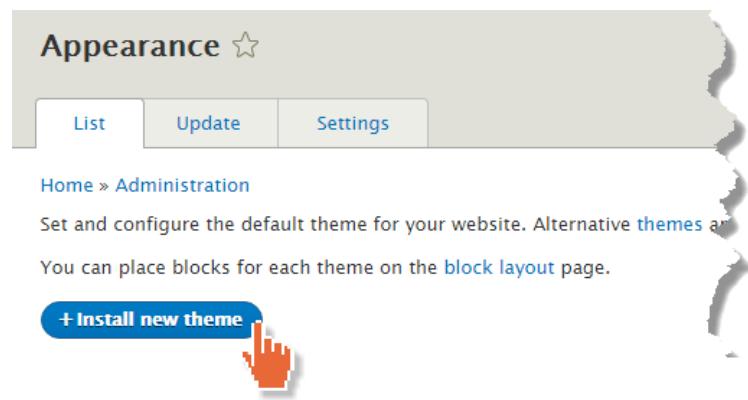
As a rule, using the administration theme will give you a better editing experience.

Advanced themes from the Drupal community

The Drupal community is starting to publish a number of Drupal 8 compliant themes on the drupal.org site. We'll take a look now at one such theme and how you can download, install, and use it to restyle your site.

Installing a new theme

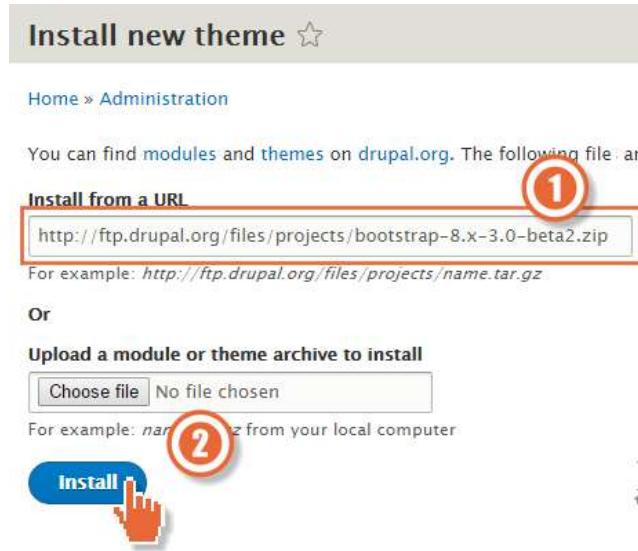
From the main menu, click on Appearance and then click on **Install new theme**:



In our example, we will download the popular Bootstrap theme from drupal.org and then copy the theme to the correct place in our installation before installing it. Go to <https://www.drupal.org/project/bootstrap> and install the latest version:



Enter the URL of the project from drupal.org just as you did when you installed new modules back in *Chapter 12, Extending Drupal*.



The theme will be physically downloaded into the themes folder and will then be available for installation on the Appearance page.

Locate the new theme on the page and click **Install and set as default** theme:



Bootstrap is actually a public frontend framework released by the Twitter team (<http://getbootstrap.com/>). This theme is a Drupal implementation of the Bootstrap framework, and you may recognize some styles from the other websites you have used.

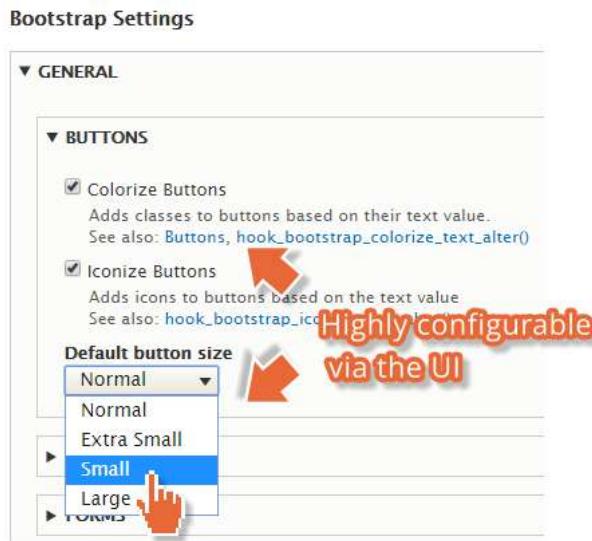
If you don't want to create your own bespoke theme, Bootstrap is a great way to give your website a professional look and gives you a great deal of scope for customization using theme settings.

Here is a screenshot of what the **Contact** form looks like when viewed in the newly applied theme:

The screenshot shows a contact form titled "Website feedback". It includes fields for "Your name" (admin), "Your email address" (YourEmail@example.com), and "Subject*" (with a red arrow pointing to the input field). The "Message*" field contains the text "The Bootstrap styled contact form" (also highlighted with a red box). Below the message field is a checkbox labeled "Send yourself a copy". At the bottom are two buttons: "Send message" (blue) and "Preview".

Now, go back to the settings page, and you will see a huge array of settings that were not available with the Bartik theme.

There are too many to go through in detail here, and it's quite likely there will be even more by the time you read this. So for the purpose of this chapter, we'll just point out where those settings appear and leave you to explore the effect of them on your site:



This should give you an idea of how much you can do through configuration of a Drupal theme without actually having to create your own from scratch.

Custom themes

If you are working on a website for commercial use, it's probable that you will have designs or at least design guidelines to follow. It is possible to create your own custom themes for Drupal to achieve the exact look and feel you need.

See more on creating a custom theme in *Module 2, Drupal 8 Development Cookbook* and *Module 3, Drupal 8 Theming with Twig* of this course, but before we conclude this topic, it's worth introducing a couple of other concepts important to Drupal theming when you take the next step.

Base themes and subthemes

Drupal themes are designed to be extendible so that you can build a theme of your own that only changes some small aspects of a parent theme. It's not best practice to edit the content of any part of Drupal core, but you may, for example, want to modify the CSS of the Bartik theme for your own purposes.

In this scenario, you would make a subtheme of the Bartik theme and then edit the CSS in your version of the theme. As mentioned earlier, some of the other themes provided in Drupal core such as Classy is not designed to be used directly, but are intended to be the Base theme for you to build on top of.

In the same way, you would not edit the contents of these themes in Drupal core. In order to make a change to their design, you would create a subtheme of one of these in the custom theme folder and continue your work in this area.

Summary

In this chapter, we introduced some of the key ideas for applying a different look and feel to your website by switching themes.

We looked at the region architecture provided by the Bartik theme and we experimented with making small theme adjustments via the UI including toggling display elements such as Title and Logo, and color schemes.

We re-themed your Drupal site using the Bootstrap community theme that was published on the drupal.org website and we saw how such themes can provide some significant configuration tools via the administrative UI.

14

Getting Support

We have now reached the end of the instructional part of the module. However, Drupal 8 is an extensive software framework and you should not expect to find all of the answers in a "learning" module. In the following chapter, we discuss how the open source Drupal Community works and how you can engage with the community to get help and support for your Drupal project.

In particular, we will cover the following topics:

- What is open source and how does the Drupal Community work together?
- How can you report bugs and request changes?
- Where else can you find help?
- How can you help?

What is open source?

The dictionary definition of open source is:

denoting software for which the original source code is made freely available and may be redistributed and modified.

– New Oxford American Dictionary

While there are no license fees associated with Drupal, the most important part is the fact that you are free to modify and extend the code base to meet your own needs.

Open source means that you can also see the full history of the code – you can see when changes were made, for what reason, and by whom. There is no "black box" technology – everything is open for you to investigate if you desire to.

The Drupal community

One of the key benefits of using Drupal is the community behind the software. Drupal 8 represents the freely donated time of thousands of individuals and companies. The ethos of the community is giving back, and you will find that there are many people willing to help you get the most out of Drupal.

Many people are very successfully using Drupal for their projects without any involvement in the community. However, should you choose to, there are plenty of ways you can find help and support.

Drupal.org

The central location of the Drupal project is the main website, that is, Drupal.org. Every Drupal module has its own project page, run by the module maintainer. Each module page explains the purpose of the project and contains links to further documentation.

The following screenshot is from the popular **Rules** module:

The screenshot shows the Drupal.org project page for the 'Rules' module. At the top, there's a navigation bar with tabs: 'View' (which is highlighted in green), 'Version control', 'Revisions', and 'Automated Testing'. Below the navigation, it says 'Posted by fago on November 7, 2007 at 1:34pm'. The main content area starts with a brief description of what the Rules module does: 'The Rules module allows site administrators to define conditionally executed actions based on occurring events (known as reactive or ECA rules). It's a replacement with more features for the trigger module in core and the successor of the Drupal 5 workflow_ng module.' To the right of this text is a large teal square icon containing a white stylized 'R' with arrows pointing outwards. To the right of the icon is a sidebar titled 'Maintainers for Rules' which lists 'klausi - 228 commits' (last: 3 days ago, first: 6 years ago) and 'fago - 976 commits' (last: 1 week ago, first: 7 years ago), with links to 'View all committers' and 'View commits'. Below this is another sidebar titled 'Issues for Rules' with a link to 'Avoid duplicates, implement...'.

On the right-hand side bar, you can find the following:

- **Maintainer:** This is the developer who is responsible for the code and documentation of this module.
- **Co-maintainers:** There are some larger and more complex modules that have multiple maintainers who have permission to update the code.

- **Links to commits:** As all of Drupal is open source, you can see the history of all the files that make up the project. Every change to every file is visible in a commit log, so you can see what was changed and why.
- **Links to issue queues:** This consists of every support question or bug that has been raised for this module.
- **Links to documentation**
- **Change records:** When a change to a module is more significant than a bug fix and may need some action on your part in order to upgrade, a change record is created.

Issue queues

For Drupal core and all Drupal contributed modules there is an issue queue, which you can use to request support or provide feedback. This is the most reliable way of communicating with the maintainers of the modules in question.

Remember, those individuals looking after different modules have assumed the responsibility of responding to questions in the issue queues, but they are still volunteers. Some more popular modules generate more questions than it would be possible to process as a full-time job.

Issues for Rules

Create a new issue Advanced search E-mail notifications						
Search for	Status	Priority	Category	Version	Component	Prior
	- Open issues -	- Any -				
	- 8.x issues -	- Any -				
Summary	Status	Priority	Category	Version	Component	Prior
Implement the data selector / direct input mode switcher in the UI <small>new</small>	Active	Normal	Task	8.x-3.x-dev	Rules Core	2
[META] Rules 8.x UI <small>new</small>	Active	Normal	Task	8.x-3.x-dev	User Interface	15
Ban on condition causes <small>new</small>	Active	Normal	Bug report	8.x-3.x-dev	Rules Core	1

Reporting a bug or requesting support

If you want to communicate with the module maintainer or Drupal core team, create a new issue using the project page.

[Rules](#) » [Issues](#)

Create Issue

▼ Issue metadata

Title *

Category * **Priority *** **Status ***
Bug report ▾ Normal ▾ Active ▾

Descriptions of the Priority and Status values can be found in the Issue queue help topic.

▼ Issue tags

Before adding tags [read the issue tag guidelines](#). Do NOT use tags for adding links. Separate tags with a space, not a space.

▼ Issue summary & relationships

Issue summary
 [B](#) [I](#) [ABC](#) [H2](#) [H3](#) [H4](#) [?](#) [code](#) [PHP](#) [Image](#) [Attachment](#) [List](#) [Table](#) [?](#)

► Files

[Save](#) [Preview](#)

In order to get attention for your issue, be as clear and descriptive as possible. If you are reporting a bug, provide the steps to reproduce it.

The best practice for getting help in the Drupal issue queues is as follows:

1. Always search first, in case someone has experienced the problem before.
2. Be clear and descriptive when you file an issue.
3. Try to include steps to reproduce the problem if you are reporting a bug.
4. Be considerate of the module owner in the phrasing you use – these are volunteers giving up their own time.

5. Do not reopen an old issue if it has been closed for a long period of time.
6. Think carefully about the priority you set—just because it is your number one priority does not mean it will be for other users.
7. If you think you have a solution to the issue you are reporting, suggest it or, even better, attach a patch file that fixes it.

The Drupal security team

If you come across an issue in Drupal core or in a contributed module that you think is a potential security risk for other users, there is a slightly different process. If there is a genuine vulnerability, it is best to report it privately to the Drupal security team who will work with the module maintainer to resolve it if necessary.

When a fix is available, a security notification is sent out explaining the vulnerability and the mitigating factors involved.

There is a special link to report a security issue on all module pages:



api.drupal.org

The developer documentation for Drupal can be found at `api.drupal.org`. If you are not carrying out any custom development, you may not need to go there, but if you are working on a custom module or theme, this is an essential reference for every internal function within Drupal.

IRC chat

A popular place to find help is to use **Internet Relay Chat (IRC)**. Many members of the Drupal community can be found in Drupal-specific chat rooms and may be able to offer assistance when you need it.

There are some general channels such as #drupal and some very specific channels where groups meet to discuss their areas of interest. You may find #drupal-support, a useful channel if you need help.

The Drupal Association

The **Drupal Association** (or DA as it is often referred to) is a not-for-profit organization that looks after the interests of the Drupal Community.

You can become a member of the association as an individual or a company, and the membership fees go toward maintaining the infrastructure of drupal.org as well as other educational and promotional initiatives. The association is also directly involved in organizing the annual Drupal conference known as *DrupalCon*.

DrupalCon

One of the key events in the Drupal community calendar is the annual *DrupalCon*. Historically, there is one event in North America and one in Europe each year, and more recently, there are events being arranged in South America and Asia.

The North American event attracts 3500-4000 delegates, and the European event attracts over 2000. These events are an opportunity for core and contributed module developers to meet and work together in person. The conference usually lasts a week with a day of training followed by 2 days of presentations in multiple topic tracks. After this, there are traditionally code sprints where community members work together on areas of interest improving the Drupal code base.

To fully understand the Drupal community, it is highly recommended to attend one of these conferences. Here, you will experience the passion and enthusiasm of the community, and if our experience is anything to go by, feel very welcome.

Many of the Drupal development companies and agencies sponsor these events, so it is also a good place to meet prospective employers.

DrupalCamps

By contrast, **DrupalCamps** are much smaller, locally organized events. Depending on the location, these can be small groups of enthusiasts meeting in the local pub or conferences with as many as 1000 delegates (for example, DrupalCamp London or NYCamp in New York).

There are also specialist camps such as FrontEnd United, which focuses on the Drupal theme layer, and DevDays, which is more focused on back end and core development.

The **Drupical** website (<http://www.drupical.com>) is a community project, which maps the locations of upcoming Drupal events.

Specialist Drupal companies

If the needs of your project are more extensive, there are many companies around the world that specialize in Drupal development. The authors of this module both work for **iKOS**—a company specializing in e-commerce with Drupal.

The largest of these organizations is **Acquia**, which was founded by the original developer of Drupal, Dries Buytaert. Acquia's mission is to promote Drupal to the enterprise market.

Other specialist companies such as **Commerce Guys** are dedicated to the development of the Drupal commerce platform.

Training

Some people are very successful at teaching themselves using books or courses such as this, others prefer a classroom-led approach. There are many Drupal training courses offered around the world from beginners "Drupal from Scratch" courses to intensive boot camp style courses.

Certification

There is no officially recognized certification for Drupal development like there is for many other technologies. However, recently Acquia announced their own certification program which is becoming quite popular.

Summary

Drupal is a complex software platform, and while you may experience no problems at all when you build your first website, you may need to reach out for help. The Drupal Community is a very welcoming place, and you should be able to find the help you are looking for through one of the many channels discussed here.

As with any technology, there are no stupid questions. It's important to remember that everyone is learning all the time, and Drupal itself is evolving every day.

Enjoy building your projects with Drupal and come back to this module as reference when you need to. There are lots of people in the community ready and willing to lend a hand if you need them.

Module 2

Drupal 8 Development Cookbook

Over 60 hands-on recipes that get you acquainted with
Drupal 8's features and help you harness its power

1

Up and Running with Drupal 8

In this chapter we'll get familiar with Drupal 8 and cover:

- ▶ Installing Drupal
- ▶ Using a distribution
- ▶ Installing modules and themes
- ▶ Using multisites in Drupal 8
- ▶ Tools for setting up an environment
- ▶ Running Simpletest and PHPUnit

Introduction

This chapter will kick off with an introduction to getting a Drupal 8 site installed and running. We will walk through the interactive installer that most will be familiar with from previous versions of Drupal, and from the command line with Drush.

Once we have installed a standard Drupal 8 site, we will cover the basics of extending Drupal. We will discuss using distributions and installing contributed projects, such as modules and themes. We will also include uninstalling modules, as this has changed in Drupal 8.

The chapter will wrap up with recipes on how to set up a multisite installation in Drupal 8, getting a local development environment configured and running the available test suites.

Installing Drupal

Just like most things, there are many different methods for downloading Drupal and installing it. In this recipe, we will focus on downloading Drupal from <https://www.drupal.org/> and setting it up on a basic Linux, Apache, MySQL, PHP (LAMP) server.

In this recipe, we will set-up the files for Drupal 8 and go through the installation process.

Getting ready

Before we start, you are going to need the below mentioned development environments that meet the new system requirements for Drupal 8:

- ▶ Apache 2.0 (or higher) or Nginx 1.1 (or higher) web server.
- ▶ PHP 5.5.9 or higher.
- ▶ MySQL 5.5 or MariaDB 5.5.20 for your database. You will need a user with privileges to create databases, or a created database with a user that has privileges to make tables in that database.
- ▶ Ability to upload or move files to the server!
- ▶ Drupal also requires specific PHP extensions and configuration. Generally a default installation of PHP should suffice. See <https://www.drupal.org/requirements/php> for up to date requirements information.



Drupal 8 ships with Symfony components. One of the new dependencies in Drupal 8, to support the Symfony routing system, is that the Drupal Clean URL functionality is required. If the server is using Apache, ensure that `mod_rewrite` is enabled. If the server is using Nginx, the `ngx_http_rewrite_module` must be enabled.

We will be downloading Drupal 8 and placing its files in your web server's document root. Generally, this is the `/var/www` folder. If you used a tool such as XAMPP, WAMP, or MAPP please consult the proper documentation to know your document root.

How to do it...

We need to follow the below steps to install Drupal 8:

1. First we need to head to [Drupal.org](https://www.drupal.org) and download the latest release of Drupal 8.x! You can find the most recent and recommended release at the bottom of this page: <https://www.drupal.org/project/drupal>. Extract the archive and place the files to your document root as the folder `drupal8`:

Downloads

Recommended releases

These are stable, well-tested versions that are actively supported.

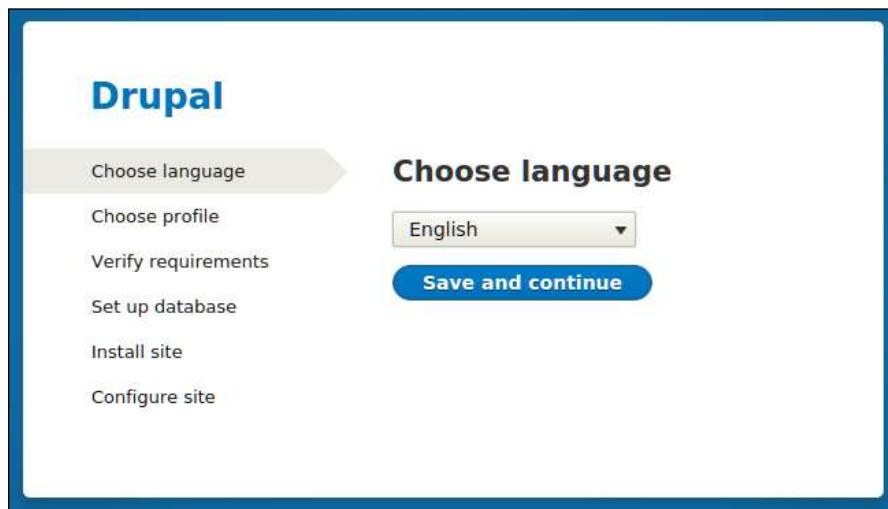
Drupal core 8.0.2
Released: Jan 06 2016

The next patch release of Drupal 8 is ready for new development and use on production sites.

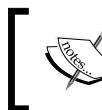
Drupal core 7.41
Released: Oct 21 2015

If you need stability and features from the widest variety of contributed modules and themes, this is the version for you.

2. Open your browser and visit your web server, for example <http://localhost/drupal8>, to be taken to the Drupal installation wizard. You will land on the new multilingual options install screen. Select your language and click **Save and continue**.



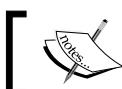
3. On the next screen keep the default **Standard** option for the installation profile. This will provide us with a standard configuration with the most commonly used modules installed.
4. The next step will verify your system requirements. If your system does not have any reportable issues, the screen will be skipped.



If you have requirement issues, the installer will report what the specific issues are. Nearly every requirement will link to a [Drupal.org handbook page](#) with solution steps.

5. Enter the database information for Drupal. In most cases, you only need to supply the username, password, and database name and leave the other defaults. If your database does not exist, the installer will attempt to create the database:

The screenshot shows the 'Database configuration' step of the Drupal 8 setup process. On the left, there's a sidebar with links: 'Choose language', 'Choose profile', 'Verify requirements', 'Set up database' (which is highlighted with a grey arrow), 'Install site', and 'Configure site'. The main area is titled 'Database configuration'. It has three fields: 'Database type *' with 'MySQL, MariaDB, Percona Server, or equivalent' selected (indicated by a blue radio button), 'Database name *' (an empty input field with an asterisk), 'Database username *' (an empty input field with an asterisk), and 'Database password' (an empty input field with an asterisk). Below these fields is a 'Save and continue' button.



See the *There's more* section in for information on setting up your database and any possible users.

6. Your Drupal 8 site will begin installing! When it is done installing the base modules, you will be taken to a site configuration screen.
7. The configure site form provides the base configuration for your Drupal site. Enter your site name and the e-mail address for the site. The site email will be used to send administrative notifications and as the originating email for outgoing emails from the Drupal site. The form allows you to set regional information regarding the country and time zone of the site. Setting the timezone ensures time values display correctly.

8. Fill in the site maintenance account information, also known as user 1, which acts in a similar way to root on Unix based systems. The site maintenance account is crucial. As stated, this acts as the first user and resembles the root user in Unix-based systems. In Drupal, the user with the user ID of 1 can bypass permission checks and have global access.
9. Enter the site's regional information and whether the site should check for updates available for modules enabled and Drupal itself. By checking for updates automatically, your site will report anonymous usage statistics to [Drupal.org](https://drupal.org) along with providing a summary of your version status. You have the option to also opt-in for the site to email you notifications of new releases, including security releases.
10. When satisfied click **Save and continue** and **Congratulations, you installed Drupal!**

How it works...

Drupal 8 supports a multilingual installation. When you visit the installer it reads the language code from the browser. With this language code, it will then select a supported language. If you choose a non-English installation the translation files will be automatically downloaded from <https://localize.drupal.org/>. Previous versions of Drupal did not support automated multilingual installations.

The installation profile instructs Drupal what modules to install by default. Contributed install profiles are termed distributions. The next recipe discusses distributions

When verifying requirements, Drupal is checking application versions and configurations. For example, if your server has the PHP Xdebug extension installed, the minimum `max_nested_value` must be 256 or else Drupal will not install.

There's more...

The Drupal installation process can be very straight forward, but there are a few items worth discussing.

Creating a database user and a database

In order to install Drupal you will need to have access to a database server and an existing (or ability to create) database (or the ability to create one). This process will depend on your server environment setup.

If you are working with a hosting provider, there is more than likely a web based control panel. This should allow you to create databases and users. Refer to your hosting's documentation.

If you are using **phpMyAdmin** on your server, often installed by MAMP, WAMP, and XAMPP, and have root access, you can create your databases and users.

- ▶ Sign into phpMyAdmin as the root user
- ▶ Click **Add a new User** from the bottom of the privileges page
- ▶ Fill in the user's information
- ▶ Select to create a database for the user with all privileges granted
- ▶ You can now use that user's information to connect Drupal to your database

If you do not have a user interface but have command line access, you can set up your database and user using the MySQL command line. These instructions can be found in the core/INSTALL.mysql.txt files:

1. Log into MySQL:

```
$ mysql -u username -p
```

2. Create the database you will use:

```
$ CREATE DATABASE database CHARACTER SET utf8 COLLATE utf8_general_ci;
```

3. Create a new user to access the database:

```
$ CREATE USER username@localhost IDENTIFIED BY 'password';
```

4. Grant the new user permissions on the database:

```
$ GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP,
INDEX, ALTER, CREATE TEMPORARY TABLES ON databasename.* TO
'username'@'localhost' IDENTIFIED BY 'password';
```



If you are installing Drupal with a PostgreSQL or SQLite database, see the appropriate installation instructions, either INSTALL.pgsql.txt or INSTALL.sqlite.txt.



Database prefixes

Drupal, like other content management systems, allows you to prefix its database tables from the database set-up form. This prefix will be placed before table names to help make them unique. While not recommended this would allow multiple installations to share one database. Utilizing table prefixes can, however, provide some level of security through obscurity since the tables will not be their default names.

ADVANCED OPTIONS

Host *
localhost *

Port number
3306

Table name prefix *

If more than one application will be sharing this database, a unique table name prefix – such as *drupal_* – will prevent collisions.

Downloading and installing with Drush

You may also install Drupal using the PHP command line tool Drush. Drush is a command line tool created by the Drupal community and must be installed. Drush is covered in *Chapter 13, Drush CLI*.

The `pm-download` command will download packages from [Drupal.org](https://www.drupal.org). The `site-install` command will allow you to specify an installation profile and other options for installing a Drupal site. The installation steps in this recipe could be run through Drush as:

```
$ cd /path/to/document/root
$ drush pm-download drupal-8 drupal8
$ cd drupal8
$ drush site-install standard --locale=en-US --account-name=admin
--account-pass=admin --account-email=demo@example.com --db-url=mysql://
user:pass@localhost/database
```

We use Drush to download the latest Drupal 8 and place it in a folder named `drupal8`. Then the `site-install` command instructs Drush to use the standard install profile, configure the maintenance account, and provides a database URI string so that Drupal can connect to its database.

Security updates

If you choose to disable the update options, you will have to check manually for module upgrades. While most upgrades are for bug fixes or features, some are for security updates. It is highly recommended that you subscribe to the Drupal security team's updates. These updates are available on Twitter at [@drupalsecurity](https://twitter.com/drupalsecurity) or the feeds on <https://www.drupal.org/security>.

See also

- ▶ For more on multilingual, see *Chapter 8, Multilingual and Internationalization*
- ▶ For more on using the command line and Drupal, see *Chapter 13, Drupal CLI*
- ▶ See the [Drupal.org handbook on installing Drupal](https://www.drupal.org/documentation/install) <https://www.drupal.org/documentation/install>
- ▶ Drush site install <http://drushcommands.com/drush-8x/site-install/site-install>

Using a distribution

A distribution is a contributed installation profile that is not provided by Drupal core. Why would you want to use a distribution? Distributions provide a specialized version of Drupal with specific feature sets. On [Drupal.org](https://www.drupal.org) when you download an installation profile it not only includes the profile and its modules but a version of Drupal core. Hence the name distribution. You can find a list of all Drupal distributions here https://www.drupal.org/project/project_distribution.

How to do it...

We will follow these steps to download a distribution to use as a customized version of Drupal 8:

1. Download a distribution from [Drupal.org](https://www.drupal.org). For this recipe let's use the Demo Framework provided by Acquia <https://www.drupal.org/project/df>.
2. Select the recommended version for the 8.x branch.
3. Extract the folder contents to your web server's document root. You'll notice there is Drupal core and, within the profiles folder, the installation profile's folder `df`.
4. Install Drupal as you would normally, by visiting your Drupal site in your browser.
5. Demo Framework declares itself as an exclusive profile. Distributions which declare this are automatically selected and assumed to be the default installation option.



The exclusive flag was added with Drupal 7.22 to improve the experience of using a Drupal distribution
<http://drupal.org/node/1961012>.

6. Follow the installation instructions and you'll have installed the distribution!

How it works...

Installation profiles work by including additional modules that are part of the contributed project realm or custom modules. The profile will then define them as dependencies to be installed with Drupal. When you select an installation profile, you are instructing Drupal to install a set of modules on installation.

There's more...

Distributions provide a specialized version of Drupal with specific feature sets, but there are a few items worth discussing.

Makefiles

The current standard for generating a built distribution is the utilization of Drush and makefiles. Makefiles allow a user to define a specific version of Drupal core and other projects (themes, modules, third party libraries) that will make up a Drupal code base. It is not a dependency management workflow, like Composer, but is a build tool.

If you look at the Demo Framework's folder you will see `drupal-org.make` and `drupal-org-core.make`. These are parsed by the `Drupal.org` packager to compile the code base and package it as a `.zip` or `.tar.gz`, like the one you downloaded.

Installing with Drush

As shown in the first recipe, you can install a Drupal site through the Drush tool. You can instruct Drush to use a specific installation profile by providing it as the first argument. The following command would install the Drupal 8 site using the Demo Framework.

```
$ drush pm-download df  
$ drush site-install df -db-url=mysql://user:pass@localhost/database
```

See also...

- ▶ See *Chapter 13, Drupal CLI*, for information on makefiles.
- ▶ Drush documentation page for drush make
<http://www.drush.org/en/master/make/>
- ▶ Distribution documentation on `Drupal.org`,
<https://www.drupal.org/documentation/build/distributions>

Installing modules and themes

Drupal 8 provides more functionality out of the box than previous versions of Drupal – allowing you to do more with less. However, one of the more appealing aspects of Drupal is the ability to extend and customize.

In this recipe, we will download and enable the Honeypot module, and tell Drupal to use the Bootstrap theme. The Honeypot module provides honeypot and timestamp anti-spam measures on Drupal sites. This module helps protect forms from spam submissions. The Bootstrap theme implements the Bootstrap front-end framework and supports using Bootswatch styles for theming your Drupal site.

Getting ready

If you have used Drupal previously, take note that the folder structure has changed. Modules, themes, and profiles are now placed in their respective folders in the root directory and no longer under sites/all. For more information about the developer experience change, see <https://www.drupal.org/node/22336>.

How to do it...

Let's install modules and themes:

1. Visit <https://www.drupal.org/project/honeypot> and download the latest 8.x release for Honeypot.
2. Extract the archive and place the `honeypot` folder inside the `modules` folder inside of your Drupal core installation:

www	14 items
core	29 items
modules	2 items
honeypot	12 items
config	3 items
src	2 items
composer.json	445 bytes
honeypot.api.php	3.3 kB
honeypot.info.yml	326 bytes
honeypot.install	1.4 kB
honeypot.links.menu.yml	215 bytes
honeypot.module	11.8 kB
honeypot.permissions.yml	240 bytes
honeypot.routing.yml	230 bytes
LICENSE.txt	18.1 kB
README.txt	1.4 kB
README.txt	1.8 kB
profiles	1 item
sites	9 items
themes	2 items
autoload.php	525 bytes

3. In Drupal, log in and select the **Extend** option to access the list of available modules.
4. Using the search text field, type in Honeypot. Check the checkbox and click **Install**.
5. Once enabled, search for it again. Clicking on the module's description will expand the row and expose links to configure permissions and module settings:

The screenshot shows the 'SPAM CONTROL' section of the Honeypot module configuration. A search bar at the top contains the text 'honeypot'. Below it is a text input field with placeholder text 'Enter a part of the module name or description'. Under the heading '▼ SPAM CONTROL', there is a list with one item: 'Honeypot' (checkbox checked). To the right of the list, a detailed description is shown: 'Mitigates spam form submissions using the honeypot method.' Below this, 'Machine name: honeypot' and 'Version: 8.x-1.19-beta14' are listed. At the bottom of the panel are two buttons: 'Permissions' and 'Configure'.

6. Visit <https://www.drupal.org/project/bootstrap> and download the latest 8.x release for Bootstrap.

7. Extract the archive and place the `bootstrap` folder inside the `themes` folder inside your Drupal core installation.

core	29 items
modules	2 items
profiles	1 item
sites	9 items
themes	2 items
bootstrap	23 items
config	2 items
css	13 items
docs	8 items
grunt	4 items
includes	8 items
js	3 items
starterkits	2 items
templates	15 items
bootstrap.api.php	6.2 kB
bootstrap.info.yml	1.4 kB
bootstrap.libraries.yml	257 bytes
bootstrap.make.example	303 bytes
bootstrap.theme	2.4 kB
bower.json	158 bytes
favicon.ico	1.1 kB

8. In Drupal, select the **Appearance** option to manage your Drupal themes.
9. Scroll down the page and click **Install and set as default** under **Bootstrap** to enable and set the theme as default:



How it works...

The following outlines the procedure for installing a module or theme and how Drupal discovers these extensions.

Discovering modules and themes

Drupal scans specific folder locations to identify modules and themes defined by the `.info.yml` file in their directory. The following is the order in which projects will be discovered:

- ▶ Respective core folder (modules, themes)
- ▶ Current installed profile
- ▶ The root `modules` or `themes` folder
- ▶ The current site directory (default or current domain)

Module installation

By placing the module inside the root `modules` folder, we are allowing Drupal to discover the module and allow it to be installed. When a module is installed, Drupal will register its code with the system through the `module_installer` service. The service will check for required dependencies and prompt for them to be enabled if required. The configuration system will run any configuration definitions provided by the module on install. If there are conflicting configuration items, the module *will not be installed*.

Theme installation

A theme is installed through the `theme_installer` service and sets any default configuration by the theme along with rebuilding the theme registry. Setting a theme to default is a configuration change in `system.theme.default` to the theme's machine name (in the recipe it would be `bootstrap`).

There's more...

The following outlines the procedure for installing a module or theme and some more information on it.

Installing a module with Drush

Modules can be downloaded and enabled through the command line using drush. The command to replicate the recipe would resemble:

```
$ drush pm-download honeypot  
$ drush pm-enable honeypot
```

It will prompt you to confirm your action. If there were dependencies for the module, it would ask if you would like to enable those, too.

Uninstalling a module

One of the larger changes in Drupal 8 is the module disable and uninstall process. Previously modules were first disabled and then uninstalled once disabled. This left a confusing process which would disable its features, but not clean up any database schema changes. In Drupal 8 modules cannot just be disabled but must be uninstalled. This ensures that when a module is uninstalled it can be safely removed from the code base.

A module can only be uninstalled if it is not a dependency of another module or does not have a configuration item in use – such as a field type – which could disrupt the installation's integrity.



With a standard installation, the `Comment` module cannot be uninstalled until you delete all `Comment` fields on the article content type. This is because the `field` type is in use.



See also...

- ▶ [Chapter 4, Extending Drupal](#), to learn about setting defaults on enabling a module
- ▶ [Chapter 9, Configuration Management – Deploying in Drupal 8](#)

Using multisites in Drupal 8

Drupal provides the ability to run multiple sites from one single Drupal code base instance. This feature is referred to as multisite. Each site has a separate database; however, projects stored in `modules`, `profiles`, and `themes` can be installed by all of the sites.

Site folders can also contain their own modules and themes. When provided, these can only be used by that one site.

The `default` folder is the default folder used if there is not a matching domain name.

Getting ready

If you are going to work with multisite functionality, you should have an understanding of how to setup virtual host configurations with your particular web server. In this recipe, we will use two subdomains under localhost called `dev1` and `dev2`.

How to do it...

We will use multisites in Drupal 8 by two subdomains under localhost:

1. Copy `sites/example.sites.php` to `sites/sites.php`.
2. Create a `dev1.localhost` and a `dev2.localhost` folder inside of the `sites` folder.
3. Copy the `sites/default/default.settings.php` file into `dev1.localhost` and `dev2.localhost` as `settings.php` in each respective folder:

www	14 items	Folder
core	29 items	Folder
modules	2 items	Folder
profiles	1 item	Folder
sites	8 items	Folder
default	5 items	Folder
dev1.localhost	2 items	Folder
files	1 item	Folder
settings.php	27.7 kB	Program
dev2.localhost	2 items	Folder
files	1 item	Folder
settings.php	27.7 kB	Program
development.services.yml	249 bytes	Text
example.settings.local.php	2.5 kB	Program
example.sites.php	2.3 kB	Program
README.txt	515 bytes	Text
sites.php	2.4 kB	Program

4. Visit `dev1.localhost` and run the installation wizard.
5. Visit `dev2.localhost` and see that you still have the option to install a site!

How it works...

The `sites.php` must exist for multisite functionality to work. By default, you do not need to modify its contents. The `sites.php` file provides a way to map aliases to specific site folders. The file contains the documentation for using aliases.

The `DrupalKernel` class provides `findSitePath` and `getSitePath` to discover the site folder path. On Drupal's bootstrap this is initiated and reads the incoming HTTP host to load the proper `settings.php` file from the appropriate folder. The `settings.php` file is then loaded and parsed into a `\Drupal\Core\Site\Settings` instance. This allows Drupal to connect to the appropriate database.

There's more...

Let's understand the security concerns of using multisite:

Security concerns

There can be cause for concern if using multisite. Arbitrary PHP code executed on a Drupal site might be able to affect other sites sharing the same code base. Drupal 8 marked the removal of the `PHP Filter` module that allowed site administrators to use PHP code in the administrative interface. While this mitigates the various ways an administrator had easy access to run PHP through an interface it does not mitigate the risk wholesale. For example, the `PHP Filter` module is now a contributed project and could be installed.

See also...

- ▶ Multi-site documentation on [Drupal.org](https://www.drupal.org/documentation/install/multi-site), <https://www.drupal.org/documentation/install/multi-site>

Tools for setting up an environment

One of the initial hurdles to getting started with Drupal is a local development environment. This recipe will cover how to set up the DrupalVM project by Jeff Geerling. DrupalVM is a VirtualBox virtual machine run through Vagrant, provisioned and configured with Ansible. It will set up all of your services and build a Drupal installation for you.

Luckily you only need to have VirtualBox and Vagrant installed on your machine and DrupalVM works on Windows, Mac OS X, and Linux.

Getting ready

To get started, you will need to install the two dependencies required for DrupalVM:

- ▶ **VirtualBox:** <https://www.virtualbox.org/wiki/Downloads>
- ▶ **Vagrant:** <http://www.vagrantup.com/downloads.html>

How to do it...

Let's set up the DrupalVM project by Jeff Geerling. DrupalVM is a VirtualBox virtual machine run through Vagrant, provisioned and configured with Ansible:

1. Download the DrupalVM archive from <https://github.com/geerlingguy/drupal-vm/archive/master.zip>.

2. Extract the archive and place the project in your directory of choice.
3. Copy `example.drupal.make.yml` to `drupal.make.yml`.
4. Copy `example.config.yml` to `config.yml`
5. Edit `config.yml` and modify the `local_path` setting to be the directory where you've placed the DrupalVM project. This will be synchronized into the virtual machine:

```
vagrant synced_folders:  
  - local_path: /path/to/drupalvm  
    destination: /var/www  
    type: nfs  
    create: true
```

6. Open a terminal and navigate to the directory where you have placed the DrupalVM project.
7. Enter the command `vagrant up` to tell Vagrant to build the virtual machine and begin the provisioning process.
8. While this process is ongoing, modify your hosts file to provide easy access to the development site. Add the line `192.168.88.88 drupalvm.dev` to your hosts file.
9. Open your browser and access <http://www.drupalvm.com/>.
10. Login to your Drupal site with the username `admin` and password `admin`.

How it works...

DrupalVM is a development project that utilizes the Vagrant tool to create a VirtualBox virtual machine. Vagrant is configured through the project's `Vagrantfile`. Vagrant then uses Ansible – an open source IT automation platform – to install Apache, PHP, MySQL, and other services on the virtual machine.

The `config.yml` file has been set up to provide a simple way to customize variables for the virtual machine and provisioning process. It also uses Drush to create and install a Drupal 8 site, or whatever components are specified in `drupal.make.yml`. This file is a Drush make file, which contains a definition for Drupal core by default and can be modified to include other contributed projects.

The `vagrant up` command tells Vagrant to either launch an existing virtual machine or create one anew in a headless manner. When Vagrant creates a new virtual machine it triggers the provisioning process. In this instance Ansible will read the `provisioning/playbook.yml` file and follow each step to create the final virtual machine. The only files needing to be modified, however, are the `config.yml` and `drupal.make.yml` files.

There's more...

The topic of automating and streamlining a local environment is quite popular right now with quite a few different options. If you are not comfortable with using Vagrant, there are a few other options that provide a server installation and Drupal.

Acquia Dev Desktop

Acquia Dev Desktop is developed by Acquia and can be found at <https://docs.acquia.com/dev-desktop2>. It is an automated environment installer for Windows and Mac. The Dev Desktop application allows you to create a regular Drupal installation or select from a distribution.

XAMPP + Bitnami

XAMPP – Apache + MySQL + PHP + Perl – is a cross platform environment installation. XAMPP is an open source project from Apache Friends. XAMPP has partnered with Bitnami to provide free all-in-one installations for common applications – including Drupal 8! You can download XAMPP at <https://www.apachefriends.org/download.html>.

Kalabox

Kalabox is developed by the Kalamuna group and intends to be a robust workflow solution for Drupal development. Kalabox is cross-platform compatible, allowing you to easily work on Windows machines. It is based for the command line and provides application binaries for you to install. You can learn more about Kalabox at <http://www.kalamuna.com/products/kalabox/>.

See also...

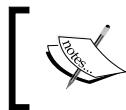
- ▶ See *Chapter 13, Drupal CLI*, for information on makefiles
- ▶ DrupalVM documentation <http://docs.drupalvm.com/en/latest/>
- ▶ Drupal.org community documentation on local environment set-up
<https://www.drupal.org/node/157602>

Running Simpletest and PHPUnit

Drupal 8 ships with two testing suites. Previously Drupal only supported Simpletest. Now there are PHPUnit tests as well. In the official change record, PHPUnit was added to provide testing without requiring a full Drupal bootstrap, which occurs with each Simpletest test. Read the change record here: <https://www.drupal.org/node/2012184>.

Getting ready

Currently core comes with Composer dependencies prepackaged and no extra steps need to be taken to run PHPUnit. This recipe will demonstrate how to run tests the same way that the QA testbot on Drupal.org does.



The process of managing Composer dependencies may change, but is currently postponed due to Drupal.org's testing and packaging infrastructure. Read more here <https://www.drupal.org/node/1475510>.



How to do it...

1. First enable the Simpletest module. Even though you might only want to run PHPUnit, this is a soft dependency for running the test runner script.
2. Open a command line terminal and navigate to your Drupal installation directory and run the following to execute all available PHPUnit tests:
`php core/scripts/run-tests.sh PHPUnit`
3. Running Simpletest tests required executing the same script, however, instead of passing PHPUnit as the argument, you must define the `url` option and `tests` option:
`php core/scripts/run-tests.sh --url http://localhost --all`
4. Review test output!

How it works...

The `run-tests.sh` script has been shipped with Drupal since 2008, then named `run-functional-tests.php`. The command interacts with the other suites in Drupal to run all or specific tests and sets up other configuration items. We will highlight some of the useful options below:

- ▶ **`--help`**: This displays the items covered in the following bullets
- ▶ **`--list`**: This displays the available test groups that can be run
- ▶ **`--url`**: This is required unless the Drupal site is accessible through `http://localhost:80`
- ▶ **`--sqlite`**: This allows you to run Simpletest without having to have Drupal installed
- ▶ **`--concurrency`**: This allows you to define how many tests run in parallel

There's more...

Is run-tests a shell script?

The `run-tests.sh` isn't actually a shell script. It is a PHP script which is why you must execute it with PHP. In fact, within core/scripts each file is a PHP script file meant to be executed from the command line. These scripts are not intended to be run through a web server which is one of the reasons for the `.sh` extension. There are issues with discovered PHP across platforms that prevent providing a shebang line to allow executing the file as a normal bash or bat script. For more info view this Drupal.org issue at <https://www.drupal.org/node/655178>.

Running Simpletest without Drupal installed

With Drupal 8, Simpletest can be run from SQLite and no longer requires an installed database. This can be accomplished by passing the `sqlite` and `dburl` options to the `run-tests.sh` script. This requires the PHP SQLite extension to be installed.

Here is an example adapted from the DrupalCI test runner for Drupal core:

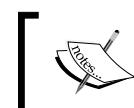
```
php core/scripts/run-tests.sh --sqlite /tmp/.ht.sqlite --die-on-fail  
--dburl sqlite://tmp/.ht.sqlite --all
```

Combined with the built in PHP webserver for debugging you can run Simpletest without a full-fledged environment.

Running specific tests

Each example thus far has used the `all` option to run every Simpletest available. There are various ways to run specific tests:

- ▶ **-module:** This allows you to run all the tests of a specific module
- ▶ **-class:** This runs a specific path, identified by a full namespace path
- ▶ **-file:** This runs tests from a specified file
- ▶ **-directory:** This run tests within a specified directory



Previously in Drupal, tests were grouped inside `module.test` files, which is where the file option derives from. Drupal 8 utilizes the PSR-4 autoloading method and requires one class per file.

DrupalCI

With Drupal 8 came a new initiative to upgrade the testing infrastructure on [Drupal.org](https://www.drupal.org). The outcome was DrupalCI. DrupalCI is open source and can be downloaded and run locally. The project page for DrupalCI is <https://www.drupal.org/project/drupalci>.

The test bot utilizes Docker and can be downloaded locally to run tests. The project ships with a Vagrant file to allow it to be run within a virtual machine or locally. Learn more on the testbot's project page: https://www.drupal.org/project/drupalci_testbot.

See also...

- ▶ PHPUnit manual: <https://phpunit.de/manual/4.8/en/writing-tests-for-phpunit.html>
- ▶ Drupal PHPUnit handbook: <https://drupal.org/phpunit>
- ▶ Simpletest from the command line: <https://www.drupal.org/node/645286>

2

The Content Authoring Experience

In this chapter we will explore what Drupal 8 brings to the content authoring experience:

- ▶ Configuring the WYSIWYG editor
- ▶ Adding and editing content
- ▶ Creating a menu and linking content
- ▶ Providing inline editing
- ▶ Creating a custom content type
- ▶ Applying new Drupal 8 core field types
- ▶ Customizing the form display of a node
- ▶ Customizing the display output of a node

Introduction

In this chapter we'll cover the Drupal 8 content authoring experience. We will show you how to configure text formats and set up the bundled CKEditor that ships with Drupal 8. We will look at how to add and manage content, along with utilizing menus for linking to content. Drupal 8 ships with inline editing for per-field modifications from the front-end.

This chapter dives into *creating custom content types* and harnessing different fields to create advanced content. We'll cover the five new fields added to Drupal 8 core and how to use them, along with getting new field types through contributed projects. We will go through customizing the content's display and modifying the new form display added in Drupal 8.

Configuring the WYSIWYG editor

Drupal 8 caused the collaboration between the Drupal development community and the CKEditor development community. Because of this, Drupal now ships with CKEditor out of the box as the default **What You See Is What You Get (WYSIWYG)** editor. The new Editor module provides an API for integrating WYSIWYG editors. Even though CKEditor is provided out of the box, contributed modules can provide integrations with other WYSIWYG editors.

Text formats control the formatting of content and WYSIWYG editor configuration for content authors. The standard Drupal installation profile provides a fully configured text format with CKEditor enabled. We will walk through the steps of recreating this text format.

In this recipe we will create a new text format with a custom CKEditor WYSIWYG configuration.

Getting ready

Before starting, make sure that the CKEditor module is enabled, which also requires Editor as a dependency. **Editor** is the module that provides an API to integrate WYSIWYG editors with text formats.

How to do it...

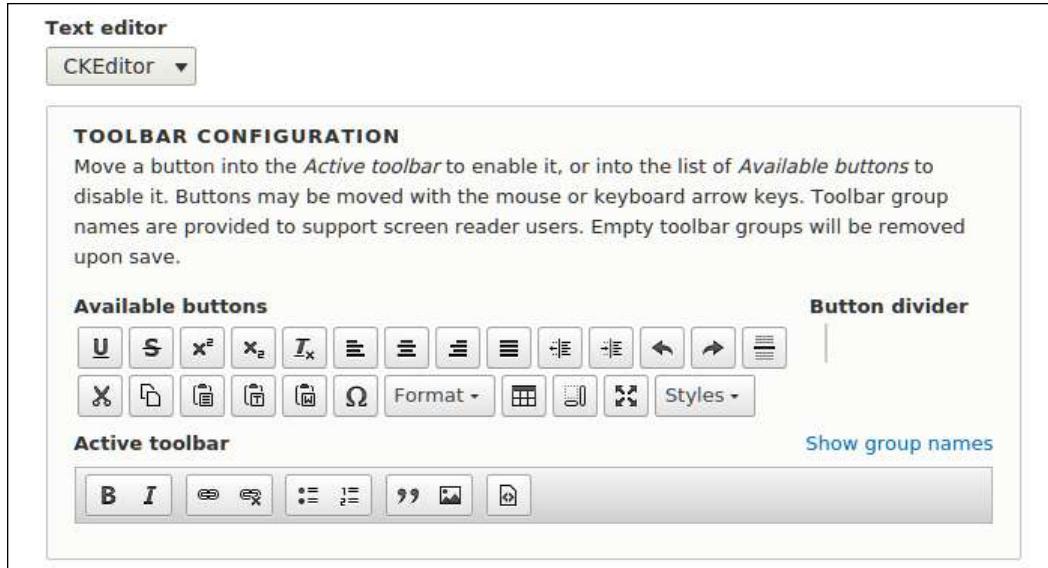
Let's create a new text format with a custom CKEditor WYSIWYG configuration:

1. Visit **Configuration** and head to **Text formats and editors** under the **Content authoring** heading.
2. Click on **Add text format** to begin creating the new text format:

The screenshot shows the 'Text formats and editors' configuration page. At the top, there is a navigation breadcrumb: Home > Administration > Configuration > Content authoring. Below the breadcrumb, a note states: 'Text formats define the HTML tags, code, and other formatting that can be used when entering text. **Improper text format configuration is a security risk.**' It also links to the 'Filter module help page'. A note below says: 'Text formats are presented on content editing pages in the order defined on this page. The first format available to a user will be selected by default.' At the bottom of the page is a blue button labeled '+ Add text format'.

3. Enter a name for the text format, such as **editor format**.
4. Select which roles have access to this format – this allows you to have granular control over what users can use when authoring content.

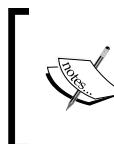
5. Select **CKEditor** from the **Text editor** select list. The configuration form for CKEditor will then be loaded.
6. You may now use an in-place editor to drag buttons onto the provided toolbar to configure your CKEditor toolbar:



7. Select any of the **Enabled** filters you would like, except for **Display any HTML as Plain text**. That would be counter intuitive to using a WYSIWYG editor!

How it works...

The Filter modules provide text formats that control over how rich text fields are presented to the user. Drupal will render rich text saved in a text area based on the defined text format for the field. Text fields with (formatted) in their title will respect text format settings, others will render in plain text.



The text formats and editors screen warns of a security risk due to improper configuration. That is because you could grant an anonymous user access to a text format that allows full HTML, or allow image sources to be from remote URLs.

The Editor module provides a bridge to WYSIWYG editors and text formats. It alters the text format form and rendering to allow the integration of WYSIWYG editor libraries. This allows each text format to have its own configuration for its WYSIWYG editor.

Out of the box the Editor module alone does not provide an editor. The CKEditor module works with the Editor API to enable usage of that WYSIWYG editor.

Drupal can support other WYSIWIG editors, such as MarkItUp or TinyMCE through contributed modules.

There's more...

Drupal provides granular control of how rich text is rendered and extensible ways as well, which we will discuss further.

Filter module

When string data is added to a field that supports text formats, the data is saved and preserved as it was originally entered. Enabled filters for a text format will not be applied until the content is viewed. Drupal works in such a way that it saves the original content and only filters on display.

With the Filter module enabled, you gain the ability to specify how text is rendered based on the roles of the user who created the text. It is important to understand the filters applied to a text format that uses a WYSIWYG editor. For example, if you selected the **Display any HTML as plain text** option, the formatting done by the WYSIWYG editor would be stripped out when viewed.

CKEditor plugins

The CKEditor module provides a plugin type called `CKeditorPlugin`. Plugins are small pieces of swappable functionality within Drupal 8. Plugins and plugin development are covered in *Chapter 7, Plug and Play with Plugins*. This type provides integration between CKEditor and Drupal 8.

The image and link capabilities are plugins defined within the CKEditor module. Additional plugins can be provided through contributed projects or custom development.

See the `\Drupal\ckeditor\Annotation\CKEditorPlugin` class for the plugin definition and the suggested `\Drupal\ckeditor\Plugin\CKEditorPlugin\DrupalImage` class as a working example.

See also

- ▶ The official blog post from CKEditor about how Drupal adopted it as the official WYSIWYG editor: <http://ckeditor.com/blog/CKEditor-Joins-Drupal>.
- ▶ *Chapter 7, Plug and Play with Plugins*.

Adding and editing content

The main functionality of a content management system is in the name itself – the ability to manage content; that is, to add, edit, and organize content. Drupal provides a central form that allows you to manage all of the content within your website and allows you to create new content. Additionally, you can view a piece of content and have the ability to click an edit link when viewing it.

Getting ready

This recipe assumes you have installed the standard installation profile and have the default node content types available for use.

How to do it...

Let's manage the content by adding, editing, and organizing the content:

1. Visit **Content** to view the content management overview from.
2. Click **Add content** to view the list of available content types. Select **article** as the piece of content you would like to make.
3. Provide a title for the piece of content. Titles are always required for content.

Fill in body text for the article:

Body (Edit summary)

B I Format Source

Text format Basic HTML About text formats ?

Tags Editor format Restricted HTML Full HTML

Enter a comma-separated list. For example: Amsterdam, Mexico City, "Cleveland, Ohio"



You may change the text format to customize what kind of text is allowed. If the user only has one format available there will be no select box, but the **About** text formats link will still be present.

4. Once you have added your text, click **Save and publish** at the bottom of the form. You will then be redirected to view the newly created piece of content.
5. Note that the URL for the piece of content is `/node/#`. This is the default path for content and can be changed by editing the content.
6. Click on **Edit** from the tabs right above the content.
7. From the right sidebar, click on **URL Path Settings** to expand the section and enter a custom alias. For example `/awesome-article` (note the required `/`):



A screenshot of a 'URL PATH SETTINGS' dialog box. It shows a 'URL alias' field containing '/awesome-article'. Below the field is a descriptive text: 'The alternative URL for this content. Use a relative path. For example, enter "/about" for the about page.'

8. Save the content and notice the URL for your article is `/awesome-article`.
9. You could also edit this article from the Content table by clicking **Edit** there instead of from viewing the content.

How it works...

The Content page is a **View**, which will be discussed in *Chapter 3, Displaying Content through Views*. This creates a table of all the content in your site that can be searched and filtered. From here you can view, edit, or delete any single piece of content.

In Drupal there are content entities that provide a method of creation, editing, deletion, and viewing. Nodes are a form of a content entity. When you create a node it will build the proper form that allows you to fill in the piece of content's data. The same process follows for editing content.

When you save the content, Drupal writes the node's content to the database along with all of its respective field data.

There's more

Drupal 8's content management system provides many features; we will cover some extra information.

Save as draft

New to Drupal 8 is the ability to easily save a piece of content as a draft instead of directly publishing it. Instead of clicking on **Save and publish**, click the arrow next to it to expand the option of **Save as unpublished**.



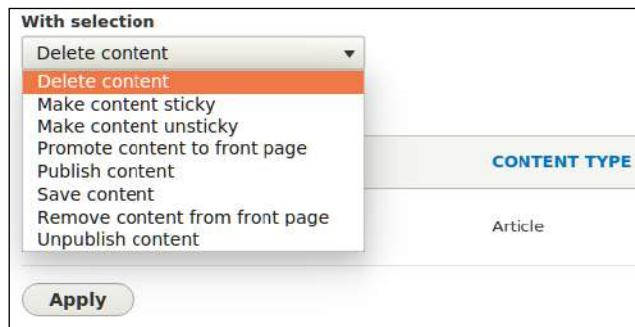
Pathauto

There is a contributed project called Pathauto that simplifies the process of providing URL aliases. It allows you to define patterns that will automatically create URL aliases for content. This module utilizes tokens to allow for very robust paths for content.

The Pathauto project can be found at <https://www.drupal.org/project/pathauto>.

Bulk moderation

You also have the capability to perform bulk actions on content. The table provides checkboxes at the beginning of each row. For each selected item, you can choose an item from **With selection** to make bulk changes – such as deleting, publishing, and unpublishing content:



See also

- ▶ Chapter 2, *The Content Authoring Experience in recipe Customizing the Form Display of a Node*

Creating a menu and linking content

Drupal provides a way to link content being authored to a specified menu on the website, generally the main menu. You can, however, create a custom menu for providing links to content. In this recipe we will show you how to create a custom menu and link content to it. We will then place the menu as a block on the page, in the sidebar.

Getting ready

This recipe assumes you have installed the standard installation profile and have the default node content types available for use. You should have some content created to create a link to.

How to do it...

1. Visit **Structure** and click on **Menus**.
2. Click on **Add Menu**.
3. Provide the title **Sidebar** and optional summary and then click on **Save**.
4. Once the menu has saved, click on the **Add link** button.
5. Enter in a link title and then begin typing the title for a piece of content. The form will provide autocomplete suggestions for linkable content:

The screenshot shows the 'Add link' configuration form for a menu item. It includes fields for 'Menu link title', 'Link', 'Description', and 'Parent link'. The 'Link' field is highlighted with a blue border, showing the title 'lo' and the placeholder 'Lorem ipsum'. The 'Enabled' checkbox is checked. The 'Parent link' dropdown is set to '<Sidebar>'.

Menu link title *
Link to content

The text to be used for this link in the menu.

Link *
lo
Lorem ipsum Also enter an internal path such as /node/add or an external URL such as http://example.com. Enter <front> to link to the front page.

Enabled
A flag for whether the link should be enabled in menus or hidden.

Description

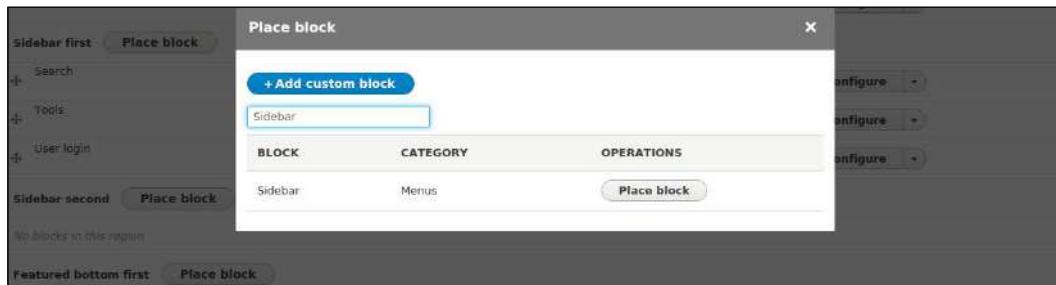
Shown when hovering over the menu link.

Show as expanded
If selected and this menu link has children, the menu will always appear expanded.

Parent link

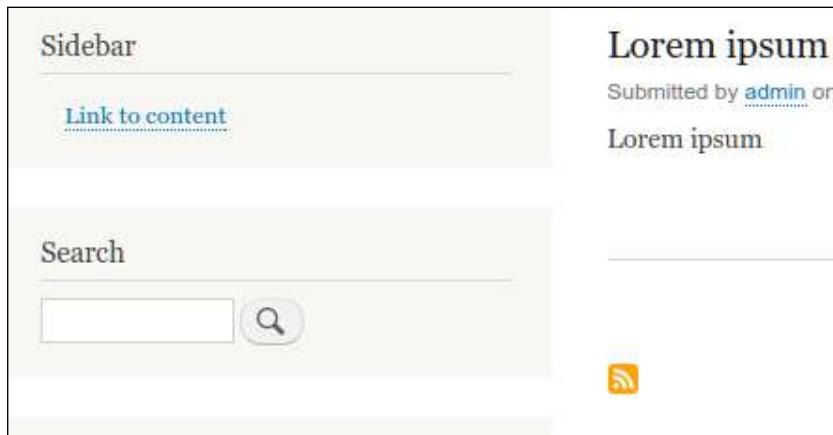
The maximum depth for a link and all its children is fixed. Some menu links may not be available as parents if selecting them would exceed this limit.

6. Click on **Save** to save the menu link.
7. With the menu link saved, visit **Structure**, and then **Block layout**.
8. Click on **Place block** next to **Sidebar first**. In the modal, search for the **Sidebar** menu and click on **Place block**:



9. Save the following forms and, at the bottom of the block list, click on **Save**.

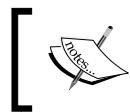
View your Drupal site and you will see the menu:



How it works...

Menus and links are part of Drupal core. The ability to make custom menus and menu links is provided through the **Menu UI** module. This module is enabled on the standard installation profile, but may not be in others.

The **Link** input of the menu link form allows you to begin typing node titles and easily link to existing content. This was a piece of functionality not available in previous versions of Drupal! It will automatically convert the title into the internal path for you. Link input also accepts a regular path, such as /node/1 or an external path.



You must have a valid path; you cannot add empty links to a menu. There is work being done to allow adding empty or ID selector link paths: <https://www.drupal.org/node/1543750>.



There's more...

Managing a contents menu link from its form

A piece of content can be linked to a menu from the **add** or **edit** form. The **menu settings** section allows you to toggle the availability of a menu link. The menu link title will reflect the content's title by default.

The parent item allows you to decide which menu and which item it will appear under. By default content types only have the main menu allowed. Editing a content type can allow multiple menus, or only choosing a custom menu.

This allows you to populate a main menu or complimentary menu without having to visit the menu management screens.

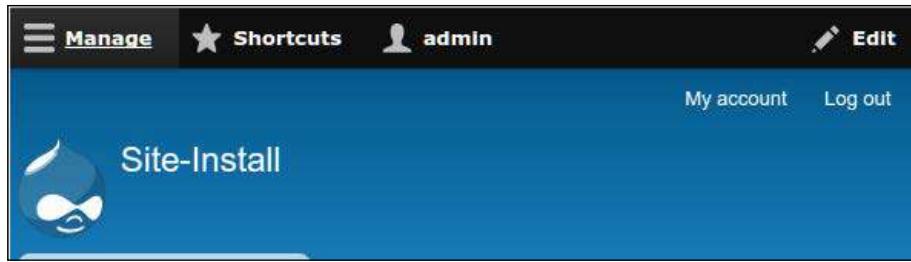
Providing inline editing

A touted feature of Drupal 8 is the ability to provide inline editing. Inline editing is enabled by default with the standard installation profile through the **Quick Edit** module. The Quick Edit module allows for editing individual fields while viewing a piece of content and integrates with the Editor module for WYSIWYG editors!

How to do it...

Let's provide inline editing:

1. Visit a piece of created content.
2. In order to enable inline editing, you must toggle contextual links on the page by clicking **Edit** in the upper right of the administrative toolbar:



3. This will toggle the contextual links available on the page. Click on the **context link** next to the content and select **Quick edit**:

A screenshot of a Drupal content page titled 'Lorem ipsum'. The page has a text field containing 'Lorem ipsum'. Above the text field are three buttons: 'View', 'Edit', and 'Delete'. To the right of the text field, a contextual menu is open, showing options: 'Quick edit' (with a pencil icon), 'Edit', and 'Delete'. Below the text field, there is a link 'Add new comment'.

4. Hover over the body text and click to **Edit**.
5. You can now edit the text with a minimal version of the WYSIWYG editor toolbar.
6. Once you have changed the text, click **Save**.
7. The changes will be saved immediately.

How it works...

The **Contextual links** module provides privileged users with shortcut links to modify blocks or content. The contextual links are toggled by clicking **Edit** in the toolbar. The **Edit** link toggles the visibility of contextual links on the page. Previously, in Drupal 7, contextual links appeared as cogs when a specific region was hovered over.

The Quick Edit module builds on top of the contextual links features. It allows field formatters, which display field data, to describe how they will interact. By default Quick Edit sets this to a form. Clicking on an element will use JavaScript to load a form and save data via AJAX calls.

Quick Edit will not work on administrative pages.

Creating a custom content type

Drupal excels in the realm of content management by allowing different types of content. In this recipe we will walk through creating a custom content type. We will create a **Services** type that has some basic fields and would be used in a scenario that brings attention to a company's provided services.

You will also learn how to add fields to a content type in this recipe; which generally goes hand in hand when making a new content type on a Drupal site.

How to do it...

1. Visit **Structure** and then **Content types**. Click **Add content type** to begin creating a new content type.
2. Enter **Services** as the name and an optional description.
3. Select **Display settings** and uncheck the **Display author and date information** checkbox. This will hide the author and submitted time from services pages.

The screenshot shows the 'Add content type' form. At the top, there is a breadcrumb navigation: Home > Administration > Structure > Content types. Below the breadcrumb, a message states: 'Individual content types can have different fields, behaviors, and permissions assigned to them.' The 'Name *' field is filled with 'Services' and has a machine name of 'services'. The 'Description' field is empty. Under 'Submission form settings', the 'Title' field is checked. Under 'Publishing options', 'Published' and 'Promoted to front page' are selected. Under 'Display settings', the 'Don't display post information' option is selected. A note next to it says: 'Author username and publish date will be displayed.' At the bottom, there is a 'Save and manage fields' button.

4. Press the **Save and manage fields** button to save our new content type and manage its fields.

5. By default, new content types have a **Body** field automatically added to them. We will keep this field in place.
6. We will add a field that will provide a way to enter a marketing headline for the service. Click on **Add field**.

Select **Text (plain)** from the drop down and enter **Marketing headline** as the label:

Add field

Home » Administration » Structure » Content types » Services » Manage fields

Add a new field **Re-use an existing field**
 Text (plain) - Select an existing field -

Label *
 Marketing headline Machine name: field_marketing_headline [Edit]

Save and continue



Text (plain) is a regular text field. The **Text (formatted)** option will allow you to use text formats on the displayed text in the field.

7. Save the field settings on the next form. On the next form you may hit **Save settings** to finish adding the field.

The field has now been added and content of this type can be created:

Manage fields

Manage fields ...

Home » Administration » Structure » Content types » Services

LABEL	MACHINE NAME	FIELD TYPE	OPERATIONS
Body	body	Text (formatted, long, with summary)	Edit
Marketing headline	field_marketing_headline	Text (plain)	Edit

How it works...

In Drupal there are entities that have bundles. A bundle is just a type of entity that can have specific configuration and fields attached. When working with nodes, a bundle is generally referred to as a content type.

Content types can be created as long as the Node module is enabled. When a content type is created through the user interface, it invokes the `node_add_body_field()` function. This function adds the default body field for content types.

Fields can only be managed or added if the Field UI module is enabled. The Field UI module exposes the **Manage Fields**, **Manage Form Display**, and **Manage Display** for entities – such as nodes and blocks.

Applying new Drupal 8 core field types

The field system is what makes creating content in Drupal so robust. With Drupal 8 some of the most used contributed field types have been merged into Drupal core as their own module. In fact, Entity reference is no longer a module but part of the main Field API now.

This recipe is actually a collection of mini-recipes to highlight the new fields: Link, Email, Telephone, Date, and Entity reference!

Getting ready

The standard installation profile does not enable all of the modules that provide these field types by default. For this recipe you will need to manually enable select modules so you can create the field. The module that provides the field type and its installation status in the standard profile will be highlighted.

Each recipe will start off expecting that you have enabled the module, if needed, and to be at the **Manage fields** form of a content type and have clicked on **Add field** and provided a field label. The recipes cover the settings for each field.

How to do it...

This section contains a series of mini recipes, showing how to use each of the new core field types.

Link

The Link field is provided by the Link module. It is enabled by default with the standard installation profile. It is a dependency of the **Menu UI**, **Custom Menu Links**, and **Shortcut module**.

1. The Link field type does not have any additional field level settings that are used across all bundles.
2. Click **Save field settings** to begin customizing the field for this specific bundle.
3. Using the **Allowed link type** setting, you can control whether provided URLs can be external, internal, or both. Selecting **Internal or Both** will allow linking to content by autocompleting the title.
4. The **Allow link** text defines if a user must provide text to go along with the link. If no text is provided, then the URL itself is displayed.
5. The field formatter for a Link field allows you to specify `rel="nofollow"` or if the link should open in a new window.

The e-mail field

The **Email** field is provided by core and is available without enabling additional modules:

1. The **Email** field type does not have any additional field level settings that are used across all bundles.
2. Click **Save field settings** to begin customizing the field for this specific bundle.
3. There are no further settings for an **Email** field instance. This field uses the HTML5 e-mail input, which will leverage browser input validation.
4. The field formatter for an **Email** field allows you to display the e-mail as plain text or a `mailto:` link.

The Telephone field

The **Telephone** field is provided by the **Telephone** module. It is not enabled by default with the standard installation profile:

1. The **Telephone** field type does not have any additional field level settings that are used across all bundles.
2. Click **Save field settings** to begin customizing the field for this specific bundle.
3. There are no further settings for a **Telephone** field instance. This field uses the HTML5 e-mail input, which will leverage browser input validation.
4. The field formatter for a **Telephone** field allows you to display the telephone number as a plain text item, or using the `tel: [link]` with an optional replacement title for the link.

Date

The **Date** field is provided by the Datetime module. It is enabled by default with the standard installation profile.

1. The Date module has a setting that defines what kind of data it will be storing: date and time, or date only. This setting cannot be changed once field data has been saved.
2. Click **Save field settings** to begin customizing the field for this specific bundle.
3. The **Date** field has two ways of providing a default value. It can either be the current date or a relative date using PHP's date time modifier syntax.
4. By default, **Date** fields use the HTML5 date and time inputs, resulting in a native date and time picker provided by the browser.
5. Additionally, **Date** fields can be configured to use a select list for each date and time component:



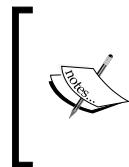
6. The default date field formatter display uses Drupal's time formats to render the time format. These are configured under **Configuration** and **Regional and language** in the **Date and time formats** form.
7. Dates and times can be displayed as **Time ago** to provide a semantic display of how far in the future or past a time is. The formats for both are customizable in the display settings.
8. Finally, dates and times can be displayed using a custom format as specified by the PHP date formats.

The Entity Reference

The Entity Reference field is part of core and is available without enabling additional modules. Unlike other fields, Entity Reference appears as a grouping of specific items when adding a field. That is because you must pick a type of entity to reference!

1. The interface allows you to select a **Content**, **File**, **Image**, **Taxonomy term**, **User**, or **Other**. Selecting one of the predefined options will preconfigure the field's target entity type.

2. When creating an Entity Reference field using the Other choice, you must specify the type of item to reference. This option cannot be changed once data is saved.



You will notice there are two groups: **content** and **configuration**. Drupal uses configuration entities. Even though configuration is an option, you may not benefit from referencing those entity types. Only content entities have a way to be viewed. Referencing configuration entities would fall under an advanced use case implementation.

3. Click **Save field settings** to begin customizing the field for this specific bundle.
4. The Entity Reference field has two different methods for allowing users to search for content: using the default autocomplete or a View.
5. Depending on the type of entity you are referencing, there will be different entity properties you may sort the results based on.
6. The default field widget for an Entity Reference field is to use autocomplete, however there is the option to use a select list or checkboxes for the available options.
7. The values of an Entity Reference field can display the referenced entity's label or the rendered output. When rendering a label it can be optionally linked to the entity itself. When displaying a rendered entity you may choose a specific view mode.

How it works...

When working with fields in Drupal 8, there are two steps. When you first create a field you are defining a base field to be saved. This configuration is a base that specifies how many values a field can support and any additional settings defined by the field type. When you attach a field to a bundle it is considered a field storage and contains configuration unique to that specific bundle. If you have the same **Link** field on the **Article** and **Page** content type, the label, link type, and link text settings are for each instance.

Each field type provides a method for storing and presents a specific type of data. The benefit of using these fields comes from validation and data manipulation. It also allows you to utilize HTML5 form inputs. By using HTML5 for telephone, e-mail, and date the authoring experience uses the tools provided by the browser instead of additional third party libraries. This also provides a more native experience when authoring with mobile devices.

There's more...

Having Drupal 8 released with new fields was a large feature and we will cover some additional topics.

Upcoming updates

Each of the recipes covers a field type that was once part of the contributed project space. These projects provided more configuration options than are found in core at the time of writing this module. Over time more and more features will be brought into core from their source projects.

For instance, the Datetime module is based on the contributed date project. However not all of the contributed project's features have made it to Drupal core. Each minor release of Drupal 8 could see more features moved to core.

Views and Entity Reference

Using a View with an Entity Reference field is covered in *Chapter 3, Displaying Content through Views*. Using a View, you can customize the way results are fetched for a reference field.

See also

- ▶ *Chapter 3, Displaying Content through Views*, providing an entity reference result view

Customizing the form display of a node

New in Drupal 8 is the availability of form display modes. Form modes allow a site administrator to configure different field configurations for each content entity bundle edit form. In the case of nodes, you have the ability to rearrange and alter the display of fields and properties on the node edit form.

In this recipe we'll modify the default form for creating the **Article** content type that comes with the standard installation profile.

The screenshot shows the 'Create Article' page. At the top, there's a title field and a rich text editor for the body. On the right, a sidebar displays user information ('Last saved: Not saved yet', 'Author: admin') and several expandable sections: 'MENU SETTINGS', 'COMMENT SETTINGS', 'URL PATH SETTINGS', 'AUTHORING INFORMATION', and 'PROMOTION OPTIONS'. At the bottom, there are text format dropdowns ('Text format' and 'Basic HTML') and a link to 'About text formats'.

How to do it...

1. To customize the form display mode, visit **Structure** and then **Content Types**.
2. We will modify the **Article** content type's form. Click on the **expand the operations** drop down and select **Manage form display**.

The screenshot shows the 'Content types' administration page. It lists two content types: 'Article' and 'Basic page'. The 'Article' row has a detailed description: 'Use *articles* for time-sensitive content like news, press releases or blog posts.' To the right of each content type, there's a vertical operations menu. For the 'Article' type, the 'Manage form display' option is highlighted, indicating it's the current target of modification.

3. First we will modify the **Comments** field. From the **Widget** dropdown choose the **Hidden** option to remove it from the form. Follow the same steps for the **sticky at top of lists** field.
4. Click on the settings cog for the **Body** field. Enter in a placeholder for the field, such as **Enter your article text here**. Click on **Update**.



Note:

The placeholder will only appear on a `textarea` using a text format that does not provide a WYSIWYG editor.

5. Click the **Save** button at the bottom of the page to save your changes. You have now customized the form display!
6. Visit **Content, Add Content**, and then **Article**. Note that the comment settings are no longer displayed, nor the sticky options under promotion options:

The screenshot shows the 'Create Article' interface. At the top, there's a title field and a body rich-text editor with a toolbar. To the right, a sidebar displays 'Last saved: Not saved yet' and 'Author: admin'. It also contains expandable sections for 'MENU SETTINGS', 'URL PATH SETTINGS', 'AUTHORING INFORMATION', and 'PROMOTION OPTIONS'.

How it works...

Entities in Drupal have various view modes for each bundle. In Drupal 7 there were only display view modes, which are covered in the next recipe. Drupal 8 brings in new form modes to allow for more control of how an entity edit form is displayed.

Form display modes are configuration entities. Form display modes dictate how the `\Drupal\Core\EntityContentEntityForm` class will build a form when an entity is edited. This will always be set to default unless changed or specified specifically to a different mode programmatically.

Since form display modes are configuration entities they can be exported using configuration management.

Hidden field properties will have no value, unless there is a provided default value. For example, if you hide the Authoring information without providing code to set a default value the content will be authored by anonymous (no user).

There's more...

Managing form display modes

Form display modes for all entities are managed under one area and are enabled for each bundle type. You must first create a display mode and then it can be configured through the bundle manage interface.

Programmatically providing a default to hidden form items

In *Chapter 6, Creating Forms with the Form API*, we will have a recipe that details altering forms. In order to provide a default value for an entity property hidden on the form display, you will need to alter the form and provide a default value. The Field API provides a way to set a default value when fields are created.

See also

- ▶ *Chapter 10, The Entity API*
- ▶ *Chapter 6, Creating Forms with the Form API*

Customizing the display output of a node

Drupal provides display view modes that allow for customization of the fields and other properties attached to an entity. In this recipe we will adjust the teaser display mode of an **Article**. Each field or property has a control for displaying the label, the format to display the information in, and additional settings for the format.

Harnessing view displays allows you to have full control over how content is viewed on your Drupal site.

How to do it...

1. Now it is time to customize the form display mode by visiting **Structure** and then **Content Types**.
2. We will modify the **Article** content type's display. Click on the dropdown button arrow and select **Manage display**.

The Content Authoring Experience

3. Click on the **Teaser** view mode option to modify it. The teaser view mode is used in node listings, such as the default home page.

Screenshot of the 'Manage display' interface for the 'Article' content type, specifically the 'Teaser' view mode. The interface shows field settings for four fields: Image, Body, Tags, and Links. The 'Image' field is set to 'Hidden' and 'Image' format. The 'Body' field is set to 'Hidden' and 'Summary or trimmed' format with a trim limit of 600 characters. The 'Tags' field is set to 'Above' and 'Label' format. The 'Links' field is set to 'Visible' format. A link 'Show row weights' is visible at the top right of the table.

4. Change the format for **Tags** to be **Hidden**. Additionally, this can be accomplished by dragging it to the hidden section. The tags on an article will no longer be displayed when viewing a teaser view mode.
5. Click on the settings cog for the **Body** field to adjust the trimmed limit. The trim limit is a fallback for **Summary or trimmed** when the summary of a `textarea` field is not provided. Modify this from 600 to 300.
6. Press **Save** to save all of your changes that you have made.
7. View the home page and see that your changes have taken affect!

Screenshot of a Drupal node preview showing the changes made to the 'Teaser' view mode. The node title is 'Fusce bibendum finibus risus'. It was submitted by 'admin' on Sun, 10/04/2015 - 13:16. The node content includes a image of various colorful vegetables (peppers, cucumbers, etc.) and a large amount of text: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi eu dolor vehicula, ullamcorper quam sed, bibendum diam. Proin cursus euismod nisi sit amet ultricies. Suspendisse tincidunt vitae sem nec facilisis. Sed hendrerit risus eros, quis fringilla ligula cursus et.' Below the text are 'Read more' and 'Add new comment' links. At the bottom left is a small orange RSS icon.

How it works...

View display modes are configuration entities. View display modes dictate how the `\Drupal\Core\EntityContentEntityForm` class will build a view display when an entity is viewed. This will always be set to default unless changed or specified as a different mode programmatically.

Since view display modes are configuration entities they can be exported using configuration management.

3

Displaying Content through Views

This chapter will cover the Views module and how to use a variety of its major features:

- ▶ Listing content
- ▶ Editing the default admin interfaces
- ▶ Creating a block from a View
- ▶ Utilizing dynamic arguments
- ▶ Adding a relationship in a View
- ▶ Providing an Entity Reference result View

Introduction

For those who have used Drupal previously, Views is in core for Drupal 8! If you are new to Drupal, Views has been one of the most used contributed projects for Drupal 6 and Drupal 7.

To briefly describe Views, it is a visual query builder, allowing you to pull content from the database and render it in multiple formats. Select administrative areas and content listings provided out of the box by Drupal are all powered by Views. We'll dive into how to use Views to customize the administrative interface, customize ways to display your content, and interact with the entity reference field.

Listing content

Views does one thing, and it does it well: listing content. The power behind the Views module is the amount of configurable power it gives the end user to display content in various forms.

This recipe will cover creating a content listing and linking it in the main menu. We will use the **Article** content type provided by the standard installation and make an articles landing page.

Getting ready

The Views UI module must be enabled in order to manipulate Views from the user interface. By default this is enabled with the standard installation profile.

How to do it...

Let's list the Views listing content:

1. Visit **Structure** and then **Views**. This will bring you to the administrative overview of all the views that have been created:

The screenshot shows the 'Views' administrative page. At the top, there are tabs for 'List' (selected) and 'Settings'. Below the tabs, the breadcrumb navigation shows 'Home > Administration > Structure > Views'. A prominent blue button labeled '+ Add new view' is centered above a search bar with the placeholder 'Filter by view name or description'. The main content area is titled 'Enabled' and lists three views in a table format:

VIEW NAME	DESCRIPTION	TAG	PATH	OPERATIONS
Content Displays: Page Machine name: content	Find and manage content.	default	/admin/content	Edit More
Custom block library Displays: Page Machine name: block_content	Find and manage custom blocks.	default	/admin/structure/block/block-content	Edit More
Files Displays: Page, Page Machine name: files	Find and manage files.	default	/admin/content/files;/admin/content/files/usage/%	Edit More

2. Click on **Add new view** to begin creating a new view.
3. The first step is to provide the **View name** of **Articles**, which will serve as the administrative and (by default) displayed title.
4. Next, we modify the **VIEW SETTINGS**. We want to display **Content** of the type **Articles** and leave the **tagged with** empty. This will force the view to only show content of the **article** content type.
5. Choose to **Create a page**. The **Page title** and **Path** will be auto populated based on the view name and can be modified as desired. For now, leave the display and other settings at their default values.

VIEW BASIC INFORMATION

View name *
Articles Machine name: articles [Edit]

Description

VIEW SETTINGS

Show: Content ▾ of type: Article ▾ tagged with: sorted by: Newest first ▾

PAGE SETTINGS

Create a page

Page title
Articles

Path
articles

PAGE DISPLAY SETTINGS

Display format:
Unformatted list ▾ of: teasers ▾

6. Click on **Save and edit** to continue modifying your new view.
7. In the middle column, under the **Page settings** section we will change the **Menu item** settings. Click on **No menu** to change the default.

8. Select **Normal menu entry**. Provide a menu link title and optional description. Keep the **Parent** set to **<Main Navigation>**.

The screenshot shows the 'Page: Menu item entry' configuration dialog. It has a dark header bar with the title. Below it, there are two columns of settings:

- Type:** A radio button group with "Normal menu entry" selected.
- Menu link title:** A text input field containing "Articles".
- Description:** A text input field containing "View all articles". Below it is a note: "Shown when hovering over the menu link."
- Parent:** A dropdown menu showing "<Main navigation>". Below it is a note: "The maximum depth for a link and all its children is fixed. Some menu links may not be available as parents if selecting them would exceed this limit."
- Weight:** An input field containing "0". Below it is a note: "In the menu, the heavier links will sink and the lighter links will be positioned nearer the top."

At the bottom are two buttons: "Apply" (highlighted in blue) and "Cancel".

9. Click on **Apply** at the bottom of the form.
10. Once the view is saved you will now see the link in your Drupal site's main menu.

How it works...

The first step for creating a view involves selecting the type of data you will be displaying. This is referred to as the base table, which can be any type of entity or data specifically exposed to Views.



Nodes are labeled as Content in Views and you will find throughout Drupal this interchanged terminology.

When creating a Views page we are adding a menu path that can be accessed. It tells Drupal to invoke Views to render the page, which will load the view you create and render it.

There are display style and row plugins that format the data to be rendered. Our recipe used the **unformatted list** style to wrap each row in a simple `div` element. We could have changed this to a table for a formatted list. The row display controls how each row is output.

There's more...

Views has been one of the must-use modules since it first debuted, to the point that almost every Drupal 7 site used the module. In the following section we will dive further into Views.

Views in Drupal Core Initiative

Views has been a contributed module up until Drupal 8. In fact, it was one of the most used modules. Although the module is now part of Drupal core it still has many improvements that are needed and are being committed.

Some of these changes will be seen through minor Drupal releases, such as 8.1.x and 8.2.x, as development progresses and probably not through patch releases (8.0.10).

Views and displays

When working with Views, you will see some different terminology. One of the key items to grasp is what a display is. A view can contain multiple displays. Each display is of a certain type. Views comes with the following display types:

- ▶ **attachment:** This is a display that becomes attached to another display in the same view
- ▶ **block:** This allows you to place the view as a block
- ▶ **embed:** The display is meant to be embedded programmatically
- ▶ **Entity Reference:** This allows Views to provide results for an entity reference field
- ▶ **feed:** This display returns an XML based feed and can be attached to another display to render a feed icon
- ▶ **page:** This allows you to display the view from a specific route

Each display can have its own configuration, too. However, each display will share the same base table (content, files, etc.). This allows you to take the same data and represent it in different ways.

Format style plugins: style and row

Within Views there are two types of style plugins that represent how your data is displayed – style and row.

- ▶ The **style** plugin represents the overall format
- ▶ The **row** plugin represents each result row's format

For example, the `grid` style will output multiple `div` elements with specified classes to create a responsive grid. At the same time, the `table` style creates a tabular output with labels used as table headings.

Row plugins define how to render the row. The default content will render the entity as defined by its selected display mode. If you choose **Fields** you manually choose which fields to include in your view.

Each format style plugin has a corresponding Twig file that the theme layer uses. You can define new plugins in custom modules or use contributed modules to access different options.

Using the Embed display

Each of the available display types has a method to expose itself through the user interface, except for **Embed**. Often, contributed and custom modules use Views to render displays instead of manually writing queries and rendering the output. Drupal 8 provides a special display type to simplify this.

If we were to add an Embed display to the view created in the recipe, we could pass the following render array to output our view programmatically.

```
$view_render = [
  '#type' => 'view',
  '#name' => 'articles',
  '#display_id' => 'embed_1',
];
```

When rendered, the `#type` key tells Drupal this is a view element. We then point it to our new display `embed_1`. In actuality, the Embed display type has no special functionality, in fact it is a simplistic display plugin. The benefit is that it does not have additional operations conducted for the sake of performance.

See also

- ▶ VDC Initiative:
<https://www.drupal.org/community-initiatives/drupal-core/vdc>
- ▶ *Chapter 7, Plug and Play with Plugins*, to learn more about plugins

Editing the default admin interfaces

With the addition of Views in Drupal core, many of the administrative interfaces are powered by Views. This allows customization of default admin interfaces to enhance site management and content authoring experiences.



In Drupal 7 and 6 there was the administrative Views module, which provided a way to override administrative pages with Views. This module is no longer required, as the functionality comes with Drupal core out of the box!

In this recipe we will modify the default content overview form that is used to find and edit content. We will add the ability to filter content by the user who authored it.

How to do it...

1. Visit **Structure** and then **Views**. This will bring you to the administrative overview of all existing views.
2. From the **Enabled** section, select the **Edit** option for the **Content** view. This is the view displayed on /admin/content when managing content.
3. In order to filter by the content author, we must add a **FILTER CRITERIA** to our view, which we will expose the following for users to modify:

FILTER CRITERIA

Add ▾

- Content: Published status or admin user
- Content: Publishing status (grouped)
- Content: Type (exposed)
- Content: Title (exposed)
- Content: Translation language (exposed)

4. Click on **Add** to add a new filter. In the search text box type **Authored by** to search the available options. Select **Content: Authored by** and click **Apply (all displays)**:

Add filter criteria

For: All displays Search: authored by Type: - All -

Content: Authored by
The user authoring the content. If you need more fields than the uid add the content: author relationship

Content revision: Authored by
The username of the content author.

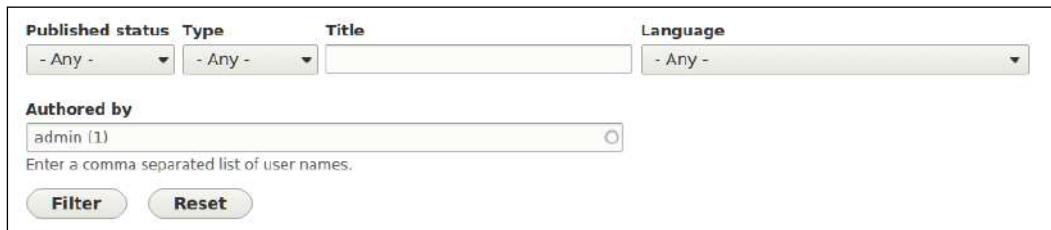
Selected: Content: Authored by

Apply (all displays) Cancel

5. Check **Expose this filter to visitors**, to allow them to change it via checkbox. This will allow users to modify the data for the filter.
6. You may modify the **Label** and add a **Description** to improve the usability of the filter option for your use case.

Displaying Content through Views

7. Click on **Apply (all displays)** once more to finish configuring the filter. It will now show up in the list as filter criteria active. You will also see the new filter in the preview below the form.
8. Click on **Save** to commit all changes to the view.
9. View /admin/content and you will have your filter. Content editors will be able to search for content authored by a user through autocompleted username searches:



The screenshot shows a Drupal administrative interface for filtering content. At the top, there are dropdown menus for 'Published status' (set to '- Any -'), 'Type' (set to '- Any -'), 'Title' (empty), and 'Language' (set to '- Any -'). Below these is a section titled 'Authored by' containing a dropdown menu with 'admin (1)' selected. A placeholder text 'Enter a comma separated list of user names.' is visible below the dropdown. At the bottom of the form are two buttons: 'Filter' and 'Reset'.

How it works...

When a view is created that has a path matching an existing route, it will override it and present itself. That is how the /admin/content and other administrative pages are able to be powered by Views.



If you were to disable the Views module you can still manage content and users. The default forms are tables that do not provide filters or other extra features.



Drupal uses the overridden route and uses Views to render the page. From that point on the page is handled like any other Views page would be rendered.

There's more...

We will dive into additional features available through Views that can enhance the way you use Views and present them on your Drupal site.

Exposed versus non-exposed

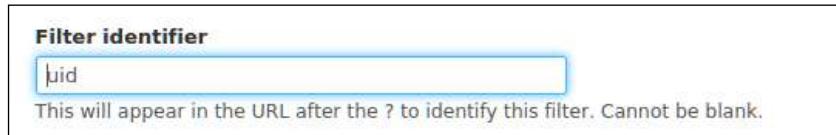
Filters allow you to narrow the scope of the data displayed in a view. Filters can either be exposed or not; by default a filter is not exposed. An example would be using the **Content: Publishing status** set to **Yes (published)** to ensure a view always contains published content. This is an item you would configure for displaying content to site visitors. However, if it were for an administrative display, you may want to expose that filter. This way content editors have the ability to view, easily, what content has not been published yet or has been unpublished.

All filter and sort criteria can be marked as exposed.

Filter identifiers

Exposed filters work by parsing query parameters in the URL. For instance, on the content management form, changing the Type filter will add type=Article amongst others to the current URL.

With this recipe the author filter would show up as **uid** in the URL. Exposed filters have a **Filter identifier** option that can change the URL component.



This could be changed to author or some other value to enhance the user experience behind the URL, or mask the Drupal-ness of it.

Overriding routes with Views

Views is able to replace administrative pages with enhanced versions due to the way the route and module system works in Drupal. Modules are executed in order of the module's weight or alphabetical order if weights are the same. Naturally, in the English alphabet, the letter V comes towards the end of the alphabet. That means any route that Views provides will be added towards the end of the route discovery cycle.

If a view is created and it provides a route path, it will override any that exist on that path. There is not a collision checking mechanism (and there was not in Views before merging into Drupal core) that prevents this.

This allows you to easily customize most existing routes. But, beware that you could easily have conflicting routes and Views will normally override the other.

Creating a block from a View

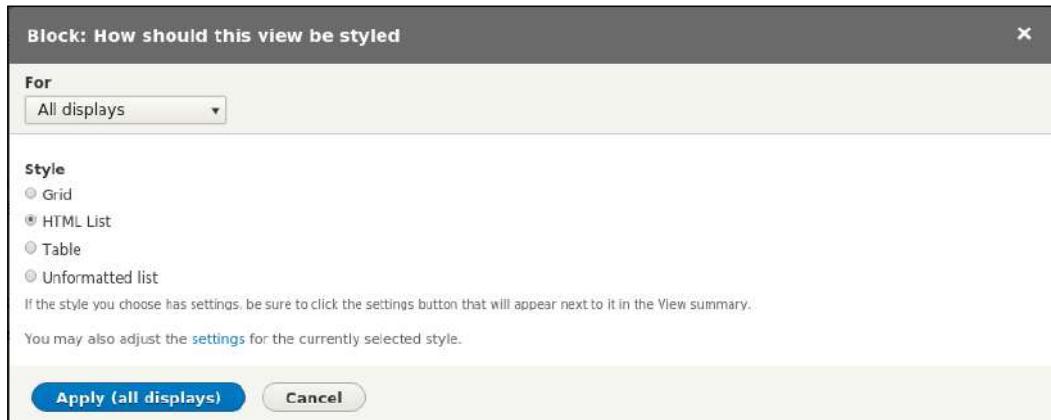
Previous recipes have shown how to create and manipulate a page created by a view. Views provides different display types that can be created, such as a block. In this recipe we will create a block powered by Views. The Views block will list all Tag taxonomy terms that have been added to the Article content type.

Getting ready

This recipe assumes you have installed the standard installation profile and have the default node content types available for use.

How to do it...

1. Visit **Structure** and then **Views**. This will bring you to the administrative overview of all the views that have been created.
2. Click on **Add new** view to begin creating a new view.
3. The first step is to provide the View name of Tags, which will serve as the administrative and (by default) displayed title.
4. Next, we modify the **View** settings. We want to display Taxonomy terms of the type Tags. This will make the view default to only displaying taxonomy terms created under the Tags vocabulary
5. Check the **Create a block** checkbox.
6. Choose the **HTML List** option from the **Display** format choices. Leave the **row style** as **Fields**.



7. We want to display all of the available tags. To change this, click the current pager style link. Pick the **Display all items** radio and click **Apply (all displays)**. On the next model, click **Save** to keep the offset at 0.
8. Next we will sort the view by tag name instead of order of creation. Click **Add** on the **Sort criteria** section. Add **Taxonomy term: Name** and click **Apply (all displays)** to use the default **sort by ascending**.

The screenshot shows the 'Displays' configuration page for a 'Block' display. The left sidebar includes sections for 'TITLE' (Title: Tags), 'FORMAT' (Format: HTML List | Settings, Show: Fields | Settings), 'FIELDS' (Taxonomy term: Name), 'FILTER CRITERIA' (Add), and 'SORT CRITERIA' (Add). The right sidebar includes sections for 'BLOCK SETTINGS' (Block name: None, Block category: Lists (Views), Allow settings: Items per page, Access: Permission | View published content), 'HEADER' (Add), 'FOOTER' (Add), 'NO RESULTS BEHAVIOR' (Add), and 'PAGER' (Use pager: Display all items | All items, More link: No, Link display: None).

9. Press **Save** to save the view.
10. Visit **Structure** and **Block layout** to place the block on your Drupal site. Press **Place block** for the **Sidebar** region in the **Bartik** theme.
11. Filter the list by typing your view's name. Press **Place block** to add your view's block to the block layout.
12. Finally click on **block** to commit your changes!

How it works...

In the Drupal 8 plugin system there is a concept called **Derivatives**. Plugins are small pieces of swappable functionality within Drupal 8. Plugins and plugin development are covered in *Chapter 7, Plug and Play with Plugins*. A derivative allows a module to present multiple variations of a plugin dynamically. In the case of Views, it allows the module to provide variations of a `ViewsBlock` plugin for each view that has a block display. Views implements the `\Drupal\views\Plugin\Block\ViewsBlock\ViewsBlock` class, providing the base for the dynamic availability of these blocks. Each derived block is an instance of this class.

When Drupal initiates the block, Views passes the proper configuration required. The view is then executed and the display is rendered whenever the block is displayed.

There's more...

We will explore some of the other ways in which Views interacts with blocks.

Exposed forms as blocks

Pages and feeds have the ability to provide blocks, however not for the actual content displayed. If your view utilizes exposed filters you have the option to place the exposed form in a block. With this option enabled you may place the block anywhere on the page, even pages not for your view!



To enable the exposed filters as a block, you must first expand the **Advanced** section on the right side of the Views edit form. Click on the **Exposed form in block** option from the **Advanced** section. In the options modal that opens, select the **Yes** radio button and click **Apply**. You then have the ability to place the block from the **Block layout** form.

An example for using an exposed form in a block is for a search result view. You would add an exposed filter for keywords that control the search results. With the exposed filters in a block you can easily place it in your site's header. When an exposed filters block is submitted, it will direct users to your view's display.

See also

- ▶ Chapter 7, *Plug and Play with Plugins*, to learn more about derivatives

Utilizing dynamic arguments

Views can be configured to accept **contextual filters**. Contextual filters allow you to provide a dynamic argument that modifies the view's output. The value is expected to be passed from the URL; however, if it is not present there are ways to provide a default value.

In this recipe we will create a new page called `My Content`, which will display a user's authored content on the route `/user/%/content`.

How to do it...

1. Visit **Structure** and then **Views**. This will bring you to the administrative overview of all the views created. Click on **Add new view** to begin creating a new view.
2. Set the **View name** to **My Content**.
3. Next, we modify the **View** settings. We want to display **Content of the type All** and leave the **Tagged** with empty. This will allow all content to be displayed.
4. Choose to **Create** a page. Keep the page title the same. We need to change the path to be `/user/%/content`. Click **Save and edit** to move to the next screen and add the contextual filter.



When building a views page, adding a percentage sign to the path identifies a route variable.

5. Toggle the Advanced portion of the form on the right hand side of the page. Click on **Add in the Contextual filters** section.
6. Select **Content: Authored by** and then click **Apply (all displays)**.
7. Change the default value When the filter is not in the URL to **Display "Access Denied"** to prevent all content from being displayed with a bad route value.

▼ WHEN THE FILTER VALUE IS NOT IN THE URL

Default actions

- Display all results for the specified field
- Provide default value
- Show "Page not found"
- Display a summary
- Display contents of "No results found"
- Display "Access Denied"

► EXCEPTIONS

8. Click **Apply (all displays)** and save the form.
9. Visit `/user/1/content` and you will see content created by the root admin!

How it works...

Contextual filters mimic the route variables found in the Drupal routing system. Variables are represented by percentage signs as placeholders in the view's path. Views will match up each placeholder with contextual filters by order of their placement. This allows you to have multiple contextual filters; you just need to ensure they are ordered properly.

Views is aware of how to handle the placeholder because the type of data is selected when you add the filter. Once the contextual filter is added there are extra options available for handling the route variable.

There's more...

We will explore extra options available when using contextual filters.

Previewing with contextual filters

You are still able to preview a view from the edit form. You simply add the contextual filter values in to the text form concatenated by a forward slash (/). In this recipe you could replace visiting /user/1/content with simply inputting 1 into the preview form and updating the preview.

Displaying as a tab on the user page

Even though the view created in the recipe follows a route under /user, it will not show up as a local task tab until it has a menu entry defined. From the **Page settings** section you will need to change **No menu** from the **Menu** option. Clicking on that link will open the menu link settings dialog.

Select **Menu tab** and provide a **Menu link title**, such as **My Content**. Click on **Apply** and save your view.

Altering the page title

With contextual filters you have the ability to manipulate the current page's title. When adding or editing a contextual filter you can modify the page title. From the When the filter value is present in the URL or a default is provided section, you may check the **Override title** option.

This text box allows you to enter in a new title that will be displayed. Additionally, you can use the information passed from the route context using the format of %# where the # is the argument order.

Validation

Contextual filters can have validation attached. Without specifying extra validation, Views will take the expected argument and try to make it *just work*. You can add validation to help limit this scope and filter out invalid route variables.

You can enable validation by checking **Specify validation criteria** from the When the filter value is present in the URL or a default is provided section. The default is set to – Basic Validation – which allows you to specify how the view should react if the data is invalid; based on our recipe, if the user was not found.

The list of **Validator** options is not filtered by the contextual filter item you selected, so some may not apply. For our recipe one might want **User ID** and select the **Validate user has access to the User**. This validator would make sure the current user is able to view the route's user's profile. Additionally, it can be restricted further based on role.

The screenshot shows a configuration dialog for validation criteria. It includes sections for 'Validator' (set to 'User ID'), 'Access operation to check' (set to 'View'), 'Multiple arguments' (set to 'Single ID'), and 'Action to take if filter value does not validate' (set to 'Display "Access Denied"').

Specify validation criteria

Validator
User ID

Validate user has access to the *User*

Access operation to check

View

Edit

Delete

Multiple arguments

Single ID

One or more IDs separated by , or +

Restrict user based on role

Action to take if filter value does not validate

Display "Access Denied"

This gives you more granular control over how the view operates when using contextual filters for route arguments.

Multiple and exclusion

You may also configure the contextual filter to allow AND or OR operations along with exclusion. These options are under the **More** section when adding or editing a contextual filter.

The Allow multiple values option can be checked to enable AND or OR operations. If the contextual filter argument contains a series of values concatenated by plus (+) signs it acts as an OR operation. If the values are concatenated by commas (,) it acts as an AND operation.

When the **Exclude** option is checked the value will be excluded from the results rather than the view being limited by it.

Adding a relationship in a View

As stated at the beginning of the chapter, Views is a visual query builder. When you first create a view, a base table is specified to pull data from. Views automatically knows how to join tables for field data, such as body text or custom attached fields.

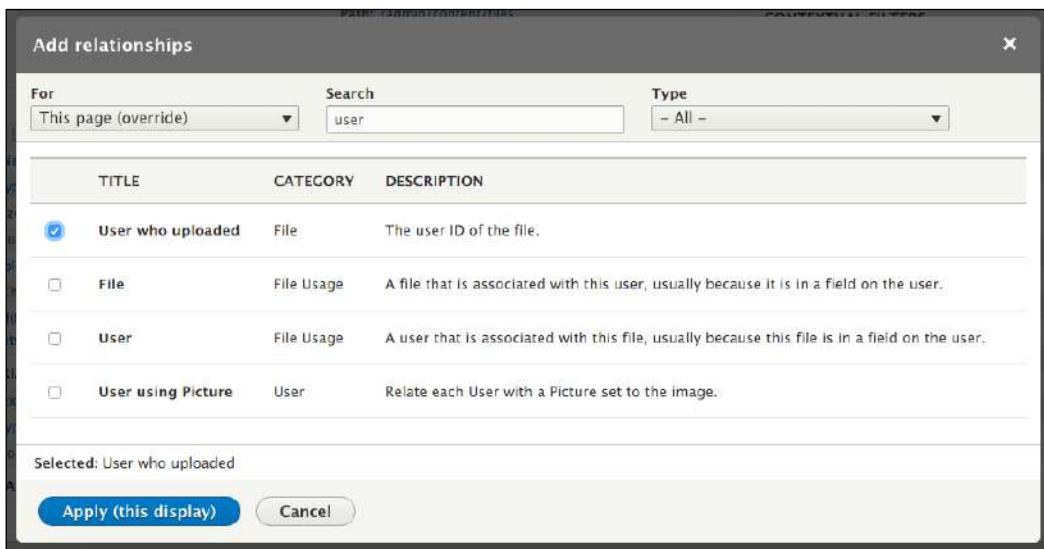
Displaying Content through Views

When using an entity reference field, you have the ability to display the value as the raw identifier, the referenced entity's label, or the entire rendered entity. However, if you add a Relationship based on a reference field you will have access to display any of that entity's available fields.

In this recipe, we will update the Files view, used for administering files, to display the username of the user who uploaded the file.

How to do it...

1. Visit **Structure** and then **Views**. This will bring you to the administrative overview of all the views that have been created
2. Find the **Files** view and click **Edit**.
3. Click on **Advanced** to expand the section and then click **Add** next to **Relationships**.
4. Search for **user**. Select the **User** who uploaded **relationship option** and click **Apply (this display)**.



5. Next we will be presented with a configure form for the relationship. Click **Apply (this display)** to use the defaults.
6. Add a new field by clicking **Add** in the **Fields** section.
7. Search for **name** and select the **Name** field and click **Apply (this display)**.

8. This view uses aggregation, which presents a new configuration form when first adding a field. Click **Apply and continue** to use the defaults.



We will discuss Views and aggregation in the *There's more...* section.

9. We will use the default field settings that will provide the label **Name** and format it as the username and link to the user's profile. Click **Apply (all displays)**.

The screenshot shows the 'Configure field: User: Name' dialog box. The 'For' dropdown is set to 'All displays (except overridden)'. The 'Relationship' dropdown is set to 'User who uploaded'. Under 'Label', there is a checked checkbox for 'Create a label' and a text input field with 'Name'. There is also a checked checkbox for 'Place a colon after the label'. Under 'Formatter', the dropdown is set to 'User name'. A checked checkbox for 'Link to the user' is present. At the bottom, there are 'Apply (all displays)', 'Cancel', and 'Remove' buttons.

10. Click on **Save** to finish editing the view and commit your changes.

How it works...

Drupal stores data in a normalized format. Database normalization, in short, involves the organization of data in specific related tables. Each entity type has its own database table and all fields have their own database table. When you create a view and specify what kind of data will be shown, you are specifying a base table in the database that Views will query. Views will automatically associate fields that belong to the entity and the relationship to those tables for you.

When an entity has an Entity reference field you have the ability to add a relationship to the referenced entity type's table. This is an explicit definition, whereas fields are implicit. When the relationship is explicitly defined all of the referenced entity type's fields come into scope. The fields on the referenced entity type can then be displayed, filtered, and sorted by.

There's more...

Using relationships in Views allows you to create some powerful displays. We will discuss aggregation and additional information about relationships.

Relationships provided by entity reference fields

Views uses a series of hooks to retrieve data that it uses to represent ways to interact with the database. One of these is the `hook_field_views_data` hook, which processes a field storage configuration entity and registers its data with Views. The Views module implements this on behalf of Drupal core to add relationships, and reverse relationships, for Entity reference fields.

Since Entity reference fields have set schema information, Views can dynamically generate these relationships by knowing the field's table name, destination entity's table name, and the destination entity's identifier column.

Relationships provided through custom code

There are times where you would need to define a relation on your own with custom code. Typically, when working with custom data in Drupal, you would more than likely create a new entity type, covered in *Chapter 9, Configuration Management – Deploying in Drupal 8*. This is not always the case, however, and you may just need a simple method of storing data. An example can be found in the Database Logging module. The Database Logging module defines schema for a database table and then uses `hook_views_data` to expose its database table to Views.

The `dblog_schema` hook implementation returns a `uid` column on the `watchdog` database table created by the module. That column is then exposed to Views with the following definition:

```
$data['watchdog']['uid'] = array(
  'title' => t('UID'),
  'help' => t('The user ID of the user on which the log entry
    was written...'),
  'field' => array(
    'id' => 'numeric',
  ),
  'filter' => array(
    'id' => 'numeric',
  ),
  'argument' => array(
    'id' => 'numeric',
  ),
```

```
) ,  
  'relationship' => array(  
    'title' => t('User'),  
    'help' => t('The user on which the log entry was written.'),  
    'base' => 'users',  
    'base field' => 'uid',  
    'id' => 'standard',  
  ),  
);
```

This array tells Views that the `watchdog` table has a column named `uid`. It is numeric in nature for its display, filtering capabilities and sorting capabilities. The `relationship` key is an array of information that instructs Views how to use this to provide a relationship (LEFT JOIN) on the `users` table. The `User` entity uses the `users` table and has the primary key of `uid`.

Using Aggregation and views.

There is a view setting under the **Advanced** section that allows you to enable aggregation. This feature allows you to enable the usage of SQL aggregate functions, such as MIN, MAX, SUM, AVG, and COUNT. In this recipe, the Files view uses aggregation to SUM the usage counts of each file in the Drupal site.

Aggregation settings are set for each field and when enabled have their own link to configure the settings.

The screenshot shows a 'FIELDS' configuration section. At the top right are 'Add' and '▼' buttons. Below is a list of fields with their aggregation settings:

- File: File ID (Fid) [hidden] | [Aggregation settings](#)
- File: Filename (Name) | [Aggregation settings](#)
- File: File MIME type (Mime type) | [Aggregation settings](#)
- File: File size (Size) | [Aggregation settings](#)
- File: Status (Status) | [Aggregation settings](#)
- File: Created (Upload date) | [Aggregation settings](#)
- File: Changed (Changed date) | [Aggregation settings](#)
- (File usage) SUM(File Usage: Use count) (Used in) | [Aggregation settings](#)

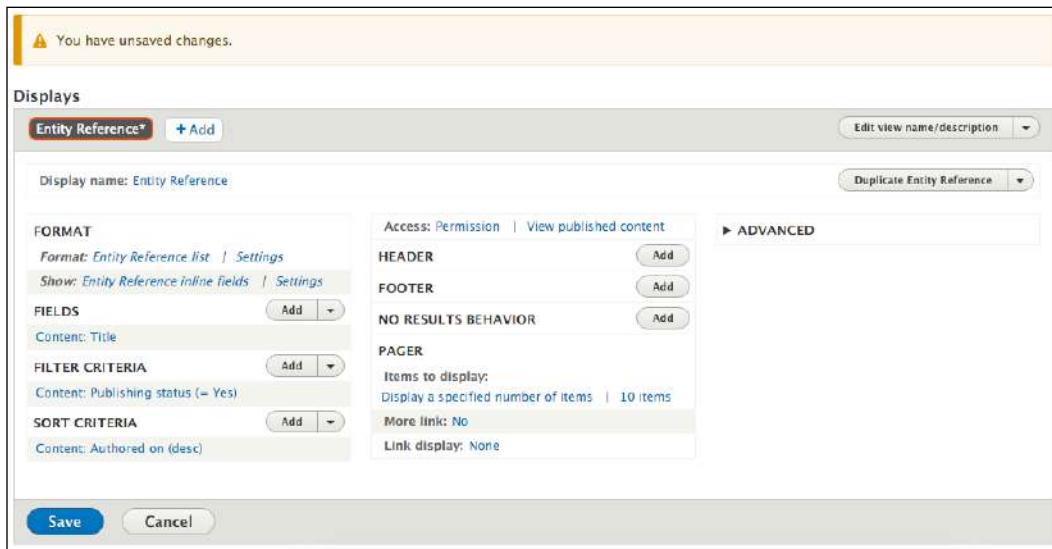
Providing an Entity Reference result View

The Entity reference field, covered in *Chapter 2, The Content Authoring Experience*, can utilize a custom view for providing the available field values. The default entity reference field will display all available entities of the type it is allowed to reference. The only available filter is based on the entity bundle, such as only returning Article nodes. Using an entity reference view you can provide more filters, such as only content that user has authored.

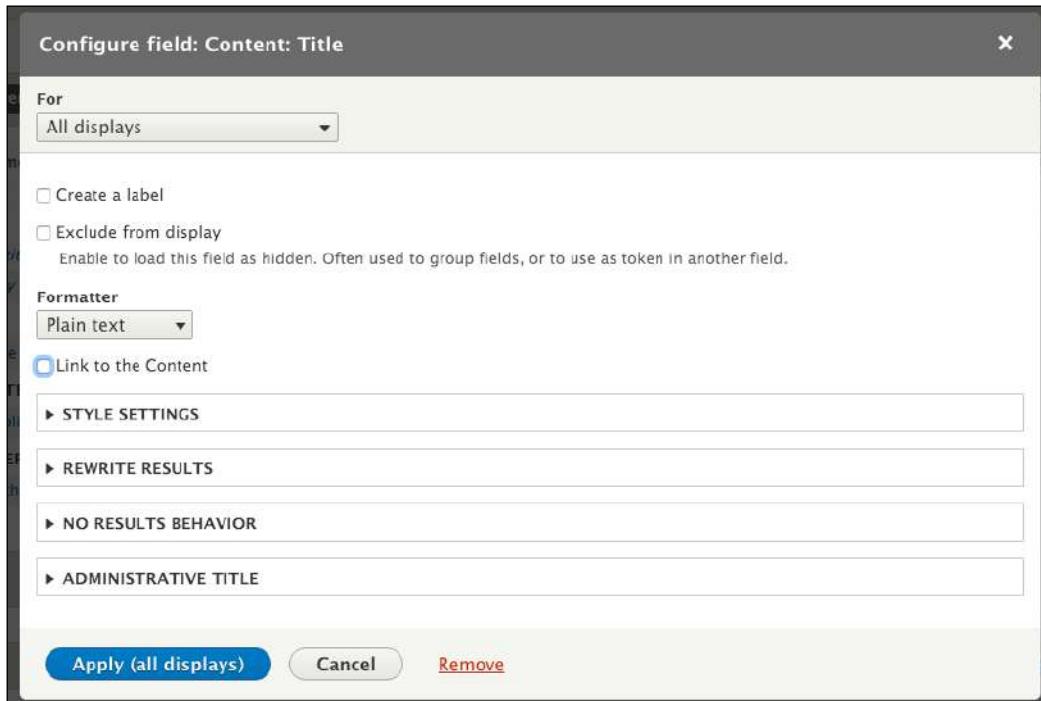
In this recipe we will create an entity reference view that filters content by the author. We will add the field to the user account form, allowing users to select their favorite contributed content.

How to do it...

1. Visit **Structure** and then **Views**. This will bring you to the administrative overview of all the views that have been created. Click on **Add new view** to begin creating a new view.
2. Set the **View name** to **My Content Reference View**. Modify the **View** settings. We want to display **Content of the type All** and leave the Tagged with empty.
3. Do not choose to create a page or block. Click **Save and edit** to continue working on your view.
4. Click on the **Add** button to create a new display. Select the **Entity Reference** option to create the display.



5. The **Format** will be automatically set to **Entity Reference List**, which utilizes fields. Click on **Settings** next to it to modify the style format.
6. For **Search Fields**, check the **Content:Title** option then click **Apply**. This is what the field will autocomplete search on.
7. You will need to modify the **Content: Title** field to stop it from wrapping the result as a link. Click on the field label and uncheck **Link to the Content**. Click **Apply (all displays)** to save.



8. Click on **Save** to save the view.
9. Go to **Configuration** and then **Account settings** to be able to **Manage fields** on user accounts.
10. Add a new **Entity Reference** field that references **Content**, call it **Highlighted contributions**, and allow it to have unlimited values. Click the **Save field settings** button.

11. Change the **Reference type** method to use **View: Filter by an entity reference view** and select the view we have just created:

The screenshot shows the 'REFERENCE TYPE' configuration screen. It includes fields for 'Reference method' (set to 'Views: Filter by an entity reference view'), 'View used to select the entities' (set to 'my_content_reference_view - Entity Reference'), and a note about choosing a view with a display type of 'Entity Reference'. There is also a 'View arguments' section with a text input field for comma-separated arguments.

How it works...

The entity reference field definition provides selection plugins. Views provides an entity reference selection plugin. This allows entity reference to feed data into a view to receive available results.

The display type for Views requires you to select which fields will be used to search against when using the autocomplete widget. If not using the autocomplete widget and using the select list or checkboxes and radio buttons, then it will return the view's entire results.

There's more...

View arguments

Entity reference view displays can accept contextual filter arguments. These are not dynamic, but can be passed manually through the field's settings. The **View arguments** field allows you to add a comma separated list of arguments that are passed to the view. The order should match the order of the contextual filters as configured.

In this recipe we could have added a **Content: type** contextual filter that fell back to **Display all results** if the argument was missing. This allows the view to be reused in multiple references. Perhaps there is one view that should limit the available references to all Articles created by the current user. You would then add **Article** to the text field and pass the argument to the view.

See also

- ▶ Chapter 7, *Plug and Play with Plugins*, to learn more about plugins

4

Extending Drupal

This chapter dives into extending Drupal using a custom module:

- ▶ Creating a module
- ▶ Defining a custom page
- ▶ Defining permissions
- ▶ Providing the configuration on installation or update
- ▶ Using Features 2.x

Introduction

A feature of Drupal that makes it desirable is the ability to customize it through modules. Whether custom or contributed, modules extend the functionalities and capabilities of Drupal. Modules can be used to not only extend Drupal, but also to create a way to provide configuration and reusable features.

This chapter will discuss how to create a module and allow Drupal to discover it, allowing it to be installed from the extend page. Permissions, custom pages, and default configurations all come from modules. We will explore how to provide these through a custom module.

In addition to creating a module, we will discuss the Features module that provides a set of tools for exporting the configuration and generating a module.

Creating a module

The first step to extend Drupal is to create a custom module. Although the task sounds daunting, it can be accomplished in a few simple steps. Modules can provide functionalities and customizations to functionalities provided by other modules, or they can be used as a way to contain the configuration and a site's state.

In this recipe, we will create a module by defining an `info` file, a file containing information that Drupal uses to discover extensions, and enabling the module.

How to do it...

1. Create a folder named `mymodule` in the `modules` folder in the base directory of your Drupal site. This will be your module's directory.

2. Create a `mymodule.info.yml` file in your module's directory. This contains metadata that identifies the module to Drupal.

3. Add a line to the `name` key to provide a name for the module:

```
name: My Module!
```

4. We need to provide the `type` key to define the type of extension. We provide the value `module`:

```
type: module
```

5. The `description` key allows you to provide extra information about your module, which will be displayed on the module's list page:

```
description: This is an example module from the Drupal 8 Cookbook!
```

6. All modules need to define the `core` key in order to specify a major release compatibility:

```
core: 8.x
```

7. Save the `mymodule.info.yml` file, which resembles the following code:

```
name: My Module!
```

```
type: module
```

```
description: This is an example module from the Drupal 8 Cookbook!
```

```
core: 8.x
```

8. Log in to your Drupal site and visit **Extend** from the administrative toolbar.

9. Search for **My Module** to filter the list of options.

10. Check the checkbox and click on **Install** to enable your module.

The screenshot shows the 'Administer' section of the Drupal interface, specifically the 'Contributed modules' page. At the top, there's a search bar with the placeholder 'mymodule'. Below the search bar, there's a note about maintaining the site by regularly reviewing updates and running the update script. A list of modules is shown, with one item highlighted: 'My Module!' which has a checked checkbox next to it. The 'OTHER' category is selected.

How it works...

Drupal utilizes `info.yml` files to define extensions. Drupal has a discovery system that locates these files and parses them to discover modules. The `info_parser` service, provided by the `\Drupal\Core\Extension\InfoParser` class, reads the `info.yml` file. The parser guarantees that the required type, core, and name keys are present.

When a module is installed, it is added to the `core.extension` configuration object, which contains a list of installed modules and themes. The collection of modules in the `core.extension` module array will be installed and will have PHP namespaces resolved, services loaded, and hooks registered.

When Drupal prepares to execute a hook or register services, it will iterate across the values in the `module` key in `core.extension`.

There's more...

There are more details that we can explore about module `info` files.

Module namespaces

Drupal 8 uses the PSR-4 standard developed by the PHP Framework Interoperability Group. The PSR-4 standard is for package-based PHP namespace autoloading. It defines a standard to understand how to automatically include classes based on a namespace and class name. Drupal modules have their own namespaces under the Drupal root namespace.

Using the module from the recipe, our PHP namespace will be `Drupal\mymodule`, which represents the `modules/mymodule/src` folder.

With PSR-4, files need to contain only one class, interface, or trait. These files need to have the same filename as the containing class, interface, or trait name. This allows a class loader to resolve a namespace as a directory path and know the class's filename. The file can then be automatically loaded when it is used in a file.

Module discovery locations

Drupal supports multiple module discovery locations. Modules can be placed in the following directories and discovered:

- ▶ /profiles/CURRENT PROFILE/modules
- ▶ /sites/all/modules
- ▶ /modules
- ▶ /sites/default/modules
- ▶ /sites/example.com/modules

The \Drupal\Core\Extension\ExtensionDiscovery class handles the discovery of extensions by types. It will iteratively scan each location and discover modules that are available. The discovery order is important. If the same module is placed in /modules but also in the sites/default/modules directory, the latter will take precedence.

Defining a package group

Modules can define a package key to group modules on the module list page:

The screenshot shows a list of modules categorized under a package group named 'MULTILINGUAL'. The package group is indicated by a downward-pointing triangle icon next to the category name. The modules listed are: Configuration Translation, Content Translation, Interface Translation, and Language. Each module has a corresponding description to its right. The entire list is contained within a light gray box.

MULTILINGUAL	Description
Configuration Translation	Provides a translation interface for configuration.
Content Translation	Allows users to translate content entities.
Interface Translation	Translates the built-in user interface.
Language	Allows users to configure languages and apply them to content.

Projects that include multiple submodules, such as Drupal commerce, specify packages to normalize the modules' list form. Contributed modules for the Drupal commerce project utilize a package name, Commerce (contrib), to group on the module list page.

Module dependencies

Modules can define dependencies to ensure that those modules are enabled before your module can be enabled.

Here is the `info.yml` for the Responsive Image module:

```
name: Responsive Image
type: module
description: 'Provides an image formatter and breakpoint mappings to output responsive images using the HTML5 picture tag.'
package: Core
version: VERSION
core: 8.x
dependencies:
  - breakpoint
  - image
```

The `dependencies` key specifies that the Breakpoint and Image modules need to be enabled first before the Responsive Image module can be enabled. When enabling a module that requires dependencies that are disabled, the installation form will provide a prompt asking you if you would like to install the dependencies as well. If a dependency module is missing, the module cannot be installed. The dependency will show a status of (`missing`).

A module that is a dependency of another module will state the information in its description, along with the other module's status. For example, the Breakpoint module will show that the Responsive Image module requires it as a dependency and is disabled:

Breakpoint Manage breakpoints and breakpoint groups for responsive designs.
Machine name: breakpoint
Version: 8.0.0-rc4
Required by: Responsive Image (disabled), Toolbar
[Help](#)

Responsive Image Provides an image formatter and breakpoint mappings to output responsive images using the HTML5 picture tag.
Machine name: responsive_image
Version: 8.0.0-rc4
Requires: Breakpoint, Image, File, Field

Specifying the module's version

There is a `version` key that defines the current module's version. Projects on Drupal.org do not specify this directly, as the packager adds it when a release is created. However, this key can be important for private modules to track the release information.

Versions are expected to be single strings, such as 1.0-alpha1, 2.0.1. You can also pass VERSION, which will resolve to the current version of Drupal core.



Drupal.org does not currently support semantic versioning for contributed projects. There is an ongoing policy discussion in the issue queue, which can be found at <https://www.drupal.org/node/1612910>.

See also...

- ▶ Refer to the PSR-4: Autoloader specification at <http://www.php-fig.org/psr/psr-4/>

Defining a custom page

In Drupal, there are routes that represent URL paths that Drupal interprets to return content. Modules have the ability to define routes and methods that return data to be rendered and then displayed to the end user.

In this recipe, we will define a controller that provides an output and a route. The route provides a URL path that Drupal will associate with our controller to display the output.

Getting ready

Create a new module like the one in the first recipe. We will refer to the module as `mymodule` throughout the recipe. Use your module's name as appropriate.

How to do it...

1. Firstly, we'll set up the controller. Create a `src` folder in your module's base directory and another folder named `Controller` inside it.
 2. Create `MyPageController.php` that will hold the route's controller class.



3. The PSR-4 standard states that filenames match the class names they hold, so we will create a `MyPageController` class:

```
<?php
/**
 * @file
 * Contains \Drupal\mymodule\Controller\MyPageController class.
 */

namespace Drupal\mymodule\Controller;

use Drupal\Core\Controller\ControllerBase;

/**
 * Returns responses for My Module module.
 */
class MyPageController extends ControllerBase {
```

This creates the `MyPageController` class, which extends the `\Drupal\Core\Controller\ControllerBase` class. This base class provides a handful of utilities for interacting with the container.

The `Drupal\\mymodule\\Controller` namespace allows Drupal to automatically load the file from `/modules/mymodule/src/Controller`.

4. Next, we will create a method that returns a string of text in our class:

```
/**  
 * Returns markup for our custom page.  
 */  
public function customPage() {  
    return [  
        '#markup' => t('Welcome to my custom page!'),  
    ];  
}  
}
```

The `customPage` method returns a render array that the Drupal theming layer can parse. The `#markup` key denotes a value that does not have any additional rendering or theming processes.

5. Create a `mymodule.routing.yml` in the base directory of your module so that a route can be added to this controller and method.

6. The first step is to define the route's internal name for the route to be referenced by:

```
mymodule.mypage:
```

7. Give the route a path (`mypage`):

```
mymodule.mypage:
```

```
  path: /mypage
```

8. The `defaults` key allows us to provide the controller through a fully qualified class name, the method to use, and the page's title:

```
mymodule.mypage:
```

```
  path: /mypage
```

```
  defaults:
```

```
    _controller: '\Drupal\mymodule\Controller\'
```

```
MyPageController::customPage'
```

```
    _title: 'My custom page'
```

You need to provide the initial \ when providing the fully qualified class name.

9. Lastly, define a `requirements` key to set the access callback:

```
mymodule.mypage:
```

```
  path: /mypage
```

```
  defaults:
```

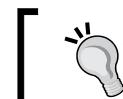
```
    _controller: '\Drupal\mymodule\Controller\'
```

```
MyPageController::customPage'
```

```
    _title: 'My custom page'
```

```
  requirements:
```

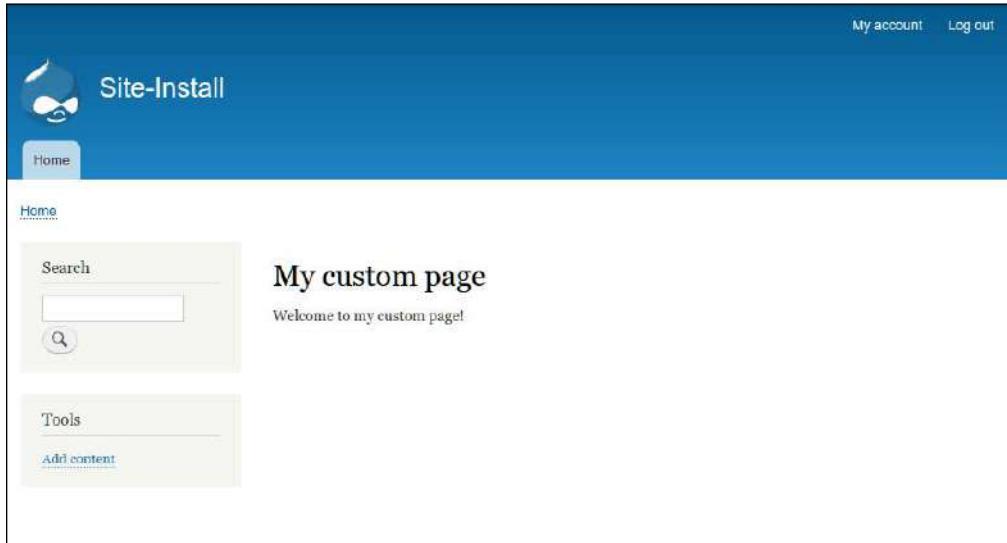
```
    _permission: 'access content'
```



Defining `_access` to TRUE for the requirements means that access is always granted.

10. Visit **Configuration** and then **Development** to rebuild Drupal's caches.

11. Visit /mypage on your Drupal site and view your custom page:



How it works...

Drupal uses routes, which define a path, that return content. Each route has a method in a controller class that generates the content, in the form of a render array, to be delivered to the user. When a request comes to Drupal, the system makes an attempt to match the path to known routes. If the route is found, the route's definition is used to deliver the page. If the route cannot be found, the 404 page is displayed.

The HTTP kernel takes the request and loads the route. It will invoke the defined controller method or procedural function. The result of the invoked method or function is then handed to the presentation layer of Drupal to be rendered into the content that can be delivered to the user.

Drupal 8 builds on top of the Symfony HTTP kernel to provide the underlying functionality of its route system. It has added the ability to provide access requirements, casting placeholders into loaded objects, and partial page responses.

There's more...

Routes have extra capabilities that can be configured; we will explore those in the next section.

Parameters in routes

Routes have the ability to accept dynamic arguments that can be passed to the route controller's method. Placeholder elements can be defined in the route using curly brackets in the URL that denote dynamic values.

An example of a route might look like the following code:

```
mymodule.cats:  
  path: '/cat/{name}'  
  defaults:  
    _controller: '\Drupal\mymodule\Controller\MyPageController::cats'  
  requirements:  
    _permission: 'access content'
```

This route specifies the /cat/{name} path. The {name} placeholder will accept dynamic values and pass them to the controller's method:

```
class MyPageController {  
  // ...  
  public function cats($name) {  
    return [  
      '#markup' => t('My cat's name is: @name', [  
        '@name' => $name,  
      ]),  
    ];  
  }  
}
```

This method accepts the name variable from the route and substitutes it into the render array to display it as text.

Drupal's routing system provides a method of upcasting a variable into a loaded object. There are a set of parameter converter classes under the \Drupal\Core\ParamConverter namespace. The EntityConverter class will read options defined in the route and replace a placeholder value with a loaded entity object.

If we have an entity type called **cat**, we can turn the name placeholder into a method to be provided the loaded the cat object in our controller's method:

```
mymodule.cats:  
  path: '/cat/{name}'  
  defaults:
```

```

_controller: '\Drupal\mymodule\Controller\MyPageController::cats'
requirements:
  _permission: 'access content'
options:
parameters:
  name:
    type: entity:cat

```

 This is not required for entities as the defined entity route handler can automatically generate this. Entities are covered in *Chapter 10, The Entity API*.

Validating parameters in routes

Drupal provides regular expression validation against route parameters. If the parameter fails the regular expression validation, a 404 page will be returned. Using the recipe's example route, we can add the validation to ensure that only alphabetical characters are used in the route parameter:

```

mymodule.cats:
  path: '/cat/{name}'
  defaults:
    _controller: '\Drupal\mymodule\Controller\MyPageController::cats'
  requirements:
    _permission: 'access content'
    name: '[a-zA-Z]+'

```

Under the `requirements` key, you can add a new value that matches the name of the placeholder. You then set it to have the value of the regular expression you would like to use.

Route requirements

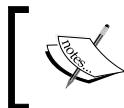
Routes can define different access requirements through the `requirements` key. Multiple validators can be added. However, there must be one that provides a true result or else the route will return **403, access denied**. This is true if the route defines no requirement validators.

Route requirement validators are defined by implementing `\Drupal\Core\Routing\Access\AccessInterface`. Here are some of the common requirement validators defined throughout Drupal core:

- ▶ `_entity_access` validates that the current user has the ability to perform `entity_type.operation`, such as `node.view`
- ▶ `_permission` checks whether the current user has the provided permission
- ▶ `_user_is_logged_in` validates that the current user is logged in, which is defined with a Boolean value in the `routing.yml`

Providing dynamic routes

The routing system allows modules to define routes programmatically. This can be accomplished by providing a `routing_callbacks` key that defines a class and method that will return an array of the `\Symfony\Component\Routing\Route` objects.



If you are working with entities, refer to *Chapter 10, The Entity API*, to learn about overriding the default route handler to create dynamic routes.

In the module's `routing.yml`, you will define the `routing_callbacks` key and related class:

```
route_callbacks:
  - '\Drupal\mymodule\Routing\CustomRoutes::routes'
```

The `\Drupal\mymodule\Routing\CustomRoutes` class will then have a method named `routes`, which returns an array of Symfony route objects:

```
<?php

namespace Drupal\mymodule\Routing;
use Symfony\Component\Routing\Route;

class CustomRoutes {
  public function routes() {
    $routes = [];

    // Create mypage route programmatically
    $routes['mymodule.mypage'] = new Route(
      // Path definition
      'mypage',
      // Route defaults
      [
        '_controller' => '\Drupal\mymodule\Controller\MyPageController::customPage',
        '_title' => 'My custom page',
      ],
      // Route requirements
      [
        '_permission' => 'access content',
      ]
    );
    return $routes;
  }
}
```

If a module provides a class that interacts with routes, the best practice is to place it in the Routing portion of the module's namespace. This helps you identify its purpose.

The invoked method is expected to return an array of initiated Route objects. The Route class takes the following arguments:

- ▶ **Path:** This represents the route
- ▶ **Defaults:** This is an array of default values
- ▶ **Requirements:** This is an array of required validators
- ▶ **Options:** This is an array that can be passed and used optionally

Altering existing routes

When Drupal's route system is rebuilt due to a module being enabled or caches being rebuilt, an event is fired that allows modules to alter routes defined statically in YAML or dynamically. This involves implementing an event subscriber by extending \Drupal\Core\Routing\RouteSubscriberBase, which subscribes the RoutingEvents::ALTER event.

Create `src/Routing/RouteSubscriber.php` in your module. It will hold the route subscriber class:

```
<?php

namespace Drupal\mymodule\Routing;

use Drupal\Core\Routing\RouteSubscriberBase;
use Symfony\Component\Routing\RouteCollection;

class RouteSubscriber extends RouteSubscriberBase {

    /**
     * {@inheritDoc}
     */
    public function alterRoutes(RouteCollection $collection) {
        // Change path of mymodule.mypage to use a hyphen
        if ($route = $collection->get('mymodule.mypage')) {
            $route->setPath('/my-page');
        }
    }
}
```

The preceding code extends `RouteSubscriberBase` and implements the `alterRoutes()` method. We make an attempt to load the `mymodule.mypage` route and, if it exists, we change its path to `my-page`. Since objects are always passed by reference, we do not need to return a value.

For Drupal to recognize the subscriber, we need to describe it in the module's `services.yml` file. In the base directory of your module, create a `mymodule.services.yml` file and add the following code:

```
services:  
  mymodule.route_subscriber:  
    class: Drupal\mymodule\Routing\RouteSubscriber  
    tags:  
      - { name: event_subscriber }
```

This registers our route subscriber class as a service to the container so that Drupal can execute it when the event is fired.

See also

- ▶ Refer to the Symfony routing documentation at
<http://symfony.com/doc/current/book/routing.html>
- ▶ *Chapter 10, The Entity API*
- ▶ Refer to access checking on routes community documentation at
<https://www.drupal.org/node/2122195>

Defining permissions

In Drupal, there are roles and permissions used to define robust access control lists for users. Modules use permissions to check whether the current user has access to perform an action, view specific items, or other operations. Modules then define the permissions used so that Drupal is aware of them. Developers can then construct roles, which are made up of enabled permissions.

In this recipe, we will define a new permission to view custom pages defined in a module. The permission will be added to a custom route and restrict access to the route path to users who have a role containing the permission.

Getting ready

Create a new module like the one in the first recipe. We will refer to the module as `mymodule` throughout the recipe. Use your module's name as appropriate.

This recipe also modifies a route defined in the module. We will refer to this route as `mymodule.mypage`. Modify the appropriate path in your module's `routing.yml` file.

How to do it...

1. Permissions are stored in a `permissions.yml` file. Add a `mymodule.permissions.yml` to the base directory of your module.
2. First, we need to define the internal string used to identify this permission, such as `view mymodule pages`:

```
view mymodule pages:
```

3. Each permission is a YAML array of data. We need to provide a `title` key that will be displayed on the permissions page:

```
view mymodule pages:  
  title: 'View my module pages'
```

4. Permissions have a `description` key to provide details of the permission on the permissions page:

```
view mymodule pages:  
  title: 'View my module pages'  
  description: 'Allows users to view pages provided by My Module'
```

5. Save your `permissions.yml`, and edit the module's `routing.yml` to add the permission.

6. Modify the route's `requirements` key to have a `_permissions` key that is equal to the defined permission:

```
mymodule.mypage:  
  path: /mypage  
  defaults:  
    _controller: '\Drupal\mymodule\Controller\  
MyPageController::customPage'  
    _title: 'My custom page'  
  requirements:  
    _permission: 'view mymodule pages'
```

7. Visit **Configuration** and then **Development** to rebuild Drupal's caches.
8. Visit **People** and then **Permissions** to add your permission to the authenticated user and anonymous user roles.

The screenshot shows the Drupal 'Manage' menu with 'Shortcuts' and 'admin' selected. The main content area is titled 'PERMISSION'. It lists a permission named 'Create and modify styles for generating image modifications such as thumbnails.' under the 'My Module!' module. This permission is assigned to three roles: 'ANONYMOUS USER', 'AUTHENTICATED USER', and 'ADMINISTRATOR'. Below this, there are two sections: 'Node' and 'Administer content'. The 'Node' section contains the permission 'Access the Content overview page' which is assigned to 'ADMINISTRATOR'. The 'Administer content' section contains the permission 'Administer content' which is also assigned to 'ADMINISTRATOR'. A warning message states: 'Warning: Give to trusted roles only; this permission has security implications. Promote, change ownership, edit revisions, and perform other tasks across all content types.'

9. Log out of your Drupal site, and view the `/mypage` page. You will see the content and not receive an access denied page.

How it works...

Permissions and Roles are provided by the `User` module. The `user.permissions` service discovers `permissions.yml` defined in installed modules. By default, the service is defined through the `\Drupal\user\PermissionHandler` class.

Drupal does not save a list of all permissions that are available. The permissions for a system are loaded when the permissions page is loaded. Roles contain an array of permissions.

When checking a user's access for a permission, Drupal checks all of the user's roles to see whether they support that permission.



You can pass an undefined permission to a user access check and not receive an error. The access check will simply fail, unless the user is UID 1, which bypasses access checks.

There's more...

Restrict access flag for permissions

Permissions can be flagged as having a security risk if enabled. This is the restrict access flag. When this flag is set to `restrict_access: TRUE`, it will add a warning to the permission description.

This allows module developers to provide more context to the amount of control a permission may give a user:

PERMISSION	ANONYMOUS USER	AUTHENTICATED USER	ADMINISTRATOR
Administer content <i>Warning: Give to trusted roles only; this permission has security implications. Promote, change ownership, edit revisions, and perform other tasks across all content types.</i>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

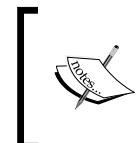
Defining permissions programmatically

Permissions can be defined by a module programmatically or statically in a YAML file. A module needs to provide a `permission_callbacks` key in its `permissions.yml` that contains an array of classes and their methods or a procedural function name.

For example, the Filter module provides granular permissions based on the different text filters created in Drupal:

```
permission_callbacks:
  - Drupal\filter\FilterPermissions::permissions
```

This tells the `user_permissions` service to execute the `permissions` method of the `\Drupal\Filter\FilterPermissions` class. The method is expected to return an array that matches the same structure as that of the `permissions.yml` file.



Dynamically defined permissions cannot use the restrict access flag and need to manually add the security warning to the description, just as `\Drupal\Filter\FilterPermissions` does.

Checking whether a user has permissions

The user account interface provides a method for checking whether a user entity has a permission. To check whether the current user has a permission, you will get the current user, and you need to invoke the `hasPermission` method:

```
\Drupal::currentUser()->hasPermission('my permission');
```

The `\Drupal::currentUser()` method returns the current active user object. This allows you to check whether the active user has permissions to perform some sort of actions.

Providing the configuration on installation or update

Drupal provides a configuration management system, which is discussed in *Chapter 9, Configuration Management – Deploying in Drupal 8*, and modules are able to provide configuration on an installation or through an update system. Modules provide the configuration through `YAML` files when they are first installed. Once the module is enabled, the configuration is then placed in the configuration management system. Updates can be made to the configuration, however, in code through the Drupal update system.

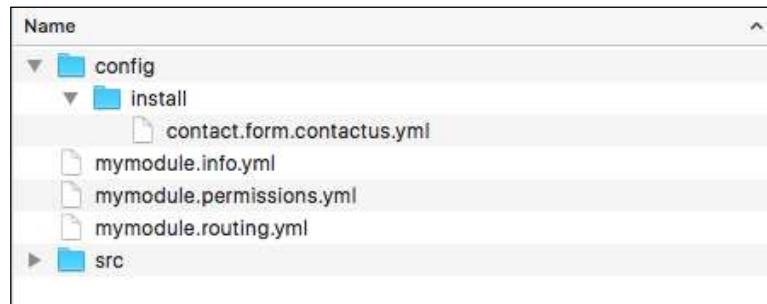
In this recipe, we will provide configuration `YAML`s that create a new contact form and then manipulate them through a schema version change in the update system.

Getting ready

Create a new module like the one in the first recipe. We will refer to the module as `mymodule` throughout the recipe. Use your module's appropriate name.

How to do it...

1. Create a `config` folder in your module's base directory. All configuration `YAML`s should be in a subfolder of `config`.
2. Create a folder named `install` in the `config` folder. Configuration `YAML`s in this folder will be imported on module installation.
3. In the `install` folder, create a `contact.form.contactus.yml` to store the `YAML` definition of the contact form, **Contact Us**:



- We will define the configuration of a contact form based on the `contact.schema.yml` file provided by the Contact module:

```
langcode: en
status: true
dependences: {}
id: contactus
label: 'Contact Us'
recipients:
  - webmaster@example.com
reply: ''
weight: 0
```

The configuration entry is based on a schema definition, which we will cover in *Chapter 9, Configuration Management – Deploying in Drupal 8*. The `langcode`, `status`, and `dependences` are the required configuration management keys.

The `id` is the contact form's machine name and the `label` is the human display name. The `recipients` key is a YAML array of valid e-mail addresses. The `reply` key is a string of text for the **Auto-reply** field. And, finally, the `weight` defines the form's weight in the administrative list.

- Visit **Extend** and enable your module to import the configuration item.
- The **Contact Us** form will now be located on the **Contact** forms overview page, located under Structure:

FORM	RECIPIENTS	SELECTED	OPERATIONS
Contact Us	webmaster@example.com	No	<button>Edit</button>
Personal contact form	Selected user	No	<button>Manage fields</button>
Website feedback	admin@example.com	Yes	<button>Edit</button>

- Create a `mymodule.install` file in the module's base directory. We will create an update hook to set a reply message for the contact form.
- We will create a function called `mymodule_update_8001()` that will be read by the update system and make our configuration changes:

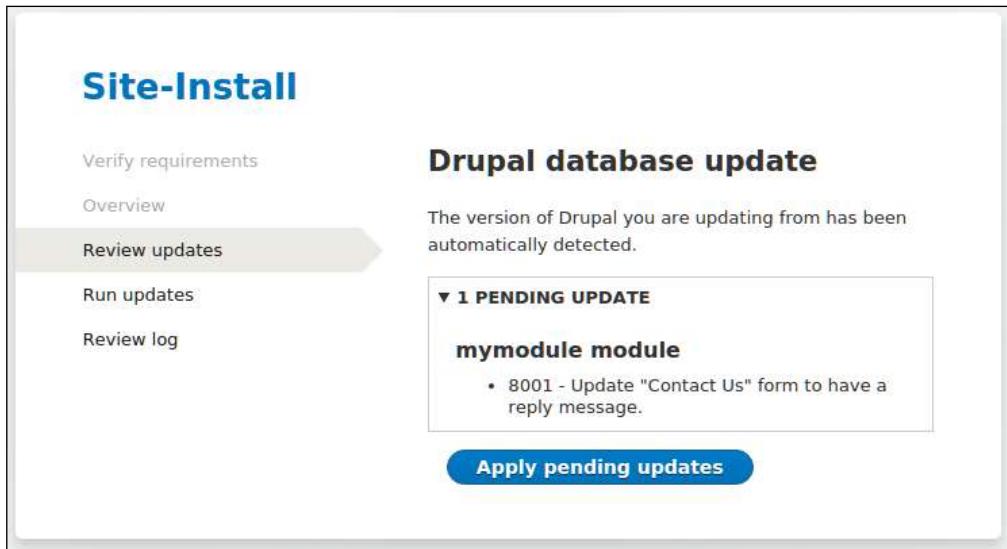
```
<?php

/**
 * Update "Contact Us" form to have a reply message.
```

```
/*
function mymodule_update_8001() {
  $contact_form = \Drupal\contact\Entity\ContactForm::load('contactus')
  $contact_form->setReply(t('Thank you for contacting us, we will
  reply shortly'));
  $contact_form->save();
}
```

This function uses the entity's class to load our configuration entity object. It loads `contact_us`, which our module has provided, and sets the `reply` property to a new value.

9. Visit `/update.php` in your browser to run the Drupal's database update system:



10. Review the **Contact Us** form settings and verify that the reply message has been set.

How it works...

Drupal's `moduler_installer` service, provided through `\Drupal\Core\Extension\ModuleInstaller`, ensures that configuration items defined in the module's `config` folder are processed on installation. When a module is installed, the `config.installer` service, provided through `\Drupal\Core\Config\ConfigInstaller`, is called to process the module's default configuration.

In the event, the `config.installer` service makes an attempt to import the configuration from the `install` folder that already exists and an exception will be thrown. Modules cannot provide changes made to the existing configuration through static YAML definitions.

Since modules cannot adjust configuration objects through static YAML definitions provided to Drupal, modules can utilize the database update system to modify the configuration. Drupal utilizes a schema version for modules. The base schema version for a module is 8000. Modules can provide update hooks in the form of `hook_update_N`, where N represents the next schema version. When Drupal's updates are run, they will execute the proper update hooks and update the module's schema version.

Configuration objects are immutable by default. In order to edit a configuration, a mutable object needs to be loaded through the configuration factory service.

There's more...

We will discuss the configuration in *Chapter 9, Configuration Management – Deploying in Drupal 8*, however, we will now dive into some important notes when working with modules and configurations.

Configuration subdirectories

There are three directories that the configuration management system will inspect in a module's `config` folder, which are as follows:

- ▶ `install`
- ▶ `optional`
- ▶ `schema`

The `install` folder specifies the configuration that will be imported. If the configuration object exists, the installation will fail.

The `optional` folder contains the configuration that will be installed if the following conditions are met:

- ▶ The configuration does not already exist
- ▶ It is a configuration entity
- ▶ Its dependencies can be met

If any one of the conditions fail, the configuration will not be installed, but it will not halt the module's installation process.

The `schema` folder provides definitions of configuration object definitions. This uses YAML definitions to structure configuration objects and is covered in depth in *Chapter 9, Configuration Management*.

Modifying the existing configuration on installation

The configuration management system does not allow modules to provide configuration on an installation that already exists. For example, if a module tries to provide `system.site` and defines the site's name, it would fail to install. This is because the System module provides this configuration object on installation.

Drupal provides `hook_install()` that modules can implement in their `.install` file. This hook is executed during the module's installation process. The following code will update the site's title to *Drupal 8 Cookbook!* on the module's installation:

```
/**  
 * Implements hook_install().  
 */  
function mymodule_install() {  
    // Set the site name.  
    \Drupal::configFactory()  
        ->getEditable('system.site')  
        ->set('name', 'Drupal 8 Cookbook!')  
        ->save();  
}
```

Configurable objects are immutable by default when loaded by the default config service. In order to modify a configuration object, you need to use the configuration factory to receive a mutable object. The mutable object can have `set` and `save` methods that are executed to update the configuration in a configuration object.

See also

- ▶ [Chapter 9, Configuration Management – Deploying in Drupal 8](#)

Using Features 2.x

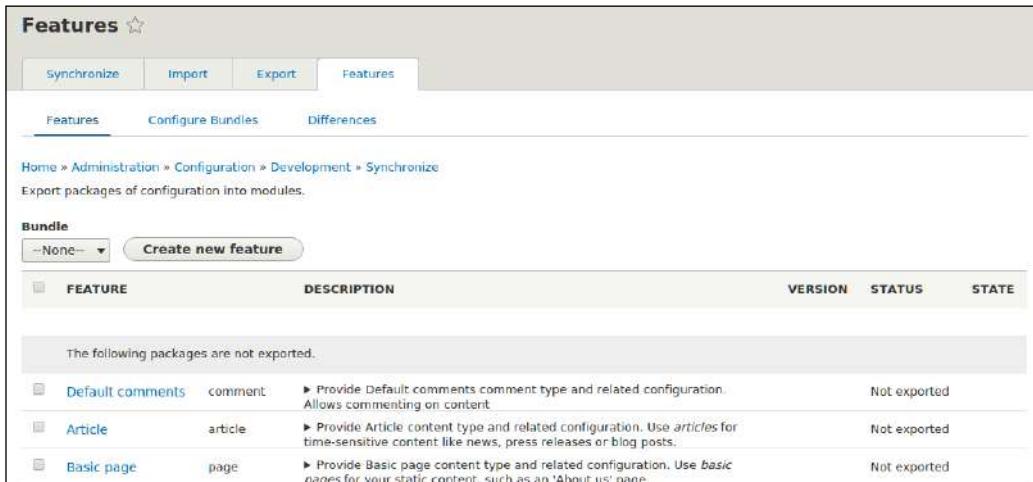
Many Drupal users create custom modules to provide specific sets of features that they can reuse across multiple sites. In fact, there is a module for the sole purpose of providing a means to export configuration and create modules that provide features. This is how the Features module received its name, in fact.

The Features module has two sub-modules. The main Features module provides all the functionalities. The Features UI module provides a user interface for creating and managing features.

We will use Features to export a module with a configuration that contains the default page and article content types provided by the standard installation, so they can be used on other installation profiles.

How to do it...

1. The Features module requires **Configuration Update Manager** as a dependency. Visit https://www.drupal.org/project/config_update and download the latest Drupal 8 release and place it in your Drupal site's /modules folder.
2. Now, visit <https://www.drupal.org/project/features> and download the latest Drupal 8 release and place it in your Drupal site's /modules folder.
3. Visit **Extend** and install the Features UI module, confirming the requirements to install Features and Configuration Update Manager as well.
4. Visit **Configuration** and then **Configuration Synchronization**. The user interface for **Features** is accessed as a tab from this page:

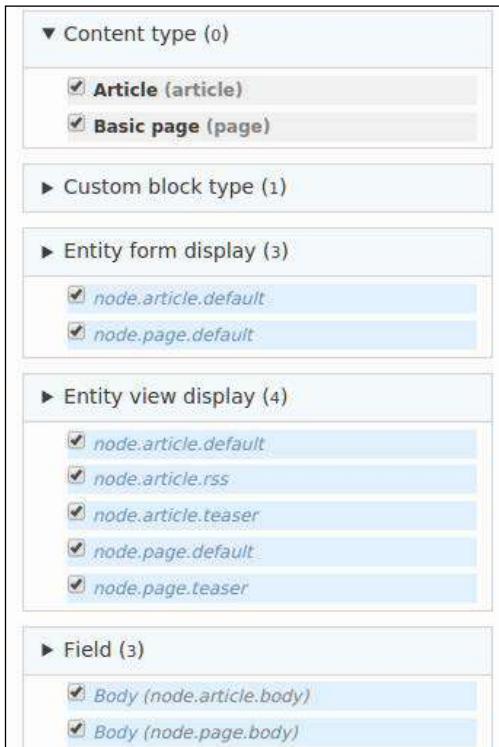


The screenshot shows the 'Features' configuration synchronization page. At the top, there are tabs for 'Synchronize', 'Import', 'Export', and 'Features'. The 'Features' tab is selected. Below the tabs, there are three navigation links: 'Features', 'Configure Bundles', and 'Differences'. The main content area has a breadcrumb trail: 'Home > Administration > Configuration > Development > Synchronize'. A sub-instruction below the breadcrumb says 'Export packages of configuration into modules.' There is a 'Bundle' dropdown set to 'None' and a 'Create new feature' button. A table lists content types: 'Default comments' (comment), 'Article' (article), and 'Basic page' (page). Each row includes a description and a 'Not exported' status indicator. The table has columns for 'FEATURE', 'DESCRIPTION', 'VERSION', 'STATUS', and 'STATE'.

FEATURE	DESCRIPTION	VERSION	STATUS	STATE
Default comments (comment)	Provide Default comments comment type and related configuration. Allows commenting on content.		Not exported	
Article (article)	Provide Article content type and related configuration. Use articles for time-sensitive content like news, press releases or blog posts.		Not exported	
Basic page (page)	Provide Basic page content type and related configuration. Use basic pages for your static content, such as an "About us" page.		Not exported	

5. Click on **Create new feature** to start making a custom Feature module.
6. Provide a Name for the feature, such as **Content Authoring**.
7. Optionally, you can provide a description. This acts as the description key in the module's `info.yml`.
8. Toggle the **Content types** grouping and check **Article** and **Basic Page** to mark them for export.

9. The features module will automatically add detected dependencies or important configuration items to also be exported, such as fields and view modes.



10. Click on **Write** to write the module to export the module and configuration to the /modules/custom directory in your Drupal site.
11. Visit **Extend** to enable your newly created module.

How it works...

Features exports static YAML configuration files into the module's config/install folder. Features modifies the standard configuration management workflow by ensuring that a specific kind of configuration exists. Configuration management does not allow modules to overwrite existing configuration objects but Features manages and allows this to happen.

To accomplish this, Features provides \Drupal\features\FeaturesConfigInstaller, which extends the default config.install service class. It then alters the services definition to use its FeaturesConfigInstaller class instead of the default \Drupal\Core\Config\ConfigInstaller class.

Beyond adjusting the `config.install` service, Features harnesses all the functionalities of the configuration management system to provide a simpler way to generate modules.



Any module can be considered a Feature's module by adding the `features: true` key to its `info.yml`. This will allow it to be managed through the Features UI.



There's more...

Suggested feature modules

The features module provides an intelligent bundling method that reviews the current Drupal site's configuration and suggests feature modules that should be created to preserve the configuration. These are provided through package assignment plugins. These plugins use logic to assign configurations to specific packages.

The screenshot shows the Features UI for the 'User' entity. At the top, there is a checkbox labeled 'User' and the text 'Provide User related configuration.' Below this, there are several sections:

- Action:** Buttons for 'Remove the Administrator role from the selected users' and 'Add the Administrator role to the selected users'.
- Dependencies:** Buttons for 'File', 'Image', and 'User'. Underneath, it lists 'user.user.default'.
- Entity form display:** Buttons for 'user.user.default' and 'user.user.compact'.
- Entity view display:** Buttons for 'user.user.default' and 'user.user.compact'.
- Field:** Button for 'Picture'.
- Field storage:** Button for 'user.user_picture'.

When you visit the Features UI, it will present you with suggested feature modules to be exported. Expanding the items will list the configuration items that will be bundled. Clicking on the suggested feature's link opens the creation form. Or the checkbox can be used in conjunction with the download archive or write buttons at the bottom of the form.



The unpackaged section shows a configuration, which has not met any of the packaging rules to group the configuration into a specified module. This will need to be manually added to a created feature module.



Features bundles

In the Features module, there are bundles and bundles have their own assignment method configurations. The purpose of bundles inside Features is to provide an automatic assignment of configuration that can be grouped into exported modules:

ASSIGNMENT METHOD	DESCRIPTION	ENABLED	OPERATIONS
<input checked="" type="checkbox"/> Packages	Detect and add existing package modules.	<input checked="" type="checkbox"/>	
<input checked="" type="checkbox"/> Exclude	Exclude configuration items from packaging by various methods including by configuration type.	<input checked="" type="checkbox"/>	Configure

A bundle has a human display name and machine name. The bundle's machine name will be prefixed on all feature modules generated under this bundle. You also have the ability to specify the bundle to act as an install profile. Features UI was heavily used in Drupal 7 to construct distributions and spawn the concept of the bundle functionality.

Assignment methods can be rearranged and configured to your liking.

Managing the configuration state of Features

The Features UI provides a means to review changes to the feature's configuration that may have been made. If a configuration item controlled by a Feature module has been modified, it will show up under the differences section of the Features UI. This will allow you to import or update the Feature module with the change.

The **Import** option will force the site to use the configuration defined in the module's configuration YAML files. For example, we have an exported content type whose description was modified in the user interface after being exported.

The screenshot shows a comparison table between 'ACTIVE SITE CONFIG' and 'FEATURE CODE CONFIG' for a 'NODE.TYPE ARTICLE' content type. The 'ACTIVE SITE CONFIG' column contains several configuration items, some of which are highlighted in yellow. The 'FEATURE CODE CONFIG' column contains a single item with a green background. A button at the bottom left says 'Import changes'.

▼ NODE.TYPE ARTICLE	
ACTIVE SITE CONFIG	FEATURE CODE CONFIG
- dependencies	+ description : Use articles for time-sensitive content like news, press releases or blog posts.
- dependencies::module	
- dependencies::module::0 : menu_ui	
- description : Use articles for content like news, press releases or blog posts.	
- third_party_settings	
- third_party_settings::menu_ui	
- third_party_settings::menu_ui::available_menus	
- third_party_settings::menu_ui::available_menus::0 : main	
- third_party_settings::menu_ui::parent : main:	

Import changes Import the selected changes above into the active configuration.

The difference created by the Feature module is highlighted. If the difference was checked, and if you click on **Import changes**, the content type's description would be reset to that defined in the configuration.

From the main features overview table, the Feature module can be reexported to include the change and update the exported YAML files.

See also

- ▶ Refer to Features for the Drupal 8 session by Mike Potter at DrupalCon Los Angeles at <https://events.drupal.org/losangeles2015/sessions/features-drupal-8>

5

Frontend for the Win

In this chapter, we will explore the world of frontend development in Drupal 8:

- ▶ Creating a custom theme based on Classy
- ▶ Using the new asset management system
- ▶ Twig templating
- ▶ Using the Breakpoint module
- ▶ Using the Responsive Image module module

Introduction

Drupal 8 brings many changes with regard to the frontend. It is now focused on the mobile-first responsive design. Frontend performance has been given a high priority, unlike in the previous versions of Drupal. There is a new asset management system based around libraries that will deliver only the minimum required assets for a page that comes with Drupal 8.

In Drupal 8, we have a new feature, the Twig templating engine, that replaces the previously used PHPTemplate engine. Twig is part of the large PHP community and embraces more of Drupal 8's *made elsewhere* initiative. Drupal 7 supported libraries to define JavaScript and CSS resources. However, it was very rudimentary and did not support the concept of library dependencies.

There are two modules provided by Drupal core that implement the responsive design with server-side components. The Breakpoint module provides a representation of media queries that modules can utilize. The Responsive Image module implements the HTML5 picture tag for image fields.

This chapter dives into harnessing Drupal 8's frontend features to get the most out of them.

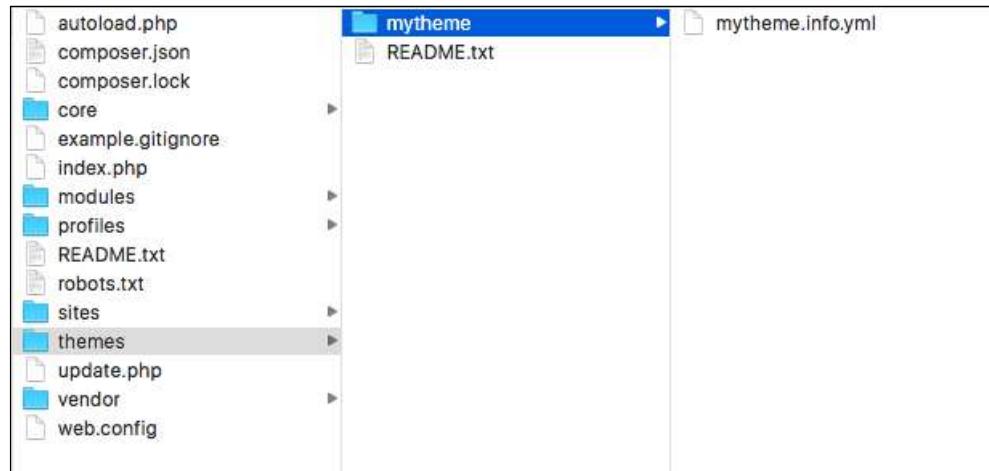
Creating a custom theme based on Classy

Drupal 8 ships with a new base theme that is intended to demonstrate the best practice and CSS class management. The Classy theme is provided by Drupal core and is the base theme for the default frontend theme, Bartik, and the administrative theme, Seven.

Unlike the previous versions of Drupal, Drupal 8 provides two base themes: Classy and Stable as a means to jump start Drupal theming. Stable provides a more lean approach to frontend theming with fewer classes and wrapping elements. In this recipe, we will create a new theme called `mytheme` that uses Classy as its base.

How to do it...

1. In the root directory of your Drupal site, create a folder called `mytheme` in the `themes` folder.
2. Inside the `mytheme` folder, create a `mytheme.info.yml` file so that Drupal can discover the theme. We will then edit this file:



3. First, we need to define the themes name using the `name` key:

```
name: My Theme
```

4. All the themes need to provide a `description` key, which will be displayed on the **Appearance** page:

```
description: My custom theme
```

5. Next, we need to define the type of extension, that is, a theme, and the version of core that is supported:

```
type: theme
core: 8.x
```

6. The `base theme` call allows us to instruct Drupal to use a specific theme as a base:

```
base theme: classy
```

7. The last item is a `regions` key that is used to define the regions of the blocks that can be placed, which is a YAML-based array of key/value pairs:

```
regions:
  header: Header
  primary_menu: 'Primary menu'
  page_top: 'Page top'
  page_bottom: 'Page bottom'
  breadcrumb: Breadcrumb
  content: Content
```

8. Regions are rendered in the page template file, which will be covered in the next recipe, Twig templates.

9. Log in to your Drupal site, and go to **Appearance** from the administrative toolbar.

10. Click on **Install and set default** in the **My theme** entry in order to enable and use the new custom theme:

Uninstalled themes

My Theme My custom theme Install Install and set as default	Stable 8.0.0-rc3 A default base theme using Drupal 8.0.0's core markup, CSS, and JavaScript. Install Install and set as default	Stark 8.0.0-rc3 An intentionally plain theme with no styling to demonstrate default Drupal's HTML and CSS. Learn how to build a custom theme from Stark in the Theming Guide . Install Install and set as default
------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

How it works...

In Drupal 8, the `info.yml` files define Drupal themes and modules. The first step to create a theme is to provide the `info.yml` file so that the theme can be discovered. Drupal will parse these values and register the theme.

The following keys are required, as a minimum, when you define a theme:

- ▶ name
- ▶ description
- ▶ type
- ▶ base theme
- ▶ core

The `name` key defines the human-readable name of the theme that will be displayed on the **Appearance** page. The `description` will be shown under the themes display name on the **Appearance** page. All Drupal projects need to define the `type` key to indicate the kind of extension that is being defined. For themes, the type must always be `theme`. You need to also define which version of Drupal the project is compatible with using the `core` value. All Drupal 8 projects will use the `core: 8.x` value. When you define a theme, you need to also provide the `base theme` key. If your theme does not use a base theme, then you need to set the value to `false`.

The `libraries` and `region` keys are optional, but these are keys that most themes provide. Drupal's asset management system parses a theme's `info.yml` and adds those libraries, if required. Regions are defined in an `info.yml` file and provide the areas into which the Block module may place blocks.

There's more...

Next, we will dive into some additional information about themes.

Theme screenshots

Themes can provide a screenshot that shows up on the **Appearance** page. A theme's screenshot can be provided by placing a `screenshot.png` in the theme folder or a file specified in the `info.yml` file under the `screenshot` key.

If the screenshot is missing, a default is used, as seen with the Classy and Stark themes. Generally, a screenshot is a Drupal site with generic content using the theme.

Themes, logos, and favicons

Drupal controls the site's favicon and logo settings as a theme setting. Theme settings are active on a theme-by-theme basis and are not global. Themes have the ability to provide a default logo by providing a `logo.svg` in the theme root folder. A `favicon.ico` placed in a theme folder will also be the default value of the `favicon` for the website.



Currently, there is no way to specify a logo of a different file type for a theme. Previous versions of Drupal looked for `logo.png`. A feature has been postponed for Drupal 8.1 to allow the themes to have the ability to define the logo's filename and extension. Refer to the core issue for more information at <https://www.drupal.org/node/1507896>.

You can change the site's logo and favicon by going to **Appearance** and then clicking on **Settings** for your current theme. Unchecking the `use default` checkboxes for the favicon and logo settings allows you to provide custom files.

▼ LOGO IMAGE SETTINGS

Use the default logo supplied by the theme

Path to custom logo

Examples: `logo.svg` (for a file in the public filesystem), `public://logo.svg`, or `core/themes/seven/logo.svg`.

Upload logo image
 No file chosen
If you don't have direct file access to the server, use this field to upload your logo.

▼ SHORTCUT ICON SETTINGS

Your shortcut icon, or 'favicon', is displayed in the address bar and bookmarks of most browsers.

Use the default shortcut icon supplied by the theme

Path to custom icon

Examples: `favicon.ico` (for a file in the public filesystem), `public://favicon.ico`, or `core/themes/seven/favicon.ico`.

Upload icon image
 No file chosen
If you don't have direct file access to the server, use this field to upload your shortcut icon.

Base themes and shared resources

Many content management systems that have a theme system support base (or parent) themes differ mostly in the terminology used. The concept of a base theme is used to provide established resources that are shared, reducing the amount of work required to create a new theme.

All libraries defined in the base theme will be inherited and used by default, allowing subthemes to reuse existing styles and JavaScript. This allows frontend developers to reuse work and only create specific changes that are required for the subtheme.

The Subthemes will also inherit all Twig template overrides provided by the base theme. This was one of the initiatives used for the creation of the Classy theme. Drupal 8 makes many fewer assumptions compared to previous version as to what class names to provide on elements. Classy overrides all of the core's templates and provides sensible default classes, giving themes the ability to use them and accept those class names or be given a blank slate.

CKEditor stylesheets

As discussed in *Chapter 2, The Content Authoring Experience*, Drupal ships with the WYSIWYG support and CKEditor as the default editor. The CKEditor module will inspect the active theme, and its base theme if provided, and loads any stylesheets defined in the `ckeditor_stylesheets` key as an array of values.

For example, the following code can be found in `bartik.info.yml`:

```
ckeditor_stylesheets:
  - css/base/elements.css
  - css/components/captions.css
  - css/components/table.css
```

This allows themes to provide style sheets that will style elements within the CKEditor module to enhance the *what you see is what you get* element of the editor.

See also

- ▶ To define a theme with an `info.yml` file, refer to
<https://www.drupal.org/node/2349827>
- ▶ To use Classy as a base theme, refer to the community documentation at
<https://www.drupal.org/theme-guide/8/classy>
- ▶ To create a Drupal 8 subtheme, refer to the community documentation at
<https://www.drupal.org/node/2165673>

Using the new asset management system

New to Drupal 8 is the asset management system. The asset management system allows modules and themes to register libraries. Libraries define CSS stylesheets and JavaScript files that need to be loaded with the page. Drupal 8 takes this approach for the frontend performance. Rather than loading all CSS or JavaScript assets, only those required for the current page in the specified libraries will be loaded.

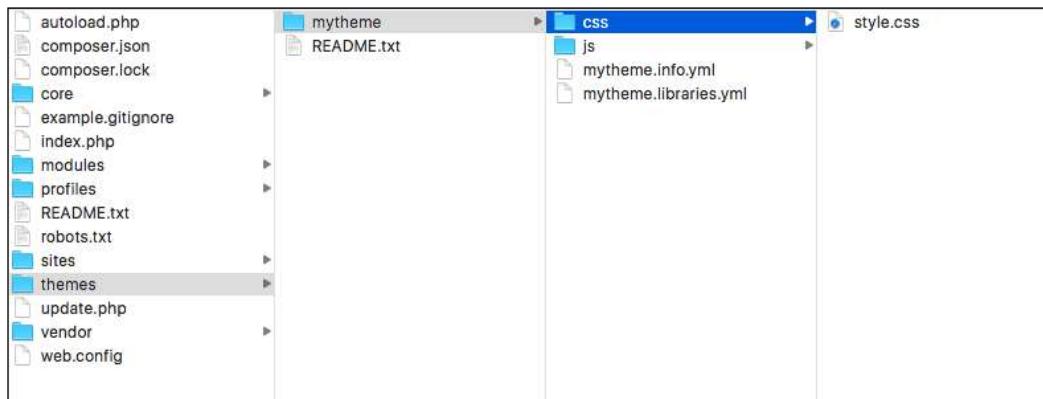
In this recipe, we will define a `libraries.yml` file that will define a CSS stylesheet and JavaScript file provided by a custom theme.

Getting ready

This recipe assumes that you have a custom theme created, such as the one you created in the first recipe. When you see `mytheme`, use the machine name of the theme that you have created.

How to do it...

1. Create a folder named `css` in your themes base directory.
 2. In your `css` folder, add a `style.css` file that will hold the theme's CSS declarations. For demonstration purposes, add the following CSS declaration to `style.css`:
- ```
body {
 background: cornflowerblue;
}
```
3. Then, create a `js` folder, and add a `scripts.js` file that will hold the themes JavaScript items.
  4. In your theme folder, create a `mytheme.libraries.yml` file and edit it, as shown in the following screenshot:

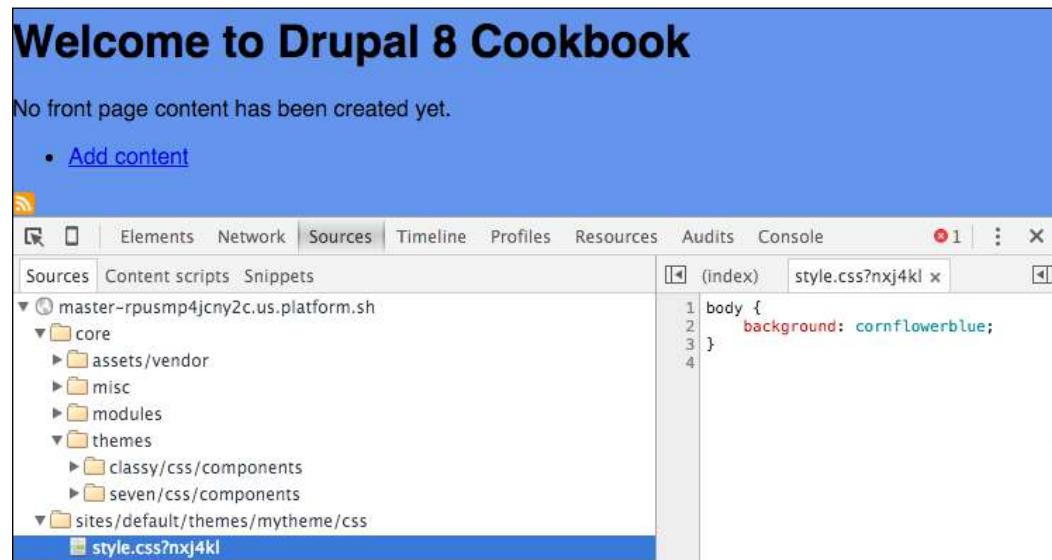


5. Add the following YAML text to define the global-styling library for your theme that will load the CSS file and JavaScript file:

```
global-styling:
 version: VERSION
 css:
 theme:
 css/style.css: {}
 js:
 js/scripts.js: {}
```

6. This tells Drupal that there is a `global-styling` library. You have the ability to specify a library version and use the `VERSION` defaults for your themes. It also defines the `css/styles.css` stylesheet as part of the library under the theme group.
7. Edit your `mytheme.info.yml`, and we need to add the declaration to our `global-styling` library:

```
libraries:- mytheme/global-styling
```
8. Themes are able to specify a `libraries` key that defines the libraries that should always be loaded. This YAML array lists libraries to be loaded for each page.
9. Go to **Configuration** and then to **Development** to rebuild Drupal's caches.
10. With your theme set to the default, go to your Drupal site.
11. Your theme's `global-styling` library will be loaded and the page's background color will be styled appropriately:



The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. On the left, the file tree shows the project structure with 'style.css?nxj4kl' highlighted at the bottom. On the right, the code editor displays the following CSS:

```
1 body {
2 background: cornflowerblue;
3 }
4
```

## How it works...

Drupal aggregates all the available `library.yml` files and passes them to the `library.discovery.parser` service. The default class for this service provider is `\Drupal\Core\Asset\LibraryDiscoveryParser`. This service reads the library definition from each `library.yml` and returns its value to the system. Before parsing the file, the parser allows themes to provide overrides and extensions to the library.

Libraries are enqueueers as they are attached to rendered elements. Themes have the ability to generically add libraries through their `info.yml` files via the `libraries` key. These libraries will always be loaded on the page when the theme is active.

CSS stylesheets are added to the data, which will build the head tag of the page. JavaScript resources, by default, are rendered in the footer of the page for performance reasons.

## There's more...

We will explore the options surrounding libraries in Drupal 8 in more detail.

### CSS groups

With libraries, you have the ability to specify CSS by different groups. Drupal's asset management system provides the following CSS groups:

- ▶ base
- ▶ layout
- ▶ component
- ▶ state
- ▶ theme

Stylesheets are loaded in the order in which the groups are listed. Each one of them relates to a PHP constant defined in `/core/includes/common.inc`. This allows separation of concerns when working with stylesheets. Drupal 8's CSS architecture borrows concepts from the SMACSS system to organize CSS declarations.

### Library asset options

Library assets can have configuration data attached to them. If there are no configuration items provided, a simple set of empty brackets is added. This is why, in each example, files end with `{ }`.

The following example, taken from `core.libraries.yml`, adds `HTML5shiv`:

```
assets/vendor/html5shiv/html5shiv.min.js: { weight: -22, browsers: {
 IE: 'lte IE 8', '!IE': false }, minified: true }
```

Let's take a look at the attributes of `html5shiv.min.js`:

- ▶ The `weight` key ensures that the script is rendered earlier than other libraries
- ▶ The `browsers` tag allows you to specify conditional rules to load the scripting
- ▶ You should always pass `minified` as `true` if the asset has already been minified

For CSS assets, you can pass a `media` option to specify a media query for the asset. Reviewing classes which implement `\Drupal\Core\Asset\AssetCollectionRendererInterface`.

## Library dependencies

Libraries have the ability to specify other libraries as dependencies. This allows Drupal to provide a minimum footprint on the frontend performance.



jQuery is only loaded if a JavaScript library specifies it as a dependency.  
Refer to <https://www.drupal.org/node/1541860>.



Here's an example from the Quick Edit module's `libraries.yml` file:

```
quickeedit:
 version: VERSION
 js:
 ...
 css:
 ...
 dependencies:
 - core/jquery
 - core/jquery.once
 - core/underscore
 - core/backbone
 - core/jquery.form
 - core/jquery.ui.position
 - core/drupal
 - core/drupal.displace
 - core/drupal.form
 - core/drupal.ajax
 - core/drupal.debounce
 - core/drupalSettings
 - core/drupal.dialog
```

The Quick Edit module defines jQuery, the jQuery Once plugin, Underscore, and Backbone, and selects other defined libraries as dependencies. Drupal will ensure that these are present whenever the `quickeedit/quickeedit` library is attached to a page.

A complete list of the default libraries provided by Drupal core can be found in `core.libraries.yml`, which is in `core/core.libraries.yml`.

## Overriding and extending other libraries

Themes have the ability to override libraries using the `libraries-override` and `libraries-extend` keys in their `info.yml`. This allows themes to easily customize the existing libraries without having to add the logic for conditionally removing or adding their assets when a particular library has been attached to a page.

The `libraries-override` key can be used to replace an entire library, replace selected files in a library, remove an asset from a library, or disable an entire library. The following code will allow a theme to provide a custom jQuery UI theme:

```
libraries-override:
 core/jquery.ui:
 css:
 component:
 assets/vendor/jquery.ui/themes/base/core.css: false
 theme:
 assets/vendor/jquery.ui/themes/base/theme.css: css/jqueryui.
css
```

The override declaration mimics the original configuration. Specifying `false` will remove the asset or else a supplied path will replace that asset.

The `libraries-extend` key can be used to load additional libraries with an existing library. The following code will allow a theme to associate a CSS stylesheet with selected jQuery UI declaration overrides, without always having them included in the rest of the theme's assets:

```
libraries-extend:
 core/jquery.ui:
 - mytheme/jqueryui-theme
```

## Using a CDN or external resource as a library

Libraries also work with external resources, such as assets loaded over a CDN. This is done by providing a URL for the file location along with selected file parameters.

Here is an example to add the `FontAwesome` font icon library from the `BootstrapCDN` provided by MaxCDN:

```
mytheme-fontawesome:
 remote: http://fontawesome.io/
 version: 4.4.0
 license:
 name: SIL OFL 1.1
 url: http://fontawesome.io/license/
 gpl-compatible: true
 css:
 base:
 https://maxcdn.bootstrapcdn.com/font-awesome/4.4.0/css/font-
 awesome.min.css: { type: external, minified: true }
```

Remote libraries require additional meta information to work properly:

```
remote: http://fontawesome.io/
```

The `remote` key describes the library as using external resources. While this key is not validated beyond its existence, it is best to define it with the external resource's primary website:

```
version: 4.4.0
```

Like all libraries, a version is required. This should match the version of the external resource being added:

```
license:
 name: SIL OFL 1.1
 url: http://fontawesome.io/license/
 gpl-compatible: true
```

If a library defines the `remote` key, it needs to also define the `license` key. This defines the license name, the URL for the license, and checks whether it is GPL compatible. If this key is not provided, a `\Drupal\Core\Asset\Extension\LibraryDefinitionMissingLicenseException` will be thrown:

```
css:
 base:
 https://maxcdn.bootstrapcdn.com/font-awesome/4.4.0/css/font-
 awesome.min.css: { type: external, minified: true }
```

Finally, specific external resources are added as normal. Instead of providing a relative file path, the external URL is provided.

## Manipulating libraries from hooks

Modules have the ability to provide dynamic library definitions and alter libraries. A module can use the `hook_library_info()` hook to provide a library definition. This is not the recommended way to define a library, but it is provided for edge use cases.

Modules do not have the ability to use `libraries-override` or `libraries-extend`, and need to rely on the `hook_library_info_alter()` hook. The hook is documented in `core/lib/Drupal/Core/Render/theme.api.php` or at [https://api.drupal.org/api/drupal/core!lib!Drupal!Core!Render!theme.api.php/function/hook\\_library\\_info\\_alter/8](https://api.drupal.org/api/drupal/core!lib!Drupal!Core!Render!theme.api.php/function/hook_library_info_alter/8).

## Placing JavaScript in the header

By default, Drupal ensures that JavaScript is placed last on the page. This improves the page, load performance by allowing the critical portions of the page to load first. Placing JavaScript in the header is now an opt-in option.

In order to render a library in the header, you need to add the `header: true` key/value pair:

```
js-library:
 header: true
```

```
js:
 js/myscripts.js: {}
```

This will load a custom JavaScript library and its dependencies into the header of a page.

## See also

- ▶ Refer to the CSS architecture for Drupal 8: Separate concerns at  
<https://www.drupal.org/node/1887918#separate-concerns>
- ▶ SMACSS (<http://smacss.com/book/>)

## Twig templating

Drupal 8's theming layer is complemented by Twig, a component of the Symfony framework. Twig is a template language that uses a syntax similar to Django and Ninja templates. The previous version of Drupal used PHPTemplate that required frontend developers to have a rudimentary understanding of PHP.

In this recipe, we will override the Twig template to provide customizations for the e-mail form element. We will use the basic Twig syntax to add a new class and provide a default placeholder.

## Getting ready

This recipe assumes that you have a custom theme created, such as the one you created in the first recipe. When you see `mytheme`, use the machine name of the theme you created.



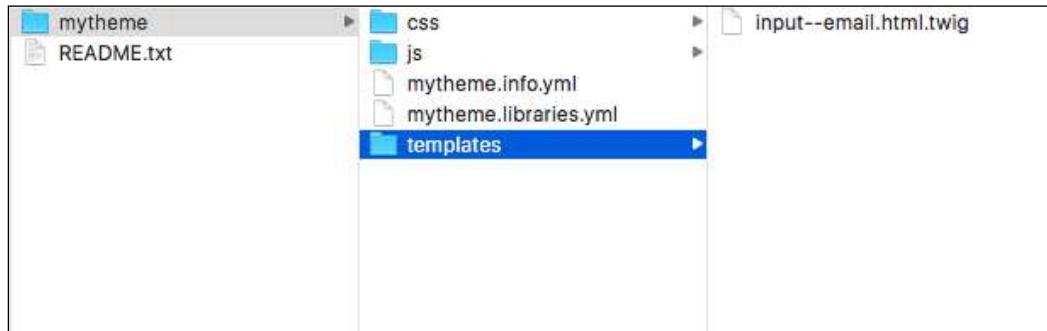
At the time of writing this module, the Classy theme does not provide a template suggestion for the e-mail input nor any customizations to the input template that differ from core.



## How to do it...

1. Create a `template` folder in your theme's base directory to hold your Twig templates.
2. To begin, you need to copy the `input.html.twig` file from `core/modules/system/templates/input.html.twig` to your theme's `template` folder.

3. Rename the `input.html.twig` file to `input--email.html.twig` in order to use the proper theme hook suggestion, as shown in the following screenshot:



4. We will use the `addClass` twig function to add an `input_email` class:

```
<input{{ attributes.addClass('input_email') }}/>{{ children }}
```

5. Above the previous line, we will create a Twig variable using ternary operators to provide a customer placeholder:

```
{% set placeholder = attributes.placeholder ? attributes.
placeholder : 'email@example.com' %}
```

This creates a new variable called `placeholder` using the `set` operator. The question mark (?) operator checks whether the `placeholder` property is empty in the `attributes` object. If it is not empty, it uses the existing value. If the value is empty, it provides a default value.

6. Go to the **Configuration** tab and then to **Development** to rebuild Drupal's cache. We need to do this because Drupal caches the generated Twig output. Any changes made to a Twig template require a cache rebuild.
7. View an `email` field or `form` element and find the modification:

```
<!-- END OUTPUT from 'core/themes/classy/templates/form/form-element-label.html.twig' -->
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'input_email' -->
<!-- FILE NAME SUGGESTIONS:
 x input--email.html.twig
 x input--email.html.twig
 * input.html.twig
-->
<!-- BEGIN OUTPUT from 'sites/default/themes/mytheme/templates/input--email.html.twig' -->
<input data-drupal-selector="edit-field-email-0-value" type="email" id="edit-field-email-0-value" name="field_email[0]
[value]" value size="60" maxlength="254" placeholder="email@example.com" class="form-email input_email">
<!-- END OUTPUT from 'sites/default/themes/mytheme/templates/input--email.html.twig' -->
```

## How it works...

Drupal's theme system is built around hooks and hook suggestions. The element definition of the e-mail input element defines the `input_email` theme hook. If there is no `input_email` hook implemented through a Twig template or PHP function, it will step down to just `input`.



Drupal theme hooks are defined with underscores (\_) but use hyphens (-) when used in Twig template files.



A processor, such as Drupal's theme layer, passes variables to Twig. Variables or properties of objects can be printed by wrapping the variable name with curly brackets. All of core's default templates provide information in the file's document block that details the available Twig variables.

Twig has a simplistic syntax with basic logic and functions. The `addClass` method will take the `attributes` variable and add the class provided in addition to the existing contents.

When providing a theme hook suggestion or altering an existing template, you will need to rebuild Drupal's cache. The compiled Twig template, as PHP, is cached by Drupal so that Twig does not need to compile each time the template is invoked.

## There's more...

### Security first

Twig automatically escapes the output by default, making Drupal 8 one of the most secure versions yet. For Drupal 7, as a whole, most security advisors were for **cross-site scripting (XSS)** vulnerabilities in contributed projects. With Drupal core, using Twig, these security advisories should be severely reduced.

### Theme hook suggestions

Drupal utilizes theme hook suggestions as ways to allow output variations based on different conditions. It allows site themes to provide a more specific template for certain instances.

When a theme hook has double underscores (\_), Drupal's theme system understands this, and it can break apart the theme hook to find a more generic template. For instance, the e-mail element definition provides `input_email` as its theme hook. Drupal understands this as follows:

- ▶ Look for a Twig template named `input--email.html.twig` or a theme hook that defines `input_email`

- ▶ If you are not satisfied, look for a Twig template named `input.html.twig` or a theme hook that defines the input

Theme hook suggestions can be provided by the `hook_theme_suggestions()` hook in a `.module` or `.theme` file.

## Debugging template file selection and hook suggestions

Debugging can be enabled to inspect the various template files that make up a page and their theme hook suggestions, and check which are active. This can be accomplished by editing the `sites/default/services.yml` file. If a `services.yml` file does not exist, copy the `default.services.yml` to create one.

You need to change `debug: false` to `debug: true` under the `twig.config` section of the file. This will cause the Drupal theming layer to print out the source code comments containing the template information. When debug is on, Drupal will not cache the compiled versions of Twig templates and render them on the fly.

There is another setting that prevents you from having to rebuild Drupal's cache on each template file change, but do not leave debug enabled. The `twig.config.auto_reload` boolean can be set to `true`. If this is set to `true`, the Twig templates will be recompiled if the source code changes.

## The Twig logic and operators

The Twig has ternary operators for logic. Using a question mark (?), we can perform a basic *is true or not empty* operation, whereas a question mark and colon (? :) performs a basic *is false or is empty* operation.

You may also use the `if` and `else` logic to provide different outputs based on variables.

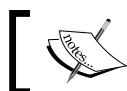
## See also

- ▶ Refer to the Twig documentation at <http://twig.sensiolabs.org/documentation>
- ▶ Refer to the API documentation for `hook_theme_suggestions` at [https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Render%21theme.api.php/function/hook\\_theme\\_suggestions\\_HOOK/8](https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Render%21theme.api.php/function/hook_theme_suggestions_HOOK/8)

## Using the Breakpoint module

The Breakpoint module provides a method for creating media query breakpoint definitions within Drupal. These can be used by other components, such as the responsive image and toolbar modules, to make Drupal responsive.

Breakpoints are a type of plugin that can be defined in a module's or theme's `breakpoints.yml` in its directory. In this recipe, we will define three different breakpoints under a custom group.



Breakpoints are defined solely in YAML files from installed modules and themes and are not configurable through the user interface.

### Getting ready

Ensure that the Breakpoint module is enabled. If you have used the standard Drupal installation, the module is enabled.

This recipe assumes that you have a custom module created. When you see `mymodule`, use the machine name of the module that you created.

### How to do it...

1. Create `mymodule.breakpoints.yml` in your module's base directory. This file will hold the breakpoint configurations.
2. Firstly, we will add a standard mobile breakpoint that does not have a media query, following mobile first practices:

```
mymodule.mobile:
 label: Mobile
 mediaQuery: ''
 weight: 0
```

3. Secondly, we will create a standard breakpoint that will run on a larger viewport:

```
mymodule.standard:
 label: Standard
 mediaQuery: 'only screen and (min-width: 60em)'
 weight: 1
```

4. Thirdly, we will create a wide breakpoint for devices that have a large viewport:

```
mymodule.wide:
 label: Wide
```

```
mediaQuery: 'only screen and (min-width: 70em)'
weight: 2
```

5. Go to the **Configuration** tab and then to **Development** to rebuild Drupal's cache and make the system aware of the new breakpoints.

## How it works...

The Breakpoint module defines the breakpoint configuration entity. Breakpoints do not have any specific form of direct functionalities beyond providing a way to save media queries and grouping them.

The Breakpoint module provides a default manager service. This service is used by other modules to discover breakpoint groups and then all of the breakpoints within a group.

## There's more...

### Caveat for providing breakpoints from themes

Themes have the ability to provide breakpoints; however, they cannot be automatically discovered if new ones are added once they have been installed. Drupal only reads breakpoints provided by themes when a theme is either installed or uninstalled.

Inside `breakpoint.manager`, there are two hooks: one for the theme `install` and one for the theme `uninstall`. Each hook retrieves the breakpoint manager service and rebuilds the breakpoint definitions. Without any extra deployment steps, new breakpoints added to a theme will not be discovered unless these hooks are fired.

### Accessing breakpoints programmatically

Breakpoints are utility configurations for other modules. Breakpoints can be loaded by using the breakpoint manager service and specifying a group. For example, the following code returns all breakpoints used by the Toolbar module:

```
\Drupal::service('breakpoint.manager')
 getBreakpointsByGroup('toolbar');
```

This code invokes the Drupal container to return the service to manage breakpoints, which, by default, is `\Drupal\breakpoint\BreakpointManager`. The `getBreakpointsByGroup` method returns all breakpoints within a group, which are initiated as the `\Drupal\breakpoint\BreakpointInterface` objects.

The Toolbar element class utilizes this workflow to push the breakpoint media query values as JavaScript settings for the JavaScript model to interact with.

## Multippliers

The multipliers value is used to support pixel resolution multipliers. This multiplier is used in coordination with *retina* displays. It is a measure of the viewport's device resolution as a ratio of the device's physical size and independent pixel size. The following is an example of standard multipliers:

- ▶ 1x is normal
- ▶ 1.5x supports Android
- ▶ 2x supports Mac retina devices

### See also

- ▶ To work with breakpoints in Drupal 8, refer to the community documentation at <https://www.drupal.org/documentation/modules/breakpoint>

## Using the Responsive Image module

The Responsive Image module provides a field formatter for image fields that use the HTML5 picture tag and source sets. Utilizing the Breakpoint module, mappings to breakpoints are made to denote an image style to be used at each breakpoint.

The responsive image field formatter works with using a defined responsive image style. Responsive image styles are configurations that map image formats to specific breakpoints and modifiers. First, you need to define a responsive image style, and then you can apply it to an image field.

In this recipe, we will create a responsive image style set called `Article image` and apply it to the `Article` content type's image field.

### Getting ready

You will need to enable the `Responsive Image` module as it is not automatically enabled with the standard installation.

### How to do it...

1. Go to **Configuration** and then to **Responsive image styles** under the **Media** section. Click on **Add responsive image style** to begin creating a new style set.
2. Provide a label that will be used to administratively identify the **Responsive image style** set.

3. Select a breakpoint group that will be used as a source of breakpoints to define the image style map.
4. Each breakpoint will have a **fieldset**. Expand the **fieldset** and **select a single image style**, and then, pick an appropriate image style:

The screenshot shows the 'Breakpoint group' configuration page. It lists two breakpoint groups: '1X WIDE [ALL AND (MIN-WIDTH: 851PX)]' and '1X NARROW [ALL AND (MIN-WIDTH: 560PX) AND (MAX-WIDTH: 850PX)]'. Each group has a 'Type' section with three options: 'Select multiple image styles and use the sizes attribute.', 'Select a single image style.' (which is selected), and 'Do not use this breakpoint.'. Below each type section is a link to the 'Responsive Image help page'. Each group also has an 'Image style' section with a dropdown menu. In the '1X WIDE' group, 'Large (480x480)' is selected. In the '1X NARROW' group, 'Medium (220x220)' is selected. A blue box highlights the 'Medium (220x220)' option in the second group's dropdown.

**Breakpoint group \***

Bartik

Select a breakpoint group from the installed themes and modules.

▼ 1X WIDE [ALL AND (MIN-WIDTH: 851PX)]

Type

Select multiple image styles and use the sizes attribute.

Select a single image style.

Do not use this breakpoint.

See the [Responsive Image help page](#) for information on the sizes attribute.

Image style

Large (480x480)

Select an image style for this breakpoint.

▼ 1X NARROW [ALL AND (MIN-WIDTH: 560PX) AND (MAX-WIDTH: 850PX)]

Type

Select multiple image styles and use the sizes attribute.

Select a single image style.

Do not use this breakpoint.

See the [Responsive Image help page](#) for information on the sizes attribute.

Image style

Medium (220x220)

Select an image style for this breakpoint.

5. Additionally, choose a fallback image style in the event of a browser that doesn't support source sets, such as Internet Explorer 8.

6. Click on **Save** to save the configuration, and add the new style set:

| LABEL         | MACHINE NAME  | OPERATIONS           |
|---------------|---------------|----------------------|
| Article image | article_image | <a href="#">Edit</a> |

7. Go to **Structure** and **Content types**, and select **Manage Display** from the **Article** content type's drop-down menu.
8. Change the **Image** field's formatter to **Responsive image**.
9. Click on the **Settings** tab of the field formatter to choose your new **Responsive image style** set. Select **Article image** from the **Responsive image style** dropdown:

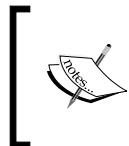
10. Click on **Update** to save the field formatter settings, and then click on **Save** to save the field display settings.

## How it works...

The **Responsive image style** provides three components: a responsive image element, the responsive image style configuration entity, and the responsive image field formatter. The configuration entity is consumed by the field formatter and displayed through the responsive image element.

The responsive image style entity contains an array of breakpoints to image style mappings. The available breakpoints are defined by the selected breakpoint groups. Breakpoint groups can be changed anytime; however, the previous mappings will be lost.

The responsive image element prints a `picture` element with each breakpoint defining a new source element. The breakpoint's media query value is provided as the `media` attribute for the element.



For Internet Explorer 9, Drupal 8 ships with the `picturefill` polyfill. Internet Explorer 9 does not recognize source elements wrapped by a `picture` element. The polyfill wraps the sources around a `video` element within the `picture` element.



## There's more...

### Performance first delivery

A benefit of using the responsive image formatter is performance. Browsers will only download the resources defined in the `srcset` of the appropriate source tag. This not only allows you to deliver a more appropriate image size but also a smaller payload on smaller devices.

### Removing picturefill polyfill

The Responsive Image module attaches the `picturefill` library to the responsive image element definition. The element's template also provides HTML to implement the polyfill. The polyfill can be removed by overriding the element's template and overriding the `picturefill` library to be disabled.

The following snippet, when added to a theme's `info.yml`, will disable the `picturefill` library:

```
libraries-override:
 core/picturefill: false
```

Then, the `responsive-image.html.twig` must be overridden by the theme to remove the extra HTML generated in the template for the polyfill:

1. Copy `responsive-image.html.twig` from `core/modules/responsive_image/templates` to the theme templates folder.
2. Edit `responsive-image.html.twig` and delete the Twig comment and IE conditional to output the initial `video` tag.
3. Remove the last conditional, which provides the closing `video` tag.

## **See also**

- ▶ Refer to the picture element on the Mozilla Developer Network at <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/picture>
- ▶ Refer to picturefill for IE9 at <http://scottjehl.github.io/picturefill/#ie9>



# 6

## Creating Forms with the Form API

In this chapter, we will explore the various recipes to work with forms in Drupal:

- ▶ Creating a form
- ▶ Using new HTML5 elements
- ▶ Validating form data
- ▶ Processing submitted form data
- ▶ Altering other forms

### Introduction

Drupal provides a robust API for creating and managing forms without writing any HTML. Drupal handles form building, validation, and submission. Drupal handles the request to either build the form or process the HTTP POST request. This allows developers to simply define the elements in a form, provide any additional validation if needed, and then handle a successful submission through specific methods.

This chapter contains various recipes to work with forms in Drupal through the Form API. In Drupal 8, forms and form states are objects.

## Creating a form

In this recipe, we will create a form, which will be accessible from a menu path. This will involve creating a route that tells Drupal to invoke our form and display it to the end user.

Forms are defined as classes, which implement `\Drupal\Core\Form\FormInterface`. The `\Drupal\Core\Form\FormBase` serves as a utility class that is intended to be extended. We will extend this class to create a new form.

### Getting ready

Since we will be writing the code, you will want to have a custom module. Creating a custom module in Drupal is simply creating a folder and an `info.yml` file. For this recipe, we will create a folder under `/modules` in your Drupal folder called `drupalform`.

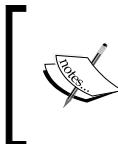
In the `drupalform` folder, create `drupalform.info.yml`. The `info.yml` file is what Drupal will parse to discover modules. An example of a module's `info.yml` file is as follows:

```
name: Drupal form example
description: Create a basic Drupal form, accessible from a route
type: module
version: 1.0
core: 8.x
```

The name will be your module's name, and the description will be listed on the Extend page. Specifying the core tells Drupal what version of Drupal it is built for. *Chapter 4, Extending Drupal* covers how to create a module in depth.

### How to do it...

1. Create an `src` folder in your module directory. In this directory, create a `Form` directory, which will hold the class that defines your form.
2. Next, create a file called `ExampleForm.php` in your module's `src/Form` directory.



Drupal utilizes PSR4 to discover and autoload classes. For brevity, this defines that there should be one class per file, with each filename matching the class name. The folder structure will also mimic the namespace expected.

3. We will edit the `ExampleForm.php` file and add the proper PHP namespace, classes used, and the class itself:

```
<?php

/**
 * @file
 * Contains \Drupal\drupalform\Form\ExampleForm.
 **/

namespace Drupal\drupalform\Form;

use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;

class ExampleForm extends FormBase {
```

```
}
```

The namespace defines the class in your module's `Form` directory. The autoloader will now look into the `drupalform` module path and load the `ExampleForm` class from the `src/Form` directory.

The `use` statement allows us to use just the class name when referencing `FormBase` and, in the next steps, `FormStateInterface`. Otherwise, we would be forced to use the fully qualified namespace path for each class whenever it is used.

4. `\Drupal\Core\Form\FormBase` is an abstract class and requires us to implement four remaining interface methods: `getFormId`, `buildForm`, `validateForm`, and `submitForm`. The latter two are covered in their own recipes; however, we will need to define the method stubs:

```
class ExampleForm extends FormBase {
```

```
 /**
 * {@inheritDoc}
 */
```

```
 public function getFormId() {
 return 'drupalform_example_form';
 }
```

```
 /**
 * {@inheritDoc}
 */
```

```
 public function buildForm(array $form, FormStateInterface $form_state) {
```

```
// Return array of Form API elements.
}

/**
 * {@inheritDoc}
 */
public function validateForm(array &$form, FormStateInterface
$form_state) {
 // Validation covered in later recipe, required to satisfy
interface
}

/**
 * {@inheritDoc}
 */
public function submitForm(array &$form, FormStateInterface
$form_state) {
 // Validation covered in later recipe, required to satisfy
interface
}
}
```

- This code flushes out the initial class definition from the previous step. `FormBase` provides utility methods and does not satisfy the interface requirements for `FormStateInterface`. We define those here, as they are unique across each form definition.
  - The `getFormId` method returns a unique string to identify the form, for example, `site_information`. You may encounter some forms that append `_form` to the end of their form ID. This is not required, and it is just a naming convention often found in previous versions of Drupal.
  - The `buildForm` method is covered in the following steps. The `validateForm` and `submitForm` methods are both called during the Form API processes and are covered in later recipes.
5. The `buildForm` method will be invoked to return Form API elements that are rendered to the end user. We will add a simple text field to ask for a company name and a submit button:

```
/**
 * {@inheritDoc}
 */
public function buildForm(array $form, FormStateInterface $form_
state) {
```

```
$form['company_name'] = array(
 '#type' => 'textfield',
 '#title' => $this->t('Company name'),
);
$form['submit'] = array(
 '#type' => 'submit',
 '#value' => $this->t('Save'),
);
return $form;
}
```

We have added a form element definition to the `form` array. Form elements are defined with a minimum of a type to specify what the element is and a title to act as the label. The title uses the `t` method to ensure that it is translatable.

Adding a **submit** button is done by providing an element with the type `submit`.

6. To access the form, we will create `drupalform.routing.yml` in the module's folder. A route entry will be created to instruct Drupal to use `\Drupal\Core\Form\FormBuilderInterface` to create and display our form:

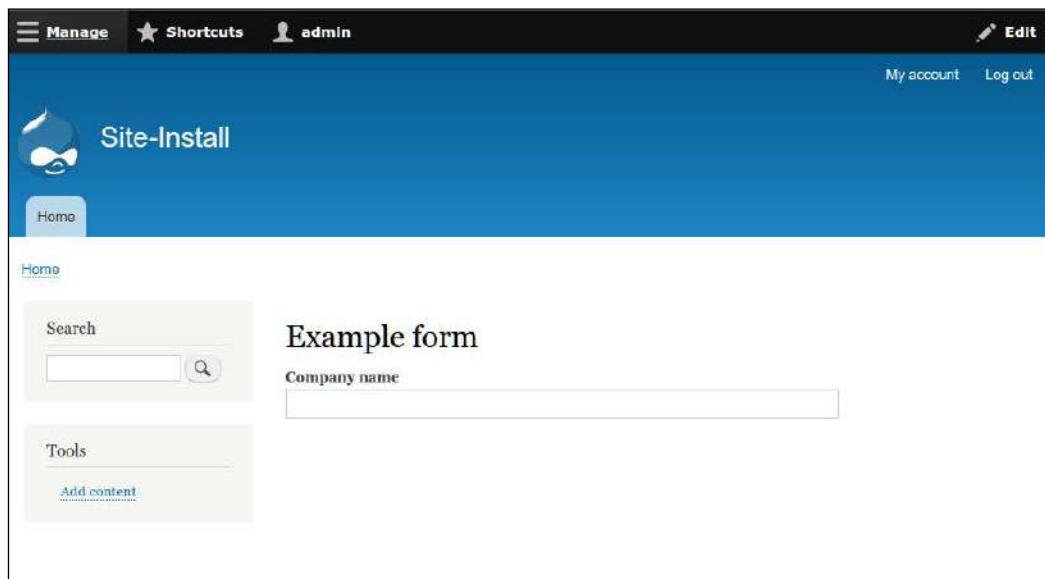
```
drupalform.form:
 path: '/drupal-example-form'
 defaults:
 _title: 'Example form'
 _form: '\Drupal\drupalform\Form\ExampleForm'
 requirements:
 _access: 'TRUE'
```

In Drupal, all routes have a name, and this example defines it as `drupalform.form`. Routes then define a path attribute and override default variables. This route definition has altered the route's title, specified it as a form, and given the fully qualified namespace path to this form's class.

Routes need to be passed a `requirements` property with specifications or else the route will be denied access.

7. Visit the **Extend** page and enable the Drupal form example module that we created.

8. Visit /drupal-example-form and the form is now visible, as shown in the following screenshot:



## How it works...

This recipe creates a route to display the form. By passing the `_form` variable in the defaults section of our route entry, we are telling the route controller how to render our route's content. The fully qualified class name, which includes the namespace, is passed to a method located in the form builder. The route controller will invoke `\Drupal::formBuilder()->getForm(\Drupal\drupalform\Form\ExampleForm)` based on the recipe. At the same time, this can be manually called to embed the form elsewhere.

A form builder instance that implements `\Drupal\Core\Form\FormBuilderInterface` will then process the form by calling `buildForm` and initiate the rendering process. The `buildForm` method is expected to return an array of form elements and other API options. This will be sent to the render system to output the form as HTML.

## There's more...

Many components make up a form created through Drupal's Form API. We will explore a few of them in depth.

## Form element definitions

A form is a collection of form elements, which are types of plugin in Drupal 8. Plugins are small pieces of swappable functionalities in Drupal 8. Plugins and plugin development are covered in *Chapter 7, Plug and Play with Plugins*. At the time of writing this module, the Drupal.org Form API reference table was severely out of date and did not reflect all of the form element types available.

Here are some of the most common element properties that can be used:

- ▶ `weight`: This is used to alter the position of a form element in a form. By default, elements will be displayed in the order in which they were added to the form array. Defining a weight allows a developer to control element positions.
- ▶ `default_value`: This gives a developer the ability to prefill the element with a value. For example, when building configuration forms that have existing data or when editing an entity.
- ▶ `placeholder`: This is new to Drupal 8. Drupal 8 provides a new HTML5 support, and this attribute will set the placeholder attribute on the HTML input.

## The form state

The `\Drupal\Core\Form\FormStateInterface` object represents the current state of the form and its data. The form state contains user-submitted data for the form along with build state information. Redirection after form submission is handled through the form state as well. You will interact more with the form state during the validation and submission recipes.

## The form cache

Drupal utilizes a cache table for forms. This holds the build table, as identified by form build identifiers. This allows Drupal to validate forms during AJAX requests and easily build them when required. It is important to keep the form cache in persistent storage; otherwise, there may be repercussions, such as loss of form data or invalidating forms.

### See also

- ▶ Form API in Drupal 8 at <https://www.drupal.org/node/2117411>
- ▶ The Drupal 8 Form API reference at [https://api.drupal.org/api/drupal/developer!topics!forms\\_api\\_reference.html/8](https://api.drupal.org/api/drupal/developer!topics!forms_api_reference.html/8)
- ▶ *Chapter 4, Extending Drupal*
- ▶ *Chapter 7, Plug and Play with Plugins*, to learn more about derivatives

## Using new HTML5 elements

With the release of Drupal 8, Drupal has finally entered into the realm of HTML5. The Form API now allows utilization of HTML5 input elements out of the box. These include the following element types:

- ▶ tel
- ▶ email
- ▶ number
- ▶ date
- ▶ url
- ▶ search
- ▶ range

This allows your forms in Drupal to leverage native device input methods along with native validation support.

### Getting ready

This recipe will walk you through adding elements to a Drupal form. You will need to have a custom form implemented through a module, such as the one created in the *Creating a form* section.

### How to do it...

1. In order to use the telephone input, you need to add a new `form` element definition of the `tel` type to your `buildForm` method:

```
$form['phone'] = array(
 '#type' => 'tel',
 '#title' => t('Phone'),
);
```

2. In order to use the e-mail input, you need to add a new `form` element definition of the `email` type to your `buildForm` method. It will validate the format of e-mail addresses in the Form API:

```
$form['email'] = array(
 '#type' => 'email',
 '#title' => t('Email'),
);
```

3. In order to use the number input, you need to add a new `form` element definition of the number type to your `buildForm` method. It will validate the range and format of the number:

```
$form['integer'] = array(
 '#type' => 'number',
 '#title' => t('Some integer'),
 // The increment or decrement amount
 '#step' => 1,
 // Minimum allowed value
 '#min' => 0,
 // Maximum allowed value
 '#max' => 100,
);
```

4. In order to use the date input, you need to add a new `form` element definition of the date type to your `buildForm` method. You can also pass the `#date_date_format` option to alter the format used by the input:

```
$form['date'] = array(
 '#type' => 'date',
 '#title' => t('Date'),
 '#date_date_format' => 'Y-m-d',
);
```

5. In order to use the URL input, you need to add a new `form` element definition of the `url` type to your `buildForm` method. The element has a validator to check the format of the URL:

```
$form['website'] = array(
 '#type' => 'url',
 '#title' => t('Website'),
);
```

6. In order to use the search input, you need to add a new `form` element definition of the `search` type to your `buildForm` method. You can specify a route name that the search field will query for autocomplete options:

```
$form['search'] = array(
 '#type' => 'search',
 '#title' => t('Search'),
 '#autocomplete_route_name' => FALSE,
);
```

7. In order to use the `range` input, you need to add a new `form` element definition of the `range` type to your `buildForm` method. It is an extension of the `number` element and accepts a `min`, `max`, and `step` property to control the values of the range input:

```
$form['range'] = array(
 '#type' => 'range',
 '#title' => t('Range'),
 '#min' => 0,
 '#max' => 100,
 '#step' => 1,
) ;
```

## How it works...

Each type references an extended class of `\Drupal\Core\Render\Element\FormElement`. It provides the element's definition and additional functions. Each element defines a `prerender` method in the class that defines the `input` type attribute along with other additional attributes.

Each input defines its theme as `input__TYPE`, allowing you to copy the `input.html.twig` base to `input.TYPE.html.twig` for templating. The template then parses the attributes and renders the HTML.

Some elements, such as e-mails, provide validators for the element itself. The `e-mail` element defines the `validateEmail` method. Here is an example of the code from `\Drupal\Core\Render\Element>Email::validateEmail`:

```
/**
 * Form element validation handler for #type 'email'.
 *
 * Note that #maxlength and #required is validated by _form_
 * validate() already.
 */
public static function validateEmail(&$element, FormStateInterface
$form_state, &$complete_form) {
 $value = trim($element['#value']);
 $form_state->setValueForElement($element, $value);

 if ($value !== '' && !\Drupal::service('email.validator')-
>isValid($value)) {
 $form_state->setError($element, t('The email address %mail is
not valid.', array('%mail' => $value)));
 }
}
```

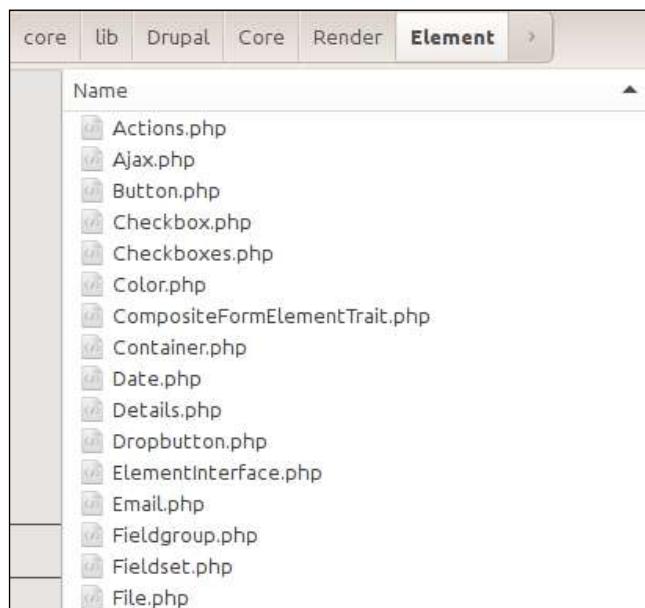
This code will be executed on form submission and validate the provider's e-mail. It does this by taking the current value and trimming any whitespaces and using the form state object to update the value. The `email.validator` service is invoked to validate the e-mail. If this method returns `false`, the form state is invoked to mark the element as the one that has an error. If the element has an error, the form builder will prevent form submission, returning the user to the form to fix the value.

### There's more...

Elements are provided through Drupal's plugin system and are explored in detail in the next section.

### Specific element properties

Elements can have their own unique properties along with individual validation methods. At the time of writing, the Drupal 8 Form Reference table is incomplete and does not highlight these new elements nor their properties. However, the classes can be examined and the definition method can be read to learn about the properties of each element. These classes are under the `\Drupal\Core\Render\Element` namespace located in `/core/lib/Drupal/Core/Render/Element`:



## **Creating new elements**

Each element used in the Form API extends the \Drupal\Core\Render\Element\FormElement class, which is a plugin. Modules can provide new element types by adding classes to their Plugins/Element namespace. Refer to *Chapter 7, Plug and Play with Plugins* for more information on how to implement a plugin.

### **See also**

- ▶ Form API in Drupal 8 at <https://www.drupal.org/node/2117411>
- ▶ *Chapter 7, Plug and Play with Plugins*

## **Validating form data**

The Form API requires all form classes to implement the \Drupal\Core\Form\FormInterface. The interface defines a validation method. The validateForm method is invoked once a form has been submitted and provides a way to validate the data and halt the processing of the data if required. The form state object provides methods for marking specific fields as having the error, providing a user experience tool to alert your users specifically to the problem input.

This recipe will be based on the custom module and form created in the *Creating a form* section of this chapter. We will be validating the length of the submitted field.

### **Getting ready**

This recipe will be using the module and custom form created in the first *Creating a form* recipe.

### **How to do it...**

1. Open and edit the \Drupal\drupalform\Form\ExampleForm class in the src/Form directory of the module.
2. Before validating the company\_name value, we need to check whether the value is empty using the isValueEmpty() method from the \Drupal\Core\Form\FormStateInterface object:

```
/**
 * {@inheritDoc}
 */
public function validateForm(array &$form, FormStateInterface
$form_state) {
 if (!$form_state->isValueEmpty('company_name')) {
```

```

 // Value is set, perform validation
 }
}

```

3. The `\Drupal\Form\FormStateInterface::isEmpty` method takes the key name of the form element. For example, `$form['company_name']` from the `buildForm` method is referenced through `company_name` in the `isEmpty` method.
4. Next, we will check whether the value's length is greater than five:

```

/**
 * {@inheritDoc}
 */
public function validateForm(array &$form, FormStateInterface
$form_state) {
 if (!empty($form_state->isEmpty('company_name'))) {
 if (strlen($form_state->getValue('company_name')) <= 5) {
 // Set validation error.
 }
 }
}

```

The `getValue` takes a form element's key and returns the value. Since we have already verified that the value is not empty, we can retrieve the value.



If you had any experience with previous versions of Drupal, note that the form state is now an object and not an array.

5. If the logic check finds a value with a length of five or fewer characters, it will throw a form error to prevent submission:

```
$form_state->setErrorByName('company_name', t('Company name is less than 5 characters'));
```

We can place the `setErrorByName` method in our `strlen` logic check. If the string is fewer than five characters, an error is set on the element. The first parameter is the element's key and the second parameter is the message to be presented to the user.

6. The entire validation method will resemble the following code:

```

/**
 * {@inheritDoc}
 */
public function validateForm(array &$form, FormStateInterface
$form_state) {
 if (!empty($form_state->isEmpty('company_name'))) {

```

## Creating Forms with the Form API

---

```
if (strlen($form_state->getValue('company_name')) <= 5) {
 $form_state->setErrorByName('company_name', t('Company
name is less than 5 characters'));
}
```

7. When the form is submitted, the **Company name** text field will have to have more than five characters or be empty in order to be submitted.

The screenshot shows a Drupal administrative interface. At the top, a red header bar displays the message "1 error has been found: Company name". Below this, the page title is "Example form". On the left, there's a sidebar with links for "Home", "Search" (with a search input field), "Tools" (with a "Save" button), and "Add content". The main content area contains a form field labeled "Company name" with the value "Four". This field is highlighted with a red border, indicating it is invalid. The overall layout is clean and follows the standard Drupal design conventions.

## How it works...

Before the form builder service invokes the form object's `submitForm` method, it invokes the object's `validateForm` method. In the validation method, the form state can be used to check values and perform logic checks. In the event that an item is deemed *invalid* and an error is set on an element, the form cannot submit and will show errors to the user.

When an error is added to an element, an overall counter for the number of errors on the form is incremented. If the form has any errors, the form builder service will not execute the `submit` method.

This process is executed through the `\Drupal\Core\Form\FormValidator` class, which is run through the form builder service.

## There's more...

## Multiple validation handlers

A form can have multiple validation handlers. By default, all forms come with at least one validator, which is its own `validateForm` method. There is more that can be added. However, by default, the form will merely execute `::validateForm` and all element validators. This allows you to invoke static methods on other classes or other forms.

If a class provides `method1` and `method2`, which it would like to execute as well, the following code can be added to the `buildForm` method:

```
$form_state->setValidateHandlers([
 [':::validateForm'],
 [':::method1'],
 [$this, 'method2'],
]);
```

This sets the validator array to execute the default `validateForm` method and the two additional methods. You can reference a method in the current class using two colons (`:::`) and the method name. Or, you can use an array consisting of a class instance and the method to invoke.

## Accessing multidimensional array values

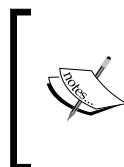
Forms support nested form elements in the form array. The default `\Drupal\Core\Form\FormStateInterface` implementation, `\Drupal\Core\Form\FormState`, supports accessing multidimensional array values. Instead of passing a string, you can pass an array that represents the parent array structure in the form array.

If the element is defined in `$form['company']['company_name']`, then we will pass `array('company', 'company_name')` to the form state's methods.

## Element validation methods

Form elements can have their own validators. The form state will aggregate all of the element validation methods and pass them to the form validation service. This will run with the form's validation.

There is a `limit_validation_errors` option, which can be set to allow selected invalid errors to be passed. This option allows you to bypass validation on specific elements in your form. This attribute is defined in the submit button, also known as the **triggering** element in the form state. It is an array value consisting of form element keys.



The triggering element value does not operate in the same fashion as the form state's methods in order to access multidimensional array values. In order to access a nested value, you need to provide a partially constructed string, representing the nested value. For example, `$form['company']['company_name']` will have to be added as `company] [company_name]`.

## Processing submitted form data

A form's purpose is to collect data and do something with the data that was submitted. All forms need to implement the `\Drupal\Core\Form\FormInterface`. The interface defines a submit method. Once the Form API has invoked the class's validation method, the submit method can be run.

This recipe will be based on the custom module and form created in the *Creating a form* recipe of this chapter. We will convert the form to `\Drupal\Core\Form\ConfigBaseForm`, allowing easy storage of the field element.

### Getting ready

In this recipe, we will be using the module and custom form created in the first *Creating a form* recipe.

### How to do it...

1. In your module's directory, create a `config` directory, and then create a directory inside it named `install`.
2. Create a file named `drupalform.schema.yml`; this file will tell Drupal about the configuration item that we want to save.
3. Add the following configuration schema definition to `drupalform.schema.yml`:

```
drupalform.company:
 type: config_object
 label: 'Drupal form settings'
 mapping:
 company_name:
 type: string
 label: 'A company name'
```

This tells Drupal that we have the configuration with the name `drupalform.company` and it has a valid option of `company_name`. We will cover this in more detail in *Chapter 9, Configuration Management – Deploying in Drupal 8*.

4. Replace the `FormBase` use statement to use the `ConfigFormBase` class:

```
<?php

/**
 * @file
 * Contains \Drupal\drupalform\Form\ExampleForm.
 */
```

```
namespace Drupal\drupalform\Form;

use Drupal\Core\Form\ConfigFormBase;
use Drupal\Core\Form\FormStateInterface;

5. Update the ExampleForm class to extend ConfigFormBase instead of harness
its implementations:
```

```
class ExampleForm extends ConfigFormBase {
```

This allows us to reuse methods from the ConfigFormBase class and write less about our own implementation.

- ```
6. For ExampleForm to implement ConfigFormBase, the
getEditableConfigNames method needs to be implemented to
satisfy the \Drupal\Core\Form\ConfigBaseTrait trait:
```

```
/**
 * {@inheritDoc}
 */
protected function getEditableConfigNames() {
  return ['drupalform.company'];
}

/**
 * {@inheritDoc}
 */
public function getFormId() {
  return 'drupalform_example_form';
}
```

This function defines the configuration names, which will be editable by the form. This brings all the configurations under drupalform [company] to be editable when accessed through the form with the config method provided by ConfigFormBaseTrait.

- ```
7. Remove the submit form element. Update the buildForm method to return data
from the parent's method rather than from $form itself. We also need to add a
#default_value option to company_name so that it uses an existing value the
next time our form is loaded:
```

```
/**
 * {@inheritDoc}
 */
public function buildForm(array $form, FormStateInterface $form_
state) {
 $config = $this->config('drupalform.company');
 $form['company_name'] = array(
 '#type' => 'textfield',
 '#title' => $this->t('Company name'),
 '#default_value' => $config->get('company_name'),
```

```
) ;

return parent::buildForm($form, $form_state);
}
```

The ConfigFormBase class implements the buildForm method to provide a reusable submit button. It also unifies the presentation across Drupal configuration forms:



The screenshot shows a simple configuration form titled "Example form". It contains a single text input field labeled "Company name" and a "Save configuration" button at the bottom.

8. The ConfigFormBase provides a configuration factory method. We will add a default\_value property to our element with the currently saved item:

```
/**
 * {@inheritDoc}
 */
public function validateForm(array &$form, FormStateInterface
$form_state)
{
// Validation covered in later recipe, required to satisfy
interface
}
```

The #default\_value key is added to the element's definition. It invokes the config method provided by ConfigFormBaseTrait to load our configuration group and access a specific configuration value.

9. The final step is to save the configuration in the submitForm method:

```
/**
 * {@inheritDoc}
 */
public function submitForm(array &$form, FormStateInterface $form_
state) {
$config = $this->config('drupalform.company');
parent::submitForm($form, $form_state);
$config->set('company_name',
$form_state->getValue('company_name'))->save();
}
```

The `config` method is invoked by specifying our configuration group. We then use the `set` method to define name as the value from the `company_name` text field.

10. When you edit your form and click on the **submit** button, the value that you entered in the **Company name** field will now be saved in the configuration.

## How it works...

The `ConfigFormBase` utilizes the `ConfigFormBaseTrait` to provide easy access to a configuration factory. The class's implementation of `buildForm` also adds a `submit` button and theme styling to forms. The submit handler displays a configuration saved message but relies on implementing a module to save the configuration.

The form saves its data under the `drupalform.company` namespace. The company name value is stored as `name` and can be accessed as `drupalform.company.name`. Note that the configuration name does not have to match the form element's key.

## There's more...

### Multiple submit handlers

A form can have multiple submit handlers. By default, all forms implement a submit handler, which is its own `submitForm` method. The form will execute `::submitForm` automatically and any defined on the triggering element. There is more that can be added. However, this allows you to invoke `static` methods on other classes or other forms.

If a class provides `method1` and `method2`, which it would like to execute as well, the following code can be added to the `buildForm` method:

```
$form_state->setSubmitHandlers([
 [':::submitForm'],
 [':::method1'],
 [$this, 'method2']
]);
```

This sets the submit handler array to execute the default `submitForm` method and two additional methods. You can reference a method in the current class using two colons (`::`) and the method name. Or, you can use an array consisting of a class instance and the method to be invoked.

## See also

- ▶ *Chapter 9, Configuration Management- Deploying in Drupal 8*

## Altering other forms

Drupal's Form API does not just provide a way to create forms. There are ways to alter forms through a custom module that allows you to manipulate the core and contributed forms. Using this technique, new elements can be added, default values can be changed, or elements can even be hidden from view to simplify the user experience.

The altering of a form does not happen in a custom class; this is a hook defined in the module file. In this recipe, we will use the `hook_form_FORM_ID_alter()` hook to add a telephone field to the site's configuration form.

### Getting ready

This recipe assumes that you have a custom module to add the code to.

### How to do it...

1. In the modules folder of your Drupal site, create a folder named `mymodule`.
2. In the `mymodule` folder, create a `mymodule.info.yml`, containing the following code:

```
name: My module
description: Custom module that uses a form alter
type: module
core: 8.x
Next, create a .module file in your module's directory:
<?php
```

```
/**
 * @file
 * Custom module that alters forms.
 */
```

As a best practice, files have document block headers that describe the purpose of the file and what it pertains to.

3. Add the `mymodule_form_system_site_information_settings_alter()` hook. The form ID can be found by viewing the form's class and reviewing the `getFormId` method:

```
/**
 * Implements hook_form_FORM_ID_alter().
 */
```

```
function mymodule_form_system_site_information_settings_
alter(&$form, \Drupal\Core\Form\FormStateInterface $form_state) {
 // Code to alter form or form state here
}
```

Drupal will call this hook and pass the current form array and its form state object. The form array is passed by reference, allowing our hook to modify the array without returning any values. This is why the `$form` parameter has the ampersand (&) before it. In PHP, all objects are passed by reference.



When calling a class in a normal file, such as the module file, you need to either use the fully qualified class name or add a use statement at the beginning of the file. In this example, we can add `\Drupal\Core\Form\FormStateInterface`.

4. Next, we add our telephone field to the form so that it can be displayed and saved:

```
/**
 * Implements hook_form_FORM_ID_alter().
 */
function mymodule_form_system_site_information_settings_
alter(&$form, \Drupal\Core\Form\FormStateInterface $form_state) {
 $form['site_phone'] = array(
 '#type' => 'tel',
 '#title' => t('Site phone'),
 '#default_value' => Drupal::config('system.site')-
>get('phone'),
);
}
```

We retrieve the current phone value from `system.site` so that it can be modified if already set.

5. Visit the **Extend** page and enable the module **My module** that we created.

6. Review the **Site Information** form under **Configuration** and test setting the site telephone number:

The screenshot shows the 'Site Information' configuration page under 'Configuration'. It features a 'Site phone' field with a placeholder '(Optional)'. Below the field is a blue 'Save configuration' button.

7. We need to add a submit handler in order to save the configuration for our new field. We will need to add a submit handler to the form and a submit handler callback:

```
/**
 * Implements hook_form_FORM_ID_alter().
 */
function mymodule_form_system_site_information_settings_
alter(&$form, \Drupal\Core\Form\FormStateInterface $form_state) {
 $form['site_phone'] = array(
 '#type' => 'tel',
 '#title' => t('Site phone'),
 '#default_value' => Drupal::config('system.site')-
>get('phone'),
);
 $form['#submit'][] = 'mymodule_system_site_information_phone_
submit';
}

/**
 * Form callback to save site_phone
 * @param array $form
 * @param \Drupal\Core\Form\FormStateInterface $form_state
 */
function mymodule_system_site_information_phone_submit(array
&$form, \Drupal\Core\Form\FormStateInterface $form_state) {
```

```
$config = Drupal::configFactory()->getEditable('system.site');
$config
->set('phone', $form_state->getValue('site_phone'))
->save();
}
```

The `$form['#submit']` modification adds our callback to the form's submit handlers. This allows our module to interact with the form once it has been submitted.

The `mymodule_system_site_information_phone_submit` callback is passed the form array and form state. We load the current configuration factory to receive the configuration that can be edited. We then load `system.site` and save phone based on the value from the form state.

8. Submit the form and verify that the data has been saved.

## How it works...

The `\Drupal\system\Form\SiteInformationForm` class extends `\Drupal\Core\Form\ConfigFormBase` to handle the writing of form elements as individual configuration values. However, it does not write the values automatically to the form state. In this recipe, we need to add a submit handler to manually save our added field.

The form array is passed by reference, allowing modifications to be made in the hook to alter the original data. This allows us to add an element or even modify existing items, such as titles or descriptions.

## There's more...

### Adding additional validate handlers

Using a form alter hook, we can add additional validators to a form. The proper way to do this is to load the current validators and add the new one to the array and reset the validators in the form state:

```
$validators = $form_state->getValidateHandlers();
$validators[] = 'mymodule_form_validate';
$form_state->setValidateHandlers($validators);
```

First, we receive all of the currently set validators from the form state as the `$validators` variable. We then append a new callback to the end of the array. Once the `$validators` variable has been modified, we override the form state's validator array by executing the `setValidateHandlers` method.



You can also use PHP array manipulation functions to add your validators in different execution orders. For example, `array_unshift` will place your validator at the beginning of the array so that it can run first.



## **Adding additional submit handlers**

Using a form alter hook, we can add additional submit handlers to a form. The proper way to do this is to load the current submit handlers, add the new one to the array, and reset the validators in the form state:

```
$submit_handlers = $form_state->getSubmitHandlers();
$submit_handlers [] = 'mymodule_form_submit';
$form_state->setSubmitHandlers($submit_handlers);
```

First, we receive all of the currently set submit handlers from the form state as the `$submit_handlers` variable. We then append a new callback to the end of the array. Once the `$submit_handlers` variable has been modified, we override the form state's submit handler array by executing the `setSubmitHandlers` method.



You can also use PHP array manipulation functions to add your callback in different execution orders. For example, `array_unshift` will place your callback at the beginning of the array so that it can run first.



# 7

## Plug and Play with Plugins

In this chapter, we will dive into the new Plugin API provided in Drupal 8:

- ▶ Creating blocks using plugins
- ▶ Creating a custom field type
- ▶ Creating a custom field widget
- ▶ Creating a custom field formatter
- ▶ Creating a custom plugin type

### Introduction

Drupal 8 introduces plugins. Plugins power many items in Drupal, such as blocks, field types, field formatters, and many more. Plugins and plugin types are provided by modules. They provide a swappable and specific functionality. Breakpoints, as discussed in *Chapter 5, Front End for the Win*, are plugins. In this chapter, we will discuss how plugins work in Drupal 8 and show you how to create blocks, fields, and custom plugin types.

Each version of Drupal has had subsystems, which provided pluggable components and even contributed modules. A problem arose in the implementation and management of these. Blocks, fields, and image styles each had an entirely different system to learn and understand. The Plugin API exists in Drupal 8 to mitigate this problem and provide a base API to implement pluggable components. This has greatly improved the developer experience when working with Drupal core's subsystems. In this chapter, we will implement a block plugin. We will use the Plugin API to provide a custom field type along with a widget and formatter for the field. The last recipe will show you how to create and use a custom plugin type.

## Creating blocks using plugins

In Drupal, a block is a piece of content that can be placed in a region provided by a theme. Blocks are used to present specific kinds of content, such as a user login form, a snippet of text, and many more.

Blocks are configuration entities, and the block module uses the Drupal plugin system as a way to define blocks for modules. Custom blocks are defined in the PHP code in the module's Plugin class namespace. Each class in the `Plugin/Block` namespace will be discovered by the block module's plugin manager.

In this recipe, we will define a block that will display a copyright snippet and the current year, and place it in the footer region.

### Getting ready

Create a new module like the one shown in this recipe. We will refer to the module as `mymodule` throughout the recipe. Use your module's appropriate name.

### How to do it...

1. Create the `src/Plugin/Block` directory in your module. This will translate the `\Drupal\mymodule\Plugin\Block` namespace and allow a block plugin discovery.
2. Create a `Copyright.php` file in the newly created folder so that we can define the `Copyright` class for our block:



3. The `Copyright` class will extend `\Drupal\Core\Block\BlockBase`:

```
<?php

/**
 * @file
```

- ```

 * Contains \Drupal\mymodule\Plugin\Block\Copyright.
 */
namespace Drupal\mymodule\Plugin\Block;
use Drupal\Core\Block\BlockBase;
class Copyright extends BlockBase {
}

4. We extend the BlockBase class, which implements \Drupal\Core\Block\BlockPluginInterface and provides us with an implementation of nearly all of its methods.

5. Blocks are annotated plugins. Annotated plugins use documentation blocks to provide details of the plugin. We will provide the block's identifier, administrative label, and category:
```

```
<?php
```

```

/**
 * @file
 * Contains \Drupal\mymodule\Plugin\Block\Copyright.
 */
```

```

namespace Drupal\mymodule\Plugin\Block;

use Drupal\Core\Block\BlockBase;
/**
 * @Block(
 *   id = "copyright_block",
 *   admin_label = @Translation("Copyright"),
 *   category = @Translation("Custom")
 * )
 */
class Copyright extends BlockBase {
```

```
}
```

- ```

6. The annotation document block of the class identifies the type of plugin through @Block. Drupal will parse this and initiate the plugin with the properties defined inside it. The id is the internal machine name, the admin_label is displayed on the block listing page, and category shows up in the block select list.

7. We need to provide a build method to satisfy the \Drupal\Core\Block\BlockPluginInterface interface. This creates the output to be displayed:
```

```
<?php
/**
 * @file
```

```
* Contains \Drupal\mymodule\Plugin\Block\Copyright
*/
namespace Drupal\mymodule\Plugin\Block;

use Drupal\Core\Block\BlockBase;

/**
 * @Block(
 * id = "copyright_block",
 * admin_label = @Translation("Copyright"),
 * category = @Translation("Custom")
 *)
 */
class Copyright extends BlockBase {
 /**
 * {@inheritDoc}
 */
 public function build() {
 $date = new \DateTime();
 return [
 '#markup' => t('Copyright @year© My Company', [
 '@year' => $date->format('Y'),
]),
];
 }
}
```

The `build` method returns a render array that uses Drupal's `t` function to substitute `@year` for the `\DateTime` object's output that is formatted as a full year.



Since PHP 5.4, a warning will be displayed if you have not explicitly set a timezone in your PHP's configuration.

8. Rebuild Drupal's cache so that the new plugin can be discovered.
9. In the **Footer fourth** region, click on **Place block**.
10. Review the block list and add the custom block to your regions, for instance, the footer region. Find the **Copyright** block, and click on **Place block**:

The screenshot shows the Drupal 8 Block Manager interface. On the left, there are four footer regions listed: 'Footer third' (with 'Place block' button), 'Footer fourth' (with 'Place block' button), 'Footer fifth' (with 'Place block' button), and 'Footer sixth' (with 'Place block' button). To the right of these regions is a list of available blocks categorized by module: 'Primary admin actions' (core), 'Tabs' (core), 'Copyright' (Custom), 'Execute PHP' (Devel), 'Search form' (Forms), and 'Switch user' (Forms). Each block has a 'Place block' button next to it.

11. Uncheck the **Display title** checkbox so that only our block's content can be rendered.
12. Review the copyright statement that will always keep the year dynamic:



## How it works...

The plugin system works through plugin definitions and plugin managers for those definitions. The `\Drupal\Core\Block\BlockManager` class defines the block plugins that need to be located in the `Plugin/Block` namespace. It also defines the base interface that needs to be implemented along with the `Annotation` class, which is to be used, when parsing the class's document block.

When Drupal's cache is rebuilt, all available namespaces are scanned to check whether classes exist in the given plugin namespace. The definitions, via annotation, will be processed and the information will be cached.

Blocks are then retrieved from the manager, manipulated, and their methods are invoked. When viewing the Block layout page to manage blocks, the \Drupal\Core\Block\BlockBase class's `label` method is invoked to display the human-readable name. When a block is displayed on a rendered page, the `build` method is invoked and passed to the theming layer to be output.

## There's more...

### Altering blocks

Blocks can be altered in two different ways: the plugin definition can be altered, the build array, or the view array out.

A module can implement `hook_block_alter` in its `.module` file and modify the annotation definitions of all the discovered blocks. This will allow a module to change the default `user_login_block` from user login to Login:

```
/**
 * Implements hook_block_alter().
 */
function mymodule_block_alter(&$definitions) {
 $definitions['user_login_block']['admin_label'] = t('Login');
}
```

A module can implement `hook_block_build_alter` and modify the build information of a block. The hook is passed the build array and the \Drupal\Core\Block\BlockPluginInterface instance for the current block. Module developers can use this to add cache contexts or alter the cache ability of metadata:

```
/**
 * Implements hook_block_build_alter().
 */
function hook_block_build_alter(array &$build, \Drupal\Core\Block\BlockPluginInterface $block) {
 // Add the 'url' cache the block per URL.
 if ($block->getBaseId() == 'myblock') {
 $build['#contexts'][] = 'url';
 }
}
```



You can test the modification of cache metadata by altering the recipe's block to output a timestamp. With caching enabled, you will see that the value persists on the same URL, but it will be different across each page.

Finally, a module can implement `hook_block_view_alter` in order to modify the output to be rendered. A module can add content to be rendered or remove content. This can be used to remove the contextual links item, which allows inline editing from the front page of a site:

```
/**
 * Implements hook_block_view_alter().
 */
function hook_block_view_alter(array &$build, \Drupal\Core\Block\BlockPluginInterface $block) {
 // Remove the contextual links on all blocks that provide them.
 if (isset($build['#contextual_links'])) {
 unset($build['#contextual_links']);
 }
}
```

## Block settings forms

Blocks can provide a setting form. This recipe provides the text *My Company* for the copyright text. Instead, this can be defined through a text field in the block's setting form.

Let's revisit the `Copyright.php` file that contained our block's class. A block can override the default `defaultConfiguration` method, which returns an array of setting keys and their default values. The `blockForm` method can then override the `\Drupal\Core\Block\BlockBase` empty array implementation to return a Form API array to represent the settings form:

```
/**
 * {@inheritDoc}
 */
public function defaultConfiguration() {
 return [
 'company_name' => '',
];
}

/**
 * {@inheritDoc}
 */
public function blockForm($form, \Drupal\Core\Form\FormStateInterface $form_state) {
```

```
$form['company_name'] = [
 '#type' => 'textfield',
 '#title' => t('Company name'),
 '#default_value' => $this->configuration['company_name'],
];
return $form;
}
```

The `blockSubmit` method must then be implemented, which updates the block's configuration:

```
/**
 * {@inheritDoc}
 */
public function blockSubmit($form, \Drupal\Core\Form\FormStateInterface $form_state) {
 $this->configuration['company_name'] = $form_state-
>getValue('company_name');
}
```

Finally, the `build` method can be updated to use the new configuration item:

```
/**
 * {@inheritDoc}
 */
public function build() {
 $date = new \DateTime();
 return [
 '#markup' => t('Copyright @year© @company', [
 '@year' => $date->format('Y'),
 '@company' => $this->configuration['company_name'],
]),
];
}
```

You can now go back and visit the `Block layout` form, and click on **Configure** in the **Copyright** block. The new setting will be available in the block instance's configuration form.

## Defining access to a block

Blocks, by default, are rendered for all users. The default access method can be overridden. This allows a block to only be displayed to authenticated users or based on a specific permission:

```
/**
 * {@inheritDoc}
 */

```

```

protected function blockAccess(AccountInterface $account) {
 $route_name = $this->routeMatch->getRouteName();
 if ($account->isAnonymous() && !in_array($route_name,
 array('user.login', 'user.logout'))) {
 return AccessResult::allowed()
 ->addCacheContexts(['route.name',
 'user.roles:anonymous']);
 }
 return AccessResult::forbidden();
}

```

The preceding code is taken from the `user_login_block`. It allows access to the block if the user is logged out and is not on the login or logout page. The access is cached based on the current route name and the user's current role being anonymous. If these are not passed, the access returned is forbidden and the block is not built.

Other modules can implement `hook_block_access` to override the access of a block:

```

/**
 * Implements hook_block_access().
 */
function mymodule_block_access(\Drupal\block\Entity\Block $block,
 $operation, \Drupal\Core\Session\AccountInterface $account) {
 // Example code that would prevent displaying the Copyright' block
 // in
 // a region different than the footer.
 if ($operation == 'view' && $block->getPluginId() == 'copyright') {
 return \Drupal\Core\Access\AccessResult::forbiddenIf($block-
 >getRegion() != 'footer');
 }

 // No opinion.
 return \Drupal\Core\Access\AccessResult::neutral();
}

```

A module implementing the preceding hook will deny access to our **Copyright** block if it is not placed in the footer region.

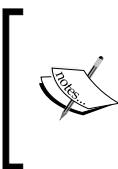
## See also

- ▶ Refer to *Creating a custom plugin type* recipe of this chapter
- ▶ **block.api.php** at <https://api.drupal.org/api/drupal/core%21modules%21block%21block.api.php/8>

## Creating a custom field type

Fields are powered through the plugin system in Drupal. Field types are defined using the plugin system. Each field type has its own class. A new field type can be defined through a custom class that will provide schema and property information.

In this example, we will create a simple field type called "real name" to store the first and last names.



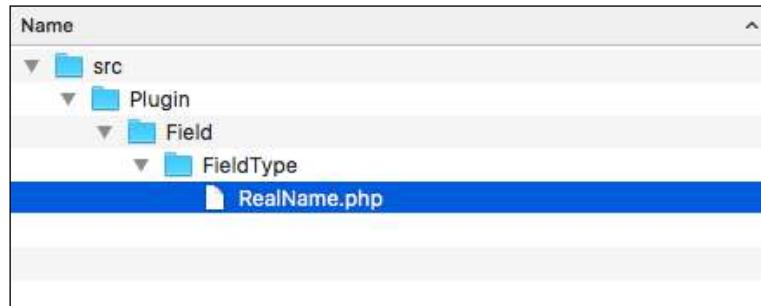
Field types just define ways in which data can be stored and handled through the Field API. Field widgets provide means for editing a field type in the user interface. Field formatters provide means for displaying the field data to users. Both are plugins and will be covered in later recipes.

### Getting ready

Create a new module like the one existing in the first recipe. We will refer to the module as `mymodule` throughout the recipe. Use your module's appropriate name.

### How to do it...

1. We need to create the `src\Plugin\Field\FieldType` directory in the module's base location. The `Field` module discovers field types in the `Plugin\Field\FieldType` namespace.
2. Create a `RealName.php` file in the newly created directory so that we can define the `RealName` class. This will provide our `real_name` field for the first and last names:



3. The RealName class will extend the \Drupal\Core\Field\FieldItemBase class:

```
<?php
/**
 * @file
 * Contains \Drupal\mymodule\Plugin\Field\FieldType\RealName.
 */

namespace Drupal\mymodule\Plugin\Field\FieldType;

use Drupal\Core\Field\FieldItemBase;
use Drupal\Core\Field\FieldStorageDefinitionInterface;
use Drupal\Core\TypedData\DataDefinition;

class RealName extends FieldItemBase {
```

The \Drupal\Core\Field\FieldItemBase satisfies methods defined by inherited interfaces except for schema and propertyDefinitions.

4. Field types are annotated plugins. Annotated plugins use documentation blocks to provide details of the plugin. We will provide the field type's identifier, label, description, category, and default widget and formatter:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\Plugin\Field\FieldType\RealName.
 */

namespace Drupal\mymodule\Plugin\Field\FieldType;

use Drupal\Core\Field\FieldItemBase;
use Drupal\Core\Field\FieldStorageDefinitionInterface;
use Drupal\Core\TypedData\DataDefinition;

/**
 * Plugin implementation of the 'realname' field type.
 *
 * @FieldType(
 * id = "realname",
 * label = @Translation("Real name"),
 * ...
 *)
```

```
* description = @Translation("This field stores a first and
last name."),
* category = @Translation("General"),
* default_widget = "string_textfield",
* default_formatter = "string"
*)
class RealName extends FieldItemBase {

}
```

The `@FieldType` tells Drupal that this is a `FieldType` plugin. The following properties are defined:

- ❑ `Id`: This is the plugin's machine name
- ❑ `Label`: This is the human-readable name for the field
- ❑ `description`: This is the human-readable description of the field
- ❑ `category`: This is the category where the field shows up in the user interface
- ❑ `default_widget`: This is the default form widget to be used for editing
- ❑ `default_formatter`: This is the default formatter with which you can display the field

5. The `RealName` class needs to implement the `schema` method defined in the `\Drupal\Core\Field\FieldItemInterface`. This returns an array of the database API schema information:

```
/**
 * {@inheritDoc}
 */
public static function schema(\Drupal\Core\Field\FieldStorageDefinitionInterface $field_definition) {
 return array(
 'columns' => array(
 'first_name' => array(
 'description' => 'First name.',
 'type' => 'varchar',
 'length' => 255,
 'not null' => TRUE,
 'default' => '',
),
 'last_name' => array(
 'description' => 'Last name.',
 'type' => 'varchar',
```

```
 'length' => '255',
 'not null' => TRUE,
 'default' => '',
),
),
'indexes' => array(
 'first_name' => array('first_name'),
 'last_name' => array('last_name'),
),
);
}
```

The `schema` method defines the columns in the field's data table. We will define a column to hold the `first_name` and `last_name` values.

6. We also need to implement the `propertySchema` method to satisfy `\Drupal\Core\TypedData\ComplexDataDefinitionInterface`. This returns a typed definition of the values defined in the `schema` method:

```
/**
 * {@inheritDoc}
 */
public static function propertyDefinitions(\Drupal\Core\Field\FieldStorageDefinitionInterface $field_definition) {
 $properties['first_name'] = \Drupal\Core\TypedData\DataDefinition::create('string')
 ->setLabel(t('First name'));
 $properties['last_name'] = \Drupal\Core\TypedData\DataDefinition::create('string')
 ->setLabel(t('Last name'));

 return $properties;
}
```

This method returns an array that is keyed with the same column names provided in `schema`. It returns a typed data definition to handle the field type's values.

7. Rebuild Drupal's cache so that the plugin system can discover the new field type.

8. The field will now appear on the field type management screen:

The screenshot shows the 'Add field' interface in the Drupal admin. At the top, it says 'Home » Administration » Structure » Comment types » Edit » Manage fields'. Below that, a dropdown menu is open under 'Add a new field' with the placeholder 'Select a field type'. The menu lists several field types: General, Boolean, Comments, Date, Email, Link, Real name, Number, and List (float). The 'Real name' option is currently selected, indicated by a blue horizontal bar underneath it.

## How it works...

Drupal core defines a `plugin.manager.field.field_type` service. By default, this is handled through the `\Drupal\Core\Field\FieldTypePluginManager` class. This plugin manager defines the field type plugins that should be in the `Plugin/Field/FieldType` namespace, and all the classes in this namespace will be loaded and assumed to be field type plugins.

The manager's definition also sets `\Drupal\Core\Field\FieldItemInterface` as the expected interface that all the field type plugins will implement. This is why most field types extend `\Drupal\Core\Field\FieldItemBase` to meet these method requirements.

As field types are annotated plugins, the manager provides `\Drupal\Core\Field\Annotation\FieldType` as the class that fulfills the annotation definition.

When the user interface defines the available fields, the `plugin.manager.field.field_type` service is invoked to retrieve a list of available field types.

## There's more...

### Altering field types

The `\Drupal\Core\Field\FieldTypePluginManager` class defines the `alter` method as `field_info`. Modules that implement `hook_field_info_alter` in their `.module` files have the ability to modify field type definitions discovered by the manager:

```
/**
 * Implements hook_field_info_alter().
 */
function mymodule_field_info_alter(&$info) {
 $info['email']['label'] = t('E-mail address');
}
```

The preceding `alter` method will change the human-readable label for the `Email` field to the e-mail address.

### Defining whether a field is empty

The `\Drupal\Core\TypedData\ComplexDataInterface` interface provides an `isEmpty` method. This method is used to check whether the field's value is empty, for example, when verifying that the required field has data. The `\Drupal\Core\TypedData\Plugin\DataType\Map` class implements the method. By default, the method ensures that the values are not empty.

Field types can provide their own implementations to provide a more robust verification. For instance, the field can validate that the first name can be entered but not the last name, or the field can require both the first and the last name.

### See also

- ▶ The *Creating blocks using plugins* recipe of this chapter

## Creating a custom field widget

Field widgets provide the form interface for editing a field. These integrate with the Form API to define how a field can be edited and the way in which the data can be formatted before it is saved. Field widgets are chosen and customized through the form display interface.

In this recipe, we will create a widget for the field created in the *Creating a custom field type* recipe in this chapter. The field widget will provide two text fields for entering the first and last name items.

### Getting ready

Create a new module such as the one existing in the first recipe. We will refer to the module as `mymodule` throughout the recipe. Use your module's appropriate name.

## How to do it...

1. We need to create the `src\Plugin\Field\FieldWidget` directory in the module's base location. The `Field` module discovers field widgets in the `Plugin\Field\FieldWidget` namespace.
2. Create a `RealNameDefaultWidget.php` file in the newly created directory so that we can define the `RealNameDefaultWidget` class. This will provide a custom form element to edit the first and last name values of our field:



3. The `RealNameDefaultWidget` class will extend the `\Drupal\Core\Field\WidgetBase` class:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\Plugin\Field\FieldWidget\
RealNameDefaultWidget
 */

namespace Drupal\mymodule\Plugin\Field\FieldWidget;

use Drupal\Core\Field\WidgetBase;

class RealNameDefaultWidget extends WidgetBase {
```

4. Field widgets are like annotated plugins. Annotated plugins use documentation blocks to provide details of the plugin. We will provide the field widget's identifier, label, and supported field types:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\Plugin\Field\FieldWidget\
RealNameDefaultWidget
 */

namespace Drupal\mymodule\Plugin\Field\FieldWidget;

use Drupal\Core\Field\WidgetBase;
use Drupal\Core\Field\FieldItemListInterface;
use Drupal\Core\Form\FormStateInterface;

/**
 * Plugin implementation of the 'realname_default' widget.
 *
 * @FieldWidget(
 * id = "realname_default",
 * label = @Translation("Real name"),
 * field_types = {
 * "realname"
 * }
 *)
 */
class RealNameDefaultWidget extends WidgetBase {

}
```

The `@FieldWidget` tells Drupal that this is a field widget plugin. It defines `id` to represent the machine name, the human-readable name as `label`, and the field types that the widget interacts with.

5. We need to implement the `formElement` method to satisfy the remaining interface methods after extending `\Drupal\Core\Field\WidgetBase`:

```
 /**
 * {@inheritDoc}
 */
public function formElement(FieldItemListInterface $items,
$delta, array $element, array &$form, FormStateInterface $form_
state) {
```

```
$element['first_name'] = array(
 '#type' => 'textfield',
 '#title' => t('First name'),
 '#default_value' => '',
 '#size' => 25,
 '#required' => $element['#required'],
);
$element['last_name'] = array(
 '#type' => 'textfield',
 '#title' => t('Last name'),
 '#default_value' => '',
 '#size' => 25,
 '#required' => $element['#required'],
);
return $element;
}
```

The `formElement` method returns a Form API array that represents the widget to be set, and edits the field data.

6. Next, we need to modify our original `RealName` field type plugin class in order to use the default widget that we created. Update the `default_widget` annotation property as `realname_default`:

```
/**
 * Plugin implementation of the 'realname' field type.
 *
 * @FieldType(
 * id = "realname",
 * label = @Translation("Real name"),
 * description = @Translation("This field stores a first and last name."),
 * category = @Translation("General"),
 * default_widget = "realname_default",
 * default_formatter = "string"
 *)
 */
class RealName extends FieldItemBase {
```

7. Rebuild Drupal's cache so that the plugin system can discover the new field widget.

8. Add a Real name field and use the new Real name widget. For example, add it to a Comment type:

The screenshot shows the 'Add new comment' interface. At the top is a 'Subject' input field. Below it is a 'Comment' area with a rich text editor toolbar containing buttons for bold (B), italic (I), link (link icon), list (list icons), and other text formats. A 'Format' dropdown and a 'Source' button are also present. The main comment area is a large text input field. Below the comment area is a 'Text format' dropdown set to 'Basic HTML'. To the right of the dropdown is a link to 'About text formats'. At the bottom of the form are two text input fields: 'First name' and 'Last name'.

## How it works...

Drupal core defines a `plugin.manager.field.widget` service. By default, this is handled through the `\Drupal\Core\Field\FieldWidgetPluginManager` class. This plugin manager defines the field widget plugins that should be in the `Plugin/Field/FieldWidget` namespace, and all the classes in this namespace will be loaded and assumed to be field widget plugins.

The manager's definition also sets `\Drupal\Core\Field\FieldWidgetInterface` as the expected interface that all the field widget plugins will implement. This is why most field types extend `\Drupal\Core\Field\WidgetBase` to meet these method requirements.

As field widgets are annotated plugins, the manager provides `\Drupal\Core\Field\Annotation\FieldWidget` as the class that fulfills the annotation definition.

The entity form display system uses the `plugin.manager.field.widget` service to load field definitions and add the field's element, returned from the `formElement` method, to the entity form.

## There's more

### Field widget settings and summary

The \Drupal\Core\Field\WidgetInterface interface defines three methods that can be overridden to provide a settings form and a summary of the current settings:

- ▶ `defaultSettings`: This returns an array of the setting keys and default values
- ▶ `settingsForm`: This returns a Form API array that is used for the settings form
- ▶ `settingsSummary`: This allows an array of strings to be returned and displayed on the manage display form for the field

Widget settings can be used to alter the form presented to the user. A setting can be created that allows the field element to be limited to only enter the first or last name with one text field.

## See also

- ▶ The *Creating a custom plugin type* recipe of this chapter

## Creating a custom field formatter

Field formatters define the way in which a field type will be presented. These formatters return the render array information to be processed by the theming layer. Field formatters are configured on the display mode interfaces.

In this recipe, we will create a formatter for the field created in the *Creating a custom field type* recipe in this chapter. The field formatter will display the first and last names with some settings.

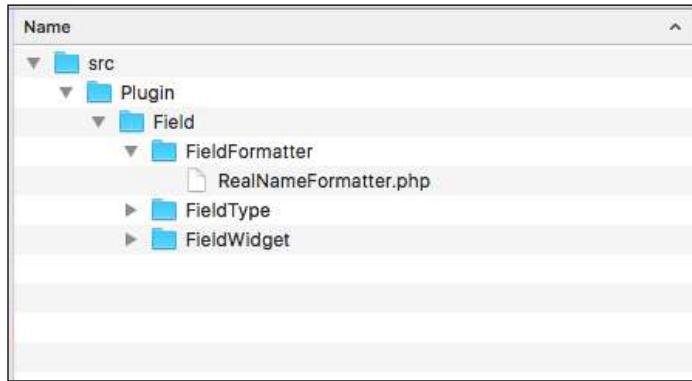
## Getting ready

Create a new module like the one existing in the first recipe. We will refer to the module as `mymodule` throughout the recipe. Use your module's appropriate name.

## How to do it...

1. We need to create the `src/Plugin/Field/FieldFormatter` directory in the module's base location. The `Field` module discovers field formatters in the `Plugin\ Field\ FieldFormatter` namespace.

2. Create a `RealNameFormatter.php` file in the newly created directory so that we can define the `RealNameFormatter` class. This will provide a custom form element to display the field's values:



3. The `RealNameFormatter` class will extend the `\Drupal\Core\Field\FormatterBase` class:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\Plugin\Field\FieldFormatter\
RealNameFormatter
 */

namespace Drupal\mymodule\Plugin\Field\FieldFormatter;

use Drupal\Core\Field\FormatterBase;
use Drupal\Core\Field\FieldItemListInterface;

class RealNameFormatter extends FormatterBase {
```

```
}
```

4. Field formatters are like annotated plugins. Annotated plugins use documentation blocks to provide details of the plugin. We will provide the field widget's identifier, label, and supported field types:

```
<?php
```

```
/***
 * @file
```

```
* Contains \Drupal\mymodule\Plugin\Field\FieldFormatter\
RealNameFormatter
*/

namespace Drupal\mymodule\Plugin\Field\FieldFormatter;

use Drupal\Core\Field\FormatterBase;
use Drupal\Core\Field\FieldItemListInterface;

/**
 * Plugin implementation of the 'realname_one_line' formatter.
 *
 * @FieldFormatter(
 * id = "realname_one_line",
 * label = @Translation("Real name (one line)"),
 * field_types = {
 * "realname"
 * }
 *)
 */

class RealNameFormatter extends FormatterBase {
}
}
```

5. We need to implement the `viewElements` method to satisfy the `\Drupal\Core\Field\FormatterInterface` interface. This is used to render the field data:

```
/**
 * {@inheritDoc}
 */
public function viewElements(FieldItemListInterface $items,
$langcode) {
 $element = [];

 foreach ($items as $delta => $item) {
 $element[$delta] = array(
 '#markup' => $this->t('@first @last', array(
 '@first' => $item->first_name,
 '@last' => $item->last_name,
))
);
 }
 return $element;
}
```

6. Next, we need to modify our original RealName field type's plugin class in order to use the default formatter that we created. Update the `default_formatter` annotation property as `realname_one_line`:

```
/**
 * Plugin implementation of the 'realname' field type.
 *
 * @FieldType(
 * id = "realname",
 * label = @Translation("Real name"),
 * description = @Translation("This field stores a first and
last name."),
 * category = @Translation("General"),
 * default_widget = "string_textfield",
 * default_formatter = "realname_one_line"
 *)
 */
```

7. Rebuild Drupal's cache so that the plugin system can discover the new field widget.
8. Update an entity view mode with a `Real name` field to use the **Real name (one line)** formatter:

The screenshot shows the 'Manage display' interface for a comment type. The interface has a header with tabs: Edit, Manage fields, Manage form display, Manage display (which is active), and Devel. Below the header, the breadcrumb navigation shows: Home > Administration > Structure > Comment types > Edit. There is a 'Show row weights' link. The main content area displays three fields in a table:

| FIELD     | LABEL      | FORMAT                               |
|-----------|------------|--------------------------------------|
| Comment   | - Hidden - | Default                              |
| Links     |            | Visible                              |
| Your name | Above      | ✓ Real name (one line)<br>- Hidden - |

Below the table, a note says 'No field is hidden.' and there is a 'CUSTOM DISPLAY SETTINGS' section with a 'Save' button.

## How it works...

Drupal core defines a `plugin.manager.field.formatter` service. By default, this is handled through the `\Drupal\Core\Field\FormatterPluginManager` class. This plugin manager defines the field formatter plugins that should be in the `Plugin/Field/FieldFormatter` namespace, and all the classes in this namespace will be loaded and assumed to be field formatter plugins.

The manager's definition also sets `\Drupal\Core\Field\FormatterInterface` as the expected interface that all field formatter plugins will implement. This is why most field formatters extend `\Drupal\Core\Field\FormatterBase` to meet these method requirements.

As field formatters are annotated plugins, the manager provides `\Drupal\Core\Field\Annotation\FieldFormatter` as the class that fulfills the annotation definition.

The entity view display system uses the `plugin.manager.field.formatter` service to load field definitions and add the field's render array, returned from the `viewElements` method, to the entity view render array.

## There's more

### Formatter settings and summary

The `\Drupal\Core\Field\FormatterInterface` interface defines three methods that can be overridden to provide a settings form and a summary of the current settings:

- ▶ `defaultSettings`: This returns an array of the setting keys and default values
- ▶ `settingsForm`: This returns a Form API array that is used for the settings form
- ▶ `settingsSummary`: This allows an array of strings to be returned and displayed on the manage display form for the field

Settings can be used to alter how the formatter displays information. For example, these methods can be implemented to provide settings to hide or display the first or last name.

## See also

- ▶ The *Creating a custom plugin type* recipe of this chapter

## Creating a custom plugin type

The plugin system provides means to create specialized objects in Drupal that do not require the robust features of the entity system.

In this recipe, we will create a new plugin type called `Unit` that will work with units of measurement and conversions. We will create a plugin manager, default plugin interface, YAML discovery method, base class, and plugin definition.

This recipe is based on the work being done to export the `Physical` module to Drupal 8. The `Physical` module provides a way to work with units of volume, weight, and dimensions and attaches them to entities. It discovers unit plugins in the same way that the `Breakpoint` module discovers breakpoint plugins.

### Getting ready

Create a new module like the one existing in the first recipe. We will refer to the module as `mymodule` throughout the recipe. Use your module's appropriate name.

### How to do it...

1. All plugins need to have a service that acts as a plugin manager. Create a new file in your module's `src` directory called `UnitManager.php`. This will hold the `UnitManager` class.
2. Create the `UnitManager` class by extending the `\Drupal\Core\Plugin\DefaultPluginManager` class:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\UnitManager.
 */

namespace Drupal\mymodule;

use Drupal\Core\Plugin\DefaultPluginManager;
use Drupal\Core\Cache\CacheBackendInterface;
use Drupal\Core\Extension\ModuleHandlerInterface;

class UnitManager extends DefaultPluginManager {
```

3. When creating a new plugin type, it is recommended that the plugin manager provides a set of defaults for new plugins, if an item is missing. This is also useful to define the default class a plugin should use:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\UnitManager.
 */

namespace Drupal\mymodule;

use Drupal\Core\Plugin\DefaultPluginManager;
use Drupal\Core\Cache\CacheBackendInterface;
use Drupal\Core\Extension\ModuleHandlerInterface;

class UnitManager extends DefaultPluginManager {
 /**
 * Default values for each unit plugin.
 *
 * @var array
 */
 protected $defaults = [
 'id' => '',
 'label' => '',
 'unit' => '',
 'factor' => 0.00,
 'type' => '',
 'class' => 'Drupal\mymodule\Unit',
];
}

}
```

4. Later, we will create the `Unit` class in our module that unit plugins will be instances of.
5. Next, we need to override the `\Drupal\Core\Plugin\DefaultPluginManager` class constructor to define the module handler and cache backend:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\UnitManager.
 */
```

```

namespace Drupal\mymodule;

use Drupal\Core\Plugin\DefaultPluginManager;
use Drupal\Core\Cache\CacheBackendInterface;
use Drupal\Core\Extension\ModuleHandlerInterface;

class UnitManager extends DefaultPluginManager {
 /**
 * Default values for each unit plugin.
 *
 * @var array
 */
 protected $defaults = [
 'id' => '',
 'label' => '',
 'unit' => '',
 'factor' => 0.00,
 'type' => '',
 'class' => 'Drupal\physical\Unit',
];

 /**
 * Constructs a new \Drupal\mymodule\UnitManager object.
 *
 * @param \Drupal\Core\Cache\CacheBackendInterface $cache_
 * backend
 * Cache backend instance to use.
 * @param \Drupal\Core\Extension\ModuleHandlerInterface $module_
 * handler
 * The module handler to invoke the alter hook with.
 */
 public function __construct(CacheBackendInterface $cache_
 backend, ModuleHandlerInterface $module_handler) {
 $this->moduleHandler = $module_handler;
 $this->setCacheBackend($cache_backend, 'physical_unit_
 plugins');
 }
}

```

6. We override the constructor so that we can specify a specific cache key. This allows plugin definitions to be cached and cleared properly; otherwise, our plugin manager will continuously read the disk to find plugins.

7. We also need to override the `getDiscovery` method. We need to implement a YAML discovery method:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\UnitManager.
 */

namespace Drupal\mymodule;

use Drupal\Core\Plugin\DefaultPluginManager;
use Drupal\Core\Cache\CacheBackendInterface;
use Drupal\Core\Extension\ModuleHandlerInterface;

class UnitManager extends DefaultPluginManager {
 /**
 * Default values for each unit plugin.
 *
 * @var array
 */
 protected $defaults = [
 'id' => '',
 'label' => '',
 'unit' => '',
 'factor' => 0.00,
 'type' => '',
 'class' => 'Drupal\mymodule\Unit',
];

 /**
 * Constructs a new \Drupal\mymodule\UnitManager object.
 *
 * @param \Drupal\Core\Cache\CacheBackendInterface $cache_
 * backend
 * Cache backend instance to use.
 * @param \Drupal\Core\Extension\ModuleHandlerInterface $module_
 * handler
 * The module handler to invoke the alter hook with.
 */
 public function __construct(CacheBackendInterface $cache_
 backend, ModuleHandlerInterface $module_handler) {
 $this->moduleHandler = $module_handler;
```

```

 $this->setCacheBackend($cache_backend, 'physical_unit_
plugins');
 }

/**
 * {@inheritDoc}
 */
protected function getDiscovery() {
 if (!isset($this->discovery)) {
 $this->discovery = new YamlDiscovery('units', $this-
>moduleHandler->getModuleDirectories());
 $this->discovery = new ContainerDerivativeDiscoveryDecorator
($this->discovery);
 }
 return $this->discovery;
}

}

```

8. The default plugin manager implementation supports an annotated plugin discovery, such as field types, field widgets, and field formatters. By setting the discovery property to `YamlDiscovery`, we are telling Drupal to look for a `*.units.yml` file in all the module directories.
9. The next step is to create a `mymodule.services.yml` in your module's directory. This will describe our plugin manager to Drupal, allowing a plugin discovery:

```

services:
 plugin.manager.unit:
 class: Drupal\mymodule\UnitManager
 arguments: ['@container.namespaces', '@cache.discovery', '@
module_handler']

```

10. Drupal utilizes services and dependency injection. By defining our class as a service, we are telling the application container how to initiate our class. This will allow us to retrieve the manager and access plugins even if another module replaces our defined plugin manager.
11. Next, we will define the plugin interface that we defined in the plugin manager. The plugin manager will validate the `Unit` plugins that implement this interface. Create a `UnitInterface.php` file in your module's `src` directory to hold the interface:

```

<?php

/**
 * @file
 * Contains \Drupal\mymodule\UnitInterface.

```

```
*/

namespace Drupal\mymodule;

/**
 * Interface UnitInterface.
 */
interface UnitInterface {

 /**
 * Returns the unit's label.
 *
 * @return string
 * The unit's label.
 */
 public function getLabel();

 /**
 * Returns the unit abbreviation.
 *
 * @return string
 * The abbreviation.
 */
 public function getUnit();

 /**
 * Returns the factor amount for conversions.
 *
 * @return int|float
 * The factor amount.
 */
 public function getFactor();

 /**
 * Converts a value to the base unit.
 *
 * @param int|float $value
 * The amount to convert.
 *
 * @return int|float
 * The converted amount.
 */
 public function toBase($value);
```

```
/**
 * Converts value from base unit to current unit.
 *
 * @param int|float $value
 * The amount to convert.
 *
 * @return int|float
 * The converted amount.
 */
public function fromBase($value);

/**
 * Rounds a value.
 *
 * @param int|float $value
 * The value to round.
 *
 * @return int|float
 * The rounded value.
 */
public function round($value);

}
```

12. We provide an interface so that we can guarantee that we have these expected methods when working with a Unit plugin and have an output, regardless of the logic behind each method. It pushes for encapsulation when working with plugins.

13. Create a mymodule.units.yml file to provide default unit plugin definitions:

```
centimeters:
 label: Centimeters
 unit: cm
 factor: 1E-2
 type: dimensions
meters:
 label: Meters
 unit: m
 factor: 1
 type: dimensions
feet:
 label: Feet
 unit: ft
 factor: 3.048E-1
 type: dimensions
```

```
inches:
 label: Inches
 unit: in
 factor: 2.54E-2
 type: dimensions

14. As defined in our plugin's default definition, we need to provide a Unit class.
Create Unit.php in your module's src directory. This class will implement
our UnitInterface interface:
<?php

/**
 * @file
 * Contains \Drupal\mymodule\Unit.
 */

namespace Drupal\mymodule;

use Drupal\Core\Plugin\PluginBase;

/**
 * Class Unit.
 */
class Unit extends PluginBase implements UnitInterface {

 /**
 * {@inheritDoc}
 */
 public function getFactor() {
 return (float) $this->pluginDefinition['factor'];
 }

 /**
 * {@inheritDoc}
 */
 public function getLabel() {
 return $this->t($this->pluginDefinition['label'], array(),
array('context' => 'unit'));
 }

 /**
 * {@inheritDoc}
 */
 public function getUnit() {
```

```
 return $this->pluginDefinition['unit'];
 }

 /**
 * {@inheritDoc}
 */
 public function toBase($value) {
 return $this->round($value * $this->getFactor());
 }

 /**
 * {@inheritDoc}
 */
 public function fromBase($value) {
 return $this->round($value / $this->getFactor());
 }

 /**
 * {@inheritDoc}
 */
 public function round($value) {
 return round($value, 5);
 }

 /**
 * Returns the unit's label.
 *
 * @return string
 * Unit label.
 */
 public function __toString() {
 return $this->getLabel();
 }
}
```

15. This class implements all the required methods defined in our interface.  
The `toBase` and `fromBase` methods allow us to convert the unit's value from its defined factor value.
16. The `Unit` plugin is now implemented and can be integrated through a custom field type or another custom code.

## How it works...

Drupal 8 implements a service container, a concept adopted from the Symfony framework. In order to implement a plugin, there needs to be a manager who can discover and process plugin definitions. This manager is defined as a service in a module's `services.yml` with its required constructor parameters. This allows the service container to initiate the class when it is required.

In our example, the `UnitManager` plugin manager discovers the `Unit` plugin definitions in YAML files that modules provide. After the first discovery, all the known plugin definitions are then cached under the `physical_unit_plugins` cache key.

Plugin managers also provide a method for returning these definitions or creating an object instance based on an available definition. The instance is created from the `class` key that we defined in our plugin's default definition. This also allows a developer to use a custom class to provide an extended `Unit` plugin as long as it extends the default `Unit` class or implements the `UnitInterface` interface.

An example usage would be to create a custom form that allows users to convert values. The following code can be placed in the `submit` method and will allow us to load our plugin for feet and return the value in meters:

```
// Load the manager service.
$unit_manager = \Drupal::service('plugin.manager.unit');

// Create a class instance through the manager.
$feet_instance = $unit_manager->createInstance('feet');

// Convert 12ft into meters.
$meters_value = $feet_instance->toBase(12);
```

## There's more

### Specifying an alter hook

Plugin managers have the ability to define an alter hook. The following line of code will be added to the `UnitManager` class's constructor to provide `hook_physical_unit_alter`. This is passed to the module handler service for invocations:

```
/**
 * Constructs a new \Drupal\mymodule\UnitManager object.
 *
 * @param \Drupal\Core\Cache\CacheBackendInterface
 * $cache_backend
```

---

```

* Cache backend instance to use.
* @param \Drupal\Core\Extension\ModuleHandlerInterface
$module_handler
* The module handler to invoke the alter hook with.
*/
public function __construct(CacheBackendInterface
 $cache_backend, ModuleHandlerInterface $module_handler) {
 $this->moduleHandler = $module_handler;
 $this->alterInfo('physical_unit');
 $this->setCacheBackend($cache_backend,
 'physical_unit_plugins');
}

```

Modules implementing `hook_physical_unit_alter` in the `.module` file have the ability to modify all the discovered plugin definitions. Modules have the ability to remove defined plugin entries or alter any information provided for the annotation definition.

## Using a cache backend

Plugins can use a cache backend to improve performance. This can be done by specifying a cache backend with the `setCacheBackend` method in the manager's constructor. The following line of code will allow the Unit plugins to be cached and only discovered on a cache rebuild.

The `$cache_backend` variable is passed to the constructor. The second parameter provides the cache key. The cache key will have the current language code added as a suffix.

There is an optional third parameter that takes an array of strings to represent cache tags that will cause the plugin definitions to be cleared. This is an advanced feature and plugin definitions should normally be cleared through the manager's `clearCachedDefinitions` method. The cache tags allow the plugin definitions to be cleared when a relevant cache is cleared as well.

## Accessing plugins through the manager

Plugins are loaded through the manager service, which should always be accessed through the service container. The following line of code will be used in your module's hooks or classes to access the plugin manager:

```
$unit_manager = \Drupal::service('plugin.manager.unit');
```

Plugin managers have various methods to retrieve plugin definitions, which are as follows:

- ▶ `getDefinitions`: This method will return an array of plugin definitions. It first makes an attempt to retrieve cached definitions, if any, and sets the cache of discovered definitions before returning them.
- ▶ `getDefinition`: This takes an expected plugin ID and returns its definition.

*Plug and Play with Plugins* —————

- ▶ `createInstance`: This takes an expected plugin ID and returns an initiated class for the plugin.
- ▶ `getInstance`: This takes an array that acts as a plugin definition and returns an initiated class from the definition.

**See also**

- ▶ Services and dependency injection at <https://www.drupal.org/node/2133171>

# 8

## Multilingual and Internationalization

In this chapter, we will cover the following recipes to make that your site is multilingual and internationalized:

- ▶ Translating administrative interfaces
- ▶ Translating configurations
- ▶ Translating content
- ▶ Creating multilingual views

### Introduction

This chapter will cover the multilingual and internationalization features of Drupal 8, which have been greatly enhanced since Drupal 7. The previous version of Drupal required many extra modules to provide internationalization efforts, but now the majority is provided by Drupal core.

Drupal core provides the following multilingual modules:

- ▶ **Language:** This provides you with the ability to detect and support multiple languages
- ▶ **Interface translation:** This takes installed languages and translates strings that are presented through the user interface
- ▶ **Configuration translation:** This allows you to translate configuration entities, such as date formats and views
- ▶ **Content translation:** This brings the power of providing content in different languages and displaying it according to the current language of the user

Each module serves a specific purpose in creating the multilingual experience for your Drupal site. Behind the scenes, Drupal supports the language code for all entities and cache contexts. These modules expose the interfaces in order to implement and deliver internationalized experiences.

## Translating administrative interfaces

The interface translation module provides a method for translating strings found in the Drupal user interface. Harnessing the Language module, interface translations are automatically downloaded from the Drupal translation server. By default, the interface language is loaded through the language code as a path prefix. With the default Language configuration, paths will be prefixed with the default language.

Interface translations are based on strings provided in the code that are passed through the internal translation functions.

In this recipe, we will enable Spanish, import the language files, and review the translated interface strings to provide missing or custom translations.

### Getting ready

Drupal 8 provides an automated installation process of translation files. For this to work, your web server must be able to communicate with <https://localize.drupal.org/>. If your web server cannot automatically download the files from the translation server, you can refer to the manual installation instructions, which will be covered in the *There's more* section.

### How to do it...

1. Go to **Extend** and install the **Interface Translation** module. It will prompt you to enable the **Language**, **File**, and **Field** modules to be installed as well if they are not.
2. After the module is installed, click on **Configuration**. Go to the **Languages** page under the **Regional and Language** section.
3. Click on **Add language** in the languages overview table:

The screenshot shows a table with the following columns: NAME, DEFAULT, INTERFACE TRANSLATION, and OPERATIONS. There is one row for the language "English". The "DEFAULT" column has a radio button next to "English", which is selected. The "INTERFACE TRANSLATION" column contains the text "not applicable". The "OPERATIONS" column has a "Edit" button with a dropdown arrow. At the top left of the table is a blue button labeled "+ Add language". At the bottom left is a blue button labeled "Save configuration".

4. The **Add language** page provides a select list of all available languages that the interface can be translated to. Select **Spanish**, and then click on **Add language**.
5. A batch process will run, install the translation language files, and import them.
6. The **INTERFACE TRANSLATION** column specifies the percentage of active translatable interface strings that have a matching translation. Clicking on the link allows you to view the **User interface translation** form:

| Show row weights |                                  |                                    |                       |
|------------------|----------------------------------|------------------------------------|-----------------------|
| NAME             | DEFAULT                          | INTERFACE TRANSLATION              | OPERATIONS            |
| ⊕ English        | <input checked="" type="radio"/> | not applicable                     | <button>Edit</button> |
| ⊕ Spanish        | <input type="radio"/>            | <a href="#">7971/8121 (98.15%)</a> | <button>Edit</button> |

**Save configuration**

7. The **Filter Translatable Strings** form allows you to search for translated strings or untranslated strings. Select **Only untranslated strings** from the **Search in** select list and click on **Filter**.
8. Using the text box on the right-hand side of the screen, a custom translation can be added to **Only untranslated strings**. Type in a translation for the item.

▼ FILTER TRANSLATABLE STRINGS

String contains

Leave blank to show all strings. The search is case sensitive.

Translation language  Search in

\* Changes made in this table will not be saved until the form is submitted.

| SOURCE STRING                                                                  | TRANSLATION FOR SPANISH                                                                         |
|--------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Use <em>basic pages</em> for your static content, such as an 'About us' page.* | Utilice <em>páginas básicos</em> para el contenido estático , como una página 'Sobre nosotros'. |

9. Click on **Save translations** to save the modification.
10. Go to /es/node/add and you will notice that the Basic page content type description will now match your translation.

## How it works...

The interface translation module provides `\Drupal\locale\LocaleTranslation`, which implements `\Drupal\Core\StringTranslation\Translator\TranslatorInterface`. This class is registered under the `string_translation` service as an available lookup method.

When the `t` function or the `\Drupal\Core\StringTranslation\StringTranslationTrait::t` method is invoked, the `string_translation` service is called to provide a translated string. The `string_translation` service will iterate through the available translators and return a translated string, if possible.



Developers need to note that this is a key reason to ensure that module strings are passed through translation functions. It allows you to identify strings that need to be translated.



The translator provided in the interface translation will then attempt to resolve the provided string against known translations for the current language. If a translation has been saved, it will be returned.

## There's more...

We will explore ways to install other languages, check translation statuses, and many more in the following sections.

### Manually installing language files

Translation files can be manually installed by downloading them from the [Drupal.org](https://localize.drupal.org) translation server and uploading them through the language interface. You can also use the import interface to upload custom **Gettext Portable Object (.po)** files.

Drupal core and most contributed projects have `.po` files available at the Drupal translations site, <https://localize.drupal.org>. On the site, click on **Download** and you will be able to download a `.po` file for Drupal core in all available languages. Additionally, clicking on a language will provide more translations for a specific language across projects.

## Spanish overview

[Overview](#) [Board](#) [Translate](#)

Spanish translation team – Grupo de traducción al Español

Nuevo! [Ayuda a probar la nueva versión de localize.drupal.org en Drupal 7!](#)

- [Diccionario](#) – Libro de estilo Wiki para crear un glosario de términos, manuales para traductores y libro de estilo en Español.
- [Traducción de Drupal core](#) Paquete de archivos .po que componen la traducción de Drupal, y los módulos del Core, al español neutro.
- [Interfaz de traducción](#): Aportar sugerencias de traducción para cadenas de texto pendientes de traducir. Los moderadores validarán las sugerencias y seleccionarán la que será finalmente utilizada por la comunidad. Permite importar nuevas cadenas de texto (actualizaciones de módulos) y exportarlas para ser utilizadas en producción.
- [Foro de traducciones](#): Iniciar y seguir debates sobre palabras o cadenas de texto concretas. Las discusiones sobre palabras establecen una base sólida sobre la que luego construir las sugerencias que serán posteadas en localize.drupal.org
- [Glosario de términos](#): Establece una relación de traducción "automática" para los términos más comunes.
- [Directrices para la traducción](#): Ofrece ideas sobre cómo realizar la traducción al español, de modo que los traductores tengamos un criterio homogéneo. Son ideas abiertas a discusión y por tanto no son realmente un "Libro de Estilo".
- [Moderadores de la traducción](#): Cómo convertirse en moderador y líneas guía para moderar las traducciones.

### Top downloads

| Drupal core                 |         |                                          |              |                  |
|-----------------------------|---------|------------------------------------------|--------------|------------------|
| Project                     | Version | Downloads                                | Date created | Up to date as of |
| <a href="#">Drupal core</a> | 5.23    | <a href="#">Download<br/>(414.14 KB)</a> | 2011-Jun-23  | 2011-Jul-14      |
| <a href="#">Drupal core</a> | 6.37    | <a href="#">Download<br/>(529.03 KB)</a> | 2015-Oct-01  | 2015-Dec-06      |
| <a href="#">Drupal core</a> | 7.41    | <a href="#">Download<br/>(679.04 KB)</a> | 2015-Nov-03  | 2015-Dec-06      |
| <a href="#">Drupal core</a> | 8.0.1   | <a href="#">Download<br/>(1.04 MB)</a>   | 2015-Dec-04  | 2015-Dec-06      |

You can import a .po file by going to the User interface translation form and selecting the Import tab. You need to select the .po file and then the appropriate language. You have the ability to treat the uploaded files as custom created translations. This is recommended if you are providing a custom translation file that was not provided by Drupal.org. If you are updating Drupal.org translations manually, make sure that you check the box that overwrites existing noncustom translations. The final option allows you to replace customized translations if the .po file provides them. This can be useful if you have translated missing strings that might now be provided by the official translation file.

## Checking translation status

As you add new modules, the available translations will grow. The Interface translation module provides a translation status report that is accessible from the Reports page. This will check the default translation server for the project and check whether there is a .po available or if it has changed. In the event of a custom module, you can provide a custom translation server, which is covered in *Providing translations for a custom module*.

If an update is available, you will be alerted. You can then import the translation file updates automatically or download and manually import them.

## Exporting translations

In the User interface translation form, there is an Export tab. This form will provide a Gettext Portable Object (.po) file. You have the ability to export all the available source text that is discovered in your current Drupal site without translations. This will provide a base .po for translators to work on.

Additionally, you can download a specific language. Specific language downloads can include noncustomized translations, customized translations, and missing translations. Downloading customized translations can be used to help make contributions to the multilingual and internationalization efforts of the Drupal community!

## Interface translation permissions

The interface translation module provides a single permission called **Translate interface text**. This permission grants users the permission to interact with all of the module's capabilities. It is flagged with a security warning as it allows users with this permission to customize all the output text presented to users.

However, it does allow you to provide a role for translators and limits their access to just translation interfaces.

## Using interface translation to customize default English strings

The interface translation module is useful beyond its typical multilingual purposes. You can use it to customize strings in the interface that are not available to be modified through typical hook methods, or if you are not a developer!

Firstly, you will need to edit the English language from the **Languages** screen. Check the checkbox for **Enable interface translation for English** and click on **Save language**. You will now have the ability to customize existing interface strings.



This is only recommended for areas of the interface that cannot already be customized through the normal user interface or provided API mechanisms.

## Interface text language detection

The Language module provides detection and selection rules. By default, the module will detect the current language based on the URL, with the language code acting as a prefix to the current path. For example, /es/node will display the node listing page in Spanish:

| Interface text language detection                                                                                                                       |                                                |                                     |                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|-------------------------------------|---------------------------|
| Order of language detection methods for interface text. If a translation of interface text is available in the detected language, it will be displayed. |                                                |                                     |                           |
| DETECTION METHOD                                                                                                                                        | DESCRIPTION                                    | ENABLED                             | OPERATIONS                |
| ⊕ Account administration pages                                                                                                                          | Account administration pages language setting. | <input type="checkbox"/>            |                           |
| ⊕ URL                                                                                                                                                   | Language from the URL (Path prefix or domain). | <input checked="" type="checkbox"/> | <a href="#">Configure</a> |
| ⊕ Session                                                                                                                                               | Language from a request/session parameter.     | <input type="checkbox"/>            | <a href="#">Configure</a> |
| ⊕ User                                                                                                                                                  | Follow the user's language preference.         | <input type="checkbox"/>            |                           |
| ⊕ Browser                                                                                                                                               | Language from the browser's language settings. | <input type="checkbox"/>            | <a href="#">Configure</a> |
| ⊕ Selected language                                                                                                                                     | Language based on a selected language.         | <input checked="" type="checkbox"/> | <a href="#">Configure</a> |

You can have multiple detection options enabled at once and use ordering to decide which takes precedence. This can allow you to use the language code in the URL first, but, if missing, a fallback to the language is specified by the user's browser.

Some detection methods have settings. For instance, the URL detection method can be based on the default path prefix or subdomains.

## Providing translations for a custom module

Modules can provide custom translations in their directories or point to a remote file. These definitions are added to the module's `info.yml` file. First, you need to specify the `interface translation project` key if it differs from the project's machine name.

You need to then specify a server pattern through the `interface translation server pattern` key. This can be a relative path to Drupal's root, such as `modules/custom/mymodule/translation.po`, or a remote file URL at `http://example.com/files/translations/mymodule/translation.po`.

Distributions (or other modules) can implement `hook_locale_translation_projects_alter` to provide this information on behalf of modules or alter defaults.

The server pattern accepts the following different tokens:

- ▶ %core for the version of a course (for example, 8.x)
- ▶ %project for the project's name
- ▶ %version for the current version string
- ▶ %language for the language code

More information on the interface translation keys and variables can be found in the `local.api.php` document file located in the interface translation module's base folder.

## See also

- ▶ Refer to the Drupal translation server at <https://localize.drupal.org/translate/drupal8>
- ▶ You can contribute using the localization server at <https://www.drupal.org/node/302194>
- ▶ Refer to the `locale.api.php` documentation at <https://api.drupal.org/api/drupal/core%21modules%21locale%21locale.api.php/8>
- ▶ Refer to PO and POT files: <https://www.drupal.org/node/1814954>

## Translating configuration

The **Configuration translation** module provides an interface for translating configurations with Interface translation and Language as dependencies. This module allows you to translate configuration entities. The ability to translate configuration entities adds an extra level of internationalization.

Interface translation allows you to translate strings provided in your Drupal site's code base. Configuration translation allows you to translate importable and exportable configuration items that you have created, such as your site title or date formats.

In this recipe, we will translate date format configuration entities. We will provide localized date formats for Danish to provide a more internationalized experience.

## Getting ready

Your Drupal site needs to have two languages enabled in order to use **Configuration Translation**. Install **Danish** from the **Languages interface**.

## How to do it...

1. Go to the **Extend** and install the **Configuration Translation** module. It will prompt you to enable the **Interface Translation**, **Language**, **File**, and **Field** modules to be installed as well if they are not.
2. After the module is installed, go to the **Configuration**. Go to the **Configuration translation** page under the **Regional and Language** section.
3. Click on the list for the **Date format** option in the configuration entity option table:

|                     |                      |
|---------------------|----------------------|
| Content type        | <a href="#">List</a> |
| Custom block fields | <a href="#">List</a> |
| Custom block type   | <a href="#">List</a> |
| Date format         | <a href="#">List</a> |
| Form mode           | <a href="#">List</a> |
| Image style         | <a href="#">List</a> |

4. We will translate the **Default long date format** to represent the **Danish** format. Click on the **Translate for the Default long date format** row.
5. Click on **Add** to create a **Danish** translation:

| Translations for <i>Default long date date format</i> ☆ |                      |
|---------------------------------------------------------|----------------------|
| LANGUAGE                                                | OPERATIONS           |
| English (original)                                      | <a href="#">Edit</a> |
| Danish                                                  | <a href="#">Add</a>  |

6. For **Danish**, we will provide the following PHP date format: I j, F, Y – H.i. This will display the day of the week, day of the month, the month, full year, and 24 hour notation for time.
7. Click on **Save translation**.
8. Whenever a user is browsing your Drupal site with **Danish** as their language, the date format will now be localized for their experience.

## How it works...

The Configuration translation module requires Interface translation; however, it does not work in the same fashion. The module modifies all entity types that extend the \Drupal\Core\Config\Entity\ConfigEntityInterface interface. It adds a new handler under the config\_translation\_list key. This is used to build a list of available configuration entities and their bundles.

The module alters the configuration schema in Drupal and updates the default configuration element definitions to use a specified class under \Drupal\config\_translation\Form. This allows \Drupal\config\_translation\Form\ConfigTranslationFormBase and its child classes proper saved translated configuration data that can be modified through the configuration translation screens.

When the configuration is saved, it is identified as being part of a collection. The collection is identified as language . LANGCODE and all translated configuration entities are saved and loaded by this identifier. Here is an example of how the configuration items are stored in the database:

| name                            | collection  |
|---------------------------------|-------------|
| block.block.bartik_account_menu |             |
| block.block.bartik_account_menu | language.es |

2 rows in set (0.00 sec)

When browsing the site in the es language code, the appropriate block.block.bartik\_account\_menu configuration entity will be loaded. If you are using the default site, or no language code, the configuration entity with an empty collection will be used.

## There's more...

Configuration entities and the ability to translate them are a big part of Drupal 8's multilingual capabilities. We'll explore them in detail in the next recipe.

### Altering configuration translation info definitions

Modules have the ability to invoke the `hook_config_translation_info_alter` hook to alter discovered configuration mappers. For instance, the Node module does this to modify the `node_type` configuration entity:

```
/**
 * Implements hook_config_translation_info_alter().
 */
function node_config_translation_info_alter(&$info) {
 $info['node_type']['class'] = 'Drupal\node\ConfigTranslation\
NodeTypeMapper';
}
```

This updates the `node_type` definition to use the `\Drupal\node\ConfigTranslation\NodeTypeMapper` custom mapper class. This class adds the node type's title as a configurable translation item.

### Translating views

Views are configuration entities. When the Configuration translation module is enabled, it is possible to translate Views. This will allow you to translate display titles, exposed form labels, and other items. Refer to the [Creating a multilingual view](#) recipe in this chapter for more information.

## See also

- ▶ In recipe [Creating a Multilingual View of Chapter 8, Multilingual and Internationalization](#)

## Translating content

The content translation module provides a method for translating content entities, such as nodes and blocks. Each content entity needs to have translation enabled, which allows you to granularly decide what properties and fields are translated.

Content translations are duplications of the existing entity but flagged with a proper language code. When a visitor uses a language code, Drupal attempts to load content entities using that language code. If a translation is not present, Drupal will render the default nontranslated entity.

## Getting ready

Your Drupal site needs to have two languages enabled in order to use Content translation. Install **Spanish** from the **Languages** interface.

## How to do it...

1. Go to **Extend**, and install the **Content translation** module. It will prompt you to enable the **Language** modules to be installed as well if they are not.
2. After the module is installed, go to **Configuration**. Go to the **Content language and translation** page under the **Regional and Language** section.
3. Check the checkbox next to the **Content to expose** settings for the current content types.
4. Enable the content translation for the **Basic** page and keep the provided default settings that enable translation for each field. Click on **Save configuration**:

The screenshot shows the 'Content language and translation' configuration page for the 'Basic page' content type. At the top, there is a 'Default language' dropdown set to 'Site's default language (English)'. Below it, a note says 'Explanation of the language options is found on the [languages list page](#)'. A checked checkbox 'Show language selector on create and edit pages' is also present. On the left, a vertical list of fields for the 'Basic page' content type is shown, each with a checked checkbox: Title, Authored by, Publishing status, Authored on, Changed, Promoted to front page, Sticky at top of lists, URL alias, Body, and Email. At the bottom, a blue 'Save configuration' button is visible.

5. First, create a new Basic page node. We will create this in the site's default language.

6. When viewing the new node, click on the **Translate** tab. From the **Spanish** language row, click on **Add** to create a translated version of the node:

The screenshot shows a table titled "Translations of About us". The columns are LANGUAGE, TRANSLATION, STATUS, and OPERATIONS. There are two rows. The first row is for "English (Original language)" with "About us" in the TRANSLATION column, "Published" in the STATUS column, and an "Edit" button in the OPERATIONS column. The second row is for "Spanish" with "n/a" in the TRANSLATION column, "Not translated" in the STATUS column, and an "Add" button in the OPERATIONS column.

| LANGUAGE                    | TRANSLATION | STATUS         | OPERATIONS           |
|-----------------------------|-------------|----------------|----------------------|
| English (Original language) | About us    | Published      | <a href="#">Edit</a> |
| Spanish                     | n/a         | Not translated | <a href="#">Add</a>  |

7. The content will be prepopulated with the default language's content. Replace the title and body with the translated text:

The screenshot shows a form titled "Crear traducción Español de About us". It has fields for "Título" (About us) and "Cuerpo (Edit summary)" (This piece of content describes the website and what we do.). Below the body is a rich text editor toolbar. At the bottom, there are buttons for "Formato de texto" (HTML Básico), "Acerca de formatos de texto", "Guardar y mantener publicado. (esta traducción)", and "Vista previa".

8. Click on **Save and keep published (this translation)** to save the new translation.

## How it works

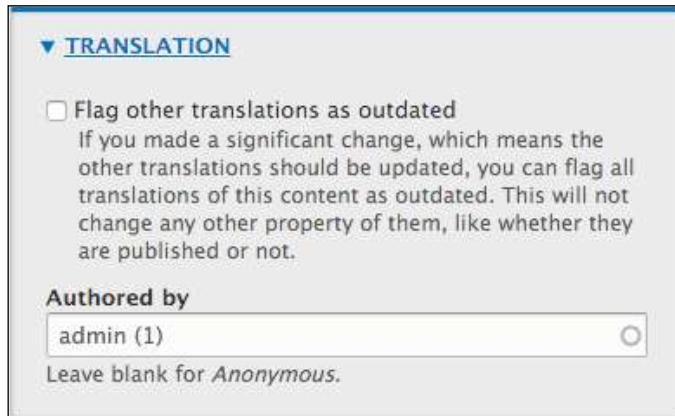
The Content translation module works by utilizing language code flags. All content entities and field definitions have a language code key. A content entity has a language code column, which specifies what language the content entity is for. Field definitions also have a language code column, which is used to identify the translation for the content entity. Content entities can provide handler definitions for handling translations, or else the Content translation module will provide its own.

Each entity and field record is saved with the proper language code to use. When an entity is loaded, the current language code is taken into consideration to ensure that the proper entity is loaded.

## There's more

### Flagging translations as outdated

The Content translation module provides a mechanism to flag translated entities as possibly being outdated. The **Flag other translations as outdated** flag provides a way to make a note of entities that will need updated translations:



This flag does not change any data but rather provides a moderation tool. This makes it easy for translators to identify content, which has been changed and requires updating. The translation tab for the content entity will highlight all translations, which are still marked as outdated. As they are changed, the editor can uncheck the flag.

## Translating content links

Mostly Drupal menus contain links to nodes. Menu links are not translated by default, and the **Custom menu links** option must be enabled under **Content translation**. You will need to translate node links manually from the menu administration interface.

Enabling a menu link from the node create and edit form will not work with translations. If you edit the menu settings from a translation, it will edit the nontranslated menu link.

## Defining translation handlers for entities

The Content translation module requires entity definitions to provide information about translation handlers. If this information is missing, it will provide its own defaults. *The Entity API* is covered in *Chapter 10*, but we will quickly discuss how the content translation module interacts with the Entity API.

Content entity definitions can provide a `translation` handler. If not provided, it will default to `\Drupal\content_translation\ContentTranslationHandler`. A node provides this definition and uses it to place the content translation information into the vertical tabs.

The `content_translation_metadata` key defines how to interact with translation metadata information, such as flagging other entities as outdated. The `content_translation_deletion` key provides a form class to handle entity translation deletion.

Currently, as of 8.0.1, no core modules provide implementations that override the default `content_translation_metadata` or `content_translation_deletion`.

### See also

- ▶ *Chapter 10, The Entity API*

## Creating multilingual views

Views, being configuration entities, are available for translation. However, the power of multilingual views does not lie just in configuration translation. Views allow you to build filters that react to the current language code. This ensures that the content, which has been translated for the user's language, is displayed.

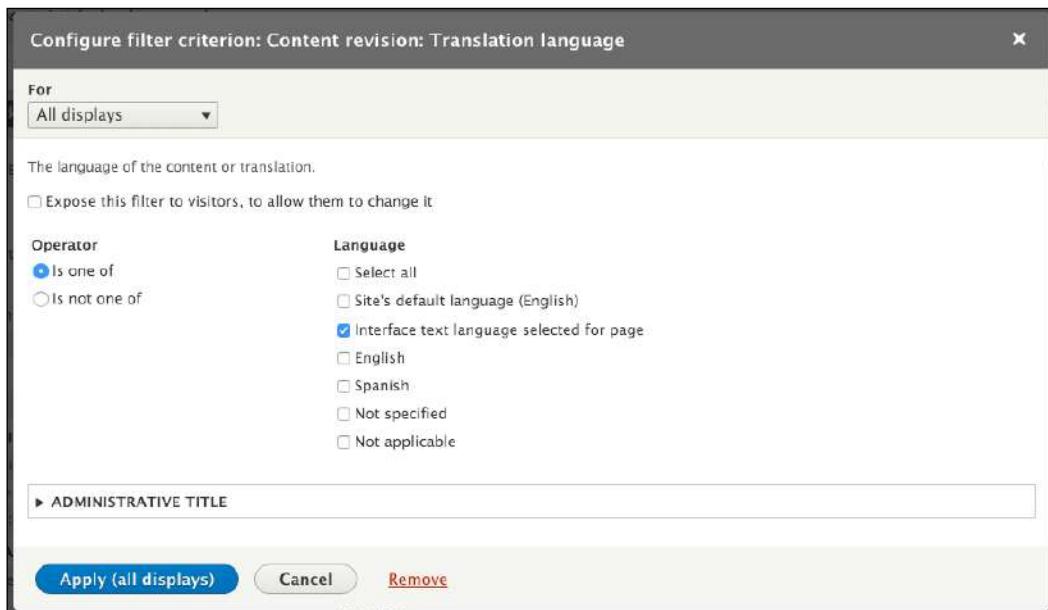
In this recipe, we will create a multilingual view that provides a block showing recent articles. If there is no content, we will display a translated `no results` message.

## Getting ready

Your Drupal site needs to have two languages enabled in order to use **Content Translation**. Install **Spanish** from the **Languages** interface. Enable content translation for **Articles**. You will also need to have some translated content as well.

## How to do it...

1. Go to **Views** from **Structure**, and click on **Add new view**.
2. Provide a view name, such as **Recent articles**, and change the type of content to **Article**. Mark that you would like to **Create a block** and then click on **Save** and **edit**.
3. Add a new **Filter criteria**. Search for **Translation language** and add the filter for **Content**. Set the filter to check the **Interface text language selected for page**. This will only display that the content that has been translated or the base language is the current language:



4. Add a **No results** behavior to the **Text** area option. Provide some sample text, such as *Currently no recent articles*.

5. Save the view.
6. Click on the **Translate** tab. Click on **Add** for the **Spanish** row to translate the view for the language.
7. Expand the **Master display settings** and then the **Recent articles** display options fieldsets. Modify the **Display title** option to provided a translated title:

|                                  |                                      |
|----------------------------------|--------------------------------------|
| Display title<br>Recent Articles | Display title<br>Artículos Recientes |
|----------------------------------|--------------------------------------|

8. Expand **No results behavior** to modify the text on the right-hand side of the screen using the textbox on the left-hand side of the screen as the source for the original text:

(EMPTY) TEXT

A string to identify the handler instance in the admin UI.  
(Empty)

THE FORMATTED TEXT OF THE AREA

The formatted text of the area

Currently no recent articles

The formatted text of the area

Actualmente no hay artículos recientes

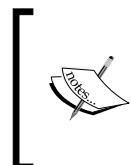
9. Click on **Save translation**.

10. Place the block on your Drupal site. Visit the site through /es and notice the translated Views block:



## How it works...

Views provide the Translation language filter that builds off of this element. The Views plugin systems provides a mechanism for gathering and displaying all available languages. These will be saved as a token internally and then substituted with the actual language code when the query is executed. If a language code is no longer available, you will see the Content language for selected page and Views will fall back to the current language when viewed.



You will come across this option when editing views provided by Drupal core or contributed modules. While this is not an option in the user interface, it is a default practice to add a language filter defined as **\*\*\*LANGUAGE\_language\_content\*\*\***, which will force the view to be multilingual.

The filter tells **Views** to query based on the language code of the entity and its fields.

Views are configuration entities. The Configuration translation module allows you to translate views. Views can be translated from the main Configuration translation screens from the Configuration area or by editing individual views.

Most translation items will be under the **Master display settings** tab unless overridden in specific displays. Each display type will also have its own specific settings.

## There's more...

### Translating exposed form items and filters

Each view has the ability to translate the exposed form from the Exposed Form section. This does not translate the labels on the form but the form elements. You have the ability to translate the submit button text, reset button label, sort label, and how ascending or descending should be translated.

You can translate the labels for exposed filters from the Filters section. Each exposed filter will show up as a collapsible fieldset allowing you to configure the administrative label and front facing label.

**▼ (EMPTY) COMBINE**

A string to identify the handler instance in the admin UI.  
(Empty)

A string to identify the handler instance in the admin UI.

**▼ NAME OR EMAIL CONTAINS EXPOSED**

|                        |                           |
|------------------------|---------------------------|
| <b>Label</b>           | <b>Label</b>              |
| Name or email contains | Nombre o correo contienen |
| <b>Description</b>     | <b>Description</b>        |
| (Empty)                |                           |

**► (EMPTY) GROUP**

By default, available translations need to be imported through the global interface translation context.

### Translating display and row format items

Some display formats have translatable items. These can be translated in each display mode's section. For example, the following items can be translated with their display format:

- ▶ The Table format allows you to translate the table summary
- ▶ The RSS feed format allows you to translate the feed description
- ▶ The Page format allows you to translate the page's title
- ▶ The Block format allows you to translate the block's title

## Translating page display menu items

Custom menu links can be translated through the Content translation module. Views using a page display; however, they do not create custom menu link entities. The Views module takes all views with a page display and registers their paths into the routing system directly, as if defined in a module's `routing.yml` file.

The screenshot shows the 'PAGE DISPLAY OPTIONS' configuration interface. It is divided into two main sections: 'LIST MENU' and 'PEOPLE TAB OPTIONS'. Each section contains a 'Title' field and a 'Description' field, with English values on the left and Spanish translations on the right.

| Section            | Title                                 | Description                                                                               |
|--------------------|---------------------------------------|-------------------------------------------------------------------------------------------|
| LIST MENU          | List                                  | Find and manage people interacting with your site.                                        |
|                    | <input type="text" value="Lista"/>    | <input type="text" value="Encontrar y gestionar qué personas interactúan con su sitio."/> |
| PEOPLE TAB OPTIONS | People                                | Manage user accounts, roles, and permissions.                                             |
|                    | <input type="text" value="Usuarios"/> | <input type="text" value="Gestionar las cuentas, roles y permisos de usuarios."/>         |

For example, the `People` view that lists all users can be translated to have an updated tab name and link description.

### See also

- ▶ [Chapter 3, Displaying Content through Views](#)

# 9

# Configuration Management – Deploying in Drupal 8

In this chapter, we will explore the configuration management system and how to deploy configuration changes. Here is a list of the recipes covered in this chapter:

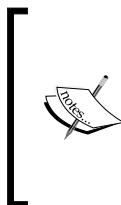
- ▶ Importing and exporting configurations
- ▶ Synchronizing site configurations
- ▶ Using command-line workflow processes
- ▶ Using the filesystem for configuration storage

## Introduction

Drupal 8 provides a new, unified system for managing configurations. In Drupal 8, all configurations are saved in configuration entities that match a defined configuration schema. This system provides a standard way of deploying the configuration between Drupal site environments and updating the site configuration.

Once the configuration has been created, or imported, it goes into an immutable state. If a module tries to install the configuration that exists, it will throw an exception and be prevented. Outside the typical user interface, the configuration can only be modified through the configuration management system.

The configuration management system can be manipulated through a user interface provided by the Configuration management module or through the command-line interface tools. These tools allow you to follow the development paradigm of utilizing a production site and development site where changes are made to the development site and then pushed to production.



Instead of creating two different Drupal sites for the recipes in this chapter, you can utilize the Drupal multisite functionality. For more information, refer to the *Installing Drupal* recipe of Chapter 1, *Up and Running with Drupal 8*. Note that if you use a multisite, you need to clone your development site's database into the site acting as your production site to replicate a realistic development and production site workflow.

## Importing and exporting configurations

Configuration management in Drupal 8 provides a solution to a common problem when working with a website across multiple environments. No matter what the workflow pattern is, at some point the configuration needs to move from one place to another, such as from production to a local environment. When pushing the development work to production, you need to have some way to put the configuration in place.

Drupal 8's user interface provides a way to import and export configuration entities via the YAML format. In this recipe, we will create a content type, export its configuration, and then import it into another Drupal site.

In this recipe, we will export a single configuration entity from a development site. The configuration YAML export will be imported into the production site in order to update its configuration.

### Getting ready

You will need a base Drupal site to act as the development site. Another Drupal site, which is a clone of the development site, must be available to act as the production Drupal site.

### How to do it...

1. To get started, create a new content type on the development site. Name the content type **Staff Page** and click on **Save and manage fields** to save the content type. We will not be adding any additional fields.
2. Once the content type has been saved, visit **Extend** and install the **Configuration Manager** module if it is not installed:

**CKEditor** ► WYSIWYG editing for rich text fields using CKEditor.

**Color** ► Allows administrators to change the color scheme of compatible themes.

**Comment** ► Allows users to comment on and discuss published content.

**Configuration Manager** ▾ Allows administrators to manage configuration changes.  
Machine name: config  
Version: 8.0.1  
Required by: Configuration Update Base (disabled), Configuration Update Reports (disabled), Features (disabled), Features UI (disabled)

**Contact** ► Enables the use of both personal and site-wide contact forms.

3. From your Drupal site's **Configuration** page, go to **Configuration synchronization** under the **Development** group. This section allows you to import and export configuration:

**Home** **Manage** **Shortcuts** **admin**

**Content** **Structure** **Appearance** **Extend** **Configuration** **People** **Reports** **Help**

**when displayed.**

**DEVELOPMENT**

- Performance**  
Enable or disable page caching for anonymous users and set CSS and JS bandwidth optimization options.
- Logging and errors**  
Settings for logging and alerts modules. Various modules can route Drupal's system events to different destinations, such as syslog, database, email, etc.
- Maintenance mode**  
Take the site offline for maintenance or bring it back online.
- Configuration synchronization**  
Import and export your configuration.

**Shortcuts**  
Add and modify shortcut sets.

**MEDIA**

- File system**  
Tell Drupal where to store uploaded files and how they are accessed.
- Image styles**  
Configure styles that can be used for resizing or adjusting images on display.
- Image toolkit**  
Choose which image toolkit to use if you have installed optional toolkits.

4. Click on the **Export** tab at the top of the page. The default page will be for a **Full archive** export, which contains the configuration of your entire Drupal site. Click on the **Single item** subtab to export a single configuration entity instead:

The screenshot shows the 'Single export' interface. At the top, there is a navigation bar with links for Home, Manage, Shortcuts, and admin. Below the navigation bar, the title 'Single export' is displayed with a star icon. A horizontal menu bar follows, containing 'Synchronize', 'Import', 'Export', and 'Features'. Underneath this, two tabs are visible: 'Full archive' and 'Single item', with 'Single item' being the active tab.

5. Select **Content type** from the **Configuration type** drop-down menu. Then, choose your content type from the **Configuration name** drop-down menu. Its configuration will populate the configuration textbox:

The screenshot shows the configuration export interface for a 'Content type'. It includes fields for 'Configuration type' (set to 'Content type') and 'Configuration name' (set to 'Staff pages (staff\_pages)'). Below these, a large text area titled 'Here is your configuration:' displays the YAML configuration for the 'Staff pages' content type. At the bottom of the text area, it says 'Filename: node.type.staff\_pages.yml'.

```
uuid: f33fa230-28ee-4f56-8751-81c30742e04d
langcode: en
status: true
dependencies:
 module:
 - menu_ui
third_party_settings:
 menu_ui:
 available_menus:
 - main
 parent: 'main'
name: 'Staff pages'
type: staff_pages
description: 'Add a new staff page to add staff profile page on the site.'
help: ''
new_revision: false
preview_mode: 1
display_submitted: true
```

6. Copy the YAML content from the textbox so that you can import it into your other Drupal site.
7. On your production Drupal site, install the Configuration management module just as you did for the development site, if it is not yet installed.
8. Visit the **Configuration synchronization** page and click on the **Import** tab.
9. Click on **Single item** and select **Content type** from the **Configuration type**:

The screenshot shows the 'Single import' page of the Configuration Synchronization module in Drupal. The top navigation bar includes 'Home', 'Manage', 'Shortcuts', and a user icon for 'admin'. Below the navigation is a title 'Single import' with a star icon. A horizontal menu bar contains 'Synchronize', 'Import' (which is selected), 'Export', and 'Features'. Underneath the menu, there are two tabs: 'Full archive' and 'Single item', with 'Single item' being the active tab. A breadcrumb trail at the top indicates the path: 'Home » Administration » Configuration » Development » Synchronize'. Below the breadcrumb, a sub-instruction reads: 'Import a single configuration item by pasting its YAML structure into the text field.' A configuration form is present, labeled 'Configuration type \*'. A dropdown menu is open, showing 'Content type' as the selected option.

10. Paste your exported configuration YAML into the textbox and click on **Import**:

The screenshot shows the 'Synchronize' configuration import interface. At the top, there are two tabs: 'Full archive' and 'Single item', with 'Single item' being selected. Below the tabs, the URL is 'Home » Administration » Configuration » Development » Synchronize'. A sub-instruction says 'Import a single configuration item by pasting its YAML structure in the text field.' Under 'Configuration type \*', a dropdown menu is set to 'Content type'. The main area contains a large text box labeled 'Paste your configuration here \*' containing the following YAML code:

```
uuid: f33fa230-28ee-4f56-8751-81c30742e04d
langcode: en
status: true
dependencies:
 module:
 - menu_ui
third_party_settings:
 menu_ui:
 available_menus:
 - main
 parent: 'main'
name: 'Staff pages'
type: staff_pages
description: 'Add a new staff page to add staff profile page on the site.'
help: ''
new_revision: false
preview_mode: 1
display_submitted: true
```

At the bottom of the form, there is an 'ADVANCED' link and a prominent blue 'Import' button.

11. Click on **Confirm** on the confirmation form to finalize your import to the production Drupal site for your custom content type.
12. Visit the **Structure** page and then the **Content Types** page to verify that your content type has been imported.

## How it works...

At the most basic level, configurations are just a mapping of keys and values, which can be represented as a PHP array and translated into YAML format.

Configuration management uses schema definitions for configuration entities. The schema definition provides a configuration namespace and the available keys and data types. The schema definition provides a typed data definition for each option that allows validation of the individual values and configuration as a whole.

The export process reads the configuration data and translates it into YAML format. The configuration manager then receives the configuration in the form of YAML and converts it back to a PHP array. The data is then updated in the database.

When importing the configuration, Drupal checks the value of the configuration YAML's `uuid` key, if present, against any current configuration with the same **Universally Unique Identifier (UUID)**. A UUID is a pattern used in software to provide a method of identifying an object across different environments. This allows Drupal to correlate a piece of data from its UUID since the database identifier can differ across environments. If the configuration item has a matching machine name but a mismatching UUID, an error will be thrown.

## There's more...

### Configuration dependencies

Configuration entities define dependencies when they are exported. The dependency definitions ensure that the configuration entity's schema is available and other module functionality.

When you review the configuration export for `field.storage.node.body.yml`, it defines `node` and `text` as dependencies:

```
dependencies:
 module:
 - node
 - text
```

If the `node` or `text` module is not enabled, the import will fail and throw an error.

## Saving to a YAML file for a module's configuration installation

*Chapter 6, Creating Forms with the Form API*, the providing configuration on install or update, discusses how to use a module to provide configurations on the module's installation. Instead of manually writing configuration YAML files for installation, the Configuration management module can be used to export configurations and save them in your module's config/install directory.

Any item exported through the user interface can be used. The only requirement is that you need to remove the `uuid` key, as it denotes the site's UUID value and invalidates the configuration when it makes an attempt at installation.

## Configuration schemas

The configuration management system in Drupal 8 utilizes the configuration schema to describe configurations that can exist. Why is this important? It allows Drupal to properly implement typed data on stored configuration values and validate them, providing a standardized way of handling configurations for translation and configuration items.

When a module uses the configuration system to store data, it needs to provide a schema for each configuration definition it wishes to store. The schema definition is used to validate and provide typed data definitions for its values.

The following code defines the configuration schema for the `navbar_awesome` module, which holds two different Boolean configuration values:

```
navbar_awesome.toolbar:
 type: config_object
 label: 'Navbar Awesome toolbar settings'
 mapping:
 cdn:
 type: boolean
 label: 'Use the FontAwesome CDN library'
 roboto:
 type: boolean
 label: 'Include Roboto from Google Fonts CDN'
```

This defines the `navbar_awesome.toolbar` configuration namespace; it belongs to the `navbar_awesome` module and has the `toolbar` configuration. We then need to have two `cdn` and `roboto` subvalues that represent typed data values. A configuration YAML for this schema would be named `navbar_awesome.toolbar.yml` after the namespace, and it contains the following code:

```
cdn: true
roboto: true
```

In turn, this is what the values will look like when represented as a PHP array:

```
[
 'navbar_awesome' => [
 'cdn' => TRUE,
 'roboto' => TRUE,
]
]
```

The configuration factory classes then provide an object-based wrapper around these configuration definitions and provide validation of their values against the schema. For instance, if you try to save the `cdn` value as a string, a validation exception will be thrown.

## See also

- ▶ *Chapter 4, Extending Drupal*
- ▶ configuration schema/metadata in the Drupal.org community handbook at <https://www.drupal.org/node/1905070>

## Synchronizing site configurations

A key component to manage a Drupal website is configuration integrity. A key part of maintaining this integrity is ensuring that your configuration changes that are made in development are pushed upstream to your production environments. Maintaining configuration changes by manually exporting and importing through the user interface can be difficult and does not provide a way to track what has or has not been exported or imported. At the same time, manually writing module hooks to manipulate the configuration can be time consuming. Luckily, the configuration management solution provides you with the ability to export and import the entire site's configuration.

A site export can only be imported into another copy of itself. This allows you to export your local development environment's configuration and bring it to staging or production without modifying the content or the database directly.

In this recipe, we will export the development site's complete configuration entities' definitions. We will then take the exported configuration and import it into the production site. This will simulate a typical deployment of a Drupal site with changes created in development that is ready to be released in production.

## Getting ready

You will need a base Drupal site to act as the development site. Another Drupal site, which is a clone of the development site, must be available to act as the production Drupal site.

You will need to get the Configuration management module enabled.

## How to do it...

1. Visit the **Configuration** page and go to **Configuration synchronization**.
2. Navigate to the **Export** tab, and click on the **Export** button to begin the export and download process:



3. Save the gzipped tarball; this contains an archive of all the site's configuration as YAML.
4. Visit your other Drupal site and navigate to its **Configuration synchronization** page.
5. Click on the **Import** tab and then on the **Full archive** tab. Use the **Configuration archive** file input, and click on **Choose File** to select the tarball you just downloaded. Click on **Upload** to begin the import process.

6. You will be taken to the **Synchronize** tab to review changes to be imported:

Compare the configuration uploaded to your sync directory with the active configuration before completing the import.

**3 changed**

| NAME             | OPERATIONS              |
|------------------|-------------------------|
| contact.settings | <b>View differences</b> |
| system.site      | <b>View differences</b> |
| user.mail        | <b>View differences</b> |

**Import all**

7. Click on **Import all** to update the current site's configuration to the items in the archive.  
 8. A batch operation will begin with the import process:

**Synchronize** ☆

- Synchronize
- Import
- Export
- Features

Home » Administration » Configuration » Development

✓ The configuration was imported successfully.

## How it works...

The **Configuration synchronization** form provides a way to interface with the config database table for your Drupal site. When you visit the **Export** page and create the tarball, Drupal effectively dumps the contents of the config table. Each row represents a configuration entity and will become its own YAML file. The contents of the YAML file represent its database value.

When you import the tarball, Drupal extracts its content. The files are placed in the available CONFIG\_SYNC\_DIRECTORY directory. The synchronization page parses the configuration entity YAMLs and provides a difference check against the current site's configuration. Each configuration item can be reviewed, and then all the items can be imported. You cannot choose to selectively import individual items.

## There's more...

### Universally Unique Identifier

When a Drupal site is installed, the UUID is set. This UUID is added to the exported configuration entities and is represented by the `uuid` key. Drupal uses this key to identify the source of the configuration. Drupal will not synchronize configurations that do not have a matching UUID in their YAML definition.

You can review the site's current UUID value by reviewing the `system.site` configuration object.

### A synchronization folder

Drupal uses a synchronization folder to hold the configuration YAML files that are to be imported into the current site. This folder is represented by the `CONFIG_SYNC_DIRECTORY` constant. If you have not defined this in the global `$config_directories` variable in your site's `settings.php`, then it will be a randomly named directory in your site's file directory.



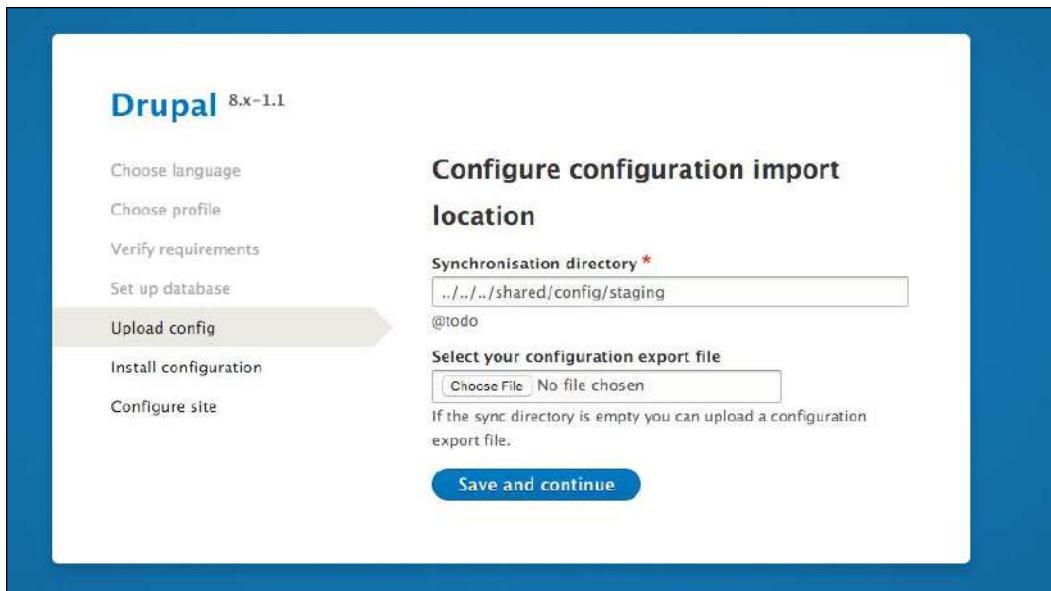
When Drupal 8 entered its beta release cycle, this folder was referenced as a staging folder and referenced by the `CONFIG_STAGING_DIRECTORY`. This is now deprecated; however, the internals of the configuration management system support reading `CONFIG_STAGING_DIRECTORY` as `CONFIG_SYNC_DIRECTORY`. This will be removed in Drupal 9.

The synchronization form will use the configuration management discovery service to look for configuration changes to be imported from this folder.

### Installing a configuration from a new site

Drupal's configuration management system will not allow the import of configuration entities that originated at a different Drupal site. When a Drupal site is installed, the `system.site` configuration entity saves a UUID for the current site instance. Only cloned versions of this site's database can accept configuration imports from it.

The configuration installer profile is a custom distribution, which will allow you to import the configuration despite the configuration's site UUID. The profile doesn't actually install itself. When you use the profile, it will provide an interface to upload a configuration export that will then be imported, as shown in the following screenshot:



The distribution can be found at [https://www.drupal.org/project/config\\_installer](https://www.drupal.org/project/config_installer).

## Using command-line workflow processes

Drupal 8's configuration systems solve many problems encountered when exporting and deploying configurations in Drupal 7. However, the task of synchronizing the configuration is still a user interface task and requires the manipulation of archive files that contain the configuration exports for a Drupal 8 site.

Configuration management can be done on the command line through Drush without requiring it to be installed. This mitigates any requirement to log in to the production website to import changes. It also opens the ability for more advanced workflows that place the configuration in version control.

In this recipe, we will use Drush to export the development site's configuration to the filesystem. The exported configuration files will then be copied to the production site's configuration directory. Using Drush, the configuration will be imported into production to complete the deployment.

## Getting ready...

You will need a base Drupal site to act as the development site. Another Drupal site, which is a clone of the development site, must be available to act as the production Drupal site.

This recipe uses Drush. If you do not have Drush installed, instructions can be found at <http://docs.drush.org/en/master/install/>. Drush needs to be installed at both the locations where your Drupal sites are located.

## How to do it...

1. For demonstration purposes, change your development site's name to `Drush Config Sync Demo!`. This way, there is at least one configuration change to be imported to the production Drupal site.
2. Open a command-line terminal and change your directory to the working directory of your development Drupal site.
3. Use the `drush config-export` command to export the configuration to a directory. The command will default to the `sync` configuration directory defined in your Drupal 8 site.



If you have not explicitly defined a `sync` directory, Drupal automatically creates a protected folder in the current site's uploaded files' directory, with a unique hash suffix on the directory name.

4. You will receive a message that the configuration has been exported to the directory.
5. Using a method of your choice, copy the contents of the configuration `sync` folder to your other Drupal sites that match the `configuration sync` folder. For example, a default folder generated by Drupal can be `sites/default/files/config_XYZ/sync`.
6. Open a command-line terminal and change your directory to your production Drupal site's working directory.
7. Use the `drush config-import` command to begin the process of importing your configuration.

8. Review the changes made to the configuration entity keys and enter `y` to confirm the changes:



The screenshot shows a terminal window titled "www — bash — 85x19". The command entered is "drush config-get system.site". The output displays various site configuration settings, including UUID, name, mail, slogan, page, admin compact mode, weight select max, langcode, and default langcode. The terminal prompt "Matts-MacBook-Air:www mglaman\$" is visible at the bottom.

```
Matts-MacBook-Air:www mglaman$ drush config-get system.site
uuid: d847568b-4c6d-4826-9f41-65d40ede72a7
name: Site-Install
mail: admin@example.com
slogan: ''
page:
 403: ''
 404: ''
 front: /node
admin_compact_mode: false
weight_select_max: 100
langcode: en
default_langcode: en

Matts-MacBook-Air:www mglaman$
```

9. Check whether your configuration changes have been imported.

## How it works...

The Drush command-line tool is able to utilize the code found in Drupal to interact with it. The config-export command replicates the functionality provided by the Configuration management module's full site export. However, you do not need to have the Configuration management module enabled for the command to work. The command will extract the available site configuration and write it to a directory, which is unarchived.

The config-import command parses the files in a directory. It will make an attempt to run a difference check against the YAML files like the Configuration management module's synchronize overview form does. It will then import all the changes.

## There's more...

### Drush config-pull

Drush provides a way of simplifying the transportation of configuration between sites. The config-pull command allows you to specify two Drupal sites and move the export configuration between them. You can either specify a name of a subdirectory under the /sites directory or a Drush alias.

The following command will copy a development site's configuration and import it into the staging server's site:

```
drush config-pull @mysite.local @mysite.staging
```

Additionally, you can specify the `--label` option. This represents a folder key in the `$config_directories` setting. The option defaults to `sync` automatically. Alternatively, you can use the `--destination` parameter to specify an arbitrary folder that is not specified in the setting of `$config_directories`.

## Using the Drupal Console

Drush has been part of the Drupal community since Drupal 4.7 and is a custom built command-line tool. The Drupal Console is a Symfony Console-based application used to interact with Drupal. The Drupal Console project provides a means for configuration management over the command line.



You can learn more about the Drupal Console in *Chapter 13, Drupal CLI* or at <http://www.drupalconsole.com/>.



The workflow is the same, except the naming of the command. The configuration export command is `config:export`, and it is automatically exported to your system's temporary folder until a directory is passed. You can then import the configuration using the `config:import` command.

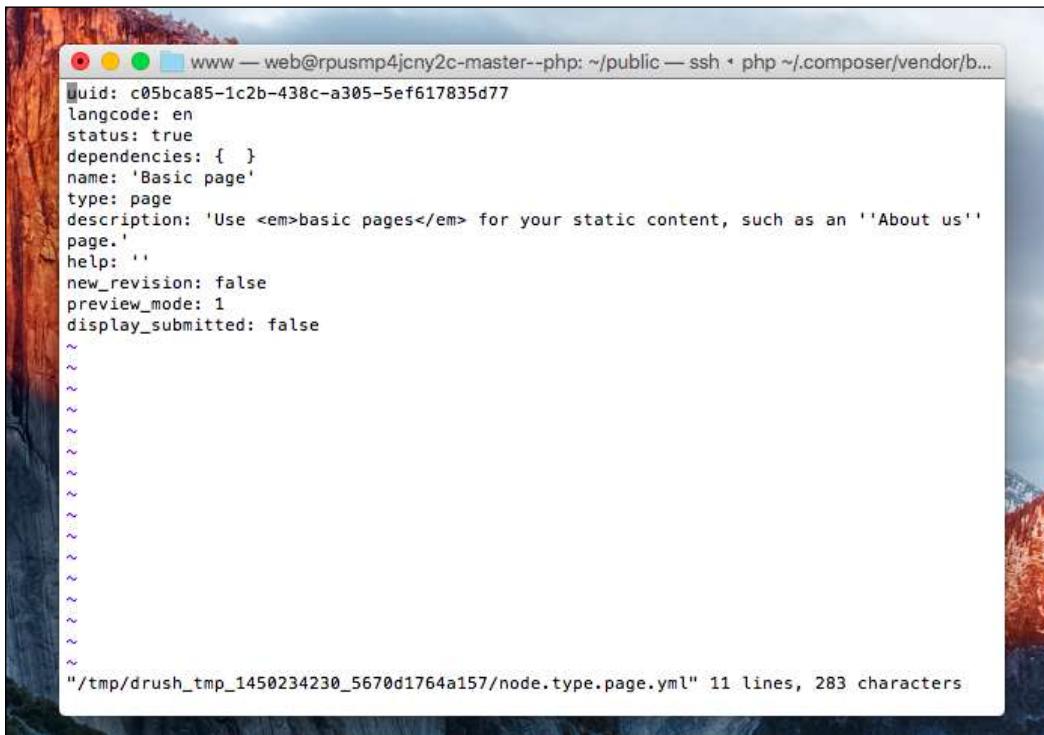
## Editing the configuration from the command line

Both Drush and Drupal Console support the ability to edit the configuration through the command line in YAML format. Both the tools operate in the same fashion and have similar command names:

- ▶ Drush: `config-edit [name]`
- ▶ Console: `config:edit [name]`

The difference is that Drush will list all the available options to be edited if you do not pass a name, while Console allows you to search.

When you edit a configuration item, your default terminal-based text editor will open. You will be presented with a YAML file that can be edited. Once you save your changes, the configuration is then saved on your Drupal site:

A screenshot of a terminal window titled "www — web@rpusmp4jcny2c-master--php: ~/public — ssh < php ~./composer/vendor/b...". The window displays a YAML configuration file for a basic page. The file contains fields such as uid, langcode, status, dependencies, name, type, description, help, new\_revision, preview\_mode, and display\_submitted. The description field includes a note about using basic pages for static content like 'About us'. The file ends with a summary line: "/tmp/drush\_tmp\_1450234230\_5670d1764a157/node.type.page.yml" 11 lines, 283 characters.

## Exporting a single configuration item

Both Drush and Console provide their own mechanisms for exporting a single configuration entity:

- ▶ Drush: config-get [name]
- ▶ Console: config:debug [name]

Drush will print the configuration's output to the terminal, while Console's default behavior is to write the output to the file disk. For example, the following commands will output the values from system.site in YAML format:

```
$ drush config-get system.site
$ drupal config:debug system.site
```

```
Matts-MacBook-Air:www mglaman$ drush config-get system.site
uid: d847568b-4c6d-4826-9f41-65d40ede72a7
name: Site-Install
mail: admin@example.com
slogan: ''
page:
 403: ''
 404: ''
 front: /node
admin_compact_mode: false
weight_select_max: 100
langcode: en
default_langcode: en
Matts-MacBook-Air:www mglaman$
```

## Using version control and command-line workflow

A benefit of having the configuration exportable to YAML files is the fact that the configuration can be kept in version control. The Drupal site's CONFIG\_SYNC\_DIRECTORY directory can be committed to version control to ensure that it is transported across environments and properly updated. Deployment tools can then use Drush or Console to automatically import changes.

The config-export command provided by Drush provides the Git integration:

```
drush config-export --add
```

Appending the --add option will run git add -p for an interactive staging of the changed configuration files:

```
drush config-export --commit --message="Updating configuration "
```

The --commit and optional --message options will stage all configuration file changes and commit them with your message:

```
drush config-export --push --message="Updating configuration "
```

Finally, you can also specify --push to make a commit and push it to the remote repository.

## See also

- ▶ [Chapter 13, The Drupal CLI](#)
- ▶ [Drush at `http://docs.drush.org/en/master/`](http://docs.drush.org/en/master/)
- ▶ [Drupal Console at `http://www.drupalconsole.com/`](http://www.drupalconsole.com/)

## Using the filesystem for configuration storage

Originally, Drupal 8 utilized the filesystem for the configuration using the database as a mere cache. During the development cycle, it was changed to keep the configuration in the database and use YAMLs and disks for synchronization. It is possible to enable this setting and have Drupal primarily utilize the disk for configuration storage. This change needs to be defined before you install your Drupal site and existing installations cannot be converted.

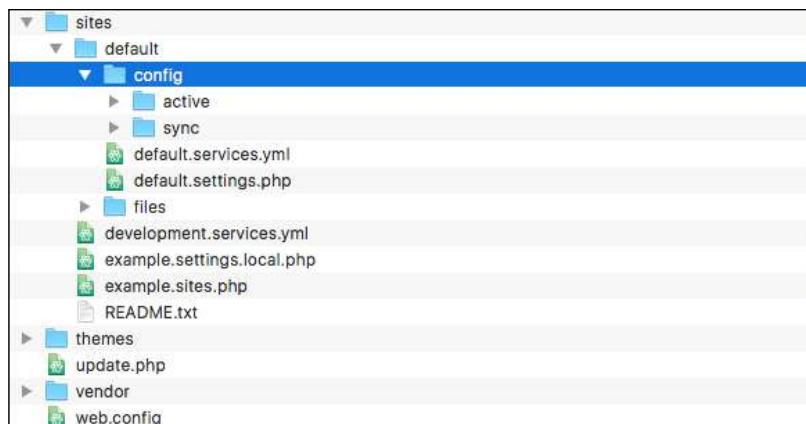
In this recipe, we will configure Drupal to write and read its configuration from the filesystem instead of the database. Configuration changes will automatically be imported on the cache rebuild.



Overall, this is an advanced topic. This recipe will explain how to use an alternative storage for the configuration in your Drupal site. For detailed information on Drupal's change from disk to database storage, refer to the issue that committed the change at <https://www.drupal.org/node/2161591>.

## How to do it...

1. Create a folder called `config` in the `sites/default` directory. Create the `active` and `sync` folders in `config`. These will hold the configuration YAMLs for your Drupal site:



2. Copy the `default.settings.php` file and name it `settings.php`.
3. In the editor of your choice, edit the newly created `settings.php`.
4. Find the empty `$config_directories` array and provide definitions that point to the newly created directories:

```
$config_directories = [
 CONFIG_ACTIVE_DIRECTORY => __DIR__ . 'config/active',
 CONFIG_SYNC_DIRECTORY => __DIR__ . 'config-sync',
];
```

The `CONFIG_ACTIVE_DIRECTORY` constant represents `active` and will be deprecated in Drupal 9, since this is an alternative workflow. `CONFIG_SYNC_DIRECTORY` represents the synchronization folder for the confirmation. The `__DIR__` PHP magic constant will represent the file's current working directory, providing an absolute path to the configuration folders.

5. Find the section for **Active configuration settings**. At the time of writing, the document block for the setting is at line 584.
6. Remove the `#` to uncomment `$settings['bootstrap_config_storage']`. This will override the default configuration storage backend to use the file storage:

```
$settings['bootstrap_config_storage'] = array('Drupal\Core\Config\BootstrapConfigStorageFactory', 'getFileStorage');
```

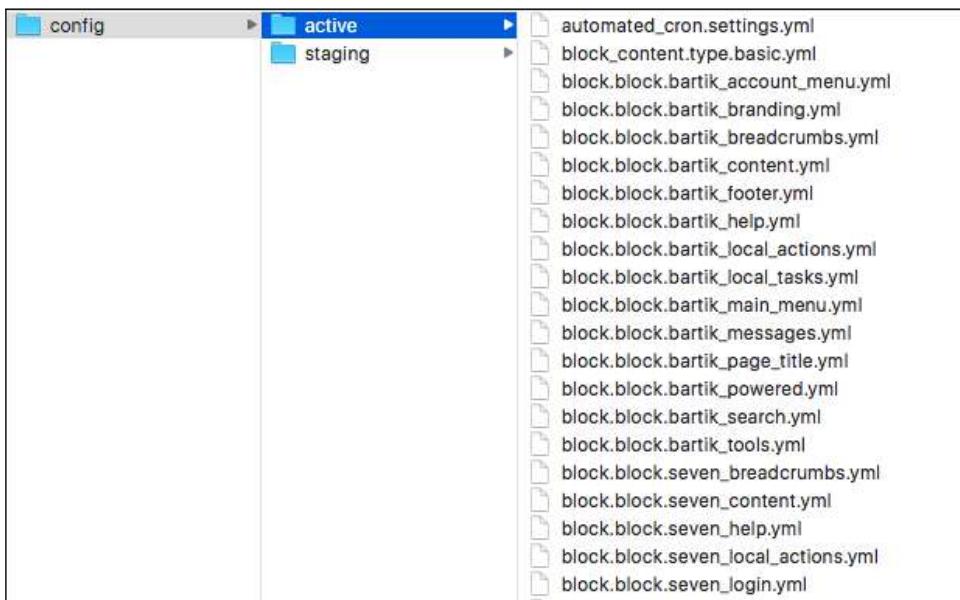
The `\Drupal\Core\Config\BootstrapConfigStorageFactory` class method's `getFileStorage` will return an initiated class that provides a storage backend for the configuration. We are telling it to return a file storage service.

7. Save your `settings.php` file.
8. Copy the `default.services.yml` file and name it `services.yml` in `sites/default` to provide a mechanism for overriding the default service definitions provided by the Drupal core.
9. Edit `services.yml` to alter the default `config.storage.active` implementation by adding the following YAML definition to the end of the file:

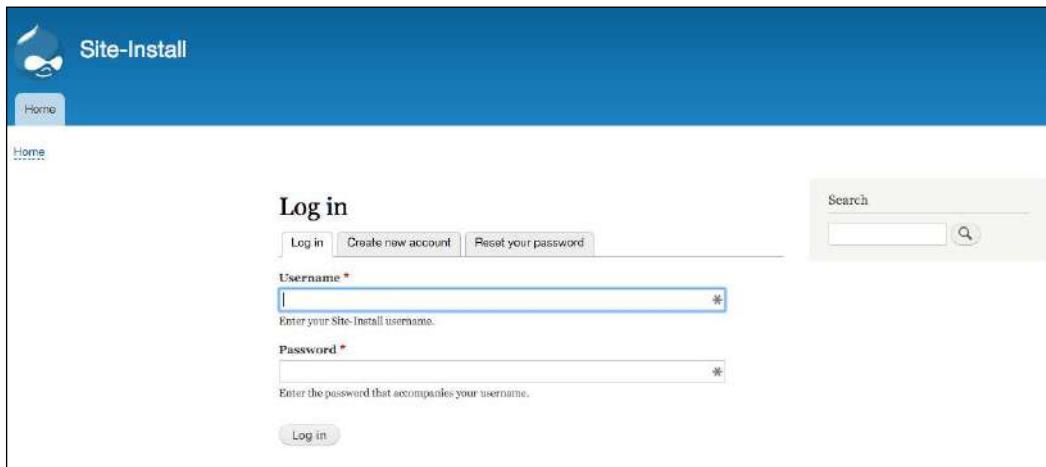
```
services:
 config.storage.active:
 class: Drupal\Core\Config\FileStorage
 factory: Drupal\Core\Config\FileStorageFactory::getActive
```

This YAML will instruct Drupal to use the `\Drupal\Core\Config\FileStorage` class and `\Drupal\Core\Config\FileStorageFactory` for active storage. This is the same definition as the `config.storage.staging` service for configuration synchronization except for the factory method call.

10. Install your Drupal site and the active configuration will be in `sites/default/config/active`. Your configuration will go live on the disk but will be cached in the database.



11. Edit the `block.block.bartik_search.yml` file to modify the block's configuration. Change the region from `sidebar_first` to `sidebar_second`.
12. Rebuild Drupal's cache and the configuration changes will be imported from the disk and displayed on your Drupal site:



## How it works...

Drupal provides a constant that identifies the active and synchronization configuration folders, `CONFIG_ACTIVE_DIRECTORY` and `CONFIG_SYNC_DIRECTORY`, respectively. Drupal uses, and expects, the `$config_directories` global variable to be an array of configuration folder names and their destinations.

The `bootstrap_config_storage` setting allows you to override the default database storage backend for the configuration. The value needs to be a factory-based static method, which returns a class implementing `\Drupal\Core\Config\StorageInterface`. The example provided in `settings.php` uses the `\Drupal\Core\Config\FileStorage` class. The `\Drupal\DrupalKernel` then caches this storage backend and uses it to retrieve configuration values.

The container's service for `config.storage.active` also needs to be overridden to point to the appropriate class. This way, when modules or internal processes invoke `\Drupal::service('config.storage.active')` they receive the proper storage backend.

## There's more...

Although deprecated, filesystem storage for the configuration explores how to provide alternative storage backends. We will explore this in more detail.

## Deprecated for Drupal 9

The concepts of an active configuration directory are deprecated and set to be removed by Drupal 9. This is due to the change in the methodology of how the configuration management works. However, just because it is deprecated in the Drupal core does not mean that it will go away. The implementation can very easily be imported into a contributed or custom project.

The `\Drupal\Core\Config\FileStorage` class, which interacts with the configuration as YAML files, will persist for synchronization purposes. To continue using filesystem-based storage, you will need to just write your own file storage factory that the service calls instead of the deprecated class provided by the core:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\MyCustomFileStorageFactory.
 */

namespace Drupal\mymodule;
```

```
/**
 * Provides a factory for creating config file storage objects.
 */
class MyCustomFileStorageFactory {

 /**
 * Returns a FileStorage object working with the active config
 * directory.
 *
 * @return \Drupal\Core\Config\FileStorage FileStorage
 * no longer creates an active directory.
 */
 static function getActive() {
 return new FileStorage(config_get_config_directory(CONFIG_ACTIVE_DIRECTORY));
 }
}
```

This class can represent a service factory replacement. The `getActive` method instructs the file storage backend to discover YAML files in the defined `CONFIG_ACTIVE_DIRECTORY` location.

## See also

- ▶ The change default active config from file storage to DB storage issue at <https://www.drupal.org/node/2161591>
- ▶ The default active config changed from file storage to DB storage change record at <https://www.drupal.org/node/2241059>



# 10

## The Entity API

In this chapter, we will explore the Entity API to create custom entities and see how they are handled:

- ▶ Creating a configuration entity type
- ▶ Creating a content entity type
- ▶ Creating a bundle for a content entity type
- ▶ Implementing custom access control for an entity
- ▶ Providing a custom storage handler
- ▶ Creating a route provider

### Introduction

In Drupal, entities are a representation of data that have a specific structure. There are specific entity types, which have different bundles and fields attached to those bundles. Bundles are implementations of entities that can have fields attached to themselves. In terms of programming, you can consider an entity that supports bundles an abstract class and each bundle a class that extends that abstract class. Fields are added to bundles. This is part of the reasoning for the term, as an entity type can contain a *bundle* of fields.

An entity is an instance of an entity type defined in Drupal. Drupal 8 provides two entity types: **configuration** and **content**. Configuration entities are not fieldable and represent a configuration within a site. Content entities are fieldable and can have bundles. Bundles are controlled through configuration entities.

In Drupal 8, the **Entity** module lives on, even though most of its functionalities from Drupal 7 are now in core. The goal of the module is to develop improvements for the developer experience around entities by merging more functionalities into core during each minor release cycle (8.1.x, 8.2.x, and so on). Each recipe will provide a *There's more* section that relates to how the Entity module can simplify the recipe.

## Creating a configuration entity type

Drupal 8 harnesses the entity API for configuration to provide configuration validation and extended functionality. Using the underlying entity structure, the configuration has a proper **Create, Read, Update, Delete (CRUD)** process that can be managed. Configuration entities are not fieldable. All the attributes of a configuration entity are defined in its configuration schema definition.

Most common configuration entities interact with Drupal core's `config_object` type, as discussed in *Chapter 4, Extending Drupal*, and *Chapter 9, Configuration Management – Deploying in Drupal 8*, to store and manage a site's configuration. There are other uses of configuration entities, such as menus, view displays, form displays, contact forms, tours, and many more, which are all configuration entities.

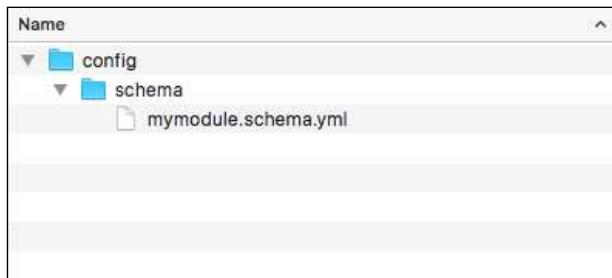
In this recipe, we will create a new configuration entity type called `SiteAnnouncement`. This will provide a simple configuration entity that allows you to create, edit, and delete simple messages that can be displayed on the site for important announcements.

### Getting ready

You will need a custom module to place code into in order to implement a configuration entity type. Create an `src` directory for your classes.

### How to do it...

1. In your module's base directory, create a `config` directory with a `schema` subdirectory. In the subdirectory, make a file named `mymodule.schema.yml` that will hold our configuration entity's schema:



2. In your `mymodule.schema.yml`, add a definition to `mymodule.announcement.*` to provide our label and message storage:

```
Schema for the configuration files of the Site Announcement.

mymodule.announcement.*:
 type: config_entity
 label: 'Site announcement'
 mapping:
 id:
 type: string
 label: 'ID'
 label:
 type: label
 label: 'Label'
 message:
 type: text
 label: 'Text'
```

We define the configuration entity's namespace as an announcement, which we will provide to Drupal in the entity's annotation block. We tell Drupal that this is a `config_entity` and provide a label for the schema.

Using the mapping array, we provide the attributes that make up our entity and the data that will be stored.

3. Create an `Entity` directory in your module's `src` folder. First, we will create an interface for our entity by making a `SiteAnnouncementInterface.php` file. The `SiteAnnouncementInterface` will extend the `\Drupal\Core\Config\Entity\ConfigEntityInterface`:

```
<?php

/**
 * @file Contains \Drupal\mymodule\Entity\SiteAnnouncementInterface.
 */

namespace Drupal\mymodule\Entity;

use Drupal\Core\Config\Entity\ConfigEntityInterface;

interface SiteAnnouncementInterface extends ConfigEntityInterface
{

 /**
 * Gets the message value.
}
```

```
*
 * @return string
 */
 public function getMessage();

}
```

This will be implemented by our entity and will be provided the method requirements. It is best practice to provide an interface for entities. This allows you to provide the required methods if another developer extends your entity or if you are doing advanced testing and need to mock an object. We also provide a method for returning our custom attribute.

4. Create `SiteAnnouncement.php` in your `Entity` directory in `src`. This file will contain the `SiteAnnouncement` class, which extends `\Drupal\Core\Config\Entity\ConfigEntityBase` and implements our entity's interface:

```
<?php

/**
 * @file Contains \Drupal\mymodule\Entity\SiteAnnouncement
 */

namespace Drupal\mymodule\Entity;

use Drupal\Core\Config\Entity\ConfigEntityBase;

class SiteAnnouncement extends ConfigEntityBase implements
SiteAnnouncementInterface {

/**
 * The announcement's message.
 *
 * @var string
 */
protected $message;

/**
 * {@inheritDoc}
 */
public function getMessage() {
 return $this->message;
}

}
```

We added the `message` property defined in our schema as a class property. Our method defined in the entity's interface is used to return that value and interact with our configuration entity.

5. Entities use annotation documentation blocks. We will start our annotation block by providing the entity's ID, label, configuration prefix, and configuration export key names:

```
<?php

/**
 * @file Contains \Drupal\mymodule\Entity\SiteAnnouncement
 */

namespace Drupal\mymodule\Entity;

use Drupal\Core\Config\Entity\ConfigEntityBase;

/**
 * @ConfigEntityType(
 * id = "announcement",
 * label = @Translation("Site Announcement"),
 * config_prefix = "announcement",
 * entity_keys = {
 * "id" = "id",
 * "label" = "label"
 * },
 * config_export = {
 * "id",
 * "label",
 * "message",
 * }
 *)
 */
class SiteAnnouncement extends ConfigEntityBase implements
SiteAnnouncementInterface {

 /**
 * The announcement's message.
 *
 * @var string
 */
protected $message;
```

```
/**
 * {@inheritDoc}
 */
public function getMessage() {
 return $this->message;
}

}
```

The annotation document block tells Drupal that this is an instance of the `ConfigEntityType` plugin. The `id` is the internal machine name identifier for the entity type and the `label` is the human-readable version. The `config_prefix` matches with how we defined our schema with `mymodule.announcement`. The `entity_keys` definition tells Drupal which attributes represent our identifiers and labels.

When specifying `config_export`, we are telling the configuration management system what properties are to be exportable when exporting our entity.

6. Next, we will add handlers to our entity. We will define the class that will display the available entity entries and the forms to work with our entity:

```
/**
 * @ConfigEntityType(
 * id = "announcement",
 * label = @Translation("Site Announcement"),
 * handlers = {
 * "list_builder" = "Drupal\mymodule\
SiteAnnouncementListBuilder",
 * "form" = {
 * "default" = "Drupal\mymodule\SiteAnnouncementForm",
 * "add" = "Drupal\mymodule\SiteAnnouncementForm",
 * "edit" = "Drupal\mymodule\SiteAnnouncementForm",
 * "delete" = "Drupal\Core\Entity\EntityDeleteForm"
 * }
 * },
 * config_prefix = "announcement",
 * entity_keys = {
 * "id" = "id",
 * "label" = "label"
 * },
 * config_export = {
 * "id",
 * "label",
 * "message",
 * }
 *)
 */
```

The `handlers` array specifies classes that provide the interaction functionality with our entity. The `list_builder` class will be created to show you a table of our entities. The `form` array provides classes for forms to be used when creating, editing, or deleting our configuration entity.

7. Lastly, for our annotation, we need to define routes for our `delete`, `edit`, and `collection` (list) pages. Drupal will automatically build the routes based on our annotation:

```
/**
 * @ConfigEntityType(
 * id = "announcement",
 * label = @Translation("Site Announcement"),
 * handlers = {
 * "list_builder" = "Drupal\mymodule\SiteAnnouncementListBuilder",
 * "form" = {
 * "default" = "Drupal\mymodule\SiteAnnouncementForm",
 * "add" = "Drupal\mymodule\SiteAnnouncementForm",
 * "edit" = "Drupal\mymodule\SiteAnnouncementForm",
 * "delete" = "Drupal\Core\Entity\EntityDeleteForm"
 * }
 * },
 * config_prefix = "announcement",
 * entity_keys = {
 * "id" = "id",
 * "label" = "label"
 * },
 * links = {
 * "delete-form" = "/admin/config/system/site-announcements/manage/{announcement}/delete",
 * "edit-form" = "/admin/config/system/site-announcements/manage/{announcement}",
 * "collection" = "/admin/config/system/site-announcements",
 * },
 * config_export = {
 * "id",
 * "label",
 * "message",
 * }
 *)
 */
```

There is a routing service for entities that will automatically provide Drupal a route with the proper controllers based on this annotation. The add form route is not yet supported and needs to be manually added.

8. Create a `mymodule.routing.yml` in your module's root directory to manually provide a route to add a Site-announcement entity:

```
entity.announcement.add_form:
 path: '/admin/config/system/site-announcements/add'
 defaults:
 _entity_form: 'announcement.add'
 _title: 'Add announcement'
 requirements:
 _permission: 'administer content'
```

9. We can use the `_entity_form` property to tell Drupal to look up the class defined in our handlers.
10. Before we implement our `list_builder` handler, we also need to add the route in `mymodule.routing.yml` for our collection link definition, as this is not auto generated by route providers:

```
entity.announcement.collection:
 path: '/admin/config/system/site-announcements'
 defaults:
 _entity_list: 'announcement'
 _title: 'Site Announcements'
 requirements:
 _permission: 'administer content'
```

11. The `_entity_list` key will tell the route to use our `list_builder` handler to build the page. We will reuse the `administer content` permission provided by the Node module.
12. Create the `SiteAnnouncementListBuilder` class defined in our `list_builder` handler by making a `SiteAnnouncementListBuilder.php` and extending the `\Drupal\Core\Config\Entity\ConfigEntityListBuilder`:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\SiteAnnouncementListBuilder.
 */

namespace Drupal\mymodule;

use Drupal\Core\Config\Entity\ConfigEntityListBuilder;
use Drupal\mymodule\Entity\SiteAnnouncementInterface;

class SiteAnnouncementListBuilder extends ConfigEntityListBuilder
{
```

```

/**
 * {@inheritDoc}
 */
public function buildHeader() {
 $header['label'] = t('Label');
 return $header + parent::buildHeader();
}

/**
 * {@inheritDoc}
 */
public function buildRow(SiteAnnouncementInterface $entity) {
 $row['label'] = $entity->label();
 return $row + parent::buildRow($entity);
}
}

```

13. In our list builder handler, we override the `buildHeader` and `builderRow` methods so that we can add our configuration entity's properties to the table.
14. Now we need to create an entity form, as defined in our form handler array, to handle our add and edit functionalities. Create `SiteAnnouncementForm.php` in the `src` directory to provide the `SiteAnnouncementForm` class that extends the `\Drupal\Core\Entity\EntityForm` class:

```

<?php

namespace Drupal\mymodule;

use Drupal\Component\Utility\Unicode;
use Drupal\Core\Entity\EntityForm;
use Drupal\Core\Form\FormStateInterface;
use Drupal\Core\Language\LanguageInterface;

class SiteAnnouncementForm extends EntityForm {
/**
 * {@inheritDoc}
 */
public function form(array $form, FormStateInterface $form_
state) {
 $form = parent::form($form, $form_state);

 /**
 * @var \Drupal\mymodule\Entity\SiteAnnouncementInterface
 */
 $entity = $this->entity;
}
}

```

```
$form['label'] = [
 '#type' => 'textfield',
 '#title' => t('Label'),
 '#required' => TRUE,
 '#default_value' => $entity->label(),
];
$form['message'] = [
 '#type' => 'textarea',
 '#title' => t('Message'),
 '#required' => TRUE,
 '#default_value' => $entity->getMessage(),
];

return $form;
}

/**
 * {@inheritDoc}
 */
public function save(array $form, FormStateInterface $form_
state) {
 $entity = $this->entity;
 $is_new = !$entity->getOriginalId();

 if ($is_new) {
 // Configuration entities need an ID manually set.
 $machine_name = \Drupal::transliteration()
 ->transliterate($entity->label(),
LanguageInterface::LANGCODE_DEFAULT, '_');
 $entity->set('id', Unicode::strtolower($machine_name));

 drupal_set_message(t('The %label announcement has been
created.', array('%label' => $entity->label())));
 }
 else {
 drupal_set_message(t('Updated the %label announcement.',

array('%label' => $entity->label())));
 }

 $entity->save();

 // Redirect to edit form so we can populate colors.
 $form_state->setRedirectUrl($this->entity-
>toUrl('collection'));
}
}
```

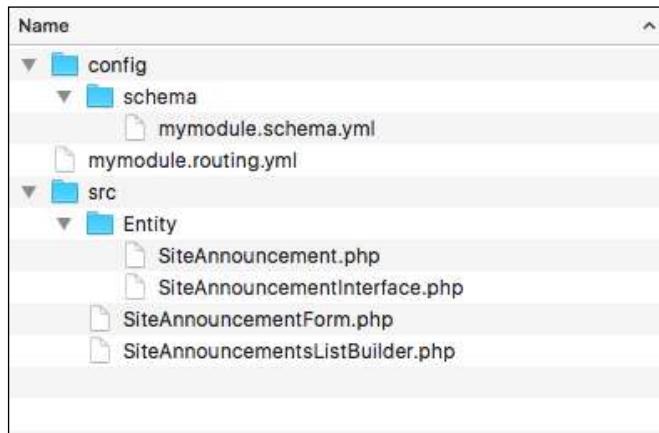
15. We override the `form` method to add Form API elements to our label and message properties. We also override the `save` method to provide user messages about the changes that are made. We utilize the entity's `toUrl` method to provide a redirect back to the collection (list) page. We use the transliteration service to generate a machine name based on the label for our entity's identifier.

16. Next, we will provide a `mymodule.links.action.yml` file in your module's directory. This will allow us to define action links on a route. We will be adding an `Add announcement` link to our entity's add form on its collection route:

```
announcement.add:
 route_name: entity.announcement.add_form
 title: 'Add announcement'
 appears_on:
 - entity.announcement.collection
```

17. This will instruct Drupal to render the `entity.announcement.add_form` link on the specified routes in the `appears_on` value.

18. Your module structure should look like the following screenshot:



19. Install your module and review the **Configuration** page. You can now manage the Site Announcement entries from the **Site Announcement** link.

## How it works

When creating a configuration schema definition, one of the first properties used for the configuration namespace is `type`. This value can be `config_object` or `config_entity`. When the type is `config_entity`, the definition will be used to create a database table rather than structure the serialized data for the `config` table.

Entities are powered by the plugin system in Drupal, which means there is a plugin manager. The default `\Drupal\Core\Entity\EntityTypeManager` provides discovery and handling of entities. The `ConfigEntityType` class for the entity type's plugin class will force the setting of the `uuid` and `langcode` in the `entity_keys` definition. The storage handler for configuration entities defaults to `\Drupal\Core\Config\Entity\ConfigEntityStorage`. The `ConfigEntityStorage` class interacts with the configuration management system to load, save, and delete custom configuration entities.

## There's more...

Drupal 8 introduces a typed data system that configuration entities, and fields use.

### Available data types for schema definitions

Drupal core provides its own configuration information. There is a `core.data_types.schema.yml` file located at `core/config/schema`. These are the base types of data that core provides and can be used when making configuration schema. The file contains YAML definitions of data types and the class which represents them:

```
boolean:
 label: 'Boolean'
 class: '\Drupal\Core\TypedData\Plugin\DataType\BooleanData'
email:
 label: 'Email'
 class: '\Drupal\Core\TypedData\Plugin\DataType>Email'
string:
 label: 'String'
 class: '\Drupal\Core\TypedData\Plugin\DataType\StringData'
```

When a configuration schema definition specifies an attribute that has an e-mail for its type, that value is then handled by the `\Drupal\Core\TypedData\Plugin\DataType>Email` class. Data types are a form of plugins and each plugin's annotation specifies constraints for validation. This is built around the Symfony Validator component.

## See also

- ▶ *Chapter 6, Creating Forms with the Form API*
- ▶ *Chapter 4, Extending Drupal*
- ▶ *Chapter 9, Configuration Management – Deploying in Drupal 8*
- ▶ Refer to configuration schema/metadata at <https://www.drupal.org/node/1905070>

## Creating a content entity type

Content entities provide base field definitions and also configurable fields through the Field module. There is also support for revisions and translations with content entities. Display modes, both form and view, are available for content entities to control how the fields are edited and displayed. When an entity does not specify bundles, there is automatically one bundle instance with the same name as the entity.

In this recipe, we will create a custom content entity that does not specify a bundle. We will create a Message entity that can serve as a content entity for generic messages.

### Getting ready

You will need a custom module to place code into in order to implement a configuration entity type. Create an `src` directory for your classes.

### How to do it...

1. Create an `Entity` directory in your module's `src` folder. First, we will create an interface for our entity by making a `MessageInterface.php` file:



2. The `MessageInterface` will extend `\Drupal\Core\Entity\ContentEntityInterface`:

```
<?php

/**
 * @file Contains \Drupal\mymodule\Entity\MessageInterface.
 */

namespace Drupal\mymodule\Entity;

use Drupal\Core\Entity\ContentEntityInterface;
```

```
interface MessageInterface extends ContentEntityInterface {

 /**
 * Gets the message value.
 *
 * @return string
 */
 public function getMessage();

}
```

This will be implemented by our entity and provide the method requirements. It is best practice to provide an interface for entities. This allows you to provide required methods if another developer extends your entity or if you are doing advanced testing and need to mock an object. We also provide a method to return our main base field definition (to be defined).

3. Create `Message.php` in your `Entity` directory in `src`. This file will contain the `Message` class, which extends `\Drupal\Core\Entity\ContentEntityBase` and implements our entity's interface:

```
<?php

/**
 * @file Contains \Drupal\mymodule\Entity\Message
 */

namespace Drupal\mymodule\Entity;

use Drupal\Core\Entity\ContentEntityBase;

class Message extends ContentEntityBase implements
MessageInterface {

}
```

4. We need to create an annotation document block to provide information about our entity, such as its ID, label, entity keys, and so on:

```
<?php

/**
 * @file Contains \Drupal\mymodule\Entity\Message
 */

namespace Drupal\mymodule\Entity;
```

---

```

use Drupal\Core\Entity\ContentEntityBase;

/**
 * Defines the message entity class.
 *
 * @ContentEntityType(
 * id = "message",
 * label = @Translation("Message"),
 * base_table = "message",
 * fieldable = TRUE,
 * entity_keys = {
 * "id" = "message_id",
 * "label" = "title",
 * "langcode" = "langcode",
 * "uuid" = "uuid"
 * },
 *)
 */
class Message extends ContentEntityBase implements
MessageInterface {

}

```

The `id` is the internal machine name identifier for the entity type and the `label` is the human-readable version. The `entity keys` definition tells Drupal which attributes represent our identifier and label.

`base_table` defines the database table in which the entity will be stored and `fieldable` allows custom fields to be configured through the Field UI module.

5. Next, we will add handlers to our entity. We will use the default handlers provided by Drupal:

```

/**
 * Defines the profile entity class.
 *
 * @ContentEntityType(
 * id = "message",
 * label = @Translation("Message"),
 * handlers = {
 * "list_builder" = "Drupal\mymodule\MessageListBuilder",
 * "form" = {
 * "default" = "Drupal\Core\Entity\ContentEntityForm",
 * "add" = "Drupal\Core\Entity\ContentEntityForm",
 * "edit" = "Drupal\Core\Entity\ContentEntityForm",
 * "delete" = "Drupal\Core\Entity\ContentEntityDeleteForm",
 * }
 * }
 */

```

```
* },
* },
* base_table = "message",
* fieldable = TRUE,
* entity_keys = {
* "id" = "message_id",
* "label" = "title",
* "langcode" = "langcode",
* "uuid" = "uuid"
* },
*)
*/
```

The handlers array specifies classes that provide the interaction functionality with our entity. The list builder class will be created to show you a table of our entities. The form array provides classes for forms to be used when creating, editing, or deleting our content entity.

6. An additional handler can be added, the `route_provider`, to dynamically generate our canonical (view), edit, and delete routes:

```
/**
 * Defines the profile entity class.
 *
 * @ContentEntityType(
 * id = "message",
 * label = @Translation("Message"),
 * handlers = {
 * "list_builder" = "Drupal\mymodule\MessageListBuilder",
 * "form" = {
 * "default" = "Drupal\Core\Entity\ContentEntityForm",
 * "add" = "Drupal\Core\Entity\ContentEntityForm",
 * "edit" = "Drupal\Core\Entity\ContentEntityForm",
 * "delete" = "Drupal\Core\Entity\ContentEntityDeleteForm",
 * },
 * "route_provider" = {
 * "html" = "Drupal\Core\Entity\Routing\DefaultHtmlRouteProvider",
 * },
 * },
 * base_table = "message",
 * fieldable = TRUE,
 * entity_keys = {
 * "id" = "message_id",
 * "label" = "title",
 * "langcode" = "langcode",
 * }
 */
```

---

```

* "uuid" = "uuid"
* },
* links = {
* "canonical" = "/messages/{message}",
* "edit-form" = "/messages/{message}/edit",
* "delete-form" = "/messages/{message}/delete",
* "collection" = "/admin/content/messages"
* },
*)
*/

```

There is a routing service for entities that will automatically provide Drupal a route with the proper controllers based on this annotation. The add form route is not yet supported and needs to be manually added.

7. We need to implement `baseFieldDefinitions` to satisfy the `FieldableEntityInterface` interface, which will provide our field definitions to the entity's base table:

```

/**
 * {@inheritDoc}
 */
public static function baseFieldDefinitions(EntityTypeInterface
$entity_type) {
 $fields['message_id'] = BaseFieldDefinition::create('integer')
 ->setLabel(t('Message ID'))
 ->setDescription(t('The message ID.'))
 ->setReadOnly(TRUE)
 ->setSetting('unsigned', TRUE);
 $fields['langcode'] = BaseFieldDefinition::create('language')
 ->setLabel(t('Language code'))
 ->setDescription(t('The message language code.'))
 ->setRevisionable(TRUE);
 $fields['uuid'] = BaseFieldDefinition::create('uuid')
 ->setLabel(t('UUID'))
 ->setDescription(t('The message UUID.'))
 ->setReadOnly(TRUE);

 $fields['title'] = BaseFieldDefinition::create('string')
 ->setLabel(t('Title'))
 ->setRequired(TRUE)
 ->setTranslatable(TRUE)
 ->setRevisionable(TRUE)
 ->setSetting('max_length', 255)
 ->setDisplayOptions('view', array(
 'label' => 'hidden',

```

```
 'type' => 'string',
 'weight' => -5,
))
->setDisplayOptions('form', array(
 'type' => 'string_textfield',
 'weight' => -5,
))
->setDisplayConfigurable('form', TRUE);

$fields['content'] = BaseFieldDefinition::create('text_long')
->setLabel(t('Content'))
->setDescription(t('Content of the message'))
->setTranslatable(TRUE)
->setDisplayOptions('view', array(
 'label' => 'hidden',
 'type' => 'text_default',
 'weight' => 0,
))
->setDisplayConfigurable('view', TRUE)
->setDisplayOptions('form', array(
 'type' => 'text_textfield',
 'weight' => 0,
))
->setDisplayConfigurable('form', TRUE);

return $fields;
}
```

8. The `FieldableEntityInterface` is implemented by the `ContentEntityBase` class through the `ContentEntityInterface`. The method needs to return an array of `BaseFieldDefinitions` for typed data definitions. This includes the keys provided in the `entity_keys` value in our entity's annotation along with any specific fields for our implementation.
9. The `content` base field definition will hold the actual text for the message.
10. Next, we will implement the `getMessage` method to satisfy our interface and provide a means to retrieve our message's text value:

```
/**
 * {@inheritDoc}
 */
public function getMessage() {
 return $this->get('content')->value;
}
```

11. This method provides a wrapper around the defined base field's value and returns it.

12. Create a `mymodule.routing.yml` to manually provide a route to add a message entity:

```
entity.message.add_form:
 path: '/messages/add'
 defaults:
 _entity_form: 'message.add'
 _title: 'Add message'
 requirements:
 _entity_create_access: 'message'
```

13. We can use the `_entity_form` property to tell Drupal to look up the class defined in our handlers.

14. Before we implement our `list_builder` handler, we also need to add the route to `routing.yml` for our collection link definition, as this is not auto generated by route providers:

```
entity.message.collection:
 path: '/admin/content/messages'
 defaults:
 _entity_list: 'message'
 _title: 'Messages'
 requirements:
 _permission: 'administer messages'
```

15. The `_entity_list` key will tell the route to use our `list_builder` handler to build the page.

16. Create the `MessageListBuilder` class defined in our `list_builder` handler by making a `MessageListBuilder.php` file and extend `\Drupal\Core\Entity\EntityListBuilder`. We need to override the default implementation to display our base field definitions:

```
<?php

/**
 * @file Contains \Drupal\mymodule\MessageListBuilder
 */

namespace Drupal\mymodule;

use Drupal\Core\Entity\EntityInterface;
use Drupal\Core\Entity\EntityListBuilder;

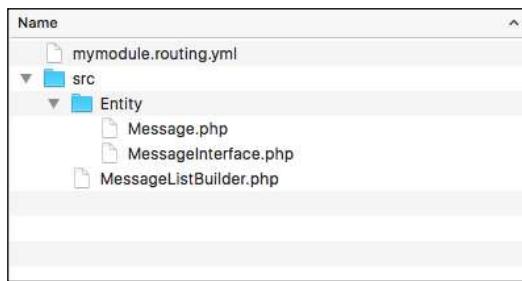
class MessageListBuilder extends EntityListBuilder {
 public function buildHeader() {
```

```
 $header['title'] = t('Title');
 return $header + parent::buildHeader();
}

public function buildRow(EntityInterface $entity) {
 $row['title'] = $entity->label();
 return $row + parent::buildRow($entity);
}

}
```

17. In our list builder handler, we override the `buildHeader` and `builderRow` methods so that we can add our configuration entity's properties to the table.
18. Your module's structure should resemble the following screenshot:



19. Install your module. Visit `/messages/add` to create your first custom content entity entry and then view it on `/admin/content/messages`:

Back to site   Manage   Shortcuts   admin

## Messages ★

Home » Administration » Content

| TITLE                    | OPERATIONS |
|--------------------------|------------|
| There is no Message yet. |            |

## How it works...

Content entities are a version of the `EntityType` plugin. When you define a content entity type, the annotation block begins with `@ContentEntityType`. This declaration, and the properties in it, represents the definition to initiate an instance of the `\Drupal\Core\Entity\ContentEntityType` class just like all other plugin annotations. The `ContentEntityType` plugin class implements a constructor to provide default storage and `view_builder` handlers, forcing us to implement the `list_builder` and `form` handler arrays.

The plugin manager for entity types lives under the `entity_type.manager` service name and is provided through `\Drupal\Core\Entity\EntityTypeManager` by default. However, while the annotation defines the plugin information, our `Message` class that extends `ContentEntityBase` provides a means for manipulating the data it represents.

## There's more...

We will discuss how to add additional functionality to your entity, and use the Entity module to simplify the developer expedience.

### Using the `AdminHtmlRouteProvider` provider

Our `Message` entity type implements the `DefaultHtmlRouteProvider` class. There is also the `\Drupal\Core\Entity\Routing\AdminHtmlRouteProvider` class. This overrides the `getEditFormRoute` and `getDeleteFormRoute` and marks them with `_admin_route`. This will cause those forms to be rendered in the administration theme.

### Simplifying base field definitions using the Entity module

Content entities need to define field definitions for each field listed in the `entity_keys` array. This often results in a lot of boilerplate code to make identifier, language code, UUID, and bundle entity reference fields. The Entity module provides the `\Drupal\entity\EntityKeysFieldsTrait` trait. A content entity type class can use this trait to provide field definitions for the possible `entity_key` values.

The `Message` content entity class can be reduced to the following code using this trait:

```
class Message extends ContentEntityBase implements MessageInterface {
 use EntityKeysFieldsTrait;

 /**
 * {@inheritDoc}
 */
 public function getMessage() {
 return $this->get('content')->value;
 }
}
```

```
/**
 * {@inheritDoc}
 */
public static function baseFieldDefinitions(EntityTypeInterface
$entity_type) {
 $fields = self::entityKeysBaseFieldDefinitions($entity_type);

 $fields['content'] = BaseFieldDefinition::create('text_long')
 ->setLabel(t('Content'))
 ->setDescription(t('Content of the message'))
 ->setTranslatable(TRUE)
 ->setDisplayOptions('view', array(
 'label' => 'hidden',
 'type' => 'text_default',
 'weight' => 0,
))
 ->setDisplayConfigurable('view', TRUE)
 ->setDisplayOptions('form', array(
 'type' => 'text_textfield',
 'weight' => 0,
))
 ->setDisplayConfigurable('form', TRUE);
 return $fields;
}
```

The `entityKeysBaseFieldDefinitions` method provided by the trait will check whether the possible `entity_key` values have been provided and adds a default base definition for them. Now, we only need to worry about implementing base fields that are unique to our entity types.

## The entity type's admin permission

The entity access handler provided by core will check whether entities implement an `admin_permission` option. If it is provided, it will be used as the basis for most access checks unless a custom access handler is implemented. This can be done by providing the following code snippet into an entity type's annotation:

```
* admin_permission = "administer messages",
```

The `\Drupal\Core\Entity\EntityAccessControlHandler` class will check whether users have this permission when validating create access or any other access operation.

## Making the collection route a local task tab

In this recipe, we specified the message collection route as `/admin/content/messages`. Without implementing this route as a local task under the `/admin/content` route, it will not show up as a tab. This can be done by creating a `links.task.yml` file for the module.

In `mymodule.links.task.yml`, add the following YAML content:

```
entity.message.collection_tab:
 route_name: entity.message.collection
 base_route: system.admin_content
 title: 'Messages'
```

This instructs Drupal to use the `entity.messages.collection` route, defined in our `routing.yml` file, to be based under the `system.admin_content` route:

### See also

- ▶ *Chapter 6, Extending Drupal*

## Creating a bundle for a content entity type

Bundles allow you to have different variations of a content entity. All bundles share the same base field definitions but not configured fields. This allows each bundle to have its own custom fields. Display modes are also dependent on a specific bundle. This allows each bundle to have its own configuration for the form mode and view mode.

Using the custom entity from the previous recipe, we will add a configuration entity to act as the bundle. This will allow you to have different message types for multiple custom field configurations.

## Getting ready

You will need a custom module to place the code into in order to implement a configuration entity type. Create an `src` directory for your classes. We need a custom content entity type to be implemented, such as the one in the *Creating a content entity type* recipe.

## How to do it...

1. Since content entity bundles are configuration entities, we need to define our configuration entity schema. Create a `config/schema` directory and a `mymodule.schema.yml` file that will contain the configuration entity's schema:

```
mymodule.message_type.*:
 type: config_entity
 label: 'Message type settings'
 mapping:
 id:
 type: string
 label: 'Machine-readable name'
 uuid:
 type: string
 label: 'UUID'
 label:
 type: label
 label: 'Label'
 langcode:
 type: string
 label: 'Default language'
```

2. We define the configuration entity's config prefix as `message_type`, which we will provide to Drupal in the entity's annotation block. We tell Drupal that this is a `config_entity` and provide a label for the schema.
3. With the mapping array, we provide the attributes that make up our entity and the data that will be stored.
4. In your module's `src/Entity` directory, create an interface for our bundle by making a `MessageTypeInterface.php` file. The `MessageTypeInterface` will extend the `\Drupal\Core\Config\Entity\ConfigEntityInterface`:

```
<?php

/**
 * @file Contains \Drupal\mymodule\Entity\MessageTypeInterface.
 */
```

```
namespace Drupal\mymodule\Entity;

use Drupal\Core\Config\Entity\ConfigEntityInterface;

interface MessageTypeInterface extends ConfigEntityInterface {
 // Empty for future enhancements.
}
```

5. This will be implemented by our entity and provide the method requirements. It is best practice to provide an interface for entities. This allows you to provide required methods if another developer extends your entity or if you are doing advanced testing and need to mock an object.
6. We will be implementing a very basic bundle. It is still wise to provide an interface in the event of future enhancements and mocking ability in tests.
7. Create a `MessageType.php` file in `src/Entity`. This will hold the `MessageType` class, which will extend `\Drupal\Core\Config\Entity\ConfigEntityBundleBase` and implement our bundle's interface:

```
<?php

/**
 * @file Contains \Drupal\mymodule\Entity\MessageType.
 */
```

```
namespace Drupal\mymodule\Entity;

use Drupal\Core\Config\Entity\ConfigEntityBundleBase;

class MessageType extends ConfigEntityBundleBase implements
MessageTypeInterface {
```

```
}
```

8. In most use cases, the bundle entity class can be an empty class that does not provide any properties or methods. If a bundle provides additional attributes in its schema definition, they would also be provided here, like any other configuration entity.
9. Entities need to be annotated. Create a base annotation for the `ID`, `label`, `entity` keys, and `configuration export` keys:

```
<?php

/**
 * @file Contains \Drupal\mymodule\Entity\MessageType.
 */
```

```
namespace Drupal\mymodule\Entity;

use Drupal\Core\Config\Entity\ConfigEntityBundleBase;

/**
 * Defines the profile type entity class.
 *
 * @ConfigEntityType(
 * id = "message_type",
 * label = @Translation("Message type"),
 * config_prefix = "message_type",
 * bundle_of = "message",
 * entity_keys = {
 * "id" = "id",
 * "label" = "label",
 * "uuid" = "uuid",
 * "langcode" = "langcode"
 * },
 * config_export = {
 * "id",
 * "label",
 * }
 *)
 */
class MessageType extends ConfigEntityBundleBase implements
MessageTypeInterface {

}
```

10. The annotation document block tells Drupal that this is an instance of the `ConfigEntityType` plugin. The `id` is the internal machine name identifier for the entity type and the `label` is the human-readable version. The `config_prefix` matches with how we defined our schema with `mymodule.message_type`. The `entity_keys` definition tells Drupal which attributes represent our identifiers and labels.
11. When specifying `config_export`, we are telling the configuration management system what properties are to be exported when exporting our entity.
12. We will then add handlers, which will interact with our entity:

```
/**
 * Defines the profile type entity class.
 *
 * @ConfigEntityType(
 * id = "message_type",
 * label = @Translation("Message type"),
 * handlers = {

```

```

* "list_builder" = "Drupal\mymodule\MessageTypeListBuilder",
* "form" = {
* "default" = "Drupal\Core\Entity\EntityForm",
* "add" = "Drupal\Core\Entity\EntityForm",
* "edit" = "Drupal\Core\Entity\EntityForm",
* "delete" = "Drupal\Core\Entity\EntityDeleteForm"
* },
* },
* config_prefix = "message_type",
* bundle_of = "message",
* entity_keys = {
* "id" = "id",
* "label" = "label",
* "uuid" = "uuid",
* "langcode" = "langcode"
* },
* config_export = {
* "id",
* "label",
* },
*)
*/

```

13. The handlers array specifies classes that provide the interaction functionality with our entity. The list builder class will be created to show you a table of our entities. The form array provides classes for forms to be used when creating, editing, or deleting our configuration entity.
14. An additional handler can be added, the route\_provider, to dynamically generate our canonical (view), edit, and delete routes:

```

/**
 * Defines the profile type entity class.
*
* @ConfigEntityType(
* id = "message_type",
* label = @Translation("Message type"),
* handlers = {
* "list_builder" = "Drupal\profile\MessageTypeListBuilder",
* "form" = {
* "default" = "Drupal\Core\Entity\EntityForm",
* "add" = "Drupal\Core\Entity\EntityForm",
* "edit" = "Drupal\Core\Entity\EntityForm",
* "delete" = "Drupal\Core\Entity\EntityDeleteForm"
* },
* }
*/

```

```
* "route_provider" = {
* "html" = "Drupal\Core\Entity\Routing\
DefaultHtmlRouteProvider",
* },
* },
* config_prefix = "message_type",
* bundle_of = "message",
* entity_keys = {
* "id" = "id",
* "label" = "label"
* },
* config_export = {
* "id",
* "label",
* },
* links = {
* "delete-form" = "/admin/structure/message-types/{message_
type}/delete",
* "edit-form" = "/admin/structure/message-types/{message_
type}",
* "admin-form" = "/admin/structure/message-types/{message_
type}",
* "collection" = "/admin/structure/message-types"
* }
*)
*/

```

15. There is a routing service for entities that will automatically provide Drupal a route with the proper controllers based on this annotation. The add form route is not yet supported and needs to be manually added.
16. We need to modify our content entity to use the bundle configuration entity that we defined:

```
/**
 * Defines the profile entity class.
 *
 * @ContentEntityType(
 * id = "message",
 * label = @Translation("Message"),
 * handlers = {...},
 * base_table = "message",
 * fieldable = TRUE,
 * bundle_entity_type = "message_type",
 * field_ui_base_route = "entity.message_type.edit_form",
 * entity_keys = {

```

```
* "id" = "message_id",
* "label" = "title",
* "langcode" = "langcode",
* "bundle" = "type",
* "uuid" = "uuid"
* },
* links = {...},
*)
*/

```

17. The `bundle_entity_type` key specifies the entity type used as the bundle. The plugin validates this as an actual entity type and marks it for configuration dependencies. With the `field_ui_base_route` key pointed to the bundle's main edit form, it will generate the Manage Fields, Manage Form Display, and Manage Display tabs on the bundles. Finally, the `bundle entity` key instructs Drupal which field definition to use in order to identify the entity's bundle, which is created in the next step.
18. A new field definition needs to be added to provide the `type` field that we defined to represent the `bundle entity` key:

```
$fields['type'] = BaseFieldDefinition::create('entity_
reference')
->setLabel(t('Message type'))
->setDescription(t('The message type.'))
->setSetting('target_type', 'message_type')
->setSetting('max_length', EntityTypeInterface::BUNDLE_MAX_
LENGTH);
```

19. The field that identifies the bundle will be typed as an entity reference. This allows the value to act as a foreign key to the bundle's base table.
20. In your `mymodule.routing.yml`, provide a route for adding a Message Type entity:

```
entity.message_type.add_form:
path: '/admin/structure/message-types/add'
defaults:
 _entity_form: 'message_type.add'
 _title: 'Add message type'
requirements:
 _entity_create_access: 'message_type'
```

21. We can use the `_entity_form` property to tell Drupal to look up the class defined in our handlers.

22. Before we implement our `list_builder` handler, we also need to add the route to `routing.yml` for our collection link definition, as this is not auto generated by route providers:

```
entity.message_type.collection:
 path: '/admin/structure/message-types'
 defaults:
 _entity_list: 'message_type'
 _title: 'Message types'
 requirements:
 _permission: 'administer message types'
```

23. The `_entity_list` key will tell the route to use our `list_builder` handler to build the page.
24. Create the `MessageTypeListBuilder` class defined in our `list_builder` handler in a `MessageTypeListBuilder.php` file and extend `\Drupal\Core\Config\Entity\ConfigEntityListBuilder`. We need to override the default implementation to display our configuration entity properties:

```
<?php

/**
 * @file Contains \Drupal\mymodule\MessageListBuilder
 */

namespace Drupal\mymodule;

use Drupal\Core\Entity\EntityInterface;
use Drupal\Core\Config\Entity\ConfigEntityListBuilder;

class MessageTypeListBuilder extends EntityListBuilder {
 public function buildHeader() {
 $header['label'] = t('Label');
 return $header + parent::buildHeader();
 }

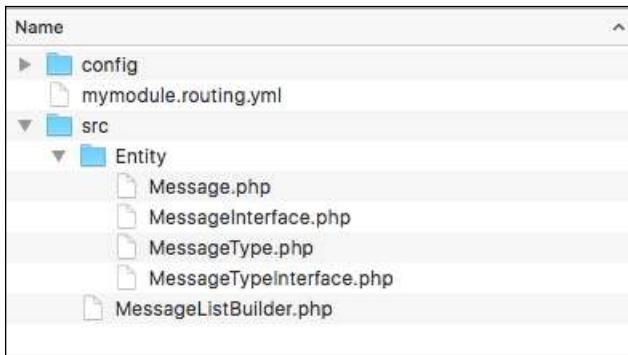
 public function buildRow(EntityInterface $entity) {
 $row['label'] = $entity->label();
 return $row + parent::buildRow($entity);
 }
}
```

25. In our list builder handler, we override the `buildHeader` and `builderRow` methods so that we can add our configuration entity's properties to the table:



The screenshot shows the Drupal administration interface. At the top, there are links for 'Back to site', 'Manage', 'Shortcuts', and a user icon for 'admin'. Below this is a title bar with 'Messages' and a star icon. Underneath is a breadcrumb trail: 'Home > Administration > Content'. A table follows, with columns labeled 'TITLE' and 'OPERATIONS'. A message in the table states: 'There is no Message yet.'

26. Your module's structure should resemble the following screenshot:



## How it works...

Bundles are most utilized in the configured field levels via the `Field` and `Field UI` modules. When you create a new field, it has a base storage item for its global settings. Once a field is added to a bundle, there is a new field configuration that is created and assigned to the bundle. Fields can then have their own settings for a specific bundle along with form and view display configurations.

Content entity bundles work just like any other configuration entity implementation, but they extend the usability of the `Field API` for your content entity types.

## There's more...

We will discuss how to add additional functionality to your entity bundle, and use the Entity module to simplify the developer expience.

### Provide action links for adding new bundles

There are special links called **action links** in Drupal. These appear at the top of the page and are generally used for links that allow the creation of an item by creating a `links.action.yml` file.

In your `mymodule.links.action.yml`, each action link defines the route it will link to, titles, and the routes it appears on:

```
message_type_add:
 route_name: entity.message_type.add_form
 title: 'Add message type'
 appears_on:
 - entity.message_type.collection
```

The `appears_on` key accepts multiple values that will allow this route link to appear on multiple pages:

## See also

- ▶ *Chapter 4, Extending Drupal*
- ▶ *Chapter 9, Configuration Management*
- ▶ *Recipe Creating a Configuration Entity Type in Chapter 10, The Entity API*

## Implementing custom access control for an entity

All entities have a set of handlers that control specific pieces of functionalities. One handler in particular handles access control. When the access handler is not specified, the base \Drupal\Core\Entity\EntityType module will implement \Drupal\Core\Entity\EntityAccessControlHandler as the access handler. By default, this will check whether any modules have implemented hook\_entity\_create\_access or hook\_entity\_type\_create\_access and use their opinions. Otherwise, it defaults to the admin permission for the entity type, if implemented.

In this recipe, we will provide an admin permission for our entity along with create, update, view, and delete permissions for each of the entity's bundles. We will base this on an entity called **Message**.

### Getting ready

You will need a custom module to place the code into in order to implement a configuration entity type. Create an `src` directory for your PSR-4 style classes. We need to implement a custom content entity type, such as the one in the *Creating a content entity type* recipe.

### How to do it...

- First, we need to define an administration permission for the entity. This is done by adding the `admin_permission` key to the entity's annotation document block:

```
/**
 * Defines the profile entity class.
 *
 * @ContentEntityType(
 * id = "message",
 * label = @Translation("Message"),
 * handlers = { ... },
 * base_table = "message",
 * fieldable = TRUE,
 * admin_permission = "administer messages",
 * entity_keys = {
 * "id" = "message_id",
 * "label" = "title",
 * "langcode" = "langcode",
 * "uuid" = "uuid"
 * },
 * links = { ... },
 *)
 */
```

2. The entity access handler provided by core will check whether entities implement this option. If it is provided, it will be used as the basis for most access checks unless a custom access handler is implemented.
3. Create a `mymodule.permissions.yml` to provide the administrative permission to Drupal. We will be defining a permission callback as well to support dynamic permissions based on current bundles:

```
administer messages:
 title: 'Administer messages'
 restrict access: true
permissions_callbacks:
 - \Drupal\mymodule\MessagePermissions::messageTypePermissions
```

4. Along with defining specific permissions, we need to specify the `class` and `static` methods to return dynamic permissions. Refer to the *Defining permissions* recipe in *Chapter 4, Extending Drupal*, for more information.
5. Create the `MessagePermissions` class in the `src` directory. This will contain the `profileTypePermissions` method that returns an array of permissions. Our class will add `create`, `view`, `update`, and `delete` permissions to our entities:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\MessagePermissions.
 */

namespace Drupal\mymodule;

use Drupal\Core\StringTranslation\StringTranslationTrait;
use Drupal\mymodule\Entity\MessageType;

/**
 * Defines a class containing permission callbacks.
 */
class MessagePermissions {
 use StringTranslationTrait;

 /**
 * Returns an array of message type permissions.
 *
 * @return array
 * Returns an array of permissions.
 */
```

```

public function messageTypePermissions() {
 $perms = [];
 // Generate message permissions for all message types.
 foreach (MessageType::loadMultiple() as $type) {
 $perms += $this->buildPermissions($type);
 }
 return $perms;
}

/**
 * Builds a standard list of permissions for a given profile
type.
 *
 * @param \Drupal\mymodule\Entity\MessageType $message_type
 * The machine name of the message type.
 *
 * @return array
 * An array of permission names and descriptions.
 */
protected function buildPermissions(MessageType $message_type) {
 $type_id = $message_type->id();
 $type_params = ['%type' => $message_type->label()];

 return [
 "add $type_id message" => [
 'title' => $this->t('%type: Add message', $type_params),
],
 "view $type_id message" => [
 'title' => $this->t('%type: View message', $type_params),
],
 "edit $type_id message" => [
 'title' => $this->t('%type: Edit message', $type_params),
],
 "delete $type_id message" => [
 'title' => $this->t('%type: Delete message', $type_params),
],
];
}
}

```

6. In our permission callback, `messageTypePermissions`, we invoke the `MessageType::loadMultiple` method with no parameters. This will return all the available entities for `message_type`. We then pass this entity to another method, which defines create, read, update, and delete permissions.

7. To utilize the dynamic permissions, we will extend the default `\Drupal\Core\Entity\EntityAccessControlHandler`. Create a `MessageAccessControlHandler` class for your module:

```
<?php

/**
 * @file Contains \Drupal\mymodule\MessageAccessControlHandler.
 */

namespace Drupal\mymodule;

use Drupal\Core\Entity\EntityAccessControlHandler;

/**
 * Defines the access control handler for the message entity type.
 */
class MessageAccessControlHandler extends EntityAccessControlHandler {

}
```

8. We will override the `checkAccess` method. The default implementation notes in the documentation state that this method is supposed to be overridden by entities using custom access checking:

```
<?php

/**
 * @file Contains \Drupal\mymodule\MessageAccessControlHandler.
 */

namespace Drupal\mymodule;

use Drupal\Core\Access\AccessResult;
use Drupal\Core\Entity\EntityAccessControlHandler;
use Drupal\Core\Entity\EntityInterface;
use Drupal\Core\Session\AccountInterface;

/**
 * Defines the access control handler for the message entity type.
 */
class MessageAccessControlHandler extends EntityAccessControlHandler {
```

```

/**
 * {@inheritDoc}
 */
protected function checkAccess(EntityInterface $entity,
$operation, AccountInterface $account) {
 // Re-use admin permission check.
 $result = parent::checkAccess($entity, $operation, $account);

 if ($result->isNeutral()) {
 // Check if user has permission: ex, "add message message".
 $result = AccessResult::allowedIfHasPermission($account,
"$operation {$entity->bundle()} message");
 }

 return $result;
}

}

```

9. In our overridden method, we check the parent class result. This handles our admin permission check and the basic *you cannot delete a new, non-saved entity* logic. If the parent class comes back neutral, we can check it based on our dynamic permissions and return that.
10. We need to follow the same pattern for the parent `checkCreateAccess` method, which is called on create. It specifies that it should be overridden if you are implementing custom access checks:

```

/**
 * {@inheritDoc}
 */
protected function checkCreateAccess(AccountInterface $account,
array $context, $entity_bundle = NULL) {
 // Re-use admin permission check.
 $result = parent::checkCreateAccess($account, $context,
$entity_bundle);

 if ($result->isNeutral()) {
 $result = AccessResult::allowedIfHasPermission($account,
"add $entity_bundle message");
 }

 return $result;
}

```

11. For this method, we follow the same pattern and reuse the parent's check for the admin permission.

12. After the access handler is created, we need to add it to the list of our entities' handlers:

```
* handlers = {
* "list_builder" = "Drupal\mymodule\ProfileTypeListBuilder",
* "access" = "Drupal\mymodule\MessageAccessControlHandler",
* "form" = {
* "default" = "Drupal\Core\Entity\EntityForm",
* "add" = "Drupal\Core\Entity\EntityForm",
* "edit" = "Drupal\Core\Entity\EntityForm",
* "delete" = "Drupal\Core\Entity\EntityDeleteForm"
* },
* "route_provider" = {
* "html" = "Drupal\Core\Entity\Routing\DefaultHtmlRouteProvider",
* },
* },
```

13. Rebuild Drupal's caches.

14. Verify that the permissions are available on the permission's overview page:

| PERMISSION                                                                                                            | ANONYMOUS<br>USER        | AUTHENTICATED<br>USER    | ADMINISTRATOR                       |
|-----------------------------------------------------------------------------------------------------------------------|--------------------------|--------------------------|-------------------------------------|
| <b>My Module</b>                                                                                                      |                          |                          |                                     |
| Administer messages<br><small>Warning: Give to trusted roles only; this permission has security implications.</small> | <input type="checkbox"/> | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Message: Add message                                                                                                  | <input type="checkbox"/> | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Message: Delete message                                                                                               | <input type="checkbox"/> | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Message: Edit message                                                                                                 | <input type="checkbox"/> | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Message: View message                                                                                                 | <input type="checkbox"/> | <input type="checkbox"/> | <input checked="" type="checkbox"/> |

## How it works...

Entities are powered by the plugin system in Drupal, which means that there is a plugin manager. The default `\Drupal\Core\Entity\EntityTypeManager` provides the discovery and handling of entities. Both the `ContentEntityType` and `ConfigEntityType` entity types and classes extend the base `\Drupal\Core\Entity\EntityType` class.

The `EntityType` class constructor provides a default access handler if it is not provided, through the `\Drupal\Core\Entity\EntityAccessControlHandler` class. There are several methods provided by the class, but the notable ones are `checkAccess`, `checkCreateAccess`, and `checkFieldAccess`. These are designed to be overridden by entity implementations that need their own access checking.

Every core module that provides an entity type implements this to override at least `checkAccess` and `checkCreateAccess`.

`\Drupal\Core\Access\AccessibleInterface` defines an access method and all the entities inherit this interface. The default implementation in `\Drupal\Core\Entity\Entity` will invoke `checkCreateAccess` if the operation is `create`; otherwise, it invokes the generic `access` method of the access controller, which will invoke entity access hooks and the class' own `checkAccess` method.

## There's more...

We will discuss how to implement custom access control for an entity, and use the Entity to simplify the controlling access.

### Controlling access to entity fields

The `checkFieldAccess` method can be overridden to control access to specific entity fields when modifying an entity. Without being overridden by a child class, the `\Drupal\Core\Entity\EntityAccessControlHandler::checkFieldAccess` will always return an allowed access result. The method receives the following parameters:

- ▶ The view and edit operations
- ▶ The current field's definition
- ▶ The user session to check access against
- ▶ And a possible list of field item values

Entity types can implement their own access control handlers and override this method to provide granular control over the modification of their base fields. A good example would be the `User` module and its `\Drupal\user\UserAccessControlHandler`.

User entities have a `pass` field that is used for the user's current password. There is also a `created` field that records when the user was added to the site.

For the `pass` field, it returns `denied` if the operation is `view`, but allows access if the operation is `edit`:

```
case 'pass':
 // Allow editing the password, but not viewing it.
 return ($operation == 'edit') ? AccessResult::allowed() :
 AccessResult::forbidden();
```

The created field uses the opposite logic. When a user became part of the site can be viewed, but should not be able to be edited:

```
case 'created':
 // Allow viewing the created date, but not editing it.
 return ($operation == 'view') ? AccessResult::allowed() :
 AccessResult::forbidden();
```

## See also

- ▶ *Chapter 4, Extending Drupal*

## Providing a custom storage handler

Storage handlers control the loading, saving, and deleting of an entity. `\Drupal\Core\Entity\ContentEntityType` provides the base entity type definition for all content entity types. If it is not specified, then the default storage handler is `\Drupal\Core\Entity\Sql\SqlContentEntityStorage`. This class can be extended to implement alternative load methods or adjustments on save.

In this recipe, we will implement a method that supports loading an entity by a specific property instead of having to write a specific `loadByProperties` method call.

## Getting ready

You will need a custom module to place the code into in order to implement a configuration entity type. Create an `src` directory for your PSR-4 style classes. A custom content entity type needs to be implemented, such as the one in the *Creating a content entity type* recipe.

## How to do it...

1. Create a `MessageStorage` class in the module's `src` directory. This class will extend the default `\Drupal\Core\Entity\Sql\SqlContentEntityStorage` class:

```
<?php

/**
 * @file Contains \Drupal\mymodule\MessageStorage.
 */

namespace Drupal\mymodule;

use Drupal\Core\Entity\Sql\SqlContentEntityStorage;

/**
 * Defines the entity storage for messages.
 */
class MessageStorage extends SqlContentEntityStorage {

}
```

2. By extending the default storage class for our entity type, we can simply add new methods that are relevant to our requirements rather than implementing the extra business logic.
3. Create a `loadMultipleByType` method and we will use this method to provide a simple way of loading all profiles of a specific bundle:

```
/**
 * Load multiple messages by bundle type.
 *
 * @param string $message_type
 * The message type.
 *
 * @return array|\Drupal\Core\Entity\EntityInterface[]
 * An array of loaded message entities.
 */
public function loadMultipleByType($message_type) {
 return $this->loadByProperties([
 'type' => $message_type,
]);
}
```

4. We pass the `type` property so that we can query it based on the message bundle and return all matching message entities.

5. Update the entity's annotation block to have the new storage handler defined:

```
* handlers = {
* "list_builder" = "Drupal\mymodule\MessageListBuilder",
* "storage" = "Drupal\mymodule\MessageStorage",
* "form" = {
* "default" = "Drupal\Core\Entity\EntityForm",
* "add" = "Drupal\Core\Entity\EntityForm",
* "edit" = "Drupal\Core\Entity\EntityForm",
* "delete" = "Drupal\Core\Entity\EntityDeleteForm"
* },
* "route_provider" = {
* "html" = "Drupal\Core\Entity\Routing\DefaultHtmlRouteProvider",
* },
* },
```

6. You can now programmatically interact with your profile entities using the following code:

```
// Get the entity type manager from the container.
Drupal::entityTypeManager()

// Access the storage handler.
->getStorage('message')

// Invoke the new method on custom storage class.
->loadMultipleByType('message');
```

## How it works...

When defining a content entity type, the annotation block begins with `@ContentEntityType`. This declaration, and the properties in it, represents the definition to initiate an instance of the `\Drupal\Core\Entity\ContentEntityType` class just like all other plugin annotations.

In the class constructor, there is a merge to provide default handlers for the storage handler if it is not provided. This will always default to `\Drupal\Core\Entity\Sql\SqlContentEntityStorage` as it provides methods and logic to help its parent class, `ContentEntityStorageBase`, interact with the SQL-based storage.



Configuration entities can have their default \Drupal\Core\Config\Entity\ConfigEntityStorage as well. However, for configuration entities, the configuration management utilizes the \Drupal\Core\Config\StorageInterface implementations for storage rather than classes, which extend ConfigEntityStorage. This logic resides in the configuration factory service.

Extending SqlContentEntityStorage reuses methods required for default Drupal implementations and provides an easy method to create custom methods to interact with loading, saving, and so on.

### There's more...

We will discuss about the custom storage handler and utilizing of different storage backend.

### Utilizing a different storage backend for an entity

Drupal provides mechanisms for supporting different database storage backends that are not provided by Drupal core, such as MongoDB. While it is not stable for Drupal 8 at the time of writing this module, there is a MongoDB module that provides storage interaction.

The module provides \Drupal\mongodb\Entity\ContentEntityStorage, which extends \Drupal\Core\Entity\ContentEntityStorageBase. This class overrides the methods used to create, save, and delete, to write them to a MongoDB collection.

While there are many more steps to provide a custom storage backend for content entities and their fields, this serves as an example for how you can choose to place a custom entity in a different storage backend.

### See also

- ▶ *Chapter 4, Extending Drupal*
- ▶ *Chapter 7, Plug and Play with Plugins*

## Creating a route provider

Entities can implement a route provider that will create the route definitions for the entity's canonical (view), edit, and delete routes. As of Drupal 8.0.1, the add path for an entity is not handled through the default route provider.

In this recipe, we will extend the default \Drupal\Core\Entity\Routing\DefaultHtmlRouteProvider and provide the add routes for our entity.

## Getting ready

You will need a custom module to place the code into in order to implement a configuration entity type. Create an `src` directory for your classes. A custom content entity type needs to be implemented, such as the one in the *Creating a content entity type* recipe.

## How to do it...

1. Create a `MessageHtmlRouteProvider` class in the `src` directory that extends `\Drupal\Core\Entity\Routing\DefaultHtmlRouteProvider`:

```
<?php

/**
 * @file Contains \Drupal\mymodule\MessageHtmlRouteProvider.
 */

namespace Drupal\mymodule;

use Drupal\Core\Entity\Routing\DefaultHtmlRouteProvider;

/**
 * Provides HTML routes for the message entity type.
 */
class MessageHtmlRouteProvider extends DefaultHtmlRouteProvider {

}
```

2. Override the provided `getRoutes` method and collect the parent class's collection of routes returned:

```
<?php

/**
 * @file Contains \Drupal\mymodule\MessageHtmlRouteProvider.
 */

namespace Drupal\mymodule;

use Drupal\Core\Entity\EntityTypeInterface;
use Drupal\Core\Entity\Routing\DefaultHtmlRouteProvider;

/**
 * Provides HTML routes for the message entity type.
 */

```

---

```
class MessageHtmlRouteProvider extends DefaultHtmlRouteProvider {

 /**
 * {@inheritDoc}
 */
 public function getRoutes(EntityTypeInterface $entity_type) {
 $collection = parent::getRoutes($entity_type);

 return $collection;
 }
}
```

3. The parent method for `getRoutes` invokes other methods that check whether the entity has defined edit, canonical, or delete route links in its annotation definition. If the entity has, it will return those as a `\Symfony\Component\Routing\RouteCollection` containing the available routes.
4. Add a new route to the collection that represents the message entity's add route. This will allow us to remove the `mymodule.routing.yml` file:

```
/**
 * {@inheritDoc}
 */
public function getRoutes(EntityTypeInterface $entity_type) {
 $collection = parent::getRoutes($entity_type);

 $route = (new Route('/messages/add'))
 ->addDefaults([
 '_entity_form' => 'message.add',
 '_title' => 'Add message',
])
 ->setRequirement('_entity_create_access', 'message');
 $collection->add('entity.message.add_form', $route);

 return $collection;
}
```

5. This section of the code defines the route programmatically. The definition created in the `routing.yml` is implemented in the `\Symfony\Component\Routing\Route` instance:

Delete the `mymodule.routing.yml` file!

6. Now, we will add routes based on each bundle, iterate through each message bundle, and add a new route that will provide a route to add a message based on the type specified in the route:

```
/**
 * {@inheritDoc}
 */
public function getRoutes(EntityTypeInterface $entity_type) {
 $collection = parent::getRoutes($entity_type);

 $route = (new Route('/messages/add'))
 ->addDefaults([
 '_entity_form' => 'message.add',
 '_title' => 'Add message',
])
 ->setRequirement('_entity_create_access', 'message');
 $collection->add('entity.message.add_form', $route);

 /** @var \Drupal\mymodule\Entity\MessageTypeInterface
 * $message_type */
 foreach (MessageType::loadMultiple() as $message_type) {
 $route = (new Route('/messages/add/{message_type}'))
 ->addDefaults([
 '_entity_form' => 'message.add',
 '_title' => "Add {$message_type->label()} message",
])
 ->setRequirement('_entity_create_access', 'message');
 $collection->add("entity.message.{\$message_type->id()}.add_form", $route);
 }

 return $collection;
}
```

7. This new code loads all the message type entities and adds a new route to each. A route will be created at /messages/add/{message\_type} that will predefine the type of message being created.

## How it works...

Entities are powered by the plugin system in Drupal, which means that there is a plugin manager. The default \Drupal\Core\Entity\EntityTypeManager provides discovery and handling of entities. The \Drupal\Core\Entity\EntityTypeManagerInterface specifies a `getRouteProviders` method that is expected to return an array of strings that provide the fully qualified class name of an implementation of the \Drupal\Core\Entity\Routing\EntityRouteProviderInterface interface.

There is an event subscriber defined in `core.services.yml` called the `entity_route_subscriber`. This service subscribes to the dynamic route event. When this happens, it uses the entity type manager to retrieve all entity type implementations, which provide route subscribers. It then aggregates all the `\Symfony\Component\Routing\RouteCollection` instances received and merges them into the main route collection for the system.

### There's more...

Drupal 8 introduces a router types and provide the add routes for our entity.

### The Entity API module provides add generation

In Drupal 8, the **Entity** module lives on, even though most of its functionalities from Drupal 7 are now in core. The goal of the module is to develop improvements for the developer experience around entities. One of these is the generation of the `add` form and its routes.

The Entity module provides two new route provider aimed specifically for `add` routes, the `\Drupal\entity\Routing\CreateHtmlRouteProvider` and `\Drupal\entity\Routing\AdminCreateHtmlRouteProvider`. The latter option forces the add form to be presented in the administrative theme.

With the Entity module installed, you can add a `create` entry for the `router_providers` array pointing to the new route provider:

```
* "route_provider" = {
* "html" = "Drupal\Core\Entity\Routing\DefaultHtmlRouteProvider",
* "create" = "Drupal\entity\Routing\CreateHtmlRouteProvider",
* },
```

Then, you just need to define the `add-form` entry in your entity's `links` definition, if not already present:

```
* links = {
* "add-form" = "/admin/structure/message-types/add",
* "delete-form" = "/admin/structure/message-types/{message_type}/delete",
* "edit-form" = "/admin/structure/message-types/{profile_type}",
* "admin-form" = "/admin/structure/message-types/{profile_type}",
* "collection" = "/admin/structure/message-types"
* }
```

This reduces the amount of boilerplate code required to implement an Entity.

## Providing a collection route

In the previous recipe, we also needed to define a collection route manually. The route provider can be used to provide this collection route:

```
$route = (new Route('/admin/content/messages'))
 ->addDefaults([
 '_entity_list' => 'message',
 '_title' => 'Messages',
])
 ->setRequirement('permission', $entity_type-
>getAdminPermission());
$collection->add('entity.message.collection', $route);
```

This route definition will replace the one in `routing.yml`. Route generation items should exist in their own handlers, even if only for a specific item. The collection route generation will go into a `CollectionHtmlRouteProvider` class and can be added as a new route handler. The reasoning is that for ease of deprecation in the event such a functionality is added to Drupal core.

### See also

- ▶ *Chapter 4, Extending Drupal*
- ▶ Refer to the Routing system in Drupal 8 at <https://www.drupal.org/developing/api/8/routing>

# 11

## Off the Drupalicon Island

In this chapter, we will see how to use third-party libraries, such as JavaScript, CSS, and PHP in detail:

- ▶ Implementing and using a third-party JavaScript library
- ▶ Implementing and using a third-party CSS library
- ▶ Implementing and using a third-party PHP library
- ▶ Using Composer manager

### Introduction

Drupal 8 comes with a *Proudly Built Elsewhere* attitude. There has been an effort made to use more components created by the PHP community at large and other communities. Drupal 8 is built with Symfony. It includes Twig as its templating system, the provided WYSIWYG editor as its CKEditor, and uses PHPUnit for testing.

How does Drupal 8 promote using libraries made elsewhere? The new asset management system in Drupal 8 makes it easier to use frontend libraries. Drupal implements PSR-0 and PSR-4 from the **PHP Framework Interoperability Group (PHP-FIG)** and **PHP Standards Recommendations (PSRs)** are suggested standards used to increase interoperability between PHP applications. This has streamlined integrating third-party PHP libraries.

Both areas will be constantly improved with each minor release of Drupal 8. These areas will be mentioned throughout the chapter.

## Implementing and using a third-party JavaScript library

In the past, Drupal has only shipped with jQuery and a few jQuery plugins used by Drupal core for the JavaScript API. This has changed with Drupal 8. `Underscore.js` and `Backbone.js` are now included in Drupal, bringing two popular JavaScript frameworks to its developers.

However, there are many JavaScript frameworks that can be used. In *Chapter 5, Frontend for the Win*, you learned about the asset management system and libraries. In this recipe, we will create a module that provides `Angular.js` as a library and a custom Angular application; the demo is available on the AngularJS home page.

### Getting ready

In this example, we will use Bower to manage our third-party `angular.js` library components. If you are not familiar with Bower, it is simply a package manager for frontend components. Instead of using Bower, you can just manually download and place the required files.

If you do not have Bower, you can follow the instructions to install it from `bower.io` at <http://bower.io/#install-bower>. If you do not want to install Bower, we will provide links to manually download libraries.

Having a background of AngularJS is not required but is beneficial. This recipe implements the example from the home page of the library.

### How to do it...

1. Create a custom module named `mymodule` that will serve the AngularJS library and its implementation:

```
name: My Module!
type: module
description: Provides an AngularJS app.
core: 8.x
```

2. Run the `bower init` to create a bower project in your module. We will use most of the default values for the prompted questions:

```
$ bower init
? name mymodule
? description Example module with AngularJS
? main file
? what types of modules does this package expose?
? keywords
```

```
? authors Matt Glaman <nmd.matt@gmail.com>
? license GPL
? homepage
? set currently installed components as dependencies? Yes
? would you like to mark this package as private which prevents it
from being accidentally published to the registry? No
{
 name: 'mymodule',
 authors: [
 'Matt Glaman <nmd.matt@gmail.com>'
],
 description: 'Example module with AngularJS',
 main: '',
 moduleType: [],
 license: 'GPL',
 homepage: '',
 ignore: [
 '**/*.',
 'node_modules',
 'bower_components',
 'test',
 'tests'
]
}

? Looks good? Yes
```

3. Next, we will install the AngularJS library using bower install:

```
$ bower install --save angular
bower angular#* cached git://github.com/angular/
bower-angular.git#1.5.0
bower angular#* validate 1.5.0 against git://github.
com/angular/bower-angular.git#*
bower angular#^1.5.0 install angular#1.5.0
angular#1.5.0 bower_components/angular
```

The `--save` option will ensure that the package's dependency is saved in the created `bower.json`. If you do not have Bower, you can download AngularJS from <https://angularjs.org/> and place it in a `bower_components` folder.

4. Create `mymodule.libraries.yml`. We will define AngularJS as its own library entry:

```
angular:
 js:
 bower_components/angular/angular.js: {}
 css:
 component:
 'bower_components/angular/angular-csp.css': {}
```

5. When the `angular` library is attached, it will add the AngularJS library file and attach the CSS stylesheet.
6. Next, create a `mymodule.module` file. We will use the theme layer's preprocess functions to add a `ng-app` attribute to the root HTML element:

```
<?php

/**
 * Implements hook_preprocess_html().
 */
function mymodule_preprocess_html(&$variables) {
 $variables['html_attributes']['ng-app'] = '';
}
```

7. AngularJS uses the `ng-app` attribute as a directive for bootstrapping an AngularJS application. It marks the root of the application.
8. We will use a custom block to implement the HTML required for the AngularJS example. Make a `src/Plugin/Block` directory and make an `AngularBlock.php` file.
9. Extend the `BlockBase` class and implement the `build` method to return our Angular app's HTML:

```
<?php

/**
 * @file
 * Contains \Drupal\mymodule\Plugin\Block\AngularBlock.
 */

namespace Drupal\mymodule\Plugin\Block;

use Drupal\Core\Block\BlockBase;
```

```

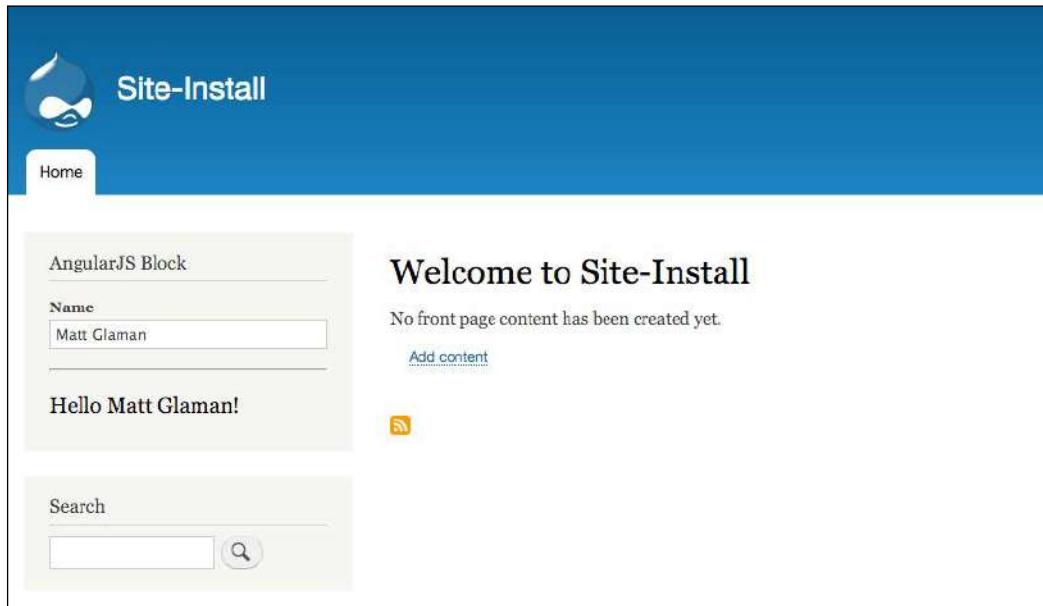
 /**
 * Provides a block for AngularJS example.
 *
 * @Block(
 * id = "mymodule-angular_block",
 * admin_label = @Translation("AngularJS Block")
 *)
 */
 class AngularBlock extends BlockBase {

 public function build() {
 return [
 'input' => [
 '#type' => 'textfield',
 '#title' => $this->t('Name'),
 '#placeholder' => $this->t('Enter a name here'),
 '#attributes' => [
 'ng-model' => 'yourName',
],
],
 'name' => [
 '#markup' => '<hr><h1>Hello {{yourName}}!</h1>',
],
 '#attached' => [
 'library' => [
 'mymodule/angular',
],
],
];
 }
 }
}

```

10. We return a render array that contains the `input`, `name`, and our `library` attachments. The `input` array returns the Form API render information for a text field. The `name` returns a regular markup that will bind Angular's changes to the `yourName` scope variable.
11. Install your custom module.
12. Visit the block layout form from the **Structure** page and place your block.

13. View your Drupal site and interact with your block, which is powered by AngularJS:



## How it works...

The simplicity of integrating with a JavaScript framework is provided by the new asset management system in Drupal 8. The usage of **Bower** is optional, but it is usually a preferred method used to manage frontend dependencies. Using Bower, we can place `bower_components` in an `ignore` file that can be used to keep third-party libraries out of version control.

## See also

- ▶ Refer to the core issue to add `Backbone.js` and `Underscore.js` at <https://www.drupal.org/node/1149866>
- ▶ The recipe *Using the new asset management system*, in *Chapter 5, Frontend for the Win*
- ▶ *Chapter 4, Extending Drupal*, in recipe *Creating a Module*

## Implementing and using a third-party CSS library

Drupal provides many things. However, one thing that it does not provide is any kind of CSS component library. In the recipe *Using the new asset management system*, in *Chapter 5, Frontend for the Win*, we added Font Awesome as a library. CSS frameworks implement robust user interface design components and they can be quite large if using a compiled version with everything bundled. The asset management system can be used to define each component as its own library to only deliver the exact files required for a strong frontend performance.

In this recipe, we will implement the Semantic UI framework, using the CSS only distribution, which provides each individual component's CSS file. We will register the `form`, `button`, `label`, and `input` components as libraries. Our custom theme will then alter the Drupal elements for buttons, labels, and inputs to have the Semantic UI classes and load the proper library.

### Getting ready

In this example, we will use Bower to manage our third-party components. If you are not familiar with Bower, it is simply a package manager used for frontend components. Instead of using Bower, you can just manually download and place the required files.

### How to do it...

1. For this recipe, create a new custom theme named `mytheme` using Classy as a base theme. This way, we can reuse some existing styling. If you are unfamiliar with creating a base theme, refer to the recipe *Creating a custom theme based on Classy*, in *Chapter 5, Frontend for the Win*.
2. Using your terminal, navigate to your theme's directory. Run `bower init` to create a `bower` project:

```
$ bower init
? name mytheme
? description Example theme with Semantic UI
? main file
? what types of modules does this package expose?
? keywords
? authors Matt Glaman <nmd.matt@gmail.com>
? license GPL
? homepage
```

```
? set currently installed components as dependencies? Yes
? would you like to mark this package as private which prevents it
from being accidentally published to the registry? No
{
 name: 'mytheme',
 authors: [
 'Matt Glaman <nmd.matt@gmail.com>'
],
 description: 'Example theme with Semantic UI',
 main: '',
 moduleType: [],
 license: 'GPL',
 homepage: '',
 ignore: [
 '**/*.',
 'node_modules',
 'bower_components',
 'test',
 'tests'
]
}
```

```
? Looks good? Yes
```

3. Next, user bower install to save the Semantic UI library:

```
$ bower install --save semantic-ui
bower semantic-ui#* not-cached git://github.com/Semantic-
Org/Semantic-UI.git#*
bower semantic-ui#* resolve git://github.com/Semantic-
Org/Semantic-UI.git#*
bower semantic-ui#* download https://github.com/
Semantic-Org/Semantic-UI/archive/2.1.8.tar.gz
bower semantic-ui#* extract archive.tar.gz
bower semantic-ui#* resolved git://github.com/Semantic-
Org/Semantic-UI.git#2.1.8
bower jquery#>=1.8 not-cached git://github.com/jquery/
jquery-dist.git#>=1.8
```

```

bower jquery#>=1.8 resolve git://github.com/jquery/
jquery-dist.git#>=1.8
bower jquery#>=1.8 download https://github.com/jquery/
jquery-dist/archive/2.2.0.tar.gz
bower jquery#>=1.8 extract archive.tar.gz
bower jquery#>=1.8 resolved git://github.com/jquery/
jquery-dist.git#2.2.0
bower semantic#^2.1.8 install semantic#2.1.8
bower jquery#>=1.8 install jquery#2.2.0

```

The `--save` option will ensure that the package's dependency is saved in the created `bower.json`. If you do not have Bower, you can download Semantic UI from <https://github.com/semantic-org/semantic-ui/> and place it in a `bower_components` folder.

4. Create `mytheme.libraries.yml` in your theme's base directory. This will hold our main Semantic UI definition along with specific component library definitions.
5. First, we will add a new library to the `form` component:

```

semantic_ui.form:
 js:
 bower_components/semantic/dist/components/form.js: {}
 css:
 component:
 bower_components/semantic/dist/components/form.css: {}

```

The `form` component for Semantic UI has a stylesheet and JavaScript file. Our library ensures that both are loaded when the library is attached.

6. The `button`, `input`, and `label` components do not have any JavaScript files. Add a library for each component:

```

semantic_ui.button:
 css:
 component:
 bower_components/semantic/dist/components/button.css: {}
semantic_ui.input:
 css:
 component:
 bower_components/semantic/dist/components/input.css: {}
semantic_ui.label:
 css:
 component:
 bower_components/semantic/dist/components/label.css: {}

```

7. Now that the libraries are defined, we can use the `attach_library` Twig function to add our libraries to the appropriate templates when we add the Semantic UI classes.
8. Copy the `form.html.twig` file from the Classy theme's `templates` folder and paste it into your theme's `templates` folder. We will attach `mytheme/semantic_ui.form` and add the `ui` and `form` classes:

```
 {{ attach_library('mytheme/semantic_ui.form') }}
<form{{ attributes.addClass(['ui', 'form']) }}>
 {{ children }}
</form>
```

The `attach_library` function will attach the specified library. Use the `addClass` method from Twig to add the `ui` and `form` classes. Semantic UI requires all elements to have the matching `ui` class.

9. Next, copy the `input.html.twig` file from the Classy theme and paste it into your theme's template folder. We will attach `mytheme/semantic_ui.input` and add the `ui` and `input` classes:

```
 {{ attach_library('mytheme/semantic_ui.input') }}
<input{{ attributes.addClass(['ui', 'input']) }} />{{ children }}
```

10. Copy the `input.html.twig` file that we just created and use it to make `input-submit.html.twig`. This template file will be used for submit and other buttons:

```
 {{ attach_library('mytheme/semantic_ui.button') }}
<input{{ attributes.addClass(['ui', 'button', 'primary']) }} />{{ children }}
```

11. Finally, copy the `form-element-label.html.twig` file from Classy to your theme. We will add the `label` library and appropriate class, along with the defaults that Classy has defined:

```
 {{ attach_library('mytheme/semantic_ui.label') }}

{
 %
 set classes = [
 title_display == 'after' ? 'option',
 title_display == 'invisible' ? 'visually-hidden',
 required ? 'js-form-required',
 required ? 'form-required',
 'ui',
 'label',
]
}
{
 % if title is not empty or required -%}
 <label{{ attributes.addClass(classes) }}>{{ title }}</label>
{%- endif %}
```

12. View a form and check whether it has been styled by the Semantic UI CSS framework:

The screenshot shows a 'Site information' configuration form. At the top, there's a section titled 'Site details' with a 'Site name \*' field containing 'My Drupal Site'. Below it is a 'Slogan' field with the placeholder 'How this is used depends on your site's theme.' Underneath is an 'Email address \*' field with 'admin@example.com' entered. A note below it says 'The *From* address in automated emails sent during registration and new password requests, and other notifications.' At the bottom of the form are two buttons: 'Front page' and 'Error pages', followed by a large blue 'Save configuration' button.

## How it works...

The simplicity of integrating with a CSS framework is provided by the new template system, Twig, and the asset management system in Drupal 8. The usage of Bower is optional, but it is usually a preferred method for managing frontend dependencies and can be used to keep third-party libraries out of version control.

While it may be a task to add each component as its own library and attach when specifically needed, it ensures optimal asset delivery. With CSS and JavaScript aggregation enabled, each page will only have the minimal resources that are needed. This is an advantage when the entire Semantic UI minified is still 524 kb.

## See also

- ▶ Refer to Semantic UI at <http://semantic-ui.com/>
- ▶ In the recipe *Creating a custom theme based on Classy*, in Chapter 5, *Frontend for the Win*
- ▶ In the recipe *Using the new asset management system*, in Chapter 5, *Frontend for the Win*
- ▶ In the recipe *Twig templating*, in Chapter 5, *Frontend for the Win*

## Implementing and using a third-party PHP library

Drupal 8 uses Composer for package dependencies and autoloading classes based on PSR standards. This allows us to use any available PHP library much more easily than in previous versions of Drupal.

In this recipe, we will add the `Stack/Cors` library to add CORS support to Drupal 8. `Stack/Cors` is a stack middleware that adds support to the **Access-Control-Allow-Origin** header used in web applications. Without specification, AJAX requests across different domains may fail.



In order to test CORS, you will need to make a cross-domain asynchronous JavaScript request. The **Access-Control-Allow-Origin** header defines domains that are allowed to perform these requests.



### Getting ready

You need to have Composer installed in order to use the Composer manager workflow. You can follow the *Getting Started* documentation at <https://getcomposer.org/doc/00-intro.md>. We will add the `asm89/stack-cors` library as a dependency to our Drupal installation.

### How to do it...

1. Using your terminal, navigate to your Drupal site's root directory.
2. Use the `require` command from Composer to add the library:  
`composer require asm89/stack-cors`
3. Composer will then add the library to the `composer.json` file and install the library along with any dependencies. Its namespace will now be registered.
4. Now, we need to implement a module that registers the `Stack/Cors` library as a middleware service. We'll call the module `asm_stack_cors`. Add the following code to the `asm_stack_cors.info.yml` file:

```
name: Stack/Cors
type: module
description: Adds CORS support to Drupal via the asm89/stack-cors
library
core: 8.x
```

5. Create `asm_stack_cors.services.yml`. This will register the library with Drupal's service container:

```
parameters:
 cors:
 enabled: true
 allowedHeaders: []
 allowedMethods: ['GET']
 allowedOrigins: ['*']
 exposedHeaders: []
 maxAge: false
 supportsCredentials: false

services:
 asm_stack_cors.cors:
 class: Asm89\Stack\Cors
 arguments: ['%cors%']
 tags:
 - { name: http_middleware }
```

6. Next, we will need to implement a compiler pass injection. This will allow us to inject our service into the container when it is compiled. Create a `src/Compiler` directory and make `CorsPass.php`.
7. The `CorsPass.php` will provide the `CorsPass` class, which implements `\Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface`:

```
<?php

/**
 * @file
 * Contains \Drupal\webprofiler\Compiler\StoragePass.
 */

namespace Drupal\asm_stack_cors\Compiler;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;

/**
 * Class CorsPass
 */
class CorsPass implements CompilerPassInterface {

 /**
```

```
* {@inheritDoc}
*/
public function process(ContainerBuilder $container) {
 if (FALSE === $container->hasDefinition('asm_stack_cors.
cors')) {
 return;
 }

 $cors_config = $container->getParameter('cors');

 if (!$cors_config['enabled']) {
 $container->removeDefinition('asm_stack_cors.cors');
 }
}
```

}

8. Enable the new Stack/Cors module. The stack middleware service will be registered and now support CORS requests. To test this, modify the allowedOrigins to only accept your Drupal 8 site's domain:

```
parameters:
 cors:
 enabled: true
 allowedHeaders: []
 allowedMethods: ['GET']
 allowedOrigins: ['http://drupal-8-cookbook.platform']
 exposedHeaders: []
 maxAge: false
 supportsCredentials: false
```

9. Make a request to your website and pass an **Origin** header for a different website, such as `http://example.com`. The request should return a `403 Forbidden` since it is not an allowed domain:

```
$ curl -I 'http://drupal-8-cookbook.platform/' --header 'origin:
http://example.com'
```

```
HTTP/1.1 403 Forbidden
Server: nginx/1.9.6
Date: Sat, 13 Feb 2016 05:04:45 GMT
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
X-Powered-By: PHP/5.6.8
Cache-Control: no-cache
```

## How it works...

Drupal 8 utilizes Symfony components. One of them is the service container and the services it has registered. During the building of the container, there is a compiler pass process that allows alterations of the container's services.

First, we need to register the service in the module's `services.yml` file. The `\Drupal\Core\DependencyInjection\Compiler\StackedKernelPass` class provided by the core will automatically load all the services tagged with `http_middleware`, such as our `asm_stack_cors.cors` service.

Our arguments definition loads items defined in the `parameters.cores` that are used for the class's constructor.

With our provided `CorePass` class, we are also tapping into the container's compile cycle. We check the parameter values for the `cors` section to see whether they are enabled. If the `enabled` setting is set to false, we remove our service from the container.

## See also

- ▶ Refer to the Cross-Origin Resource Sharing specification at  
<http://www.w3.org/TR/cors/>
- ▶ Refer to the Symfony Service Container documentation at  
[http://symfony.com/doc/current/book/service\\_container.html](http://symfony.com/doc/current/book/service_container.html)
- ▶ Refer to the Symfony Dependency Injection component documentation at  
[http://symfony.com/doc/current/components/dependency\\_injection/introduction.html](http://symfony.com/doc/current/components/dependency_injection/introduction.html)

## Using Composer manager

Drupal 8 has an interesting predicament. It utilizes third-party PHP libraries *Proudly Built Elsewhere* that are managed through Composer. However, the packages managed by Composer are committed into version control and Composer is (as of 8.0.x) not part of the Drupal build or installation process.

The Composer manager module provides a way to fully support a Composer-based workflow when working with Drupal. Drupal Commerce requires a Composer-based workflow because it uses third-party PHP libraries. In this recipe, we will examine the Drupal Commerce `composer.json` file and install the module.



Ideally, future versions of Drupal, such as 8.1.x or 8.2.x, will remove the need for the Composer manager and the previous recipe can use a `composer.json` in the module itself to define the external library dependency.

## Getting ready

You need to have Composer installed in order to use the Composer manager workflow. You can follow the *Getting Started* documentation at <https://getcomposer.org/doc/00-intro.md>.

## How to do it...

1. Download the latest version of Composer manager, and place it in your Drupal site's modules folder:

| Other releases       |                                                                    |             |
|----------------------|--------------------------------------------------------------------|-------------|
| Version              | Download                                                           | Date        |
| 8.x-1.0-rc1          | <a href="#">tar.gz (16.95 KB)</a>   <a href="#">zip (25.14 KB)</a> | 2015-Oct-18 |
| Development releases |                                                                    |             |
| Version              | Download                                                           | Date        |
| 8.x-1.x-dev          | <a href="#">tar.gz (16.96 KB)</a>   <a href="#">zip (25.15 KB)</a> | 2015-Oct-18 |

2. The Drupal 8 version will most likely remain as an **Other release**, as the goal is to improve Drupal core's Composer integration and remove the need for this module.
3. Download the Drupal Commerce module and place it in your Drupal site's modules folder. Do not install the module.
4. The Drupal Commerce module contains a `composer.json` that requires three external PHP libraries:

```
{
 "name": "drupal/commerce",
 "type": "drupal-module",
 "description": "Drupal Commerce is a flexible eCommerce
solution.",
 "homepage": "http://drupal.org/project/commerce",
 "license": "GPL-2.0+",
 "require": {
 "commerceguys/intl": "dev-master",
 "commerceguys/pricing": "dev-master",
 "commerceguys/tax": "dev-master"
 },
 "minimum-stability": "dev"
}
```

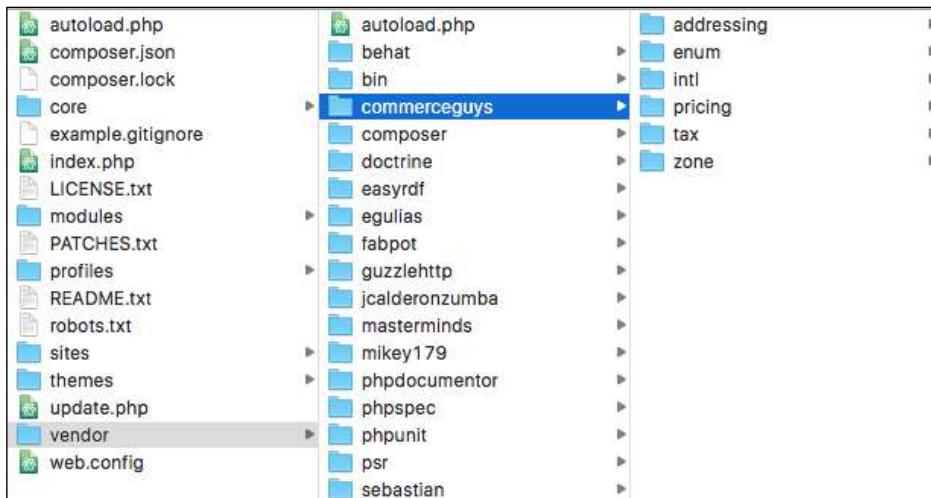
- In order to allow Composer to download our dependencies, we need to run a script provided by Composer manager that will patch Drupal core's `composer.json`. Run this command from the root of your Drupal site's directory:

```
php modules/composer_manager/scripts/init.php
```

- Now, Drupal core's `composer.json` will be aware of any module requirements. The next command will download all the required dependencies:

```
composer drupal-update
```

- The `comerceguys/intl`, `comerceguys/pricing`, and `comerceguys/tax` libraries will now be in the root `vendor` folder of your Drupal site.
- You can now successfully install Drupal Commerce and its submodules:



## How it works...

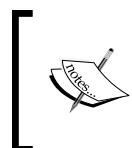
The `composer.json` in the root Drupal 8 directory actually acts as a meta configuration. It defines a requirement for the Wikimedia library that merges the `composer.json` files. The Composer manager module adds the discovered `composer.json` files used for extensions to the list of files to be merged.

When you run the `init.php` script, it updates the root `composer.json` file to manually add a namespace to the module's Composer script and two custom commands: `drupal-rebuild` and `drupal-update`:

```
"autoload": {
 "psr-4": {
 "Drupal\\Core\\Composer\\": "core/lib/Drupal/Core/
Composer",
 }
}
```

```
 "Drupal\\composer_manager\\Composer\\": "modules/contrib/composer_manager/src/Composer"
 }
},
"scripts": {
 "pre-autoload-dump": "Drupal\\Core\\Composer\\Composer::preAutoloadDump",
 "post-autoload-dump": "Drupal\\Core\\Composer\\Composer::ensureHtaccess",
 "post-package-install": "Drupal\\Core\\Composer\\Composer::vendorTestCodeCleanup",
 "post-package-update": "Drupal\\Core\\Composer\\Composer::vendorTestCodeCleanup",
 "drupal-rebuild": "Drupal\\composer_manager\\Composer\\Command::rebuild",
 "drupal-update": "Drupal\\composer_manager\\Composer\\Command::update"
},
}
```

The drupal-rebuild command updates the files that are to be merged. Then, the drupal-update command will download or update the required dependencies.



Currently, the discovery of the extension composer.json files is the major functionality provided by the module. You can follow the issue to provide automatic discovery at <https://www.drupal.org/node/2609568>.

## There's more...

Soon Drupal core will support Composer in custom and contributed modules. We will cover how to simplify the previous *Implementing and using a third-party PHP library* recipe using a composer.json file in your module.

### Updating the Stack/Cors recipe

The *Implementing and using a third-party PHP library* recipe manually adds the `asm89/stack-cors` library to the root `composer.json` as a dependency. A problem with this is that any Drupal core upgrade will remove this modification, this being one reason we require Composer manager.

This can be mitigated by adding a `composer.json` in your module's base directory file that contains the following code:

```
{
 "name": "drupal/asm_stack-cors",
```

```
"type": "drupal-module",
"description": "Implements stack middleware Stack/Cors",
"license": "GPL-2.0+",
"require": {
 "asm89/stack-cors": "^0.2.1"
},
"minimum-stability": "dev"
}
```

Drupal 8 has the Wikimedia `composer-merge-plugin` as a dependency. This package allows you to merge multiple `composer.json` files into one, such as a `composer.json` that is provided by modules. Composer manager provides the missing steps that allow the module's `composer.json` be merged into the root `composer.json` and download the PHP library.

## See also

- ▶ Refer to the Composer documentation for `replace` at  
<https://getcomposer.org/doc/04-schema.md#replace>
- ▶ Refer to the `wikimedia/composer-merge-plugin` library at  
<https://github.com/wikimedia/composer-merge-plugin>
- ▶ Refer to `wikimedia/composer-merge-plugin` manage contrib dependencies at <https://www.drupal.org/node/2609568>



# 12

## Web Services

Drupal 8 ships with the RESTful functionality, to implement web services to interact with your application. This chapter shows you how to enable these features and build your own API:

- ▶ Enabling RESTful interfaces
- ▶ Using GET to retrieve data
- ▶ Using POST to create data
- ▶ Using PATCH to update data
- ▶ Using Views to provide custom data sources
- ▶ Authentication

### Introduction

There are several modules provided by Drupal 8 that enable the ability to turn it into a web services provider. The `Serialization` module provides a means of serializing data to, or deserializing from formats such as JSON and XML. The `RESTful Web Services` module then exposes entities and other APIs through web APIs.

The `HAL` module serializes entities using **Hypertext Application Language** format. (`HAL`) is an Internet Draft standard convention used to hyperlink between resources in an API. HAL JSON is required when working with `POST` and `PATCH` methods. For authentication, the `HTTP Basic Authentication` module provides a simplistic authentication via HTTP headers.

This chapter covers how to work with the `RESTful Web Services` module and the supporting modules around developing a RESTful API powered by Drupal 8. We will cover how to use the `GET`, `POST`, and `PATCH` HTTP methods to manipulate content on the website. Additionally, we will cover how to use Views to provide custom content that lists endpoints. And finally, we will cover how to handle custom authentication for your API.



In an article, Putting off PUT, the team behind the web services initiative chose to not implement PUT and only support PATCH. For more information, refer to the original article at <https://groups.drupal.org/node/284948>. However, the API is open for contributed modules to add the PUT support for core resources or their own.

## Enabling RESTful interfaces

The RESTful Web Services module provides routes that expose endpoints for your RESTful API. It utilizes the Serialization module to handle the normalization to a response and denormalization of data from requests. Endpoints support specific formats and authentication providers.

There is one caveat: RESTful Web Services does not provide a user interface and relies on a single configuration object to enable RESTful endpoints for content entities. Individual endpoints are not their own configuration entities.

When the RESTful Web Services module is first installed, it will enable GET, POST, PATCH, and DELETE methods for the node entity. Each method will support the `hal_json` format and `basic_auth` for its support authentication methods. This ends up with a highly coupled relationship between the HAL and HTTP Basic Authentication modules.

In this recipe, we will install RESTful Web Services and enable the proper permissions to allow the retrieval of nodes via REST to receive our formatted JSON.



We will cover using GET, POST, PATCH, and DELETE in other recipes. This recipe covers the installation and configuration of the base modules to enable web services.

### Getting ready

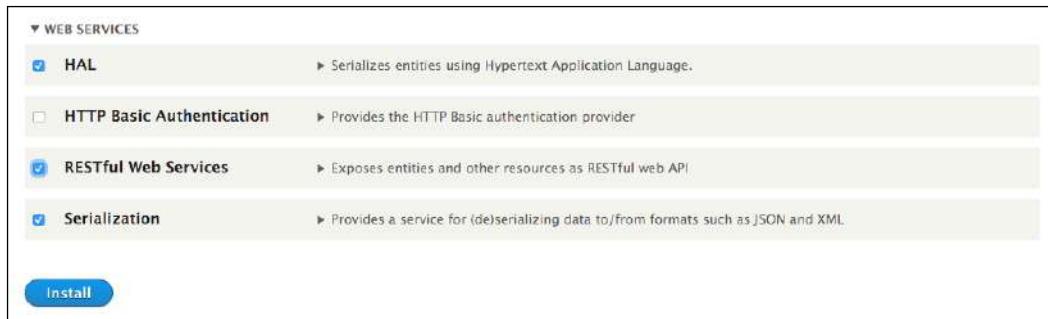
There is a configuration change that might be required if you are running PHP 5.6, the `always_populate_raw_post_data` setting. If you try to enable the RESTful Web Services module without changing the default setting, you will see the following error message on installation:

```
The always_populate_raw_post_data PHP setting should be set to -1 in
PHP version 5.6. Please check the PHP manual for information on how
to correct this. (Currently using always_populate_raw_post_data PHP
setting version Not set to -1.)
```

You will need to modify your PHP's configuration to set `always_populate_raw_post_data` to `-1`.

## How to do it...

1. Visit **Extend** from the administrative toolbar and install the **Web Services** modules: **Serialization**, **RESTful Web Services**, and **HAL**:



2. The RESTful Web Services module provides the default installation configuration in its config/install/rest.settings.yml file. This enables the entity:node endpoint, allowing it to be manipulated over a RESTful interface:

```
resources:
 entity:node:
 GET:
 supported_formats:
 - hal_json
 supported_auth:
 - basic_auth
 POST:
 supported_formats:
 - hal_json
 supported_auth:
 - basic_auth
 PATCH:
 supported_formats:
 - hal_json
 supported_auth:
 - basic_auth
 DELETE:
 supported_formats:
 - hal_json
 supported_auth:
 - basic_auth
```

## Web Services

---

3. In the `rest.settings` configuration namespace, there is a resources section. Each enabled RESTful interface resides under an `entity:ENTITY_TYPE` format with each HTTP method it supports. This YAML settings enables GET, POST, PATCH, and DELETE using HAL JSON and Basic Auth.
4. The RESTful Web Services module exposes each HTTP method as a permission for each endpoint. Visit the Permissions form from the People page.
5. Enable the **Access GET on Content resource** permission for anonymous and authenticated users:

| RESTful Web Services                     |                                     |                                     |                                     |
|------------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
| Access DELETE on <i>Content</i> resource | <input type="checkbox"/>            | <input type="checkbox"/>            | <input checked="" type="checkbox"/> |
| Access GET on <i>Content</i> resource    | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> |
| Access PATCH on <i>Content</i> resource  | <input type="checkbox"/>            | <input type="checkbox"/>            | <input checked="" type="checkbox"/> |
| Access POST on <i>Content</i> resource   | <input type="checkbox"/>            | <input type="checkbox"/>            | <input checked="" type="checkbox"/> |

6. Additionally, you can enable DELETE, PATCH, and POST on other roles, such as authenticated users.
7. Save the permissions form. Node entities are now available in REST endpoints.

## How it works...

The RESTful Web Services module works by implementing an event subscriber service, `rest.resource_routes`, that adds routes to Drupal based on implementations of its `RestResource` plugin. Each plugin returns the available routes based on HTTP methods that are enabled for the resource.

When routes are built, the `\Drupal\rest\Routing\ResourceRoutes` class uses the `RestResource` plugin manager to retrieve all the available definitions. The `rest.settings` configuration object is loaded and inspected. If the resource plugin provides an HTTP method that is enabled in the `rest.settings.resources` definitions, it begins to build a new route. Verification is done against the defined supported formats and supported auth definitions. If the basic validation passes, the new route is added to the `RouteCollection` and returned.

If you provide a `supported_formats` or `supported_auth` value that is not available, the endpoint will still be created. There will be an error, however, if you attempt to use the route with the invalid plugin. For example, routes need to define an authentication provider key, whether it is a disabled provider or an empty YAML array.

### There's more...

The RESTful Web Services module provides a robust API that has some additional items to make a note of. We will explore these in the next recipe.

#### **Soft dependency on the HAL module**

For all intents and purposes, the HAL module is not technically a dependency when you install the RESTful Web Services module. The issue, however, resides in the fact that the default installation configuration sets the allowed format to `hal_json`. In the event that the HAL module is not enabled, an error will be thrown using the default node endpoint configuration.

There is work being done in the Drupal core issue queue to resolve the high coupling of the web services modules.

#### **RestResource plugin to expose data through RESTful Web Services**

The RESTful Web Services module defines a `RestResource` plugin. This plugin is used to define resource endpoints. They are discovered in a module's `Plugin/rest/resource` namespace and need to implement the `\Drupal\rest\Plugin\ResourceInterface` interface.

Drupal 8 provides two implementations of the `RestResource` plugin. The first is the `EntityResource` class that is provided by the RESTful Web Services module. It implements a driver class that allows it to represent each entity type. The second is the Database Logging module that provides its own `RestResource` plugin as well. It allows you to retrieve logged messages by IDs.

The `\Drupal\rest\Plugin\ResourceBase` class provides an abstract base class that can be extended for `RestResource` plugin implementations. If the child class provides a method that matches the available HTTP methods, it will support them. For example, if a class has only a `get` method, you can only interact with that endpoint through HTTP GET requests. On the other hand, you can provide a `trace` method that allows an endpoint to support HTTP TRACE requests.

## The REST UI module

As stated in the recipe's introduction, the RESTful Web Services module does not have a user interface to enable, disable, or configure REST endpoints. The REST UI module provides an interface to configure the available REST endpoints. While the interface is rudimentary, it provides a way to enable and disable content entity endpoints. You can then edit the endpoints and enable or disable the specific HTTP methods and their supported formats.

| RESOURCE NAME | PATH          | DESCRIPTION | OPERATIONS                                                                                                                                                                                                                                             |
|---------------|---------------|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Content       | //node/{node} |             | GET<br>authentication: basic_auth<br>formats: hal_json<br><br>POST<br>authentication: basic_auth<br>formats: hal_json<br><br>PATCH<br>authentication: basic_auth<br>formats: hal_json<br><br>DELETE<br>authentication: basic_auth<br>formats: hal_json |

The REST UI module can be downloaded from Drupal.org at <https://www.drupal.org/project/restui>.

## Rate limiting your API

Many APIs implement a rate limit to prevent abuse of public APIs. When you have publicly exposed APIs, you need to control the amount of traffic hitting the service and prevent abusers from slowing down or stopping your service.

The **Rate Limiter** module implements multiple ways to control access to your public APIs. There is an option to control the rate limit on specific requests, IP address-based limiting, and IP whitelisting.

You can download the Rate Limiter module from [https://www.drupal.org/project/rate\\_limiter](https://www.drupal.org/project/rate_limiter).

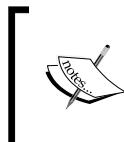
## See also

- ▶ Refer to the Drupal.org documentation for the RESTful Web Services module at <https://www.drupal.org/documentation/modules/rest>
- ▶ Change record: Accept header based routing got replaced by a query parameter, <https://www.drupal.org/node/2501221>
- ▶ *Chapter 7, Plug and Play with Plugins*
- ▶ Refer to the Rate Limiter module at [https://www.drupal.org/project/rate\\_limiter](https://www.drupal.org/project/rate_limiter)
- ▶ Refer to the REST UI module at <https://www.drupal.org/project/restui>

## Using GET to retrieve data

The RESTful Web Services module's entity resource plugin implements a `get` method that is called when an HTTP GET request is made on an appropriate route. The entity is processed and then returned in the appropriate format requested.

In this recipe, we will enable the REST endpoint for taxonomy term entities through GET through both the JSON and HAL JSON formats. Since there is no user interface provided by the Drupal core to edit the RESTful Web Services settings, we will use a command-line tool to modify the values.



Since both Drush and Console, as discussed in *Chapter 9, Configuration Management – Deploying in Drupal 8*, in the recipe *Using command-line workflow processes*, support manipulating configuration objects, this recipe will provide commands for both.

## Getting ready

We will be modifying the `rest.settings` configuration object using command-line tools. You need to have either Drush or Drupal Console installed with the ability to manipulate your Drupal site.

If you are using Mac OS X and Vim is the default editor on the command line, you may experience difficulties. Vim does not always report its exit code as expected, and the command-line tool may not recognize that you have finished editing your code. Each command-line tool provides a method used to specify an editor (such as Nano).

We need to have a taxonomy vocabulary created with some terms so that there is data to be retrieved.

## How to do it...

1. Visit **Extend** from the administrative toolbar and install the **Web Services** modules: `Serialization`, `RESTful Web Services`, and `HAL`.
2. Once the modules are installed, open a terminal and navigate to your Drupal site's directory.
3. Edit the `rest.settings` configuration by running the appropriate configuration edit command:

```
For Drush
drush config-edit rest.settings
```

```
For Console
drupal config:edit rest.settings
```

4. Once the editor is loaded, we need to add an `entity:taxonomy_term` section with a GET definition:

```
resources:
 'entity:taxonomy_term':
 GET:
 support_formats:
 - json
 - hal_json
 supported_auth:
 - cookie
```

5. The `entity:taxnomy_term` points to the entity resource plugin's derivative for the taxonomy term entity. The definitions under `GET` provide the supported formats, which can be returned, and supported authentications.
6. Commit the changes in your editor so that they can be imported into your Drupal site.
7. We need to rebuild Drupal's routes for our endpoints to be activated, since the definition only lives in a configuration object:

```
For Drush
drush cache-rebuild
```

```
For Console
drupal router:rebuild
```

8. Console provides a way to rebuild the routing system, whereas with Drush you need to rebuild all caches.
9. Visit the **Permissions** form from the **People** page. Enable the **Access GET on Taxonomy term resource** permission for anonymous and authenticated users.

10. Access a taxonomy term by visiting your Drupal site with the /taxonomy/term/1?\_format=json path. You will see the following response in your browser:

```
{ "message": "Not acceptable"}
```

11. In order to retrieve data through the endpoint, you need to pass the appropriate Accept header. You can use curl to simulate a request that passes this header:

```
curl --request GET \
--url 'http://example.com/taxonomy/term/1?_format=json' \
--header 'accept: application/json'
```

12. The command will return the formatted JSON with your taxonomy term's information.

## How it works...

The RESTful Web Services module compiles routes based on the `rest.settings.resources` values. When we implement a content entity endpoint, it actually adds a variation to the canonical URL. It allows us to specify a request format on the same path and have the data returned in that format.

The default routes provided by the `\Drupal\rest\Plugin\ResourceBase` class, the base class for resource plugins, return `\Drupal\rest\RequestHandler::handle` for the controller. This method checks the passed `_format` parameter against the configured plugin. If the format is valid, the data is passed to the appropriate serializer.

The serialized data is then returned in the request with appropriate content headers.

## There's more...

There are details that involve the way in which a request is formulated to a Drupal web service resource. We will explore these now.

### Using `_format` instead of the `Accept` header

Early in the Drupal 8 life cycle, up until 8.0.0-beta12, Drupal supported the use of the `Accept` header instead of using the `_format` parameter. Unfortunately, there were issues with external caches since HTML and other formats are served on the same path, only having different `Accept` headers. The only solution to prevent cache poisoning on these external caches, such as Varnish, was to ensure the implementation of the `Vary: Accept` header. There were, however, too many issues regarding CDNs and variance of implementation, so the `_format` parameter was introduced instead of appending extensions (`.json`, `.xml`) to paths.

A detail of the problem can be found on these core issues:

- ▶ Refer to external caches mix up response formats on URLs where content negotiation is in use at <https://www.drupal.org/node/2364011>
- ▶ Check how to implement query parameter-based content negotiation as an alternative to extensions at <https://www.drupal.org/node/2481453>

## See also

- ▶ Refer to Change record: Accept header-based routing got replaced by a query parameter at <https://www.drupal.org/node/2501221>
- ▶ *Chapter 9, Configuration Management – Deploying in Drupal 8*, in the recipe *Using command-line workflow processes*

## Using POST to create data

When working with RESTful Web Services, the HTTP POST method is used to create new entities. We will use the `Basic HTTP Authentication` to authenticate a user and create a new node.

In this recipe, we will use the exposed node endpoint to create a new piece of article content through the RESTful Web Services module. It is a requirement to use HAL JSON when making POST requests, which is provided as the default format for the node endpoint.

## Getting ready

We will be using the `Article` content type provided by the standard installation.

## How to do it...

1. Visit **Extend** from the administrative toolbar and install the `Web Services` modules: `Serialization`, `RESTful Web Services`, and `HAL`
2. We also need to install the **HTTP Basic Authentication** module. This will allow us to authenticate our requests, and it is the default method for the node endpoint.
3. Enable the **Access POST on Content resource** permission for authenticated users.
4. First, we will start constructing the pieces of our JSON payload. We need to provide a `_links` entry that contains objects defining relationship links, which is part of the **Hypertext Application Language** definition implemented by Drupal:

```
{
 "_links": {
 "type": {
```

```
 "href": "http://example.com/rest/type/node/page"
 }
}
}
```

5. The `_links` is a collection of `href` values that link to `/rest/some/path`.
6. We can now provide the `title` and `body` values after our `_links` definition:

```
{
 "_links": {
 "type": {
 "href": "http://example.com/rest/type/node/page"
 }
 },
 "title": [
 { "value" : "Article via POST!" }
],
 "body": [
 { "value" : "We created this over the RESTful API!" }
]
}
```

7. Before we send our JSON payload, we need to retrieve a CSRF token. We do this by performing a GET request against `/rest/session/token`:

```
curl --request GET \
--url http://example.com/rest/session/token \
--header 'accept: text/plain'
```

8. We can send the request containing our body payload to the `/entity/node?_format=hal_json` path through an HTTP POST to create our node:

```
curl --verbose --request POST \
--url 'http://example.com/entity/node?_format=hal_json' \
--user admin:admin \
--header 'accept: application/hal+json' \
--header 'content-type: application/hal+json' \
--data '{"_links":{"type":{"href":"http://example.com/
rest/type/node/page"}}, "title": [{"value": "Article via
POST!"}], "body": [{"value": "We created this over the RESTful
API!"}]}'
```

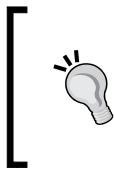
9. We have to append `?_format=hal_json` to ensure that our response comes back in a non-HTML format.
10. A successful request will return an empty message with a `201` header code.

11. View your Drupal site and verify that the node was created.



## How it works...

When working with content entities and the `POST` method, the endpoint is different to the one used for `GET` requests. The `\Drupal\rest\Plugin\rest\resource\EntityResource` class extends the `\Drupal\rest\Plugin\ResourceBase` base class, which provides a route method. If a resource plugin provides an `https://www.drupal.org/link-relations/create` link template, then that path will be used for the `POST` path.

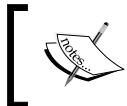


The link template is hardcoded to `https://www.drupal.org` and does not relate to your host name. I tried to research why the creation link uses the `drupal.org` domain. The information can be found at `https://www.drupal.org/node/2019123` and can be resolved by navigating to `https://www.drupal.org/node/2113345`.

The `EntityResource` class defines `/entity/{entity_type}` as the create link template. It then overrides the `getBaseRoute` method to ensure that the `entity_type` parameter is properly populated from the definition.

The `EntityResource` class will run a set of conditions for the request. First, it will validate the `POST` request by checking whether the entity is null. Then the current user is authorized to create the entity type, if the current user also has access to edit all fields provided, and finally it checks that an identifier was not passed. The last condition is important as updates are only to be made through a `PATCH` request.

If the entity is validated, it will be saved. On a successful save, an empty **HTTP 201** response will be returned.



There is currently an issue in the Drupal core issue queue to support JSON for POST and PATCH requests (<https://www.drupal.org/node/1964034>).



## There's more...

Working with POST requests requires some specific formatting to be covered in the recipe. We'll explain them in the next recipe.

### Understanding available \_links requirements

As stated previously, Drupal requires the use of HAL JSON for the format of requests using the POST method. This is done to ensure that the entity is properly created with any relationships it requires, such as the entity type for a content entities bundle. Another example would be to create a comment over a RESTful interface. You would need to provide a `_links` entry for the user owning the comment.

The `rest.link_manager` service uses the `rest.link_manager.type` and `rest.link_manager.relation` and is responsible for returning the URLs for types and relations. By default, a bundle will have a path that resembles `/rest/type/{entity_type}/{bundle}` and relations will resemble `/rest/relation/{entity_type}/{bundle}/{field_name}`.

Taking a user reference as an example; we would have to populate a `uid` field, as follows:

```
{
 "_links": {
 "type": {
 "href": "http://master-rpusmp4jcny2c.us.platform.sh/rest/type/
node/page"
 },
 "http://example.com/rest/relation/node/article/uid": [
 {
 "href": "http://example.com/user/1?_format=hal_json",
 "lang": "en"
 }
]
 }
}
```

Unfortunately, the documentation is sparse, and the best way to learn what `_links` are required is to perform a GET request and study the returned `_links` from the HAL JSON.

## Working with images

Most RESTful APIs utilize base64 encoding of files to support POST operations for uploading an image. Unfortunately, this is not supported in the Drupal core. While there is a `serializer.normalizer.file_entity.hal` service that serializes file entities into HAL JSON, it does not currently work as of 8.0.x and does not appear to be part of 8.1.x.

The `\Drupal\hal\Normalizer\FileEntityNormalizer` class supports denormalization; however, it does not handle base64 and expects binary data.

There is a Drupal core issue for this problem, which is available at <https://www.drupal.org/node/1927648>.

## Using Cross-Site Request Forgery tokens

When working with a POST request, you will need to pass a Cross-Site Request Forgery token if you are authenticating with a session cookie. The **X-CSRF-Token** header is required when using a session cookie to prevent accidental API requests.

If you are using the cookie provider for authentication, you will need to request a CSRF token from the `/rest/session/token` route:

```
curl --request GET \
 --url http://example.com/rest/session/token
```

Take the token string returned in the response and use it as the value for the **X-CSRF-Token** header in your POST request:

```
curl --request POST \
 --url 'http://example.com/entity/node/?_format=hal_json' \
 --header 'content-type: application/hal+json' \
 --header 'x-csrf-token: tmd1RcICiED9D4GCt0_npMWlIOI4MkgW_2lnYKfjlMc'
```

## See also

- ▶ Refer to the Drupal core issue to support POST with json at <https://www.drupal.org/node/1964034>
- ▶ Refer to how to serialize file content (base64) to support REST GET/POST/PATCH on file entity at <https://www.drupal.org/node/1927648>

## Using PATCH to update data

When working with RESTful Web Services, the HTTP PATCH method is used to update entities. We will use the Basic HTTP Authentication to authenticate a user and update a node.

In this recipe, we will use the exposed node endpoint to create a new piece of article content through the RESTful Web Services module. It is a requirement to use HAL JSON when making PATCH requests, which is provided as the default format for the node endpoint.

### Getting ready

We will be using the Article content type provided by the standard installation.

### How to do it...

1. Visit **Extend** from the administrative toolbar and install the **Web Services** modules: **Serialization**, **RESTful Web Services**, and **HAL**
2. We need to also install the **HTTP Basic Authentication** module. This will allow us to authenticate our requests, and it is the default method for the node endpoint.
3. Enable the **Access PATCH on Content resource** permission for authenticated users.
4. Create a sample article node on your Drupal site that we will modify using the REST endpoint:



5. First, we will start building our JSON payload. We need to provide a `_links` entry that contains objects that define relationship links, which is part of the Hypertext Application Language definition implemented by Drupal:

```
{
 "_links": {
 "type": {
 "href": "http://master-rpusmp4jcny2c.us.platform.sh/rest/
type/node/page"
 }
 }
}
```

6. The `_links` is a collection of href values that link to `/rest/some/path`.
7. We will change the node's title by adding a `title` attribute:

```
{
 "_links": {
 "type": {
 "href": "http://master-rpusmp4jcny2c.us.platform.sh/rest/
type/node/page"
 }
 },
 "title": [
 { "value" : "Node updated via REST!" }
]
}
```

8. Before we send our JSON payload, we need to retrieve a CSRF token. We do this by performing a GET request against `/rest/session/token`:

```
curl --request GET \
--url http://example.com/rest/session/token \
--header 'accept: text/plain'
```

9. We can send the request containing our body payload to the `/node/NODE_ID?_format=hal_json` path through an HTTP POST to create our node. Replace `NODE_ID` with the appropriate identifier for the node on your Drupal site:

```
curl --verbose --request PATCH \
--url 'http://example.com/node/52?_format=hal_json' \
--user admin:admin \
--header 'accept: application/hal+json' \
--header 'content-type: application/hal+json' \
--data '{"_links": {"type": {"href": "http://example.com/rest/type/
node/page"}}, "title": [{"value": "Node updated via REST!"}]}'
```

10. If it is successful, you will receive a 204 HTTP code with no content.

11. View your Drupal site and verify that the node was updated:



## How it works...

When working with content entities and the PATCH method, the endpoint is the same as the GET method path. The only validation is the matching of the content type in the headers, which needs to be application/hal+json. The current user's access is checked to see whether they have the permission to update the entity type and each of the submitted fields provided in the request body.

Each field provided will be updated on the entity and then validated. If the entity is validated, it will be saved. On a successful save, an empty HTTP 204 response will be returned.



There is currently an issue in the Drupal core issue queue to support JSON for POST and PATCH requests (<https://www.drupal.org/node/1964034>).

## See also

- ▶ Refer to the Drupal core issue to support POST with json at <https://www.drupal.org/node/1964034>

## Using Views to provide custom data sources

The RESTful Web Services module provides Views plugins that allow you to expose data over Views for your RESTful API. This allows you to create a view that has a path and outputs data using a serializer plugin. You can use this to output entities, such as JSON, HAL JSON, or XML, and it can be sent with appropriate headers.

In this recipe, we will create a view that outputs the users of the Drupal site, providing their username, e-mail, and picture if provided.

### How to do it...

1. Visit **Extend** from the administrative toolbar and install the `Web Services` modules: `Serialization`, `RESTful Web Services`, and `HAL`.
2. Visit **Structure** and then **Views**. Click on **Add new view**. Name the view `API Users` and have it show `Users`.
3. Check the **Provide a REST export** checkbox, and use the `api/users` path. This is where requests will be made:



4. Click on **Save and edit**.
5. Change the format of the row plugin from `Entity` to `Fields` instead so that we can control the specific output.
6. Ensure that your view has the following user entity fields: `Name`, `Email`, and `Picture`.
7. Change the **User: Name** field to the Plain text formatter and do not link it to the user, so the response does not contain any HTML.
8. Save your view.

9. Access your view by visiting /api/users and you will receive a JSON response containing the user information:

```
[
 {
 "name": "houotrara",
 "mail": "houotrara@example.com",
 "user_picture": " \n\n"
 },
 {
 "name": "cragedrelohi",
 "mail": "cragedrelohi@example.com",
 "user_picture": " \n\n"
 }
]
```

## How it works...

The RESTful Web Services module provides a display, row, and format plugin that allows you to export content entities to a serialized format. The REST Export display plugin is what allows you to specify a path to access the RESTful endpoint, and properly assigns the Content-Type header for the requested format.

The Serializer style is provided as the only supported style plugin for the REST Export display. This style plugin only supports row plugins that identify themselves as data display types. It expects data from the row plugin to be raw so that it can be passed to the appropriate serializer.

You then have the option of using the Data entity or Data field row plugins. Instead of returning a render array from their render method, they return raw data that will be serialized into the proper format.

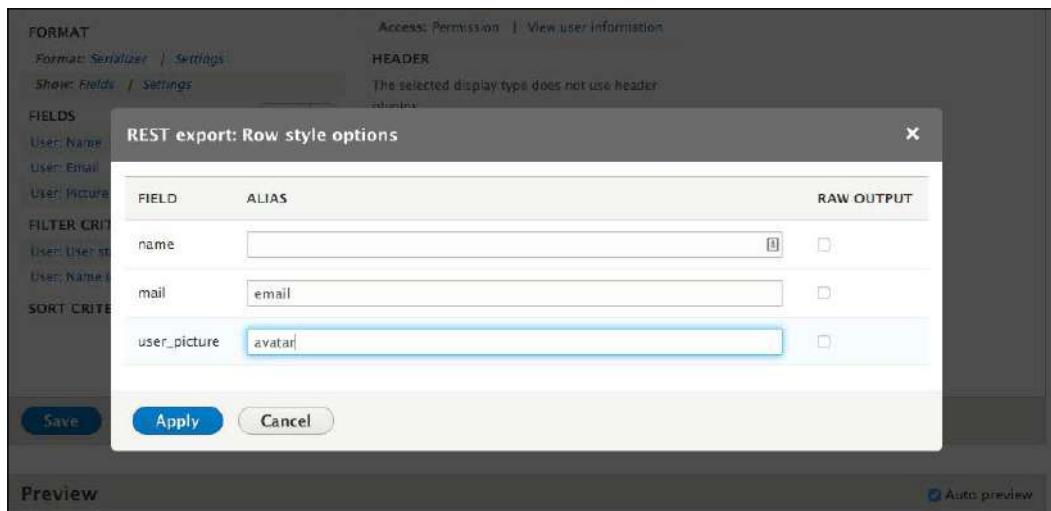
With the row plugins returning raw format data and the data then serialized by the style plugin, the display plugin will then return the response, converted into the proper format via the Serialization module.

## There's more...

Views provide a way to deliver specific RESTful endpoints. We will explore some additional features in the next recipe.

### Controlling the key name in JSON output

The Data fields row plugin allows you to configure field aliases. When the data is returned through the view, it will have Drupal's machine names. This means that custom fields will look something like `field_my_field`, which may not make sense to the consumer. By clicking on **Settings** next to Fields you can set aliases in the modal form:



When you provide an alias, the fields will match. For example, `user_picture` can be changed to `avatar` and the `mail` key can be changed to `e-mail`:

```
[{
 "name": "houotrara",
 "mail": "houotrara@example.com",
 "avatar": "
}]
```

### Controlling access of RESTful Views

When you create a RESTful endpoint with Views, you are not using the same permissions created by the RESTful Web Services module. You need to define the route permissions within the view, allowing you to specify specific roles or permissions for the request.

The default GET method provided by the `EntityResource` plugin does not provide a way to list entities, and allows any entity to be retrieved by an ID. Using Views, you can provide a list of entities, limiting them to specific bundles and many more.

Using Views, you can even provide a new endpoint to retrieve a specific entity. Using `Contextual filters`, you can add route parameters and filters to limit and validate entity IDs. For example, you may want to expose the article content over the API, but not pages.

### Add a URL formatter for the image field

As you may have noticed, our `user_picture` field returned the complete HTML for the image and not a URL for the image directly. In fact, currently, there is no option, as of 8.0.x, to return the URL or endpoint resource for the image file. There is, however, an item in the issue queue to resolve this, which is available at <https://www.drupal.org/node/2517030>, slated for 8.1.x.

You have the option of implementing your own field formatter or applying the patch in your build to get the formatter. Or, you can use the Backports module. At the time of writing this module, the URL field formatter is the only patch provided by the module. However, the purpose of the module is to implement a functionality that is not provided by Drupal but will be provided in the near future. You can get the Backports module at <https://www.drupal.org/project/backports>.

### See also

- ▶ Refer to the Backports module at <https://www.drupal.org/project/backports>

## Authentication

Using the RESTful Web Services module, we define specific supported authentication providers for an endpoint. The Drupal core provides a `cookie` provider, which authenticates through a valid cookie, such as your regular login experience. Then, there is the `HTTP Basic Authentication` module to support HTTP authentication headers.

There are alternatives that provide more robust authentication methods. With cookie-based authentication, you need to use CSRF tokens to prevent unrequested page loads by an unauthorized party. When you use the HTTP authentication, you are sending a password for each request in the request header.

A popular, and open, authorization framework is OAuth. OAuth is a proper authentication method that uses tokens and not passwords. In this recipe, we will implement the `Simple OAuth` module to provide OAuth 2.0 authentication for GET and POST requests.

## Getting ready

If you are not familiar with OAuth or OAuth 2.0, it is a standard for authorization. The implementation of OAuth revolves around the usage of tokens sent in HTTP headers. Refer to the OAuth home page for more information at <http://oauth.net/>.

## How to do it...

1. Download the Simple OAuth module and place it in your Drupal site's modules directory.
2. Visit **Extend** from the administrative toolbar and install the Web Services modules: Serialization, RESTful Web Services, HAL, and Simple OAuth.
3. Edit the `rest.settings` configuration by running the appropriate configuration edit command:

```
For Drush
drush config-edit rest.settings

For Console
drupal config:edit rest.settings
Modify the entity:node resource and replace basic_auth for the GET
and POST method with token_bearer.
resources:
 'entity:node':
 GET:
 supported_formats:
 - hal_json
 supported_auth:
 - token_bearer
 POST:
 supported_formats:
 - hal_json
 supported_auth:
 - token_bearer
```

4. Commit the changes in your editor so that they can be imported into your Drupal site.
5. We need to rebuild Drupal's routes for our endpoint to be activated, since the definition only lives in a configuration object:

```
For Drush
drush cache-rebuild

For Console
drupal router:rebuild
```

6. Enable the Access GET on Content resource and Access POST on Content resource permissions for authenticated users.
7. View your user profile, and click on the **OAuth 2 Tokens** tab.
8. Click on the **Add Access Token** button to create an OAuth token. Then, save the following form:

The screenshot shows a web form titled "Add Access Token" with a star icon. The URL in the header is "Home » Administration » Content » Access Token list". The form has three main sections: "User" (set to "admin (1)"), "Expire" (set to "01/08/2020 05:18:13 AM" with a note about the format), and "Resource" (set to "Global (global)"). At the bottom is a blue "Save" button.

9. Copy the generated token; this will allow you to authenticate requests.
10. Rest a node over REST with the Authorization: Bearer [token] header:

```
curl --request GET \
--url 'http://example.com/node/1?_format=hal_json' \
--header 'accept: application/json' \
--header 'authorization: Bearer JT9zgBgMEDlk2QIF0ecpZEocsYC7-
x649Bovo83HXQM'
How it works
```

In a typical authentication request, there is an authentication manager that uses the authentication\_collector service to collect all the tagged authentication provider servers. Based on the provider's set priority, each service is invoked to check whether it applies to the current request. Each applied authentication provider then gets invoked to see whether the authentication is invalid.

For the RESTful Web Services module, the process is more explicit. The providers identified in the supported\_auth definition for the endpoint are the only services that run through the applies and authenticates process.

## There's more...

We will explore more information on working with authentication providers and the RESTful Web Services module.

### Authentication provider services

When working with the RESTful Web Services module endpoints, the `supported_auth` values reference services tagged with `authentication_provider`. Out of the box, Drupal supports cookie authentication. The following code is provided by the `basic_auth` module to support the HTTP header authentication:

```
services:
 basic_auth.authentication.basic_auth:
 class: Drupal\basic_auth\Authentication\Provider\BasicAuth
 arguments: ['@config.factory', '@user.auth', '@flood', '@entity.
manager']
 tags:
 - { name: authentication_provider, provider_id: 'basic_auth',
 priority: 100 }
```

An authentication provider can be created by making a class in your module's `Authentication\Provider` namespace and implementing the `\Drupal\Core\Authentication\AuthenticationProviderInterface` interface. Then, register the class as a service in your module's `services.yml`.

### Page cache request policies and authenticated web service requests

When working with data that expects authenticated users, the authentication service provider needs to also provide a page cache service handler. Services that are tagged with `page_cache_request_policy` have the ability to check whether the content is cached or not.

The following code is taken from the `basic_auth` module:

```
basic_auth.page_cache_request_policy.disallow_basic_auth_requests:
 class: Drupal\basic_auth\PageCache\DisallowBasicAuthRequests
 public: false
 tags:
 - { name: page_cache_request_policy }
```

The `\Drupal\basic_auth\PageCache\DisallowBasicAuthRequests` class implements the `\Drupal\Core\PageCache\RequestPolicyInterface` interface. The `check` method allows the page cache policy to explicitly deny or remain neutral on a page's ability to be cached. The `basic_auth` module checks whether the default authentication headers are present. For the `simple_oauth` module, it checks whether a valid token is present.



This is an important security measure if you are implementing your own authentication services.

A page cache policy service can be implemented by making a class in your module's `PageCache` namespace and implementing the `\Drupal\Core\PageCache\ResponsePolicyInterface` interface. Then, we need to register the class as a service in your module's `services.yml`.

## IP Authentication Provider

Some APIs that implement server-to-server communication will authenticate using IP address whitelists. For this use case, we have the `IP Consumer Auth` module. Whitelisted IP addresses are controlled by a form that saves a configuration value. If an IP address is whitelisted, the user is authenticated as an anonymous user. While this may not be recommended for POST, PATCH, and DELETE requests, it can provide a simple way to control specific GET endpoints in a private network.

You can download `IP Consumer Auth` from its project page at [https://www.drupal.org/project/ip\\_consumer\\_auth](https://www.drupal.org/project/ip_consumer_auth).

### See also

- ▶ Refer to the OAuth Community Site at <http://oauth.net/>
- ▶ Refer to the OAuth module for OAuth 1.0 support at <https://www.drupal.org/project/oauth>
- ▶ Refer to the simple OAuth module for OAuth 2.0 support at [https://www.drupal.org/project/simple\\_oauth](https://www.drupal.org/project/simple_oauth)
- ▶ Refer to the IP Consumer Auth module at [https://www.drupal.org/project/ip\\_consumer\\_auth](https://www.drupal.org/project/ip_consumer_auth)



# 13

## The Drupal CLI

There are two command-line tools for Drupal 8: **Console** and **Drush**. In this chapter, we will discuss how they make working with Drupal easier by covering the following recipes:

- ▶ Rebuilding cache in Console or Drush
- ▶ Using Drush to interact with the database
- ▶ Using Drush to manage users
- ▶ Scaffolding code through Console
- ▶ Making a Drush command
- ▶ Making a Console command

### Introduction

In the previous chapters of this module, there have been recipes that provide ways of using command-line tools to simplify working with Drupal. There are two contributed projects that provide Drupal with a command-line interface experience.

First, there is Drush. Drush was first created for Drupal 4.7 and has become an integral tool used for day-to-day Drupal operations. However, with Drupal 8 and its integration with Symfony, there came Drupal Console. Drupal Console is a Symfony Console-based application that allows it to reuse more components and integrate more easily with contributed modules.

This chapter contains recipes that will highlight operations that can be simplified by using Drush or Console. By the end of this chapter, you will be able to work with your Drupal sites through the command line.



At the time of writing, Drush was still the primary tool of choice for Drupal 8 as it had a larger feature set. However, Console is rapidly being developed and features are been added regularly. Due to this rapid development, the commands will still exist but the output may differ.

To get started, refer to the following installation guides for each tool:

- ▶ Drush: <http://docs.drush.org/en/master/install/>
- ▶ Console: <https://hechoendrupal.gitbooks.io/drupal-console/content/en/getting/installer.html>

## Rebuilding cache in Console or Drush

Drupal utilizes caching to store plugin definitions, routes, and so on. When you add a new plugin definition or new route, you need to rebuild Drupal's cache for it to be recognized.

In this recipe, we will walk you through using both Drush and Console to clear various cache bins in Drupal. It is important to know how to clear specific cache bins so that you do not need to rebuild everything, if possible.

### How to do it...

1. Open a terminal and navigate to an installed Drupal directory.
2. We use the `cache-rebuild` command in Drush to rebuild all of Drupal's caches, including routes:

```
$ drush cache-rebuild
Cache rebuild complete.
```

3. Drush will bootstrap Drupal to a full site and invoke a full cache clear.
4. In Console, we use the `cache:rebuild` command to clear specific cache bins:

```
$ drupal cache:rebuild
Select cache. [all]:
> all
Rebuilding cache(s), wait a moment please.
[OK] Done clearing cache(s).
```

5. If you select `all`, the same operation is run in Drush. However, Console is set up to allow distinct cache bins in future development.

6. If you only need to rebuild your routes in Drupal, you can use the `router:rebuild` command in Console:

```
$ drupal router:rebuild
Rebuilding routes, wait a moment please
[OK] Done rebuilding route(s).
```

7. Instead of clearing all caches to rebuild routes, it can be done directly with this command.
8. Drush provides `twig-compile` to rebuild template changes without clearing all caches:

```
$ drush twig-compile
```

## How it works...

Both Drush and Console will load files off your Drupal site and bootstrap the application. This allows the commands to invoke functions and methods found in Drupal.

Currently, Drush does not implement the dependency injection container, and still needs to rely on procedural functions in Drupal. Console, however, harnesses the dependency injection container, allowing it to reuse Drupal's container and services.

## Using Drush to interact with the database

When working with any application that utilizes a database, there are times when you will need to export a database and import it elsewhere. Most often, you would do this with a production site to work on it locally. This way, you can create a new configuration that can be exported and pushed to production, as discussed in *Chapter 9, Configuration Management – Deploying in Drupal 8*.

In this recipe, we will export a database dump from a production site in order to set up the local development. The database dump will be imported over the command line and sanitized. We will then execute an SQL query through Drush to verify sanitization.

## Getting ready

Drush has the ability to use **site aliases**. Site aliases are configuration items that allow you to interact with a remote Drupal site. In this recipe, we will use the following alias to interact with a fictional remote site to show how a typical workflow will go to fetch a remote database.

Note that you do not need to use a Drush alias to download the database dump created in the recipe; you can use any method you are familiar with (manually, from the command line, use mysqldump or phpMyAdmin):

```
$aliases['drupal.production'] = array(
 'uri' => 'example.com',
 'remote-host' => 'example.com',
 'remote-user' => 'someuser',
 'ssh-options' => '-p 2222',
) ;
```



Read the Drush documentation for more information on site aliases at <http://docs.drush.org/en/master/usage/#site-aliases>. Site aliases allow you to interact with remote Drupal installations.



We will also assume that the local development site has not yet been configured to connect to the database.

## How to do it...

1. We will use the sql-dump command to export the database into a file. The command returns the output that needs to be redirected to a file:

```
$ drush @drupal.production sql-dump > ~/prod-dump.sql
```

This will take the data from sql-dump and save it in prod-dump.sql in your home directory.

2. Navigate to your local Drupal site's directory and copy sites/default/default.settings.php to sites/default/settings.php.
3. Edit the new settings.php file and add a database configuration array at the end of the file:

```
// Database configuration.
$databases['default']['default'] = array(
 'driver' => 'mysql',
 'host' => 'localhost',
 'username' => 'mysql',
 'password' => 'mysql',
 'database' => 'data',
 'prefix' => '',
 'port' => 3306,
 'namespace' => 'Drupal\\Core\\Database\\Driver\\mysql',
) ;
```

4. This will add our database connection information as the default database in the global \$databases variable.

5. Using the `sql-cli` command, we can import the database dump that we created:

```
$ drush sql-cli < ~/prod-dump.sql
```

This will then run the SQL dump as a set of commands on the database, importing your data.

6. The `sql-sanitize` command allows you to obfuscate user e-mails and passwords in the database:

```
$ drush sql-sanitize
```

This will update all of the users in the user table by changing their usernames and passwords.

7. To verify that our information has been sanitized, we will use the `sql-query` command to run a query against the database:

```
$ drush sql-query "SELECT uid, name, mail FROM users_field_data;"
```

The command will return a list of the results.

## How it works...

When working with Drush, we have the ability to use Drush aliases. A Drush alias contains a configuration that allows the tool to connect to a remote server and interact with that server's installation of Drush.



You need to have Drush installed on your remote server in order to use a site alias for it.

The `sql-dump` command executes the proper dump command for the database driver, which is typically MySQL and the `mysqldump` command. It streams to the terminal and must be piped to a destination. When piped to a local SQL file, we can import it and execute the `create` commands to import our database schema and data.

With the `sql-cli` command, we are able to execute SQL commands to the database through Drush. This allows us to redirect the file contents to the `sql-cli` command and run the set of SQL commands. With the data imported, the `sql-sanitize` command replaces usernames and passwords.

Finally, the `sql-query` command allows us to pass an SQL command directly to the database and return its results. In our recipe, we query the `users_field_data` to verify that e-mails have been sanitized.

## There's more...

Working with Drupal over the command line simplifies working with the database. We will explore this in more detail in the following sections.

### Using gzip with sql-dump

Sometimes databases can be quite large. The `sql-dump` command has a `gzip` option that will output the SQL dump using the `gzip` command. In order to run the command, you would simply do:

```
$ drush sql-dump --gzip dump.sql.gz
```

The end result provides a reduction in the dump file:

```
-rw-r--r-- 1 user group 3058522 Jan 14 16:10 dump.sql
-rw-r--r-- 1 user group 285880 Jan 14 16:10 dump.sql.gz
```



If you create a gzipped database dump, make sure that you unarchive it before attempting an import with the `sql-cli` command.

### Using Console to interact with the database

At the time of writing this module, Console does not provide a command for sanitizing the database. There are the `database:connect` and `database:client` commands, which will launch a database client. This allows you to be logged into the database's command-line interface:

```
$ drupal database:client
$ drupal database:connect
```

These commands are similar to the `sql-cli` and `sql-connect` commands from Drush. The `client` command will bring you to the database's command-line tool, where `connect` shows the connection string.

Console also provides the `database:dump` command. Unlike Drush, this will write the database dump for you in the Drupal directory:

```
$ drupal database:dump
[OK] Database exported to: /path/to/drupal/www/data.sql
```

## See also

- ▶ [Chapter 9, Configuration Management – Deploying in Drupal 8](#)

- ▶ Refer to Dumping Data in SQL Format with mysqldump at <http://dev.mysql.com/doc/refman/5.7/en/mysqldump-sql-format.html>

## Using Drush to manage users

When you need to add an account to Drupal, you will visit the People page and manually add a new user. Drush provides the complete user management for Drupal, from creation to role assignment, password recovery, and deletion. This workflow allows you to create users easily and provides them with a login without having to enter your Drupal site.

In this recipe, we will create a `staff` role with a `staffmember` user and log in as that user through Drush.

### How to do it...

1. Use the `role-create` command to create a new role labeled `staff`:

```
$ drush role-create staff
Created "staff"
```

2. Use the `role-lists` command to verify that the role was created in Drupal:

```
$ drush role-list
 ID Role Label
anonymous Anonymous user
authenticated Authenticated user
administrator Administrator
staff Staff
```

3. The `user-create` command will create our user:

```
$ drush user-create staffmember
 User ID : 2
 User name : staffmember
 User roles : authenticated
 User status : 1
```

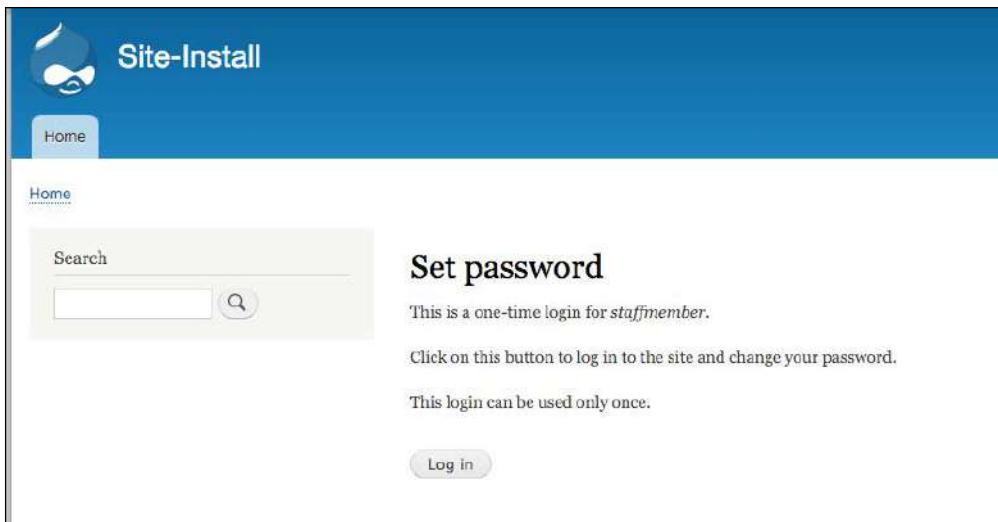
4. In order to add the role, we need to use the `user-add-role` command:

```
$ drush user-add-role staff staffmember
Added role staff role to staffmember
```

5. We will now log in as the `staffmember` user using the `user-login` command:

```
$ drush user-login staffmember --uri=http://example.com
http://example.com/user/reset/2/1452810532/IalnJvbr2UQ3Pi_
QnmIT1VgcCWzDtnKmHxf-I2eAqPE
```

6. Provide the `uri` option to ensure that a correct URL points to a one time login link.
7. Copy the link and paste it in your browser to log in as that user.



## How it works...

When you reset a password in Drupal, a special one-time login link is generated. The login link is based on a generated hash. The Drush command validates the given user, which exists in the Drupal site and then passes it to the `user_pass_reset_url` function from the User module.

The URL is made up of the user's ID, the timestamp when the link was generated, and a hash based on the user's last login time, link generation, and e-mail. When the link is loaded, this hash is rebuilt and verified. For example, if the user has logged in since the time it was generated, the link will become invalid.

When used on a machine that has a web browser installed, Drush will make an attempt to launch the link in a web browser for you. The `browser` option allows you to specify which browser should be launched. Additionally, you can use `no-browser` to prevent one from being launched.

## There's more...

The command line offers the ability to simplify user management and user administration. Next, we will explore more on this topic in detail.

## Advanced user-login use cases

The `user-login` command is a useful tool that allows some advanced use cases. For instance, you can append a path after the username and be launched to that path. You can pass a UID or e-mail instead of a username in order to log in as a user.

You can use the `user-login` to secure your admin user account. In Drupal, the user with the identifier of 1 is treated as the root, and can bypass all permissions. Many times, this is the default maintenance account used to work on the Drupal site. Instead of logging in manually, you can set the account to a very robust passphrase and use the `user-login` command when you need to access your site. With this, the only users who should be able to log in as the administrator account are those with access to run Drush commands on the website.

## Using Drupal Console

Console also provided commands to interact with users. While they do not allow the creation of users or roles, they provide basic user management.

The `user:login:url` command will generate a one time login link for the specified user ID . This uses the same methods as the Drush command:

```
$ drupal user:login:url 2
```

The `user:password:reset` command allows you to reset a user's password to the new provided password. You can provide the user ID and new password as arguments, but if missing, the values will be prompted for interactively:

```
$ drupal user:password:reset 2 newpassword
```

The `create:users` command provides an interactive way to generate bulk users, which are useful to debug. However, it cannot make individual users with specific passwords like Drush.

## Scaffolding code through Console

When Drupal Console was first introduced, one of the biggest highlights was its ability to scaffold code. The project has turned into a much larger Drupal runner over the command-line interface, but much of its resourcefulness is code generation.

As you may have noticed in the previous chapters and recipes, there can be a few mundane tasks and a bit of boilerplate code. Drupal Console enables Drupal developers to create various components without having to write all of the boilerplate code.

In *Chapter 10, The Entity API* we covered the creation of a custom entity type. In this recipe, we will automate most of that process using Console to generate our content entity.

## Getting ready

For this recipe, you need to have Drupal Console installed. The tool will generate everything else for us. You will need to have a Drupal 8 site installed. Many of Console's commands will not work (or be listed) unless they can access an installed Drupal site. This is because of the way it interacts with Drupal's service container.

## How to do it...

- From the root of your Drupal site, generate a module with the `generate:module` command and follow the interactive process. Use the defaults prompted besides giving it a module name:

```
$ drupal generate:module
```

```
Welcome to the Drupal module generator
```

```
Enter the new module name: My module
Enter the module machine name [my_module]:
Enter the module Path [/modules/custom]:
Enter module description [My Awesome Module]:
Enter package name [Other]:
Enter Drupal Core version [8.x]:
Do you want to generate a .module file (yes/no) [no]: no
Define module as feature [no]? no
Do you want to add a composer.json file to your module [yes]? yes
Would you like to add module dependencies [no]?
Do you confirm generation [yes]?
```

```
Generated or updated files
```

```
Site path: /path/to/drupal8/www
1 - modules/custom/my_module/my_module.info.yml
2 - modules/custom/my_module/composer.json
```

- The command walks you through prompts to generate the `info.yml` and will output the path of the generated `info.yml` and `composer.json` files.

3. Next, we will generate our content entity. Provide a module name:

```
$ drupal generate:entity:content
```

```
Enter the module name: my_module
Enter the class of your new entity [DefaultEntity]:
CustomContentEntity
Enter the name of your new entity [custom_content_entity]:
Enter the label of your new entity [Custom content entity]:
Do you want this (content) entity to have bundles (yes/no) [no]:
```

```
Generated or updated files
```

```
Site path: /Users/mgllaman/Drupal/sites/drupal8/www
1 - modules/custom/my_module/my_module.routing.yml
2 - modules/custom/my_module/my_module.permissions.yml
3 - modules/custom/my_module/my_module.links.menu.yml
4 - modules/custom/my_module/my_module.links.task.yml
5 - modules/custom/my_module/my_module.links.action.yml
6 - modules/custom/my_module/src/CustomContentEntityInterface.php
7 - modules/custom/my_module/src/
CustomContentEntityAccessControlHandler.php
8 - modules/custom/my_module/src/Entity/CustomContentEntity.php
9 - modules/custom/my_module/src/Entity/
CustomContentEntityViewsData.php
10 - modules/custom/my_module/src/CustomContentEntityListBuilder.
php
11 - modules/custom/my_module/src/Entity/Form/
CustomContentEntitySettingsForm.php
12 - modules/custom/my_module/src/Entity/Form/
CustomContentEntityForm.php
13 - modules/custom/my_module/src/Entity/Form/
CustomContentEntityDeleteForm.php
14 - modules/custom/my_module/custom_content_entity.page.inc
15 - modules/custom/my_module/templates/custom_content_entity.
html.twig
```

4. When the command is finished executing, it will output all of the created files.

5. Install your `my` module using Console:

```
$ drupal module:install my_module
[OK] The following module(s) were installed successfully: my_
module
Rebuilding cache(s), wait a moment please.
[OK] Done clearing cache(s).
```

6. View **Structure** and find your **Custom content entity settings**:

The screenshot shows a list of configuration options under the 'Structure' menu. The items listed are: Contact forms, Content types, Custom content entity settings, Display modes, and Menus. Each item has a brief description below it.

|                                                                                           |
|-------------------------------------------------------------------------------------------|
| <b>Contact forms</b><br>Create and manage contact forms.                                  |
| <b>Content types</b><br>Create content types and manage their default settings.           |
| <b>Custom content entity settings</b><br>Configure Custom content entity entities         |
| <b>Display modes</b><br>Configure what displays are available for your content and forms. |
| <b>Menus</b><br>Manage menus and menu links.                                              |

## How it works...

One of the biggest features of Console was its ability to reduce the time spent by developers to create code for Drupal 8. Console utilizes the Twig templating engine to provide code generation. These Twig templates contain variables and logic that are compiled into the end result code.

A set of generator classes receive specific parameters, which are received through the appropriate command, and pass them to Twig for rendering. This allows Console to easily stay up to date with changes in Drupal core and still provide valuable code generation.

## Making a Drush command

Drush provides an API that allows developers to write their own commands. These commands can be part of a module and loaded through a Drupal installation, or they can be placed in the local user's Drush folder for general purposes.

Often, contributed modules create commands to automate user interface operations. However, creating a custom Drush command can be useful for specific operations. In this recipe, we will create a command that loads all the users who have not logged in in the last 10 days and resets their password.

## Getting ready

For this recipe, you need to have Drush installed. We will be creating a command in a local user directory.

## How to do it...

1. Create a file named `disable_users.drush.inc` in the `~/.drush` folder for your user:

```
<?php

/**
 * @file
 * Loads all users who have not logged in within 10 days and
 disables them.
 */
```

2. Add the Drush command hook that will allow Drush to discover our commands provided by the file:

```
/**
 * Implements hook_drush_command().
 */
function disable_users_drush_command() {
 $items = [];
 $items['disable-users'] = [
 'description' => 'Disables users after 10 days of inactivity',
];
 return $items;
}
```

3. This hook returns an array of command configurations; the hook should be prefixed with the part of the file before `.drush.inc`.
4. Next, we will create the command callback function, which will end up holding all of our logic:

```
/**
 * Implements drush_hook_COMMAND().
 */
```

```
function drush_disable_users_disable_users() {
}
}
```

5. Since our filename is `disable_users.drush.inc` and our command is `disable-users`, the hook turns out to be `drush_disable_users_disable_users`.
6. Update the function to create a `DateTime` object, representing 10 days ago. We will use this to generate a timestamp for our query:

```
/**
 * Implements drush_hook_COMMAND().
 */
function drush_disable_users_disable_users() {
 // Get the default timezone and make a DateTime object for 10
 // days ago.
 $system_date = \Drupal::config('system.date');
 $default_timezone = $system_date->get('timezone.default') ?:
 date_default_timezone_get();
 $now = new DateTime('now', new DateTimeZone($default_timezone));
 $now->modify('-10 days');
}
```

7. We load the `system.date` configuration object to get the default time zone and properly construct a `DateTime` object, modified 10 days ago.
8. Now, we will add our query, which will query all the user entities who have a login timestamp greater than 10 days:

```
/**
 * Implements drush_hook_COMMAND().
 */
function drush_disable_users_disable_users() {
 // Get the default timezone and make a DateTime object for 10
 // days ago.
 $system_date = \Drupal::config('system.date');
 $default_timezone = $system_date->get('timezone.default') ?:
 date_default_timezone_get();
 $now = new DateTime('now', new DateTimeZone($default_timezone));
 $now->modify('-10 days');

 $query = \Drupal::entityQuery('user')
 ->condition('login', $now->getTimestamp(), '>');
 $results = $query->execute();

 if (empty($results)) {
 drush_print('No users to disable!');
 }
}
```

9. If there are no results, an empty array will be returned.
10. Next, we will iterate over the results and mark the user as disabled:

```
/***
 * Implements drush_hook_COMMAND().
 */
function drush_disable_users_disable_users() {
 // Get the default timezone and make a DateTime object for 10
 // days ago.
 $system_date = \Drupal::config('system.date');
 $default_timezone = $system_date->get('timezone.default') ?: date_default_timezone_get();
 $now = new DateTime('now', new DateTimeZone($default_timezone));
 $now->modify('-10 days');

 $query = \Drupal::entityQuery('user')
 ->condition('login', $now->getTimestamp(), '>');
 $results = $query->execute();

 if (empty($results)) {
 drush_print('No users to disable!');
 }

 foreach ($results as $uid) {
 /** @var \Drupal\user\Entity\User $user */
 $user = \Drupal\user\Entity\User::load($uid);
 $user->block();
 $user->save();
 }

 drush_print(dt('Disabled !count users', ['!count' => count($results)]));
}
```

11. The result is an array of user IDs. We loop over them to load the user, mark them as disabled, and then save them to commit the changes.
12. Drush's cache will need to be cleared in order to discover your new command:

```
$ drush cache-clear drush
```

13. Check whether the command exists:

```
$ drush disable-users --help
Disables users after 10 days of inactivity
```

## How it works...

Drush works by scanning specific directories for files that follow the `COMMANDFILE.drush.inc` pattern. You can think of `COMMANDFILE` for Drush as a representation of a module name in Drupal's hook system. When implementing a Drush hook, in the `HOOK_drush` format, you need to replace `HOOK` with your `COMMANDFILE` name, just as you would do in Drupal with a module name.

In this recipe, we created a `disable_users.drush.inc` file. This means that all hooks and commands in the file need to use `disable_users` for hook invocations. Drush uses this to load the `hook_drush_command` hook that returns our command information.

We then provide the functionality of our logic in the `drush_hook_command` hook. For this hook, we replace `hook` with our `commandfile` name. This was `disable_users`, giving us `drush_disable_users_command`. We replace `command` with the `command` that we defined in `hook_drush_command`, which was `disable-users`. We then have our final `drush_disable_users_disable_users` hook.

## There's more...

Drush commands have additional options that can be specified in their definitions. We explore their abilities to control the required level of Drupal integration for a command.

### Specifying the level of Drupal's bootstrap

Drush commands have the ability to specify the level of Drupal's bootstrap before being executed. Drupal has several bootstrap levels in which only specific parts of the system are loaded. By default, a command's bootstrap is at `DRUSH_BOOTSTRAP_DRUPAL_LOGIN`, which is at the same level as accessing Drupal over the web.

Commands, depending on their purpose, can choose to avoid bootstrapping Drupal at all or only until the database system is loaded. Drush commands that are utilities, such as the `Git Release Notes` module, provide a Drush command that does not interact with Drupal. It specifies a bootstrap of `DRUSH_BOOTSTRAP_DRUSH`, as it only interacts with repositories to generate change logs based on `git` tags and commits.

## See also

- ▶ Refer to how to creating custom Drush commands at <http://docs.drush.org/en/master/commands/>
- ▶ Refer to how to installing Drush at <http://docs.drush.org/en/master/install/>
- ▶ Refer to the Drush Bootstrap process at <http://docs.drush.org/en/master/bootstrap/>

## Making a Console command

Drupal Console makes use of the Symfony Console project and other third-party libraries to utilize modern PHP best practices. In doing so, it follows Drupal 8 practices as well.

This allows Console to use namespaces for the command detection and interaction with Drupal by reading its class loader.

This allows developers to easily create a Console command by implementing a custom class in a module.

In this recipe, we will create a command that loads all the users who have not logged in in the last 10 days and resets their password. We will generate the base of our command using the scaffolding commands.

### Getting ready

For this recipe, you need to have Drupal Console installed. The tool will generate everything else for us. You will need to have a Drupal 8 site installed.

### How to do it...

1. Create a new module that will hold your Console command, such as `console_commands`:

```
$ drupal generate:module
```

```
// Welcome to the Drupal module generator
```

```
Enter the new module name:
```

```
> Console commands
```

```
Enter the module machine name [console_commands] :
```

```
>
```

```
Enter the module Path [/modules/custom] :
```

```
>
```

```
Enter module description [My Awesome Module] :
```

```
>
```

```
Enter package name [Other] :
```

```
>
```

```
Enter Drupal Core version [8.x] :
>

Define module as feature (yes/no) [no] :
>

Do you want to add a composer.json file to your module (yes/no)
[yes] :
>

Would you like to add module dependencies (yes/no) [no] :
>

Do you confirm generation? (yes/no) [yes] :
>

Generated or updated files
Site path: /path/to/drupal8/www
1 - modules/custom/console_commands/console_commands.info.yml
2 - modules/custom/console_commands/console_commands.module
3 - modules/custom/console_commands/composer.json
```

2. Next, we will generate the command's base files using the `generate:command` command. Call the `Disable Users` command:

```
$ drupal generate:command
```

```
// Welcome to the Drupal Command generator
Enter the module name [console_commands] :
> console_commands

Enter the Command name. [console_commands:default] :
> console_commands:disable_users

Enter the Command Class. (Must end with the word 'Command') .
[DefaultCommand] :
> DisableUsersCommand

Is the command aware of the drupal site installation when
executed?. (yes/no) [yes] :
> yes

Do you confirm generation? (yes/no) [yes] :
> yes
```

```
Generated or updated files
Site path: /path/to/drupal8/www
1 - modules/custom/console_commands/src/Command/
DisableUsersCommand.php
```

3. Edit the created `DisableUsersCommand.php` file and remove the boilerplate example code from the `execute` method:

```
/**
 * {@inheritDoc}
 */
protected function execute(InputInterface $input,
OutputInterface $output) {
}
```

4. The `execute` method is invoked by Symfony Console and contains all the execution operations.
5. Update the function to create a `DateTime` object, representing 10 days ago. We will use this to generate a timestamp for our query:

```
/**
 * {@inheritDoc}
 */
protected function execute(InputInterface $input,
OutputInterface $output) {
 // Get the default timezone and make a DateTime object for 10
days ago.
 $system_date = \Drupal::config('system.date');
 $default_timezone = $system_date->get('timezone.default') ?:
date_default_timezone_get();
 $now = new \DateTime('now', new \DateTimeZone($default_
timezone));
 $now->modify('-10 days');
}
```

6. We load the `system.date` configuration object to get the default time zone and properly construct a `DateTime` object, modified for 10 days ago.
7. Now, we will add our query, which will query all the user entities who have a login timestamp greater than 10 days:

```
/**
 * {@inheritDoc}
 */
protected function execute(InputInterface $input,
OutputInterface $output) {
 // Get the default timezone and make a DateTime object for 10
days ago.
```

```
$system_date = \Drupal::config('system.date');
$default_timezone = $system_date->get('timezone.default') ?:
date_default_timezone_get();
$now = new \DateTime('now', new \DateTimeZone($default_timezone));
$now->modify('-10 days');

$query = \Drupal::entityQuery('user')
 ->condition('login', $now->getTimestamp(), '>');
$results = $query->execute();

if (empty($results)) {
 $output->writeln('<info>No users to disable!</info>');
}
}
```

8. To output to the terminal, you need to use the `write` or `writeln` functions from the `OutputInterface` object.
9. Next, we will iterate over the results and mark the user as disabled:

```
/**
 * {@inheritDoc}
 */
protected function execute(InputInterface $input,
OutputInterface $output) {
 // Get the default timezone and make a DateTime object for 10
 // days ago.
 $system_date = \Drupal::config('system.date');
 $default_timezone = $system_date->get('timezone.default') ?:
date_default_timezone_get();
 $now = new \DateTime('now', new \DateTimeZone($default_timezone));
 $now->modify('-10 days');

 $query = \Drupal::entityQuery('user')
 ->condition('login', $now->getTimestamp(), '>');
 $results = $query->execute();

 if (empty($results)) {
 $output->writeln('<info>No users to disable!</info>');
 }

 foreach ($results as $uid) {
 /** @var \Drupal\user\Entity\User $user */
 $user = \Drupal\user\Entity\User::load($uid);
```

```
 $user->block();
 $user->save();
}

$total = count($results);
$output->writeln("Disabled $total users");
}
```

10. The result is an array of user IDs. We loop over them to load the user, mark them as disabled, and then save them to commit the changes.

11. Enable the module in order to access the command:

```
$ drupal module:install console_commands
```

12. Run your command:

```
$ drupal console_commands:disable_users
Disabled 1 users
```

## How it works...

Console provides integration with modules using namespace discovery methods. When Console is run in a Drupal installation, it will discover all the available projects. It then discovers any files in the \Drupal\{ a module }\Command namespace that implement \Drupal\Console\Command\Command.

Console will rescan the Drupal directory for available commands every time it is invoked, as it does not keep a cache of available commands.

## There's more...

Drupal Console provides a much more intuitive developer experience, as it follows Drupal core's coding formats. We will touch on how Console can be used to create entities.

### Using a Console command to create entities

A benefit of Console is its ability to utilize Symfony Console's question helpers for a robust interactive experience. Drupal Commerce utilizes Console to provide a commerce:create:store command to generate stores. The purpose of the command is to simplify the creation of a specific entity.

The \Drupal\commerce\_store\Command\CreateStoreCommand class overrides the default interact method that is executed to prompt data from the user. It will prompt users to enter the store's name, e-mail, country, and currency.

Developers can implement similar commands to give advanced users a simpler way to work with modules and configuration.

## See also

- ▶ Refer to how to create custom commands at <https://hechoendrupal.gitbooks.io/drupal-console/content/en/extending/creating-custom-commands.html>
- ▶ Refer to how to installing Drupal Console at <https://hechoendrupal.gitbooks.io/drupal-console/content/en/getting/installer.html>

# Module 3

## **Drupal 8 Theming with Twig**

Master Drupal 8's new Twig templating engine to create fun and fast websites  
with simple steps to help you move from concept to completion



# 1

## Setting Up Our Development Environment

Regardless of you being a seasoned web developer or someone who is just about to start learning Drupal, there are few things that everybody needs to have in place before we can get started:

- First, is to make sure that we have an Application stack that will meet Drupal 8's system requirements. MAMP provides us with a standalone web server that is generally referred to as an AMP (Apache, MySQL, PHP) stack and is available for both OS X and Windows. We will look at installing and configuring this local web server in preparation to install Drupal 8.
- Second, is to set up a Drupal 8 instance and learn the process of installing Drupal instances into our AMP stack. There are a few changes on how the configuration and installation processes work in Drupal 8, so we will take a closer look to ensure that we all begin development from the same starting point.
- Third, we will be reviewing the Admin interface, including the new responsive Admin menu and any configuration changes that have been made as we navigate to familiar sections of our site. We will also look at how to extend our website using contributed modules, review changes to the files and folder structure that make up Drupal 8, and discuss best practices to manage your files.
- Finally, we will review the exercise files that we will be using throughout the series, including how to download and extract files, how to use phpMyAdmin—a database administration tool to back up and restore database snapshots—and how to inspect elements within our HTML structure using Google Chrome.

Let's get started by installing our web environment that we will be using as we take an exciting look at Drupal 8 theming with Twig.

## Installing an AMP (Apache, MySQL, PHP) stack

To install and run Drupal 8, our server environment must meet and pass certain requirements. These requirements include a web server (Apache, NGINX, or Microsoft IIS) that can process server-side languages such as PHP, which Drupal 8 is built on.

Our server should also contain a database that can manage the data and content that Drupal 8's content management system will store and process. The preferred database is MySQL. However, Drupal 8 can also support PostgreSQL along with Microsoft SQL Server and Oracle with an additional module support.

Finally, Drupal 8 requires PHP 5.5.9 or later, with the CURL extension.

However, because this module is not meant to be a "How-to" on installing and configuring Apache, MySQL, or PHP, we will take all the guesswork and trial by fire out of the equation and instead turn to MAMP.

## Introducing MAMP

MAMP can be found at <https://www.mamp.info/en> and is a tool that allows us to create Drupal sites locally without the need or knowledge of installing and configuring Apache, MySQL, or PHP on a specific platform.

The application stack will consist of the following:

- Apache: The world's most popular web server
- MySQL: The world's most popular database server
- PHP: Various versions of PHP
- phpMyAdmin: A tool to manage MySQL databases via your browser

## Downloading MAMP

Let's begin with the steps involved in quickly downloading, installing, and configuring our very own AMP stack along with an initial instance of Drupal that we will be using throughout the rest of this module. Begin by opening up our web browser and navigating to <https://www.mamp.info/en/downloads> and selecting either Mac OS X or Windows and then clicking on the **Download** button, as shown in the following image:



MAMP will allow us to install a local web server on either Mac or Windows and provides us with all the tools we will need to develop most open source websites and applications including Drupal 8.

## Installing MAMP

Once the download has completed, we will need to locate the `.dmg` (Mac users) or `.exe` (Windows users) installation file and double-click on it to begin the installation process. Once the executable is opened, we will be presented with a splash screen that will guide us through the process of installing and configuring MAMP.

Clicking on the **Continue** button located on the **Introduction** pane, will take us to the **Read Me** information. MAMP will notify us that two folders will be created: one for MAMP and the other for MAMP PRO. It is important to not move or rename these two folders.

Click on the **Continue** button, which will take us to the license information. Simply accept the terms of the license agreement by clicking on **Continue** and then on **Agree** when prompted.

We can finally click on the **Install** button to complete the installation process. Depending on the operating system, we may need to enter our credentials for MAMP to be able to continue and configure our local web server. Once the install has completed, we can click on the **Close** button.

## A quick tour of MAMP PRO

Let's begin by opening up MAMP and taking a quick tour of the various settings and how we can go about using our local web server to install and configure a new Drupal 8 instance.

When we first open up MAMP, we will be prompted to launch either MAMP or MAMP PRO, as shown in the following image:



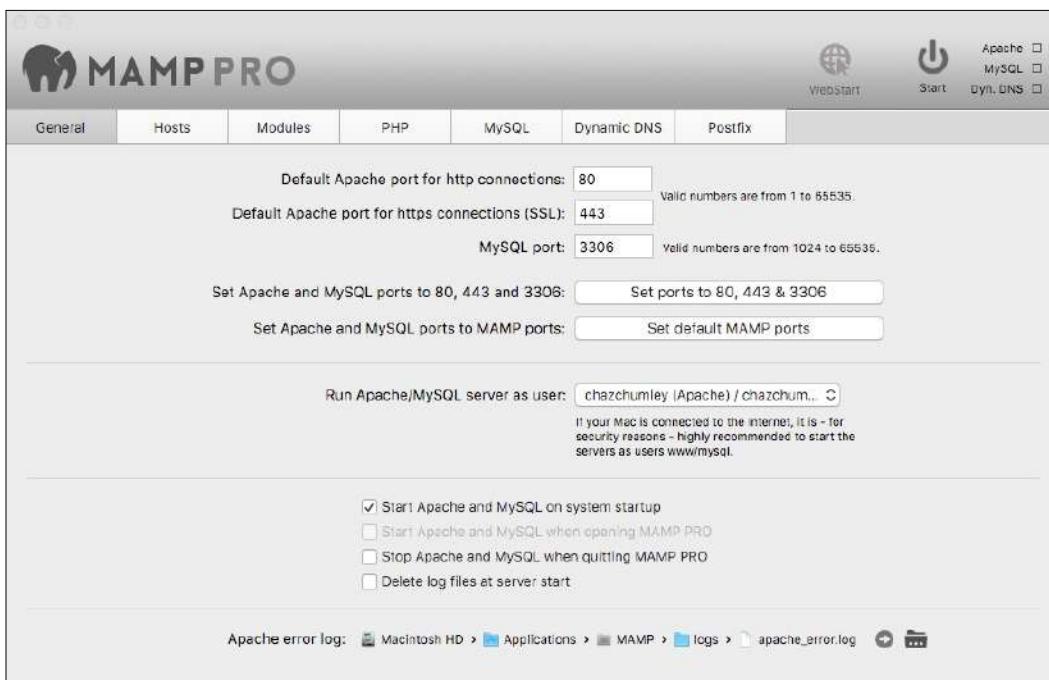
While MAMP is the free version of the local web server, it is strongly recommended that we use MAMP PRO for configuration and easy setup of a website. We can continue by clicking on **Launch MAMP PRO**, which will prompt us one more time to accept the **Initialization** of the remaining components that MAMP PRO needs before we can begin using it. Now, click on **OK** to continue.

We can use MAMP PRO free for 14 days and at the end of that period, we can decide whether to purchase a license or continue using the free version. Click on **OK** to continue.

## General settings for MAMP PRO

If this is the first time you're using MAMP PRO, then there is some quick housekeeping we will want to take care of, beginning with the general settings. MAMP PRO tries to make sure that it does not interfere with any other possible web servers we may be running by setting the default ports of Apache to 8888 and MySQL to 8889. Although this is nice, the recommendation is to click on the **Set ports to 80, 443 & 3306** button that will make sure that MAMP PRO is running on more standardized ports for web development.

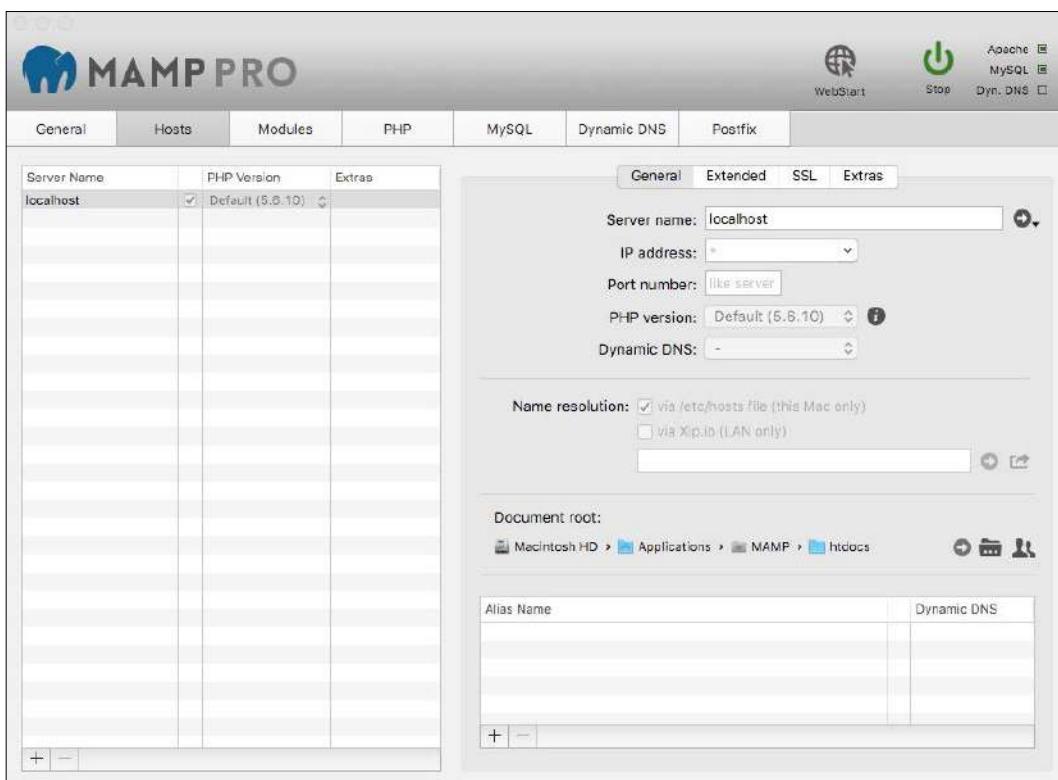
If we want to make sure that Apache or MySQL are active at all time, we will also check **Start Apache and MySQL on system startup** and uncheck **Stop Apache and MySQL when quitting MAMP PRO**. Once we have made these changes, we can click on the **Save** button. Our changes should now be applied as shown in the following image and MAMP PRO will now prompt us to **Start servers now**.



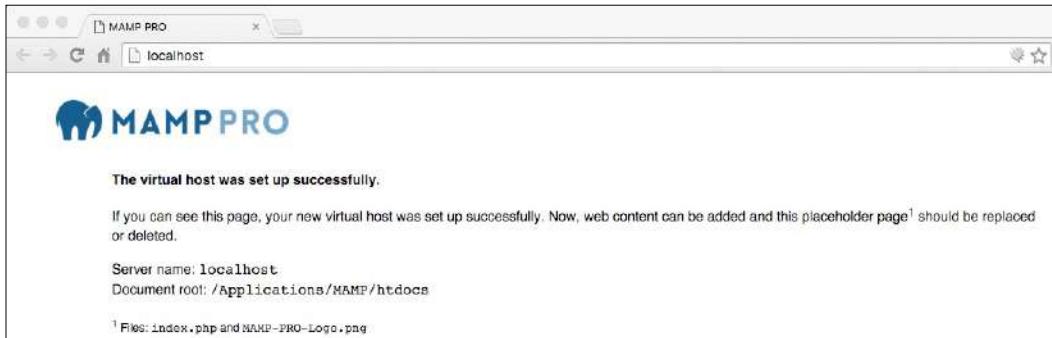
## Host settings

The next tab we will look at is the **Hosts** tab, which is where we will create and configure basic websites. By default, MAMP PRO creates a localhost entry for us, which is common when developing a web application.

We will be using the **Hosts** tab to create an additional website when we install Drupal 8, so let's take a moment to locate some of the common settings we will need to know. Take a look at the following image; we can see that **localhost** is the **Server name** of our default website, uses the default **PHP version** of **5.6.10**, and has a **Document root** pointing to the **htdocs** folder of our MAMP installation.



Another nice ability of MAMP PRO is to be able to click on the arrow icon located to the right of the **Server name** and have our default web browser open up to the localhost page, as shown in the following image:



It is important to point out that the **Server name** always equates to the name of the URL in our browser that displays our website.

MAMP PRO is quite a robust and powerful local web server and while there are many more configuration options and settings that we could spend time looking through, most of our time will be spent on working from the **Hosts** tab creating new websites or configure existing sites.

So far, MAMP PRO has configured everything for us, but how to create a new website and, in more general, install a Drupal 8 instance? Let's look at it in the following section.

## Installing Drupal 8

In order to install Drupal 8 within our local MAMP PRO server, we will need to perform a series of steps:

1. We will need to grab a copy of the latest Drupal 8 release and extract the files to a location on our computer that will be the document root of our website.
2. We will have to create a new host entry with the server name that we will want to use for our URL and point our host entry to the proper document root containing our Drupal 8 instance.
3. We will have to create a MySQL database that we can point Drupal to during the installation process.

We will walk through each of these steps in detail to ensure that we all have a copy of Drupal 8 installed properly that we will build upon as we work through each lesson.

## Downloading Drupal 8

Drupal.org is the authority on everything about Drupal. We will often find ourselves navigating to Drupal.org to learn more about the community, look for documentation, post questions within the support forum, or review contributed modules or themes that can help us extend Drupal's functionality. Drupal.org is also the place where we can locate and download the latest release of Drupal 8.

We can begin by navigating to <https://www.drupal.org/node/2627402> and locate the latest release of Drupal 8. Click on the compressed version of Drupal 8 that we prefer, which will begin downloading the files to our computer. Once we have a copy of Drupal 8 on our computer, we will want to extract the contents to a location where we can easily work with Drupal and its folders and files.

## Creating our document root

A document root is the main folder that our host entry will point to. In the case of Drupal, this will be the extracted root folder of Drupal itself. Generally, it is a best practice to maintain some sort of folder structure that is easy to manage and that can contain multiple websites.

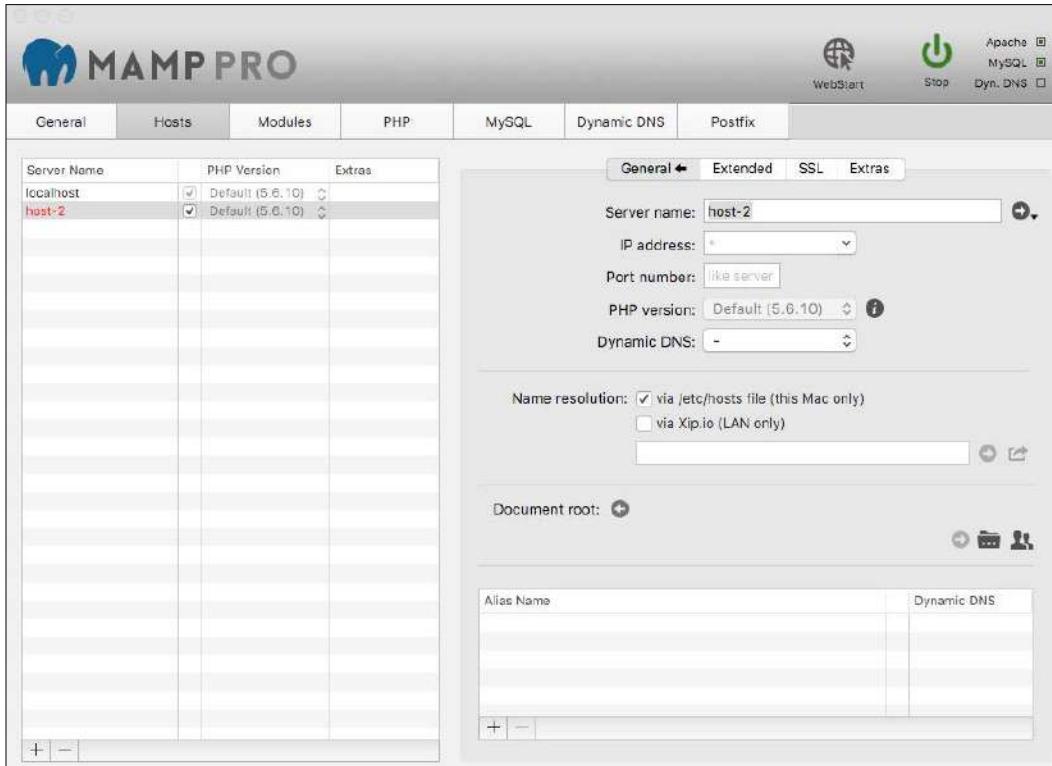
For the sake of demonstration, we will create a `sites` folder and then copy the compressed Drupal files to our new folder and extract the contents, as shown in the following image:

| ▼ | Sites             |                  |                                           |
|---|-------------------|------------------|-------------------------------------------|
| ▼ | drupal-8.0.1      |                  |                                           |
| ► | core              | Today, 9:11 PM   | Folder                                    |
| ► | modules           | 12/2/15, 9:43 AM | Folder                                    |
| ► | profiles          | 12/2/15, 9:08 AM | Folder                                    |
| ► | sites             | 12/2/15, 9:08 AM | Folder                                    |
| ► | themes            | 12/2/15, 9:08 AM | Folder                                    |
| ► | vendor            | 12/2/15, 9:08 AM | Folder                                    |
| ▀ | autoload.php      | 12/2/15, 9:08 AM | PHP: Hypertext...(PHP) document 262 bytes |
| ▀ | composer.json     | 12/2/15, 9:08 AM | Plain Text File 1.2 KB                    |
| ▀ | composer.lock     | 12/2/15, 9:08 AM | Document 132.5 KB                         |
| ▀ | example.gitignore | 12/2/15, 9:08 AM | Document 1.1 KB                           |
| ▀ | index.php         | 12/2/15, 9:08 AM | PHP: Hypertext...(PHP) document 549 bytes |
| ▀ | LICENSE.txt       | 9/23/14, 3:24 PM | Plain Text Document 18 KB                 |
| ▀ | README.txt        | 12/2/15, 9:08 AM | Plain Text Document 5.8 KB                |
| ▀ | robots.txt        | 12/2/15, 9:08 AM | Plain Text Document 1.2 KB                |
| ▀ | update.php        | 12/2/15, 9:08 AM | PHP: Hypertext...(PHP) document 547 bytes |
| ▀ | web.config        | 12/2/15, 9:08 AM | Document 3.9 KB                           |

## Creating our host entry

A host entry represents our website, which, in this case, is our Drupal 8 instance. Hosts always contain a server name that equates to the URL we will use to navigate to Drupal within our browser.

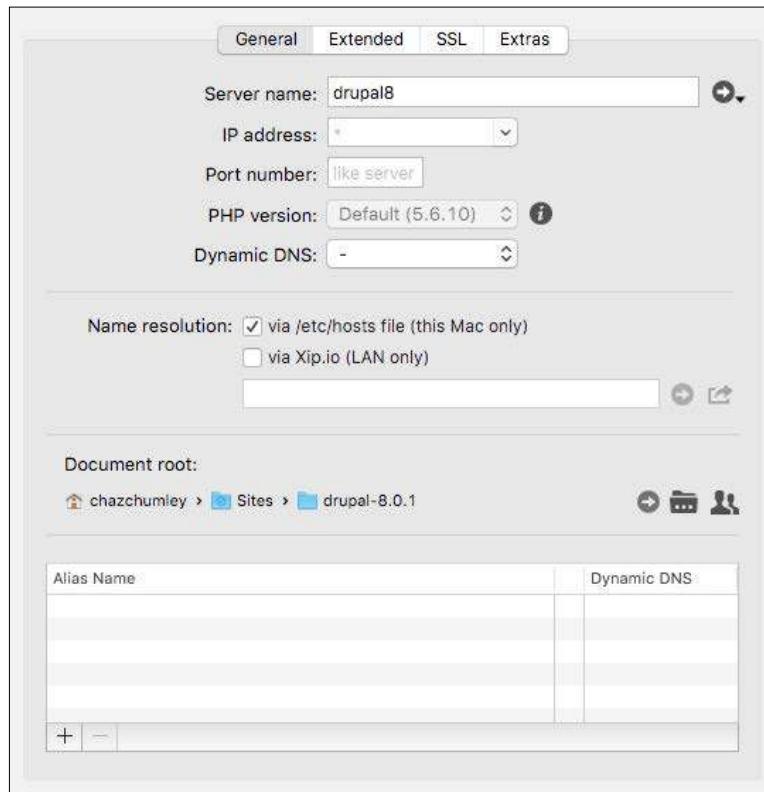
Begin by opening MAMP PRO and clicking on the **Hosts** tab. To add a new host entry, we can click on the plus icon at the bottom of the **Server Name** column, as shown in the following image:



By default, this will add a new host entry that will require us to configure with three very important pieces of information:

- We will have to change the **Server name** from the default to `drupal8`.
- Then, we will want to verify that the required version of PHP is being used; in our case, the default of **5.6.10** will work just fine.
- Finally, we will need to click on the folder icon within the **Document root** section and choose our Drupal 8 folder that we placed within our `sites` folder earlier from the **Please select a Document Root folder** dialog.

The **General** settings for our new host entry should look as shown in the following image:



We can now apply our changes by clicking on the **Save** button and then clicking on the **Yes** button when prompted to have MAMP PRO restart the servers.

## Creating a new database for Drupal

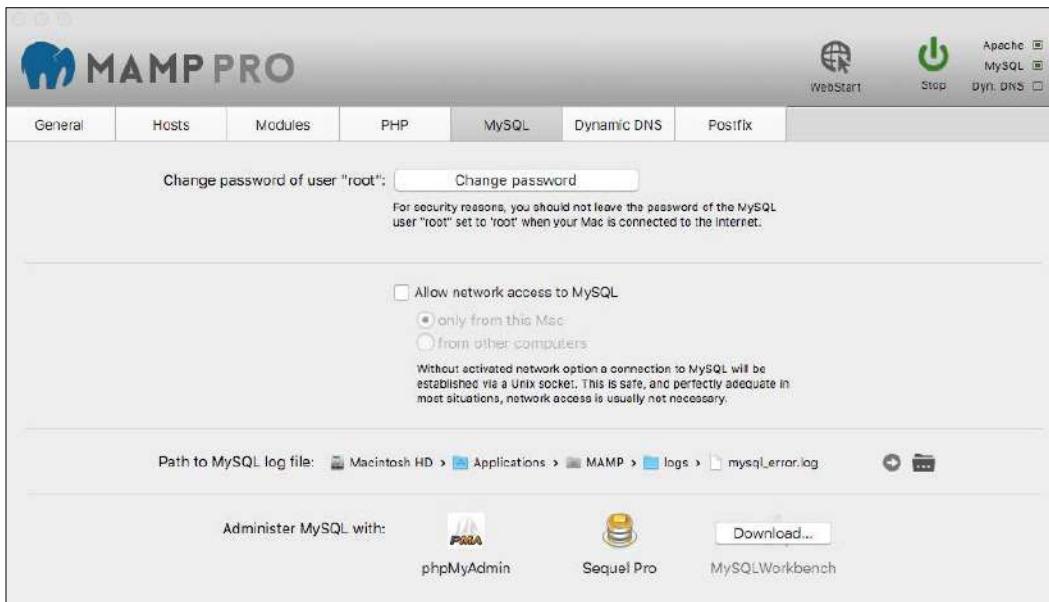
Drupal 8 requires a database available to install any tables that make up the content management system. These tables will hold configuration data, users and permissions, content, and any extendable functionality that makes Drupal 8 so powerful.

Lucky for us, MAMP PRO installs a MySQL database server that we can take advantage of to create a new database that Drupal 8 can point to. This same database server we will also be working with to back up and restore our database content as we progress through each lesson.

## Using phpMyAdmin

MAMP PRO also installs a free software tool written in PHP for the sole purpose of managing MySQL databases. phpMyAdmin allows us to perform a multitude of tasks from browsing tables, views, fields, and indexes to exporting and importing database backups and much more.

If we switch back over to MAMP PRO, we can locate the **MySQL** tab and click on the **phpMyAdmin** link located in the **Administer MySQL with** region, as shown in the following image:



## *Setting Up Our Development Environment*

---

We should now be presented with **phpMyAdmin** within our browser. Currently, we are interested in creating a new database. We will revisit phpMyAdmin a little later to learn how to back up and restore our database. The following are the steps to create a database:

1. Begin by clicking on the **New** link in the left sidebar, as shown in the following image:

The screenshot shows the phpMyAdmin interface with the title bar "Server: localhost:3306". The left sidebar has a "New" link under the "Recent" section. The main panel is titled "General Settings" and includes sections for "Appearance Settings" (Language: English, Theme: Original, Font size: 82%) and "Database server" (Server: Localhost via UNIX socket, Server type: MySQL, Server version: 5.5.42 - Source distribution, Protocol version: 10, User: root@localhost, Server charset: UTF-8 Unicode (utf8)). It also shows "Web server" information (Apache, Database client version: libmysql - mysqlnd 5.0.11-dev - 20120503 - \$Id: 3c688b5bbc30d36a13ac34fd4b7b5b787fe651\$, PHP extension: mysqli, PHP version: 5.6.10) and the "phpMyAdmin" logo.

2. Next, we will want to enter a name of **drupal8** within the **Create database** field, as shown in the following image, and then click on the **Create** button.

The screenshot shows the "Databases" page. At the top, there is a "Create database" input field containing "drupal8". Below it is a note: "Note: Enabling the database statistics here might cause heavy traffic between the web server and the MySQL server." A table lists existing databases: "information\_schema" (Collation: utf8\_general\_ci), "mysql" (Collation: latin1\_swedish\_ci), and "performance\_schema" (Collation: utf8\_general\_ci). A total of 3 databases are listed. At the bottom, there are buttons for "Check All", "With selected:", "Drop", and a link to "Enable Statistics".

We have now created our first MySQL database, which we will use when configuring Drupal 8 in the next step.

## Completing Drupal 8 installation

Now that we have all of our basic requirements completed, we can open up our favorite web browser and navigate to <http://drupal8/core/install.php> to begin the installation process.

Since this may be the first time installing Drupal 8, one thing we will notice is that the install screen looks a little different. The install screen has been given a makeover, but the steps are similar to that of Drupal 7, starting with choosing a language.

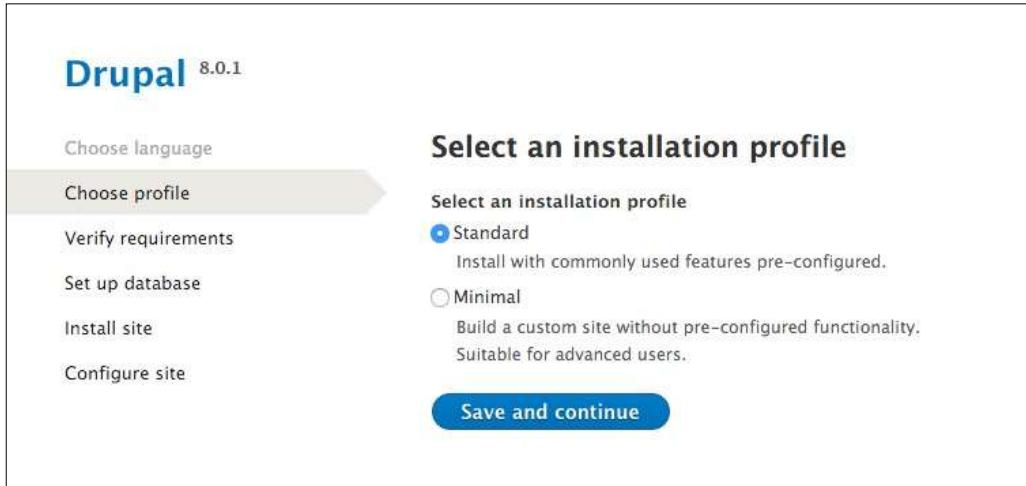
### Choosing a language

The installation process will prompt us to choose a language that we want Drupal 8 to be installed in. This language will control how the Admin area appears, and in many cases, the default of English is acceptable. We will need to click on the **Save and continue** button to proceed to the next step, as shown in the following image:



## Choosing a profile

Our next step is to choose an installation profile. We can think of this as Drupal's way of preconfiguring items for us that will make our job easier when developing. By default, we can leave the **Standard** profile selected, but if we choose to configure Drupal ourselves, we can always choose **Minimal**. Click on the **Save and continue** button to proceed to the next step, as shown in the following image:



## Verifying requirements

The next screen allows us to review any requirements that Drupal needs or recommends for optimal performance. From here, we can see web server information, PHP version, memory limits, and more:

The screenshot shows the 'Requirements review' step of the Drupal 8.0.1 installation process. On the left, a sidebar lists steps: 'Choose language', 'Choose profile', 'Verify requirements' (which is highlighted in grey), 'Set up database', 'Install site', and 'Configure site'. The main area is titled 'Requirements review' and contains a table of system requirements:

| Web server                                               | Apache                                                                                                                                                        |
|----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PHP                                                      | 5.6.10                                                                                                                                                        |
| PHP extensions                                           | Enabled                                                                                                                                                       |
| <span style="color: orange;">⚠</span> PHP Opcode caching | Not enabled<br>PHP Opcode caching can improve your site's performance considerably. It is <b>highly recommended</b> to have OPCache installed on your server. |
| Database support                                         | Enabled                                                                                                                                                       |
| PHP memory limit                                         | 128M                                                                                                                                                          |
| File system                                              | Writable ( <i>public</i> download method)                                                                                                                     |
| Unicode library                                          | PHP Mbstring Extension                                                                                                                                        |
| Settings file                                            | The <code>./sites/default/settings.php</code> exists.                                                                                                         |
| Settings file                                            | The Settings file is writable.                                                                                                                                |

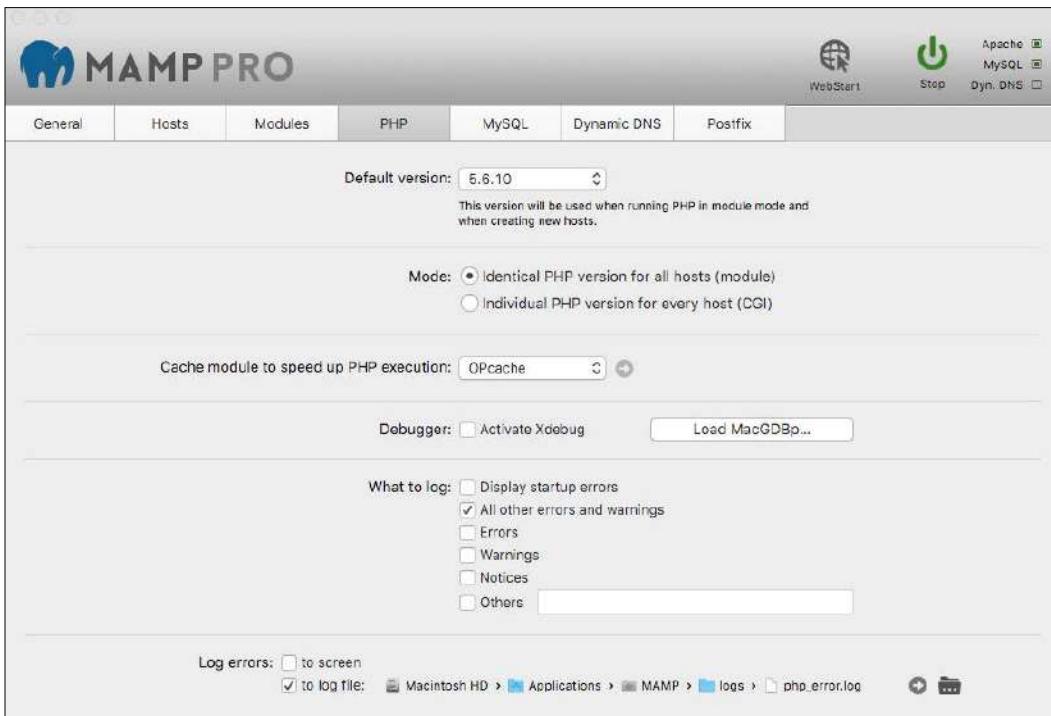
At the bottom, a message says: 'Check the messages and [retry](#), or you may choose to continue [anyway](#)'.

The requirements review can also alert us to any configuration settings that will allow Drupal to perform better. In our example, we forgot to enable OPCode caching, which allows PHP to compile down to bytecode. Without going into the details of caching, we can easily enable this feature in MAMP PRO.

## *Setting Up Our Development Environment*

---

Begin by opening up the MAMP PRO console and clicking on the **PHP** tab. Next, we will want to select **OPcache** from the **Cache module to speed up PHP execution** dropdown, as shown in the following image:



Click on the **Save** button and then allow MAMP PRO to restart servers if prompted. Now, we can refresh our Drupal install in the browser, and we will be taken to the next step in the installation process.

## **Database configuration**

Database configuration can sometimes be a tricky part of installing Drupal for the first time. This is generally due to selecting the incorrect database type, wrong database name, or password, or by not specifying the correct host or port number.

The screenshot shows the 'Database configuration' step of the Drupal 8.0.1 setup process. On the left, a sidebar lists steps: 'Choose language', 'Choose profile', 'Verify requirements', 'Set up database' (which is highlighted with a grey background), 'Install site', and 'Configure site'. The main content area is titled 'Database configuration'. It contains the following form fields:

- Database type \***: A radio button group where 'MySQL, MariaDB, Percona Server, or equivalent' is selected.
- Database name \***: An input field containing 'drupal8'.
- Database username \***: An input field containing 'root'.
- Database password**: An input field containing '\*\*\*\*\*'.

Below these fields is a link '► ADVANCED OPTIONS' and a blue 'Save and continue' button at the bottom.

The settings we will want to use are as follows:

- **Database type**: Leave the default of MySQL selected.
- **Database name**: This is the name of the database that was created upon import. In our case, it should be drupal8.
- **Database username**: root.
- **Database password**: root.

With these settings, we can click on the **Save and continue** button to proceed.

If this is successful, we can see the **Installing Drupal** screen and watch as Drupal installs the various modules and configurations. This process may take a few minutes. If this process fails in any way, please go back and review the previous steps to make sure that they match what we have used.

## Configuring the site

Before we can wrap up our Drupal 8 installation, we need to configure our site by inputting various settings for site information, site maintenance account, regional settings, and update notifications. Let's proceed now by entering our **Site Information**.

The screenshot shows the 'Configure site' page of a Drupal 8.0.1 installation. On the left, a sidebar lists steps: 'Choose language', 'Choose profile', 'Verify requirements', 'Set up database', 'Install site', and 'Configure site'. The 'Configure site' step is highlighted with a grey arrow pointing right. The main content area is titled 'Configure site' and contains two sections: 'SITE INFORMATION' and 'SITE MAINTENANCE ACCOUNT'. In the 'SITE INFORMATION' section, the 'Site name \*' field is filled with 'drupal8'. Below it, the 'Site email address \*' field is empty. A note explains that automated emails like registration info will be sent from this address, using a domain to prevent spam. In the 'SITE MAINTENANCE ACCOUNT' section, the 'Username \*' field is filled with 'admin'. A note specifies allowed characters: space, period, hyphen, apostrophe, underscore, and '@'. The 'Password \*' field contains '\*\*\*\*\*' and is marked as 'Weak'. The 'Confirm password \*' field also contains '\*\*\*\*\*'. A note at the bottom states 'Passwords match: yes'.

Drupal 8.0.1

Choose language

Choose profile

Verify requirements

Set up database

Install site

Configure site

**Configure site**

**SITE INFORMATION**

**Site name \***  
drupal8

**Site email address \***  
Automated emails, such as registration information, will be sent from this address. Use an address ending in your site's domain to help prevent these emails from being flagged as spam.

**SITE MAINTENANCE ACCOUNT**

**Username \***  
admin  
Several special characters are allowed, including space, period (.), hyphen (-), apostrophe ('), underscore (\_), and the @ sign.

**Password \***  
\*\*\*\*\*  
Password strength: Weak

**Confirm password \***  
\*\*\*\*\*  
Passwords match: yes

The site information consists of the following:

- **Site name:** To be consistent, let's call our site drupal8. We may give our new site any name that we like. As usual, we can change the site name later from the Drupal admin.
- **Site e-mail address:** Enter an e-mail address that we will want to use for automated e-mails such as registration information. It is the best practice to use an e-mail that ends in our site's domain to prevent e-mails from being flagged as spam.

Next, we will want to set up the site maintenance account. This is the primary account used to manage Drupal to perform such tasks as updating the core instance, module updates, and any development that needs user 1 permissions.

## Site maintenance account

The site maintenance account information consists of the following:

- **Username:** Because we are developing a demo site, it makes sense to generally use admin for the username. Feel free to choose whatever is easy to remember, but don't forget it.
- **Password:** Security sticklers will ask to create something strong and unique, and Drupal displays a visual interface to let us know how strong our password is. For the sake of demonstration, we will use admin as our password so that your username and password match and are easy to remember.
- **E-mail address:** Generally, using the same e-mail that is used for the site e-mail address makes for consistency but is not required as we can choose any e-mail that we don't mind receiving security and module update notices.

## Regional settings

Regional settings consist of default country and default time zone. These are often neglected and left with their defaults. The defaults are not recommended, as they are important in the development and design of Drupal 8 websites, specifically, when it comes to dates and how they are used to capture data and display dates back to the end user.

For our specific installation, choose the country and time zone for our region.

## Update notifications

At last, we have come to our final set of configurations. Update notifications should always be left-checked unless we have no reason to receive security updates to Drupal core or module updates. By default, they should be checked. Click on **Save and continue** to finalize the configuration and installation of Drupal 8.

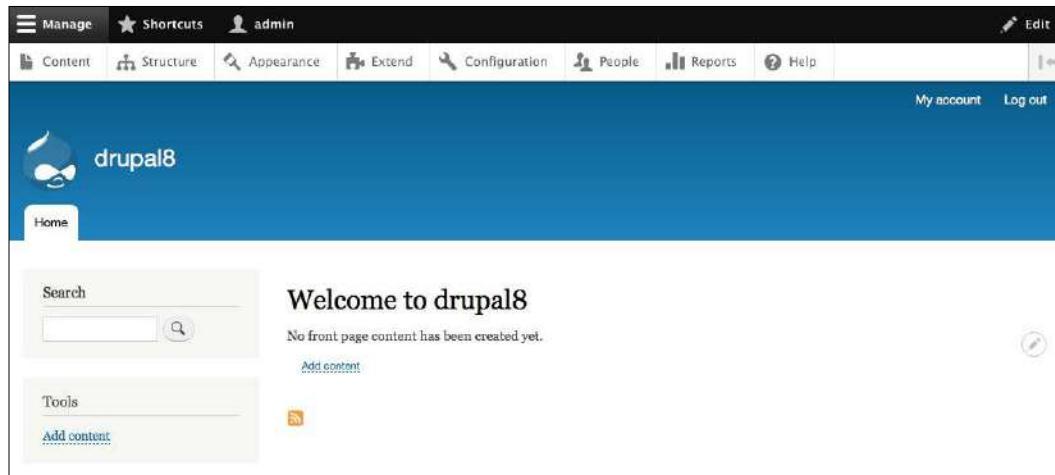
Drupal installation is now complete, and we should see the home page of our new website. Say "hello" to Drupal 8.

## Reviewing the new admin interface

Our local instance of Drupal 8 is similar to Drupal 7 when first viewing the site. You will note the default Bartik theme with the friendly Drupal drop logo and a tabbed main menu. It is here though where the similarities stop. Drupal 8 has been reworked from the ground up, including a brand new responsive layout and admin menu.

## Exploring the admin menu

One of the nice new features of Drupal 8 is the rebuilt admin menu. Everything has been moved under the **Manage** menu item. The admin menu itself is responsive and will change from text and icon to icon only, as soon as the browser is resized to tablet screen size.



The admin menu can also be pinned to the left side of the window by clicking on the arrow icon to the right of the **Help** menu item.



The flexibility of the new admin menu enables the admin user to manage Drupal 8 websites from the browser or a tablet or a smartphone very easily.

## Previewing the interface

Taking a closer look at the menu items contained in our admin menu, we begin to see some differences in how things are named and may wonder where to find once familiar settings and configurations. Let's quickly walk through these menu items now so that it is easier to find things as we progress later on in future chapters:

- **Content:** This section displays any user-generated content, comments, and files with the ability to filter by **Published status**, **Type**, **Title**, and **Language**. The display for content is also now a view and can be customized with additional fields and filters as needed.

A screenshot of the 'Content' administration view. The top navigation bar includes 'Back to site', 'Manage', 'Shortcuts', and the user 'admin'. Below the navigation is a horizontal menu with links for Content, Structure, Appearance, Extend, Configuration, People, Reports, and Help. The main content area is titled 'Content' with a star icon. It features three tabs: 'Content', 'Comments', and 'Files'. Below the tabs is a breadcrumb trail: 'Home &gt; Administration'. A large blue button labeled '+ Add content' is visible. Underneath are four dropdown filters: 'Published status' (set to '- Any -'), 'Type' (set to '- Any -'), 'Title' (an empty input field), and 'Language' (set to '- Any -'). A 'Filter' button is located below the filters. A table header follows with columns: TITLE, CONTENT TYPE, AUTHOR, STATUS, UPDATED, and OPERATIONS. The table body contains the message 'No content available.'.

## *Setting Up Our Development Environment*

---

- **Structure:** This section is to manage **Block layout, Comment types, Contact forms, Content types, Display modes, Menus, Taxonomy, and Views**. We will explore some of the changes and new functionality contained within this section later on in the module.

The screenshot shows the Drupal 8 administration interface. The top navigation bar includes links for Back to site, Manage, Shortcuts, and the current user (admin). Below the bar, a horizontal menu bar contains links for Content, Structure (which is highlighted), Appearance, Extend, Configuration, People, Reports, and Help. The main content area is titled "Structure" with a star icon. It displays a list of configuration items:

- Block layout**: Configure what block content appears in your site's sidebars and other regions.
- Comment types**: Manage form and displays settings of comments.
- Contact forms**: Create and manage contact forms.
- Content types**: Create content types and manage their default settings.
- Display modes**: Configure what displays are available for your content and forms.
- Menus**: Manage menus and menu links.
- Taxonomy**: Manage tagging, categorization, and classification of your content.
- Views**: Manage customized lists of content.

- **Appearance:** This section is to enable, disable, and configure default themes as well as administrative themes.

The screenshot shows the 'Appearance' page in the Drupal 8 administration interface. The top navigation bar includes links for 'Back to site', 'Manage', 'Shortcuts', and 'admin'. Below the navigation is a horizontal menu with links for 'Content', 'Structure', 'Appearance' (which is highlighted in blue), 'Extend', 'Configuration', 'People', 'Reports', and 'Help'. The main content area has a title 'Appearance' with a star icon. Below the title are three tabs: 'List', 'Update', and 'Settings'. Underneath the tabs, the breadcrumb navigation shows 'Home > Administration'. A message states 'Set and configure the default theme for your website. Alternative themes are available.' Below this, another message says 'You can place blocks for each theme on the block layout page.' A blue button labeled '+ Install new theme' is visible. The section titled 'Installed themes' displays the 'Bartik 8.0.1 (default theme)' with a preview image and the description: 'A flexible, recolorable theme with many regions and a responsive, mobile-first layout.' There is also a 'Settings' link.

- **Extend:** Formerly known as Modules, this section is for listing, updating, and uninstalling core, custom, and contributed modules. New is the ability to search for modules using a filter. Various contributed modules have been moved into core, including Views and CKEditor.

The screenshot shows the 'Extend' page in the Drupal 8 administration interface. The top navigation bar includes links for 'Back to site', 'Manage', 'Shortcuts', and 'admin'. Below the navigation is a horizontal menu with links for 'Content', 'Structure', 'Appearance', 'Extend' (which is highlighted in blue), 'Configuration', 'People', 'Reports', and 'Help'. The main content area has a title 'Extend' with a star icon. Below the title are three tabs: 'List', 'Update', and 'Uninstall'. Underneath the tabs, the breadcrumb navigation shows 'Home > Administration'. A message states 'Download additional contributed modules to extend your site's functionality.' Another message says 'Regularly review and install available updates to maintain a secure and current site. Always run the update script each time a module is updated.' A blue button labeled '+ Install new module' is visible. To the left of the main content area is a sidebar with a heading 'CORE' followed by a list of experimental modules: 'CORE (EXPERIMENTAL)', 'FIELD TYPES', 'MULTILINGUAL', and 'WEB SERVICES'. At the bottom of the sidebar is an 'Install' button. On the right side of the main content area is a search bar labeled 'Filter by name or description' and a text input field for entering module names.

## *Setting Up Our Development Environment*

---

- **Configuration:** This section is designed to configure both core and contributed modules. Each area is grouped into functional sections and allows us to manage site information to file system to performance tuning.

The screenshot shows the Drupal Administration interface with the 'Configuration' tab selected. The top navigation bar includes links for Back to site, Manage, Shortcuts, and the current user (admin). Below the navigation is a breadcrumb trail: Home > Administration > Configuration. A prominent red error message box states: "One or more problems were detected with your Drupal installation. Check the [status report](#) for more information." The main content area is divided into several sections:

- PEOPLE**: Contains the **Account settings** link, which allows configuring default user account settings, including fields, registration requirements, and email messages.
- SYSTEM**: Contains the **Site information** link, which allows changing site name, email address, slogan, default front page, and error pages.
- CONTENT AUTHORING**: Contains the **Text formats and editors** link, which allows selecting and configuring text editors and how content is filtered when displayed.
- Cron**: Manages automatic site maintenance tasks.
- USER INTERFACE**: Contains the **Shortcuts** link.

- **People:** This section allows us to manage users, permissions, and roles. The display for users is now a View as well and can be customized to add additional fields and filters as needed.

The screenshot shows the 'People' administration page in Drupal. At the top, there are tabs for 'List', 'Permissions', and 'Roles'. Below the tabs, there is a search/filter section with fields for 'Name or email contains', 'Role', 'Permission', and 'Status'. A 'Filter' button is present. Below the filter section, there is a dropdown menu titled 'With selection' with the option 'Add the Administrator role to the selected users'. An 'Apply' button is located next to the filter section. The main table lists users with the following columns: USERNAME, STATUS, ROLES, MEMBER FOR, LAST ACCESS, and OPERATIONS. One user, 'admin', is listed with the role 'Administrator'. The 'Edit' button is visible for the 'admin' user. There is also an 'Apply' button at the bottom of the table.

- **Reports:** This section is designed to view available updates, recent log messages, field lists, status reports, top "access denied" errors, top "page not found" errors, top search phrases, and View plugins.

The screenshot shows the 'Reports' administration page in Drupal. At the top, there are tabs for 'Content', 'Structure', 'Appearance', 'Extend', 'Configuration', 'People', 'Reports', 'Help', and a search bar. Below the tabs, there is a list of report types:
 

- Available updates**: Get a status report about available updates for your installed modules and themes.
- Recent log messages**: View events that have recently been logged.
- Field list**: Overview of fields on all entity types.
- Status report**: Get a status report about your site's operation and any detected problems.
- Top 'access denied' errors**: View 'access denied' errors (403s).
- Top 'page not found' errors**: View 'page not found' errors (404s).
- Top search phrases**: View most popular search phrases.
- Views plugins**: Overview of plugins used in all views.

- **Help:** This section is designed to obtain helpful information on functionality necessary to know in administering a Drupal 8 website. This includes a **Getting Started** section and help topics on items, such as **Block**, **Views**, **User**, and more.

The screenshot shows the Drupal 8 administration interface with the 'Help' tab selected. The main content area is titled 'Getting Started' and contains steps for setting up a website. Below this is a 'Help topics' section listing various Drupal components. The topics are organized into four columns:

| Automated Cron        | Database Logging            | Link       | Text           |
|-----------------------|-----------------------------|------------|----------------|
| Block                 | Datetime                    | Menu UI    | Text Editor    |
| Breakpoint            | Field                       | Node       | Toolbar        |
| CKEditor              | Field UI                    | Options    | Tour           |
| Color                 | File                        | Path       | Update Manager |
| Comment               | Filter                      | Quick Edit | User           |
| Configuration Manager | Help                        | RDF        | Views          |
| Contact               | History                     | Search     | Views UI       |
| Contextual Links      | Image                       | Shortcut   |                |
| Custom Block          | Internal Dynamic Page Cache | System     |                |
| Custom Menu Links     | Internal Page Cache         | Taxonomy   |                |

## Exploring Drupal 8 folder structure

There are several changes to Drupal 8 with regard to how files and folders are structured. Let's walk through the `core`, `modules`, `sites`, and `themes` folders and discuss some best practices for how each of these folders should be managed when creating a Drupal 8 website.

### The core folder

One of the first things to point out is that the files and folder structure of Drupal 8 have changed from its predecessor Drupal 7. The first change is that everything that Drupal 8 needs to run is contained within the new `core` folder. No longer is there any confusion of having the `modules` and `themes` folders contained within a `sites` folder and having to ask "did I place my files in the correct location?".

| Name               | Date Modified    | Size      | Kind                |
|--------------------|------------------|-----------|---------------------|
| core               | 12/2/15, 9:08 AM | --        | Folder              |
| assets             | 12/2/15, 9:08 AM | --        | Folder              |
| authorize.php      | 12/2/15, 9:08 AM | 7 KB      | PHP                 |
| CHANGELOG.txt      | 12/2/15, 9:08 AM | 64 KB     | Plain Text Document |
| composer.json      | 12/2/15, 9:08 AM | 6 KB      | JSON                |
| composer.txt       | 12/2/15, 9:08 AM | 886 bytes | Plain Text Document |
| config             | 12/2/15, 9:08 AM | --        | Folder              |
| COPYRIGHT.txt      | 12/2/15, 9:08 AM | 2 KB      | Plain Text Document |
| core.api.php       | 12/2/15, 9:08 AM | 118 KB    | PHP                 |
| core.libraries.yml | 12/2/15, 9:08 AM | 21 KB     | YAML                |
| core.services.yml  | 12/2/15, 9:08 AM | 62 KB     | YAML                |
| globals.api.php    | 12/2/15, 9:08 AM | 2 KB      | PHP                 |
| includes           | 12/2/15, 9:08 AM | --        | Folder              |
| INSTALL.mysql.txt  | 12/2/15, 9:08 AM | 2 KB      | Plain Text Document |
| INSTALL.pgsql.txt  | 12/2/15, 9:08 AM | 2 KB      | Plain Text Document |
| install.php        | 12/2/15, 9:08 AM | 1 KB      | PHP                 |
| INSTALL.sqlite.txt | 12/2/15, 9:08 AM | 1 KB      | Plain Text Document |
| INSTALL.txt        | 12/2/15, 9:08 AM | 18 KB     | Plain Text Document |
| lib                | 12/2/15, 9:08 AM | --        | Folder              |
| LICENSE.txt        | 12/2/15, 9:08 AM | 18 KB     | Plain Text Document |
| MAINTAINERS.txt    | 12/2/15, 9:08 AM | 17 KB     | Plain Text Document |
| misc               | 12/2/15, 9:08 AM | --        | Folder              |
| modules            | 12/2/15, 9:08 AM | --        | Folder              |
| phpcs.xml.dist     | 12/2/15, 9:08 AM | 4 KB      | Document            |
| phpunit.xml.dist   | 12/2/15, 9:08 AM | 3 KB      | Document            |
| profiles           | Today, 1:12 PM   | --        | Folder              |
| rebuild.php        | 12/2/15, 9:08 AM | 2 KB      | PHP                 |
| scripts            | 12/2/15, 9:08 AM | --        | Folder              |
| tests              | 12/2/15, 9:08 AM | --        | Folder              |
| themes             | 12/2/15, 9:08 AM | --        | Folder              |
| UPGRADE.txt        | 12/2/15, 9:08 AM | 6 KB      | Plain Text Document |
| example.gitignore  | 12/2/15, 9:08 AM | 1 KB      | Document            |
| index.php          | 12/2/15, 9:08 AM | 549 bytes | PHP                 |
| LICENSE.txt        | 9/23/14, 3:24 PM | 18 KB     | Plain Text Document |

The `core` folder consists of miscellaneous files needed by Drupal to bootstrap the content management system as well as the following folders:

- `assets`: Various external libraries used by core (jquery, modernizr, backbone, and others)
- `config`: It contains misc configuration for installation and database schema
- `includes`: It contains files and folders related to the functionality of Drupal

- lib: Drupal core classes
- misc: Various JavaScript files and images used by core
- modules: Drupal core modules and Twig templates
- profiles: Installation profiles
- scripts: Various CLI scripts
- tests: Drupal core tests
- themes: Drupal core themes

## The modules folder

While a lot of functionality, which was generally contained in a contributed module, has been moved into the core instance of Drupal 8, you will still find yourself needing to extend Drupal. Previously, you would locate a contributed module, download it, and then extract its contents into your `sites/all/modules` folder so that Drupal could then use it.

Contributed and custom modules are now placed into the `modules` folder, which is no longer contained inside your `sites` folder.

| Name              | Date Modified    | Size      | Kind                |
|-------------------|------------------|-----------|---------------------|
| autoload.php      | 12/2/15, 9:08 AM | 262 bytes | PHP                 |
| composer.json     | 12/2/15, 9:08 AM | 1 KB      | JSON                |
| composer.lock     | 12/2/15, 9:08 AM | 133 KB    | Document            |
| core              | Today, 11:35 PM  | --        | Folder              |
| example.gitignore | 12/2/15, 9:08 AM | 1 KB      | Document            |
| index.php         | 12/2/15, 9:08 AM | 549 bytes | PHP                 |
| LICENSE.txt       | 9/23/14, 3:24 PM | 18 KB     | Plain Text Document |
| modules           | 12/2/15, 9:08 AM | --        | Folder              |
| README.txt        | 12/2/15, 9:08 AM | 2 KB      | Plain Text Document |
| profiles          | 12/2/15, 9:08 AM | --        | Folder              |
| README.txt        | 12/2/15, 9:08 AM | 6 KB      | Plain Text Document |
| robots.txt        | 12/2/15, 9:08 AM | 1 KB      | Plain Text Document |
| sites             | Today, 9:31 PM   | --        | Folder              |
| themes            | 12/2/15, 9:08 AM | --        | Folder              |
| update.php        | 12/2/15, 9:08 AM | 547 bytes | PHP                 |
| vendor            | 12/2/15, 9:08 AM | --        | Folder              |
| web.config        | 12/2/15, 9:08 AM | 4 KB      | Document            |

Best practices are to create a few subdirectories inside the `modules` folder for contributed modules – the modules built by third parties that we will use to extend your project, such as `contrib`, and `custom`, for the modules that we create on a per project basis. We will also occasionally find ourselves with a `features` folder if we plan to use the Features module to break out functionality that needs to be managed in code for purposes of migrating it easily to development, staging, and production instances of our website.

## The sites folder

We are all familiar with the `sites` folder in Drupal 7. However, in Drupal 8, the `sites` folder only contains our Drupal instance configuration and files.

| Name                       | Date Modified    | Size      | Kind                |
|----------------------------|------------------|-----------|---------------------|
| autoload.php               | 12/2/15, 9:08 AM | 262 bytes | PHP                 |
| composer.json              | 12/2/15, 9:08 AM | 1 KB      | JSON                |
| composer.lock              | 12/2/15, 9:08 AM | 133 KB    | Document            |
| core                       | Today, 11:35 PM  | --        | Folder              |
| example.gitignore          | 12/2/15, 9:08 AM | 1 KB      | Document            |
| index.php                  | 12/2/15, 9:08 AM | 549 bytes | PHP                 |
| LICENSE.txt                | 9/23/14, 3:24 PM | 18 KB     | Plain Text Document |
| modules                    | 12/2/15, 9:08 AM | --        | Folder              |
| profiles                   | 12/2/15, 9:08 AM | --        | Folder              |
| README.txt                 | 12/2/15, 9:08 AM | 6 KB      | Plain Text Document |
| robots.txt                 | 12/2/15, 9:08 AM | 1 KB      | Plain Text Document |
| sites                      | Today, 11:47 PM  | --        | Folder              |
| default                    | Today, 9:31 PM   | --        | Folder              |
| default.services.yml       | 12/2/15, 9:08 AM | 6 KB      | YAML                |
| default.settings.php       | 12/2/15, 9:08 AM | 28 KB     | PHP                 |
| files                      | Today, 10:44 PM  | --        | Folder              |
| config_ac....38-1802       | Today, 9:31 PM   | --        | Folder              |
| config_c....17wiglaDQ      | Today, 10:20 PM  | --        | Folder              |
| css                        | Today, 11:11 PM  | --        | Folder              |
| js                         | Today, 11:11 PM  | --        | Folder              |
| php                        | Today, 10:20 PM  | --        | Folder              |
| styles                     | Today, 10:20 PM  | --        | Folder              |
| settings.php               | Today, 10:20 PM  | 29 KB     | PHP                 |
| development.services.yml   | 12/2/15, 9:08 AM | 249 bytes | YAML                |
| drupal8.dd                 | Today, 9:31 PM   | 7 bytes   | Alias               |
| example.settings.local.php | 12/2/15, 9:08 AM | 3 KB      | PHP                 |
| example.sites.php          | 12/2/15, 9:08 AM | 2 KB      | PHP                 |
| README.txt                 | 12/2/15, 9:08 AM | 515 bytes | Plain Text Document |
| sites.php                  | Today, 9:31 PM   | 2 KB      | PHP                 |
| themes                     | 12/2/15, 9:08 AM | --        | Folder              |
| update.php                 | 12/2/15, 9:08 AM | 547 bytes | PHP                 |
| vendor                     | 12/2/15, 9:08 AM | --        | Folder              |
| web.config                 | 12/2/15, 9:08 AM | 4 KB      | Document            |

## The themes folder

Finally, we have the `themes` folder. So, why don't we see the default themes that Drupal generally ships with inside this folder? That is because Drupal's default themes now are contained within the `core` folder. Finally, we will actually use the `themes` folder to place our custom or contributed themes inside it for use by Drupal.

We will be exploring the `themes` folder in more detail in later chapters as we begin creating custom themes.

| Name              | Date Modified    | Size      | Kind                |
|-------------------|------------------|-----------|---------------------|
| autoload.php      | 12/2/15, 9:08 AM | 262 bytes | PHP                 |
| composer.json     | 12/2/15, 9:08 AM | 1 KB      | JSON                |
| composer.lock     | 12/2/15, 9:08 AM | 133 KB    | Document            |
| ▶ core            | Today, 11:35 PM  | --        | Folder              |
| example.gitignore | 12/2/15, 9:08 AM | 1 KB      | Document            |
| index.php         | 12/2/15, 9:08 AM | 549 bytes | PHP                 |
| LICENSE.txt       | 9/23/14, 3:24 PM | 18 KB     | Plain Text Document |
| ▶ modules         | 12/2/15, 9:08 AM | --        | Folder              |
| ▶ profiles        | 12/2/15, 9:08 AM | --        | Folder              |
| README.txt        | 12/2/15, 9:08 AM | 6 KB      | Plain Text Document |
| robots.txt        | 12/2/15, 9:08 AM | 1 KB      | Plain Text Document |
| ▶ sites           | Today, 11:47 PM  | --        | Folder              |
| ▶ themes          | 12/2/15, 9:08 AM | --        | Folder              |
| ▶ README.txt      | 12/2/15, 9:08 AM | 1 KB      | Plain Text Document |
| update.php        | 12/2/15, 9:08 AM | 547 bytes | PHP                 |
| ▶ vendor          | 12/2/15, 9:08 AM | --        | Folder              |
| web.config        | 12/2/15, 9:08 AM | 4 KB      | Document            |

## Using the project files

As we work through each chapter of the module, we will be using exercise files that contain examples of how each page will be themed is laid out. This will include database snapshots, HTML, CSS, and images for our Home, About, Portfolio, Blog, and Contact pages. Before we begin using these files, we need to know where to download them from and the best location to extract them to for future use.

## Downloading and extracting the exercise files

We can find the exercise files at <https://www.packtpub.com/support>. Click on the download link and save the compressed file to the desktop. Once the download is finished, we will need to extract the contents. Let's take a quick look at what we have:

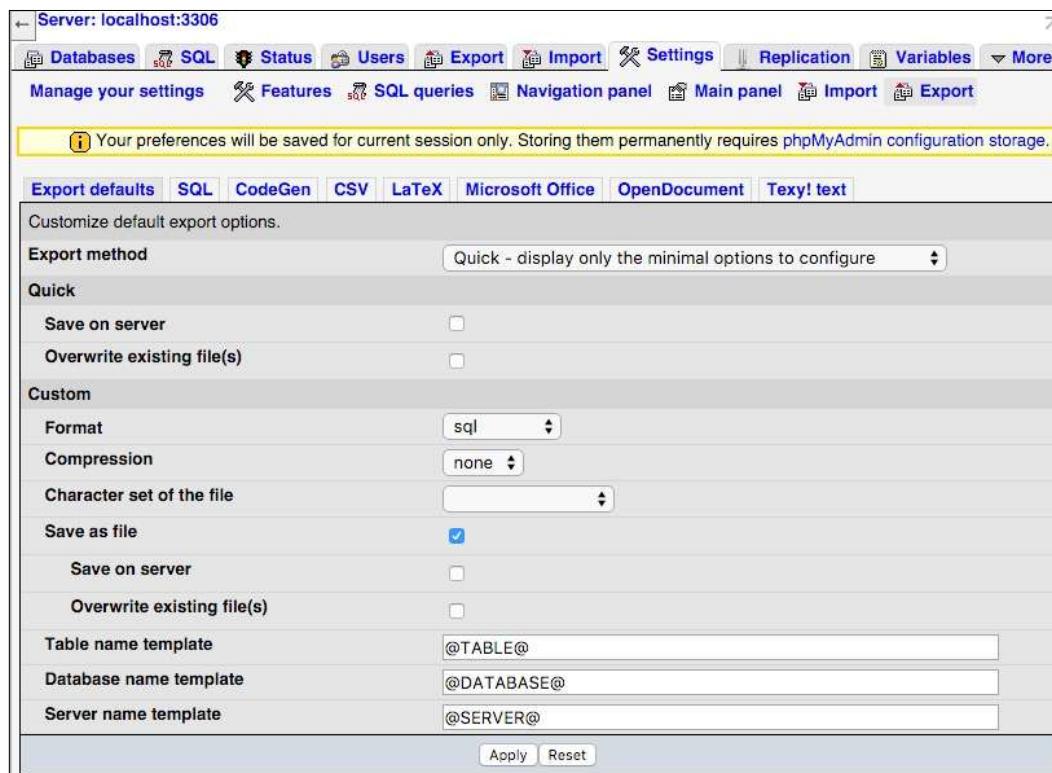
- Several chapter folders containing files that we will use for whatever task we are working on in that chapter
- A Mockup folder that contains the finished HTML version of our theme
- A database snapshot contained within various chapter folders that we will use to restore on top of our current Drupal instance to ensure that we always have the same configuration at specific points along the way

Since we will be working with database snapshots at various points, we will want to look at how we can manage these files using the MySQL database tool named phpMyAdmin.

## Database backup

It is important to know how to backup our database when working in Drupal 8 as most of our content and configurations are contained within a database. Make sure that phpMyAdmin is open in the browser.

Next, we will want to make sure that any database exports are saved as a file versus just plain SQL script. Because this is a global setting, we will need to make sure that we have not selected any specific database. We can make sure that we are affecting global settings by clicking on the house icon in the left-hand sidebar underneath the phpMyAdmin logo. Next, we can navigate to **Settings** and then **Export**, as shown in the following image:



Next, click on the **Save as file** option and the **Apply** button.

## Export settings

One little gotcha when using phpMyAdmin is making sure that when we create our database export, we ensure that the SQL also drops any tables before trying to recreate them when we do the import later.

We can create our database export by following these steps:

1. Select the drupal8 database from the left sidebar.
2. Click on the **Export** tab.

3. Select **Custom - display all possible options** as our export method.
4. Select **SQL** as our format.
5. Choose **Add DROP TABLE / VIEW / PROCEDURE / FUNCTION / EVENT / TRIGGER statement** from **Object creation options**.
6. Click on the **Go** button.

At this point, we have a new file named `drupal8.sql`, which contains a backup of our database. Next, we will use this file we just created to restore our database.

## Database restore

Restoring a database is simpler than backing our database up. Except this time, we will be using an existing database snapshot that either we have taken or that we were provided to overwrite our current database files with. Let's begin by following these steps:

1. Click on the **Import** tab.
2. Click on the **Choose File** button and locate the `drupal8.sql` file we created earlier.
3. Click on the **Go** button to begin the restoration process.

The process of restoring the file can sometimes take a minute to complete, so please be patient while the file is being restored.

While phpMyAdmin allows us to manage database operations, we can choose to use other database tools or even the command line, which is a lot faster to export and import databases.

Now that we have a good understanding of how to back up and restore our database, it's time to take a quick look at how we will be using Google Chrome to review our HTML and CSS structure within Drupal 8 and our theme Mockup.

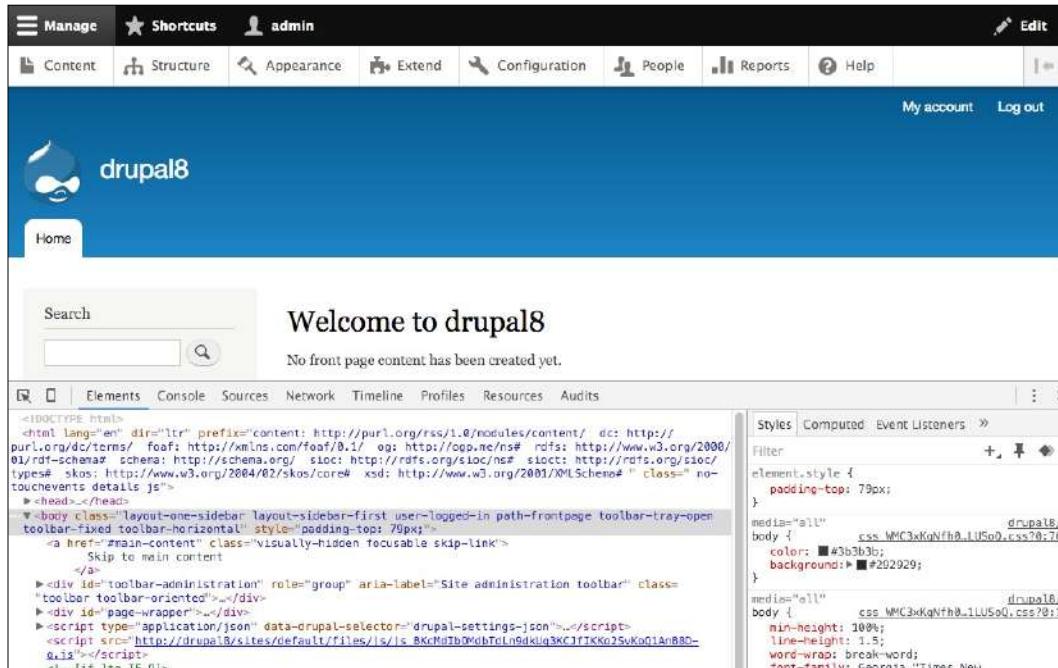
## Using Google Chrome to inspect elements

While there are many different browsers to view web content on, Google Chrome is definitely a favorite browser when theming in Drupal. It is not only standards compliant ensuring that most HTML and CSS work properly but Chrome also allows us to inspect the HTML and CSS and preview changes "live" within the browser using the **Developer Tools** option.

## *Setting Up Our Development Environment*

---

Begin by browsing to our local Drupal 8 instance in Google Chrome and then selecting **Developer Tools** from the **View | Developer | Developer Tools** menu. The **Developer Tools** will open up in the bottom of your browser, as shown in the following image:



There are several tools available for our use, but the one we will use the most is inspecting **Elements** on the page, which allows us to view the HTML structure and any CSS being applied to that element from the **Elements** and **Styles** panels. We can navigate through the HTML structure, or if we prefer to isolate an element on the page, we can place our cursor on that element and right-click to open up a context menu where we can select **Inspect**. Doing so will target that element in the **Elements** pane for us.

As we dive deeper into theming, we will use this set of tools to help preview changes as well as isolate any issue we may be experiencing as our HTML structure changes based on what Drupal 8 outputs.

## Summary

We collected a lot of information to start our series on Drupal 8 themes. Let's review exactly what we covered so far:

- We successfully configured an AMP (Apache MySQL, PHP) stack by downloading and installing MAMP PRO.
- We set up our first Drupal 8 instance by downloading the latest version from Drupal.org, importing the Drupal instance into our AMP stack, and completing the Drupal 8 install by choosing our language, profile, database settings, and site information.
- We also had our first look at Drupal 8 and some of the new responsive functionality that it provides. We familiarized ourselves with the admin menu and the new admin interface, which included the **Content, Structure, Appearance, Extend, Configuration, People, Reports, and Help** sections. Having a better knowledge of Drupal 8 and its folder structure has given us insight into how to apply best practices to manage our theme and its assets.
- By using the project files, we learned how to manage database snapshots through importing and exporting SQL files inside phpMyAdmin.
- Finally, we learned how to use Google Chrome to inspect our HTML and CSS to have a better understanding of our theme and its markup.

In the next chapter, we will take a closer look at "theme administration" and answer the question: what is a theme? We will explore the "appearance interface" and discuss how Drupal's default themes function. Finally, we will follow up with looking closer at how to use prebuilt themes and managing content with blocks and custom block layouts.



# 2

## Theme Administration

Before we can get started with creating or managing themes in Drupal 8, we need to have a better understanding of exactly what a theme is. From there we will have the basis for how we work with themes in Drupal and the various configuration and components that are considered to be part of the theming ecosystem.

Let's get started by exploring what we will be covering along the way:

- First, we will explore the Appearance interface and the core default themes. We will learn how themes are administered, how to install and uninstall themes, how they are configured, and the different settings a theme can have.
- Second, we will take a closer look at a prebuilt theme, where to find themes that we can use, and how we can easily install themes using the Drupal 8 admin.
- Third, we will take a closer look at the `themes` folder structure and how to manually install a theme in preparation to create a custom theme.
- Fourth, we will take a look at the new Block layout and how we can manage chunks of content and assign content to regions. This will include configuring a block and controlling the visibility of blocks based on certain settings.
- Finally, we will take a look at the new Custom Block library and explore how to add fields to blocks, something not previously available to us in Drupal 7.

We have a lot to cover, so let's get started by discussing, what is a theme?

## What is a theme?

In simple terms, a theme is the presentational layer to content. Regardless of you working in Drupal or another **content management system (CMS)** without a theme, all you have is content that looks very similar to a Word document.

A theme generally consists of HTML markup, CSS, JavaScript, and media (images, video, and audio). It is this combination of technologies that allow the graphic designer to build something visually rich that can then be applied on top of the logic a web developer is building in Drupal. Sometimes, a web developer may be the person who implements the theme, but in most cases, you will hear the term themer or interface engineer, which describes the person who actually fills that role.

This module helps you learn that role. So, as long as you have a good knowledge of HTML, CSS, and JavaScript, you are well on your way to filling a much-needed role in the Drupal community.

We will begin by exploring the Appearance interface in Drupal 8.

## Exploring the Appearance interface

The Appearance interface in Drupal 8 can be located by clicking on the **Manage** menu item from the administrative toolbar and then by choosing the **Appearance** link. We can also directly navigate to the Appearance interface by entering the address of `/admin/appearance` in our browser.

The screenshot shows the 'Appearance' page in the Drupal 8 admin interface. At the top, there's a navigation bar with tabs for 'List', 'Update', and 'Settings'. Below it, a breadcrumb trail shows 'Home > Administration > Appearance'. A message says 'Set and configure the default theme for your website. Alternative themes are available.' A blue button labeled '+ Install new theme' is visible. The main section is titled 'Installed themes' and features a thumbnail of the 'Bartik' theme. To the right of the thumbnail, the text 'Bartik 8.0.1 (default theme)' and 'A flexible, recolorable theme with many regions and a responsive, mobile-first layout.' is displayed. A 'Settings' link is also present below the theme description.

The **Appearance** interface allows us to work with themes in Drupal, that is, anything from installing, uninstalling, and configuring the default theme for our website. We will be exploring the various functions within this section starting with taking a look at the default themes that Drupal 8 ships with.

## Drupal's core themes

By default, Drupal 8 ships with three themes. As part of the standard installation profile, Drupal will install and configure Bartik, Seven, and Stark themes. Each of these themes serves a specific function in the workflow. Let's look at them in more detail.

### Bartik

Bartik is considered the default theme in Drupal and is familiar to most as it has been part of the Drupal ecosystem for quite a while now. We can think of Bartik as the frontend theme or what we see when we first install Drupal. The Bartik theme is what you will visually see when you are not navigating within the Drupal administrative screens.

### Seven

Seven is the default admin theme, and it provides a clean separation between the frontend and backend of Drupal. This is great as it will always allow us to navigate through the administrative areas if our default theme generates any errors that may cause a blank white screen while theming.

### Stark

Stark is an intentionally plain theme with no styling at all to help demonstrate the default HTML and CSS that Drupal will output and is great for learning how to build a custom theme.

### Classy

Wait, this is the fourth theme! Actually, Classy is a base theme that both Bartik and Seven use that provides both with clean well-documented markup and CSS classes. Classy is hidden from the Appearance admin screen by default, and we will learn more about Classy as a base theme and how to use it within our own themes later in *Chapter 4, Getting Started – Creating Themes*.

## Theme states

One of the advantages of Drupal is the ability to have multiple themes available to use at any time, and as we discussed earlier, Drupal provides us with three themes to start with. However, it is important to differentiate between installed, uninstalled, and default. We can consider these as the theme's states.

## Installed themes

Installed themes are always located in the **Installed themes** section of the Appearance admin and are available for Drupal to use for either the frontend or backend of the CMS. However, there can only be one theme set as the default at any given time. We can see a list of installed themes, as shown in the following image:

The screenshot shows the 'Installed themes' section of the Drupal Appearance admin. There are two entries:

- Bartik 8.0.1 (default theme)**: A flexible, recolorable theme with many regions and a responsive, mobile-first layout. It has a preview image of a globe and a 'Settings' link.
- Seven 8.0.1 (admin theme)**: The default administration theme for Drupal 8 was designed with clean lines, simple blocks, and sans-serif font to emphasize the tools and tasks at hand. It has a preview image of a dashboard and a 'Settings' link.

## Uninstalled themes

Uninstalled theme(s) are themes that Drupal is aware of within the core themes folder or the custom themes folder but have not been placed into an installed state. One or multiple themes can be present at any time within the **Uninstalled theme** section, as shown in the following image:

**Uninstalled theme**

**Stark 8.0.1**  
An intentionally plain theme with no styling to demonstrate default Drupal's HTML and CSS. Learn how to build a custom theme from Stark in the [Theming Guide](#).

[Install](#) | [Install and set as default](#)

## Default theme

Finally, we will often hear the term **default theme** being used, so it's important to remember that the default theme is always the current theme being displayed to users when viewing our website as an anonymous or logged out user or when logged in but not within an Administrative section of Drupal. Anytime a theme is set as default, it will always be considered installed as well.

## Installing and uninstalling themes

The act of installing or uninstalling a theme is a common practice when administering a Drupal website. Let's try installing Stark and making it our default theme in place of Bartik.

### Step one – installing a theme

Currently, the only uninstalled theme is **Stark**, and we can easily move this into the installed state by following these two steps:

1. Click on the **Install and set as default** link.
2. Scroll back to the top of **Appearance** admin.

3. If we now look at the **Installed themes** section, we should see that we now have three themes installed: Stark, Bartik, and Seven. We can also tell that Stark is now our default theme by looking to the right of the themes name, which will be marked as **(default theme)** as shown in the following image:

The screenshot shows the 'Installed themes' section of the Drupal Appearance admin. It lists three themes: Stark 8.0.1 (default theme), Bartik, and Seven. The Stark theme is highlighted, showing its thumbnail (a dark background with text), name, description ('An intentionally plain theme with no styling to demonstrate default Drupal's HTML and CSS. Learn how to build a custom theme from Stark in the Theming Guide.'), and a 'Settings' link.

We can also see what the Stark theme looks like by clicking on the **Back to site** link in the Admin menu, which will take us back to the frontend of our website. We are now presented with an unstyled page, which is to help demonstrate the clean markup of Drupal.

## Step two – uninstalling a theme

It is just as easy to uninstall a theme, and one nice feature of Drupal 8 is that it ensures that we have at least one installed theme set as default. Otherwise, we won't even have the option of uninstalling the theme.

Let's navigate back to the Appearance admin located at /admin/appearance and uninstall the Stark theme by following these two steps:

1. Locate the **Bartik** theme and click on **Set as default**.
2. Locate the **Stark** theme and click on **Uninstall**.

We saw how simple it is to install and uninstall themes within Drupal 8. Another common task we will find ourselves completing within the Appearance admin is adjusting the settings of a theme.

## Theme settings

Under the **Settings** tab of the Appearance admin, the options to control the default display settings for your entire site are located, across all themes or individually. These settings range from toggling the display of certain page elements, updating the default logo supplied by the theme, to providing a shortcut icon or favicon that is displayed in the address bar of most browsers.

Let's explore these in more detail by clicking on the **Settings** tab and previewing the interface, as shown in the following image:

The screenshot shows the 'Appearance settings' page with the 'Global settings' tab selected. The page title is 'Appearance settings' with a star icon. Below the title are three tabs: 'List', 'Update', and 'Settings'. Under the 'Global settings' tab, there is a breadcrumb trail: 'Home > Administration > Appearance'. A note below the breadcrumb states: 'These options control the default display settings for your entire site, across all themes. Unless they have been overridden by a specific theme, these settings will be used.' There are three main sections with expandable arrows:

- ▼ TOGGLE DISPLAY**: Describes enabling or disabling the display of certain page elements. It contains five checked checkboxes:
  - User pictures in posts
  - User pictures in comments
  - User verification status in comments
  - Shortcut icon
- ▼ LOGO IMAGE SETTINGS**: Contains one checked checkbox: 'Use the default logo supplied by the theme'.
- ▼ SHORTCUT ICON SETTINGS**: Describes the shortcut icon or 'favicon' being displayed in the address bar and bookmarks. It contains one checked checkbox: 'Use the default shortcut icon supplied by the theme'.

## Toggling the display of page elements

Having control over certain page elements of a theme can come in handy when we want to hide or show specific items. Most of the items listed pertain to user settings, such as user pictures in posts or comments, user verification status in comments, and the Shortcut icon from displaying.

Simply checking or unchecking an item will toggle that item on or off. Also, keep in mind that toggling the Shortcut icon will disable the ability to add a shortcut icon as the visibility of that section is also toggled on and off.

Gone are Logo, Site name, Site slogan, Main menu, and Secondary menu from the theme settings. These were present in Drupal 7 but have now been moved into Blocks and block configuration. We will be addressing each of these moved settings in a few moments.

## Logo image settings

Another nice option within the Appearance settings admin is the ability to manage the themes logo. By default, Drupal displays the infamous Drop logo, but we have the power to replace that logo with our own.

Let's begin by following these five steps:

1. Locate the **LOGO IMAGE SETTINGS** section.
2. Uncheck **Use the default logo supplied by the theme**.
3. Click on the **Choose file** button under the **Upload logo image** field.
4. Locate the exercise files and select `logo.png` from the `mockup/assets/img` folder. Click on the **Open** button.
5. Click on the **Save configuration** button.
6. Our new logo has now been placed into the `sites/default/files` folder of our Drupal installation, as shown in the following image:

The screenshot shows the 'LOGO IMAGE SETTINGS' section of the theme administration. It includes a checkbox for 'Use the default logo supplied by the theme', which is unchecked. Below it is a 'Path to custom logo' input field containing 'logo.png'. A note below the path says 'Examples: logo.png (for a file in the public filesystem), public://logo.png, or sites/default/files/logo.png.'. There is also an 'Upload logo image' section with a 'Choose File' button and a message 'No file chosen'. A note at the bottom states 'If you don't have direct file access to the server, use this field to upload your logo.'

With the path to our custom logo now pointing to our new logo, we can preview it by clicking on the **Back to site** link in the Admin menu, which will take us to the frontend of our website, as shown in the following image:



One thing to note is that there is no simple way to delete logo images we upload using the Logo image settings, so Drupal will append a sequential number to the end of the file versus overriding it if it has the same name. In the case where we ever need to delete a logo image, we would have to navigate to the `sites/default/files` directory and manually delete the file.

## Shortcut icon settings

If you are wondering what a shortcut icon is, don't worry. The shortcut icon is also known as a favicon. It is the small image located in the browser window next to the URL address or if you are using Google Chrome, next to the page title of the website you are visiting.

Often this step is overlooked when creating or working with themes in Drupal, but the steps involved in adding a shortcut icon is exactly like adding a logo. Start by navigating to `/admin/appearance/settings` and follow these five steps:

1. Locate the **SHORTCUT ICON SETTINGS** section.
2. Uncheck **Use the default icon supplied by the theme**.
3. Click on the **Choose file** button under the **Upload icon image** field.
4. Locate the exercise files and select `favicon.ico` from the `mockup/assets/img` folder; click on the **Open** button.
5. Click on the **Save configuration** button.

Our new favicon has now been placed into the `sites/default/files` folder of our Drupal installation, as shown in the following image:

The screenshot shows the 'Shortcut icon settings' section of the theme administration. It includes a note about favicons being displayed in address bars and bookmarks. There is a checkbox for using the default theme icon, which is unchecked. Below it is a 'Path to custom icon' field containing 'favicon.ico'. A note says examples include 'favicon.ico' (file in public filesystem), 'public://favicon.ico', or 'sites/default/files/favicon.ico'. There is also an 'Upload icon image' field with a 'Choose File' button and a 'No file chosen' message. A note at the bottom says if there's no direct file access to the server, use this field.

We can now preview our shortcut icon by clicking on the **Back to site** link in the Admin menu and navigating to the homepage, as shown in the following image:



So far, we have been working with Global settings. However, individual theme settings can be applied as well. In fact, if we navigate back to the Appearance settings admin located at `/admin/appearance/settings`, we will see that Bartik and Seven can each have their own settings.

## Theme-specific settings

Drupal 8 allows for the configuration of theme-specific settings. These can vary based on the theme and the amount of extensibility that a theme provides. For example, if we click on the **Bartik** theme, we will notice that it provides us with an ability to change the **COLOR SCHEME** through a series of presets, as shown in the following image:

▼ COLOR SCHEME

| Color set                | Blue Lagoon (default) |
|--------------------------|-----------------------|
| Header background top    | #055a8e               |
| Header background bottom | #1d84c3               |
| Main background          | #ffffff               |
| Sidebar background       | #f6f6f2               |
| Sidebar borders          | #f9f9f9               |
| Footer background        | #292929               |
| Title and slogan         | #ffffff               |
| Text color               | #3b3b3b               |
| Link color               | #0071b3               |

Color wheel preview: A circular color wheel showing a gradient from red to blue, with a small square color swatch in the center.

Preview

Etiam est risus      **Lorem ipsum dolor**

Feel free to experiment by selecting various color sets and then previewing what those selections would look like if applied. Outside the core themes shipped with Drupal 8, we can apply prebuilt themes to provide various features.

## Using prebuilt themes

Additional themes for Drupal 8 may be limited at first, but we can find prebuilt themes at several places. Some of these themes have to be purchased, whereas others are free to use. We will take a look at [Drupal.org](https://drupal.org) to find some prebuilt themes and how to install them using the Drupal admin, how to manually install a theme, and finally how to uninstall a theme once we're done using it.

Begin by opening up a new tab in our browser and navigating to [https://drupal.org/project/project\\_theme](https://drupal.org/project/project_theme).

The **Download & Extend** section of [Drupal.org](https://drupal.org) allows us to filter results based on various options. We can find Drupal 8-specific themes by performing the following steps:

1. Select **8.x** from the **Core compatibility** dropdown.
2. Click on the **Search** button.
3. With a selection of themes compatible with Drupal 8 to choose from, one result looks promising and that is the Bootstrap 3 theme, as shown in the following image:

**Bootstrap**

Posted by markcarver on May 18, 2008 at 1:15pm

“Sleek, intuitive, and powerful mobile first front-end framework for faster and easier web development. Bootstrap has become one of the most popular front-end frameworks and open source projects in the world.

This base theme bridges the gap between Drupal and the [Bootstrap Framework](#).

**Features**

- [jsDelivr CDN](#) for "out-of-the-box" styling and faster page load times.
- [Bootswatch theme support](#), if using the CDN.
- Glyphicons support via [Icon API](#).
- Extensive integration and template/preprocessor overrides for most of the [Bootstrap Framework CSS](#), Components and JavaScript
- Theme settings to further enhance the Drupal Bootstrap integration:
  - [Breadcrumbs](#)
  - [Navbar](#)
  - [Popovers](#)
  - [Toolips](#)
  - [Wells \(per region\)](#)

**Documentation**

Visit the project's official documentation site or the markdown files inside the `./docs` folder.



**Bootstrap 3**  
for Drupal

## Installing a new theme

At this point, we should have two tabs opened in our browser. One opened to the Bootstrap 3 theme and the other tab opened to the Appearance admin of our Drupal instance.

From the Appearance admin, we can install a new theme by clicking on the **Install new theme** button, which will take us to the **Install new theme** interface, as shown in the following image:

We will first take a look at using the **Install from a URL** option to install the Bootstrap 3 theme.

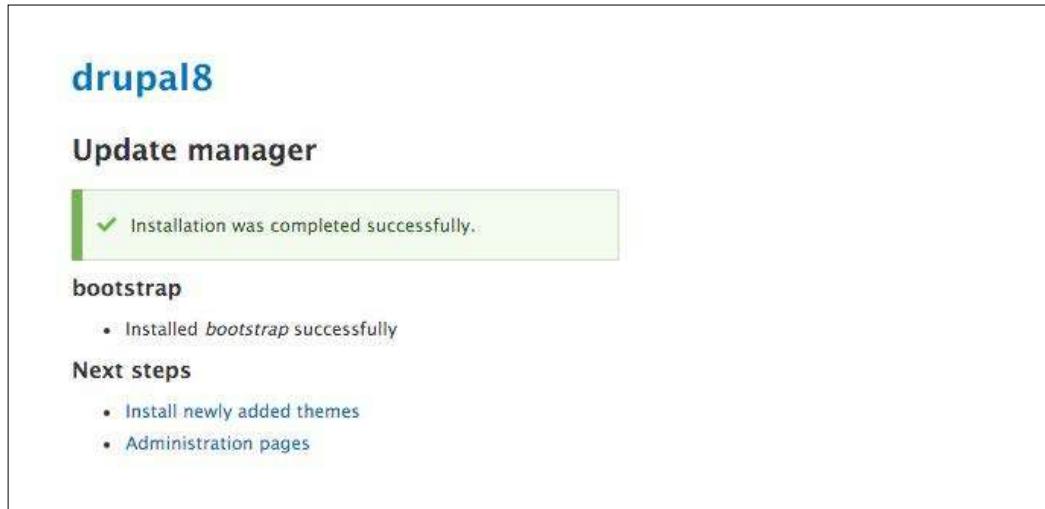
## Installing from a URL

To install a theme from a URL, we only need to know the URL path to the archived theme file. This can be obtained from the Theme project page where the theme was found.

| Version       | Download                                                             | Date        |
|---------------|----------------------------------------------------------------------|-------------|
| 8.x-3.0-beta2 | <a href="#">tar.gz (211.08 KB)</a>   <a href="#">zip (572.35 KB)</a> | 2015-Dec-19 |
| 7.x-3.4       | <a href="#">tar.gz (199.55 KB)</a>   <a href="#">zip (828.34 KB)</a> | 2015-Dec-19 |

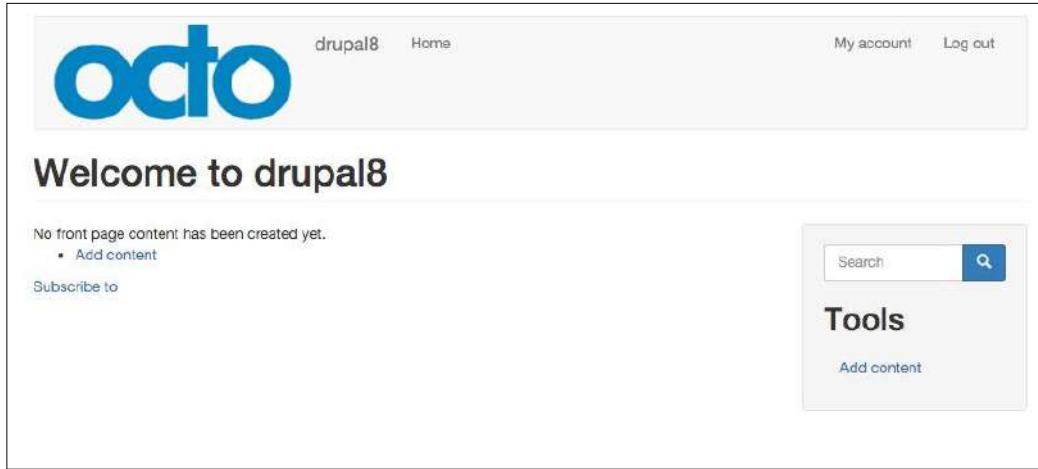
Follow these four steps:

1. Right-click on the tar.gz link located under the **Download** column of the 8.x version of the theme.
2. Select **Copy Link Address** from the context menu.
3. Paste the link into the **Install from a URL** textbox on the **Install new theme** admin screen.
4. Click on the **Install** button.
5. We should now be presented with the **Update manager** screen notifying us that the **Installation was completed successfully**, as shown in the following image:



6. Click on the **Install newly added themes** link to take us back to the **Appearance** admin.

7. If we look at the **Uninstalled themes** section, we will see the **Bootstrap 3** theme where we can click on the **Install and set as default** link.
8. To verify that our theme is installed, navigate to the frontend of our site by clicking on the **Back to site** link in the Admin menu. We should now see our new theme being displayed, as shown in the following image:



Congratulations, we have installed our first prebuilt theme. As we can see, the process of installing a theme from a URL is quite simple. In fact, the process of installing a new theme from a file is not that different, as we will see.

## **Uploading a module or theme archive to install**

If we navigate to `/admin/theme/install`, we will be back on the **Install new theme** screen. The **Upload a module or theme archive to install** option requires that we have a copy of the archived theme downloaded.

If we navigate back to the Drupal.org Theme project page and look a little further down our search results, we will see another popular theme based on the ZURB Foundation framework, as shown in the following image:

## ZURB Foundation

Posted by ishmael-sanchez on November 4, 2011 at 7:33pm

Do you like grid systems? How about rapid prototyping? Do you believe in mobile first? Then this theme is for you.

From the Zurb Foundation homepage:

- “ An easy to use, powerful, and flexible framework for building prototypes and production code on any kind of device.
- “ Mockups don't tell a story. Build a prototype in no time.
- “ The difference between a good site and a great site is iteration — building something, playing with it, refining it. In Foundation, we've included dozens of styles to help you quickly prototype a site.

See Zurb Foundation [in the wild](#).

### Read the Zurb Foundation Docs

In order to use this theme effectively, it may help if you learn how to use Zurb Foundation's built in classes and components:

- [Learn how to use the Foundation framework.](#)
- See the kitchen sink! See all of Zurb Foundation 4's components on one page



We will be downloading the current Drupal 8 Development release of the theme by clicking on either the tar.gz or zip links. This will initiate a file download to our specified downloads folder. We will use this downloaded file to perform the following steps from the **Install new theme** admin:

1. Click on the **Choose file** button from the **Upload a module or theme archive to install** input.
2. Locate the downloaded file and select it.

3. Click on the **Open** button.
4. Click on the **Install** button.
5. Click on the **Install newly added themes** link from the **Drupal 8 Update manager**.
6. If we look at the **Uninstalled themes** section, we will see the **ZURB Foundation** theme where we can click on the **Install and set as default** link.
7. Navigate back to the frontend by clicking on the **Back to site** link and verify that our new theme is being displayed, as shown in the following image:



We have now mastered using the Drupal Admin to install themes from both a URL and an archive. However, the preferred method to install a theme is by manually installing it. The benefit to choose this method is that we have full control over the themes folder, and if we are doing any type of custom theming, we will need to be familiar with this process anyway.

## Manually installing a theme

In order to manually install a theme, we will need a copy of the archive downloaded to our local machine. Start by navigating to the Drupal Theme project page and locate the Drupal 8 theme named **Neato**. Neato is based on the Neat grid system and is part of the Bourbon Sass framework.

**Neato**

Posted by [kevinquillen](#) on March 23, 2015 at 6:34pm

Neato is a theme based on the Neat grid system, incorporating Bourbon and Bitters to make grid theming with semantic markup easy and sensible.

Gulp and Bower are also included to assist in setting up for theme editing, and are required to pull in and manage assets. A Drush command and STARTER theme is included to get started right away.

**Downloads**

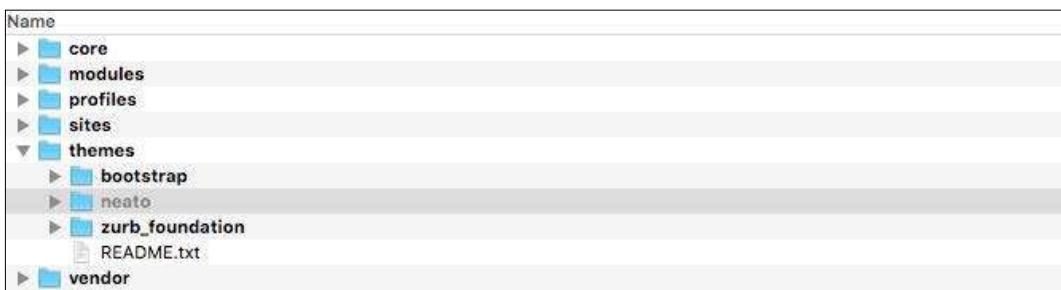
**Recommended releases**

| Version | Download                                                           | Date        |
|---------|--------------------------------------------------------------------|-------------|
| 8.x-1.2 | <a href="#">tar.gz (40.32 KB)</a>   <a href="#">zip (51.55 KB)</a> | 2015-Dec-08 |
| 7.x-1.1 | <a href="#">tar.gz (36.21 KB)</a>   <a href="#">zip (46.28 KB)</a> | 2015-Dec-03 |



Click on the tar.gz or zip file next to the 8.x version of the theme to initiate the download. Next, we need to locate the tar or zipped file on our machine and extract the contents of the file. We should now have a theme folder named neato.

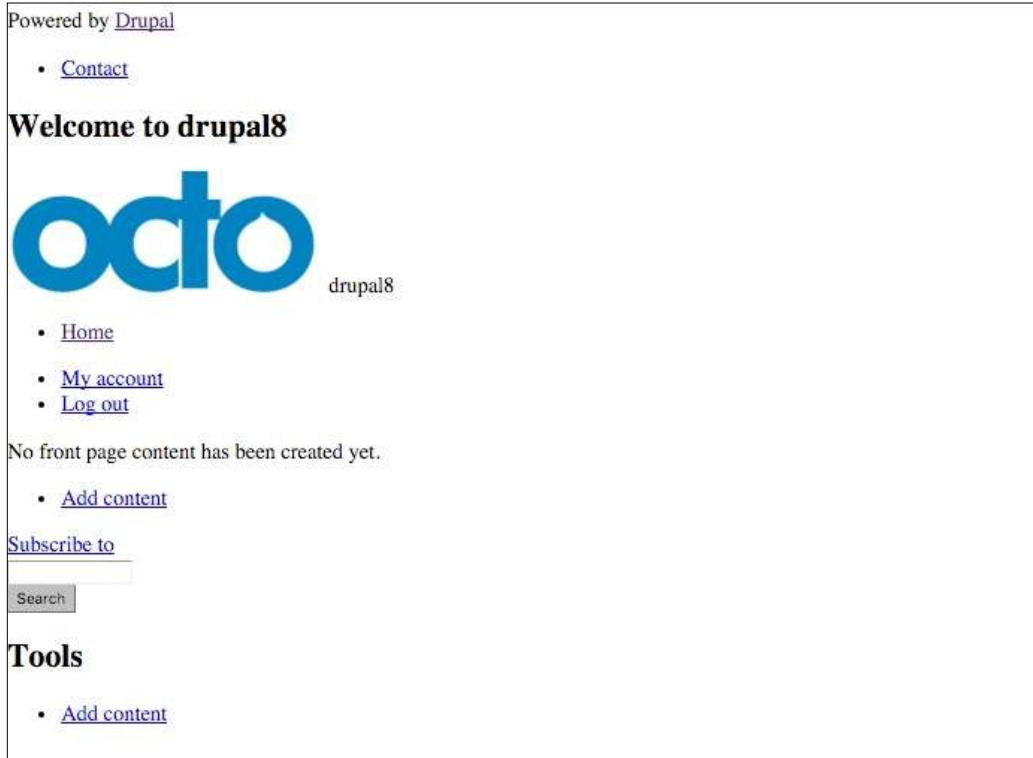
In order for Drupal 8 to recognize a new theme, all we need to do is copy the theme into the themes directory inside our Drupal 8 installation, as shown in the following image:



Our themes folder should now contain three themes. Two of the themes are installed through the Drupal admin and the third is manually placed. As long as a Drupal 8 theme is configured properly, simply placing it into the themes folder will allow it to be found by the Drupal admin. However, some themes may come with an INSTALL.txt or README.txt file that provides additional installation instructions.

Navigate back to the Appearance admin located at /admin/appearance, and we should see our Neato theme within the **Uninstalled themes** section. We can now click on the **Install and set as default** link to activate our new theme.

With our new theme enabled, we can now view it by navigating back to our homepage, as shown in the following image:



By now, we should be getting used to working with prebuilt themes. However, the more themes we play around with, the more our themes folder can become bloated. It is much easier to manage themes if we have a clean directory structure.

## Cleaning up our themes folder

To have a much more manageable theme folder, let's take a few moments to do some housecleaning. First, let's set the Bartik theme back as our default. Once that is complete, we can uninstall Bootstrap 3, ZURB Foundation, and Neato themes. Finally, once the three themes are uninstalled, we can remove them from our themes folder as we will not be using them anymore.

## Managing content with blocks

Themes are much more than just layout with their respective HTML, CSS, and JavaScript. Without content, we would not have much to display. In Drupal 8, a lot of content has been moved into blocks that are then assigned to various regions.

Think of blocks as small sections of content. These blocks can contain a menu, a search form, a listing of content, plain HTML, and more. Drupal 8 uses this content within the Block layout system that makes up a good part of a website.

If you are familiar with blocks in Drupal 7, you will be happily surprised to find that blocks have matured and are now fieldable, similar to content types. This new implementation of blocks also allows the same block to be reused and displayed using different view modes. As we explore the Block layout, we will learn how to place blocks, configure them, and create custom blocks.

## Exploring the Block layout

Begin by navigating to `/admin/structure/block` or by clicking on **Structure** and then **Block layout** from the Admin menu. The Block layout page provides an interface in order to manage block content and place them into regions, as shown in the following image:

| BLOCK             | CATEGORY    | REGION         | OPERATIONS                    |
|-------------------|-------------|----------------|-------------------------------|
| Header            | Place block | Header         | Show row weights<br>Configure |
| Site branding     | System      | Primary menu   | Configure                     |
| Main navigation   | Menus       | Secondary menu | Configure                     |
| User account menu | Menus       |                | Configure                     |

## Blocks and regions

A block can be thought of as a chunk of content as small as a single line of HTML markup or as complicated as a listing of content types. Blocks can be categorized based on their functionality with the most common types of blocks categorized:

- **Core:** Blocks contained within the core installation consisting of items such as page title, primary admin actions, and tabs.
- **System:** Blocks that provide system functionality consisting of breadcrumbs, main-page content, messages, site branding, and a few others.
- **Forms:** Blocks that contain embedded forms such as the search form and user login.
- **Menus:** Blocks that contain menus and menu items, such as Administration, Footer, Main navigation, and Tools.
- **Lists (Views):** Blocks consisting of Views generated for block content. Generally, these types of blocks will be created during configuration or site building.
- **Custom:** Blocks are created from the Custom block library consisting of fieldable blocks with one or more display modes.

If blocks consist of content, regions are the containers that hold blocks and make up a theme's layout.

Drupal 8 provides the following regions:

- Header
- Primary menu
- Secondary menu
- Highlighted
- Help
- Content
- Sidebar first
- Sidebar second
- Footer
- Breadcrumb

Also, each theme can define its own regions. An example of this is the Bartik theme, which implements additional regions for the footer. We will look at how to add custom regions later in *Chapter 3, Dissecting a Theme*.

## Demonstrating block regions

To view defined regions within a theme, we can click on the **Demonstrate block regions** link located on the **Block layout** page. Clicking on this link will take us to the homepage with the regions highlighted, as shown in the following image:



To return to the **Block layout** page, we click on the **Exit block region demonstration** link located at the top of the page.

## Placing blocks into regions

If we scroll down the **Block layout** page and locate the **Sidebar second** region, we can see that it is currently empty. Empty regions will not output anything until we have placed a block within it. To place a block, we can follow these steps:

1. Click on the **Place block** button next to the **Sidebar second** region.
2. Locate the **Powered by Drupal** block and click on the **Place block** button.
3. Leave the default settings within the **Configure block** dialog.
4. Click on the **Save block** button.
5. We now have a new **Powered by Drupal** block placed within the **Sidebar second** region, as shown in the following image:



Let's verify that the **Powered by Drupal** block is displaying properly by navigating to our homepage. Since the **Sidebar second** region now contains a block, we will see the content appear in the right-hand column of the page, as shown in the following image:



So far, we have placed a block successfully within a region, but we can also configure a block based on the type of block it is.

## Configuring a block

Although we can navigate back and forth between the specific page a block is located on and the Block layout screen, it is much easier to use the context menu provided by the block. If we hover over a block, we will see a **Configure block** link as, shown in the following image:



Clicking on this link takes us directly to the **Configure block** screen for the block, as shown in the following image:

A screenshot of the "Configure block" screen for the "Powered by Drupal" block. The title is "Powered by Drupal". The "Display title" checkbox is checked. Under "Visibility", the "Content types" section shows "Pages" and "Roles" both set to "Not restricted". The "Region" dropdown is set to "Sidebar second". At the bottom, there are "Save block" and "Delete" buttons.

All blocks contain three common areas of configuration: **Title**, **Visibility**, and **Region**. Keep in mind that additional configuration options may be available based on the type of block. We will only be covering the basic configuration options.

## Managing the title

Block content, whether system generated or custom, can have its display title changed or even suppressed. In the case of our **Powered by Drupal** block, we can change the title by simply inputting a different value in the **Title** field:

1. Locate the **Title** field.
2. Change the value to **Powered by Drupal 8**.
3. Click on the **Save block** button.
4. We return to the homepage, and we can see that our block title has changed, as shown in the following image:



Let's try suppressing the block title all together by clicking on the **Configure block** context link and following these steps:

1. Uncheck the **Display title** checkbox.
2. Click on the **Save block** button.

We should now only see the content of our block being displayed as the title is gone.

## Managing visibility settings

Sometimes, we may want to control the visibility of a particular block of content based on various contexts. If we navigate back to the **Configure block** screen and scroll down to the **Visibility** section, we will see three different contexts to restrict visibility based on **Content types**, **Pages**, and **Roles**.

## Content types restriction

Content types' visibility allows us to determine whether a block is displayed based on the content type of the node or page that the block is placed on. For example, if we have a block listing of recent articles that we only want to display in the sidebar on an Article Detail node, we could specify **Article** for the **Content types** restriction. This restriction would ensure that the same block did not accidentally show on a Basic page node.

The screenshot shows the 'Visibility' configuration page. On the left, there are three sections: 'Content types' (Not restricted), 'Pages' (Not restricted), and 'Roles' (Not restricted). On the right, under 'Content types', there is a list with two items: 'Article' and 'Basic page'. Both items have an unchecked checkbox next to them.

## Page restriction

Page restriction allows us to whitelist or blacklist blocks of content based on the path to a specific page or set of pages. The path to the page needs to be entered one line at a time and can utilize a wildcard "\*" character to specify all children pages. Once we have entered the path, we can choose to *negate the condition* by either selecting **Show for the listed pages** or **Hide for the listed pages**.

The screenshot shows the 'Visibility' configuration page. On the left, there are three sections: 'Content types' (Not restricted), 'Pages' (Not restricted), and 'Roles' (Not restricted). On the right, under 'Pages', there is a large text input field with placeholder text: 'Specify pages by using their paths. Enter one path per line. The '\*' character is a wildcard. Example paths are /user/ for the current user's page and /user/\* for every user page. <front> is the front page.' Below this field, there is a section titled 'Negate the condition' with two radio buttons: 'Show for the listed pages' (selected) and 'Hide for the listed pages'.

Page restriction visibility is probably the most common visibility setting used for blocks. Especially with Drupal 8 introducing the ability to reuse blocks, being able to control what page a block displays on is important to make sure that a block is not duplicated.

## Role restriction

The last way to restrict block content is by role. A role is defined by the site administrator and generally consists of an administrator, editor, contributor, an authenticated user, and an anonymous user. Visibility to block content can be restricted by selecting the specific role, as shown in the following image:



Role-specific visibility can be useful to display admin-only content or membership content to authenticated users without the anonymous user seeing it.

## Creating a custom block

So far, we have worked with system-generated blocks. However, with the introduction of fieldable blocks in Drupal 8, we now have the ability to create custom blocks. Custom blocks are quite powerful and will be used to display content in ways not possible previously without a contributed module.

We can create a custom block by navigating back to the **Block layout** admin located at /admin/structure/block and following these steps:

1. Locate the **Sidebar second** region.
2. Click on the **Place block** button.
3. Click on the **Add custom block** button.

We are now presented with the **Add custom block** screen that will allow us to create a default custom block that includes a **Block description** and a **Body** field, as shown in the following image:

The screenshot shows the 'Add custom block' interface. At the top, there's a title bar with the text 'Add custom block' and a star icon. Below it is a navigation bar with 'Home'. The main area has two sections: 'Block description' (marked with a red asterisk) and 'Body'. The 'Block description' section contains a text input field with placeholder text: 'A brief description of your block.' The 'Body' section contains a rich text editor toolbar with buttons for bold (B), italic (I), link (link icon), image (image icon), and other text formats. Below the toolbar is a large text area for content entry. At the bottom of the form, there's a 'Text format' dropdown set to 'Basic HTML' and a link 'About text formats'. The 'Revision information' section includes a 'New revision' button, a checked checkbox for 'Create new revision', and a 'Revision log message' text area with placeholder text: 'Briefly describe the changes you have made.' A blue 'Save' button is located at the bottom left.

We can continue filling out our custom block by entering the following values:

- **Block description:** Our custom block
- **Body:** This is some basic content

Click on the **Save** button to proceed to the **Configure block** screen as follows:

The screenshot shows the 'Configure block' interface. At the top, there's a success message: 'Basic block Our custom block has been created.' Below it, the 'Block description' is set to 'Our custom block'. The 'Title' field contains 'Our custom block' with a machine name of 'ourcustomblock'. The 'Display title' checkbox is checked. Under 'Visibility', the 'Content types' section has 'Pages' selected under 'Not restricted'. There are also sections for 'Roles' and 'Regions'. The 'Region' dropdown is currently set to '- None -'. A note below says 'Select the region where this block should be displayed.' At the bottom is a blue 'Save block' button.

We can complete the creation of our custom block by leaving the defaults and clicking on the **Save block** button.

We have now created our first custom block, and if we scroll to the bottom of the Block layout admin, we will see our block displayed in the **Disabled** section, as shown in the following image:

The screenshot shows the 'Disabled' section of the Block layout admin. It lists the 'Our custom block' entry. To its right is a 'Place block' button. Below the list are buttons for 'Custom' and 'None' regions, and a 'Configure' button with a dropdown arrow.

At this point, we can place our custom block within any region by using another method of placing blocks, the **Region** dropdown. The **Region** dropdown is visible within each region and allows a block to quickly move another region by selecting the specified region from the dropdown.

Let's move our custom block into the **Sidebar second** region by following these steps:

1. Select **Sidebar second** from the **Region** dropdown.
2. Click on the **Save blocks** button to save our changes.

If we navigate to the homepage, we will see our new custom block displayed in the right sidebar directly under our other block.

## Managing custom blocks

If we want to edit the block content of a custom block, we will not find it by selecting the **Configure block** button in its context menu or by clicking on the **Configure** button next to the block from the **Block layout** page. Custom blocks can only be configured from the custom block library located at `/admin/structure/block/block-content` or by clicking on the **Custom block library** tab from the **Block layout** admin.

## Exploring the custom block library

The **Custom block library** tab displays any custom blocks that have been created. It is from here that we can **Edit** any custom block:

The screenshot shows the 'Custom block library' page. At the top, there are tabs for 'Block layout' (which is active) and 'Custom block library'. Below the tabs, there are two buttons: 'Blocks' and 'Types'. A breadcrumb navigation path is visible: Home > Administration > Structure > Block layout. A note below the path states: 'Blocks in the block library belong to block types, each with its own fields and display settings. After creating a block, place it in a region from the Block layout page.' A blue button labeled '+ Add custom block' is present. Below this, there is a 'Block description' input field and an 'Apply' button. A table lists the existing custom blocks. The table has columns: BLOCK DESCRIPTION, BLOCK TYPE, UPDATED, and OPERATIONS. One row is shown: 'Our custom block' (BLOCK DESCRIPTION), 'Basic block' (BLOCK TYPE), '12/23/2015 - 20:55' (UPDATED), and an 'Edit' button with a dropdown arrow (OPERATIONS).

| BLOCK DESCRIPTION | BLOCK TYPE  | UPDATED            | OPERATIONS                                |
|-------------------|-------------|--------------------|-------------------------------------------|
| Our custom block  | Basic block | 12/23/2015 - 20:55 | <a href="#">Edit</a> <a href="#">More</a> |

Clicking on the **Edit** button will bring up the **Edit custom block** page where any content, including the **Block description**, **Body**, or additional fields, can be changed.

The **Custom block library** consists of both the **Blocks** tab, which displays all custom blocks, and the **Types** tab, which displays the various block types that have been created. A block type is similar to a content type and contains a lot of the same traits as a content type. Let's take a closer look at the custom block type.

## Exploring block types

Selecting the **Types** tab from the **Custom block library** exposes that we currently have a single block type named **Basic block**.

| BLOCK TYPE  | DESCRIPTION                                | OPERATIONS                    |
|-------------|--------------------------------------------|-------------------------------|
| Basic block | A basic block contains a title and a body. | <a href="#">Manage fields</a> |

A Basic block contains a title and a body field, similar to that of the Page content type. However, we now have the ability to manage both the View mode of a block as well as to manage the fields a block can have. This new functionality allows us to extend the normal block way further than before. Let's take a look at how we can manage the fields of a basic block by adding a new field.

## Managing fields

The minute we click on the **Manage fields** button, we are seeing something quite familiar to us, the Fields UI. The Fields UI allows us to manage existing fields as well as add new fields, as shown in the following image:



The screenshot shows the 'Manage fields' page for a 'Basic' block type. At the top, there are tabs: 'Edit' (selected), 'Manage fields' (highlighted in blue), 'Manage form display', and 'Manage display'. Below the tabs, the breadcrumb navigation shows: Home » Administration » Structure » Block layout » Custom block library » Basic. A prominent blue button labeled '+Add field' is visible. The main content area displays a table with one row. The columns are labeled 'LABEL', 'MACHINE NAME', 'FIELD TYPE', and 'OPERATIONS'. The row contains the values: 'Body', 'body', 'Text (formatted, long, with summary)', and an 'Edit' button with a dropdown arrow. The entire interface has a clean, modern design with a light gray background and white text.

The Fields UI consists of the following:

- **LABEL:** This is a descriptive name of our field that will be used as a label when inputting content into this field.
- **MACHINE NAME:** The machine name is a lower case field name used by Drupal to distinguish this field from others.
- **FIELD TYPE:** This allows us to choose from various field types, such as date, file, text, and more.

We can add an additional field to our basic block type now by following these steps:

1. Click on the **Add field** button.
2. Select **Image** from the **Add a new field** dropdown.
3. Enter a **Label** of **Featured Image**.
4. Click on the **Save and continue** button.

**Add field** 

Home » Administration » Structure » Block layout » Custom block library » Basic » Manage fields

Add a new field:

Image

Label \*

Featured Image Machine name: field\_featured\_image [Edit]

**Save and continue**

5. Leave the default settings on the field settings page.
6. Click on the **Save field settings** button.
7. Leave the default settings on the Edit page.
8. Click on the **Save settings** button.
9. We have successfully added a new field to the Basic block type that all future custom blocks can use to add a Featured Image to, as shown in the following image:

| LABEL          | MACHINE NAME         | FIELD TYPE                           | OPERATIONS                                                                                                 |
|----------------|----------------------|--------------------------------------|------------------------------------------------------------------------------------------------------------|
| Body           | body                 | Text (formatted, long, with summary) | <a href="#">Edit</a>  |
| Featured Image | field_featured_image | Image                                | <a href="#">Edit</a>  |

We could continue to add additional fields as needed, but we will stop at this point and focus on how to manage the display of custom blocks and their respective **View modes**.

## Managing display

The **Custom block library** not only allows us to manage fields using the Fields UI, but we can also manage the display of the fields. Continuing with **Our custom block**, we will click on the **Manage display** tab. Managing the display of a block is exactly like managing the display of a content type.

| FIELD            | LABEL      | FORMAT  |                |
|------------------|------------|---------|----------------|
| ⊕ Body           | - Hidden - | Default |                |
| ⊕ Featured Image | Above      | Image   | Original image |

No field is hidden.

► CUSTOM DISPLAY SETTINGS

**Save**

From the **Manage display** page, we can manage several display options ranging from showing or hiding the label to configuring the format of a field. The format options will vary based on the field type it is referring to. Feel free to play around with the various settings and preview the changes.

The final thing to point out is the custom display settings or view modes that a block can have. If we expand the **CUSTOM DISPLAY SETTINGS** field, we will only see one **View mode** which is called **Full**. Drupal 8 allows us to create additional View modes for use with custom blocks the same way we create content types.

If we navigate to /admin/structure/display-modes/view or using the Admin menu, click on **Structure**, **Display Modes**, and finally **View modes**. We will see all the options available to create additional **View modes** for **Content** as well as **Custom blocks**.

| Content                          |                      |
|----------------------------------|----------------------|
| NAME                             | OPERATIONS           |
| Full content                     | <a href="#">Edit</a> |
| RSS                              | <a href="#">Edit</a> |
| Search index                     | <a href="#">Edit</a> |
| Search result highlighting input | <a href="#">Edit</a> |
| Teaser                           | <a href="#">Edit</a> |

[Add new Content view mode](#)

| Custom block |                      |
|--------------|----------------------|
| NAME         | OPERATIONS           |
| Full         | <a href="#">Edit</a> |

[Add new Custom block view mode](#)

The **View modes** page contains several prebuilt displays based on **Content**, **User**, **Taxonomy term**, **Comment**, and **Custom Block**. If we scroll down to the **Custom block** section, we will see **Full** display. We can **Edit** or **Delete** this display or add additional displays by clicking on the **Add new Custom Block view mode** link. New view modes are simply containers to hold the display of our fields when we specify one to be used back on the **Manage display** page.

## Summary

We have covered a lot of information surrounding the administration of themes in Drupal 8. So, let's recap exactly what we have covered so far:

- We answered the question, "What is a theme?"
- We took an in-depth look at the Appearance interface and how we can use it to install, uninstall, and configure settings, including toggling the display of certain page elements.
- We learned how to work with the logo image settings and shortcut icon settings that can be configured for themes.
- We worked with prebuilt themes and learned where to find them, how to install them using the Drupal admin, and manually using a theme archive.
- Finally, we took a detailed look at blocks and regions, including how to configure blocks and control their appearance using the new custom blocks layout to add additional fields and view modes.

In the next chapter, we will begin dissecting a theme. This includes taking a look at the file and folder structure of a theme, configuration management, and what makes up a core theme versus a custom theme. We will also explore the new `info.yml` configuration, the role of assets in Drupal, and how templates function with an introduction to Twig. Finally, we will follow up with looking closer at the theme file and how it can control variables within our templates.

# 3

## Dissecting a Theme

Drupal 8 provides us, as developers and designers, with a unique opportunity to change the appearance of the output content. We have the ability to manage the configuration from the admin user interface as well as work with the actual templates and variables that output the HTML, CSS, and JavaScript. To get a better understanding, we will take a look at dissecting a theme, as we cover the following:

- Having a proper development environment is important when working with themes, so we will take a look at the steps involved in configuring our local environment.
- Next, we will compare the similarities and differences between core default themes and custom themes while looking at how configuration has changed in Drupal 8 with the introduction of the `info.yml` file.
- Being able to breakdown how the metadata of the `info.yml` works in conjunction with general information, libraries, and regions will ensure that we have a better understanding of Drupal's theme configuration.
- The role of templates, where to find core templates, and the process of overriding templates plays a major role in theming, so we will introduce ourselves to the Twig templating system.
- Finally, we will look at the role the theme file plays in manipulating template variables and how we can use it to our advantage when working with the content.

## Setting up a local development environment

Everything we will be creating with Drupal revolves around having a proper local development environment, and with the move from Drupal 7 to Drupal 8, there has been a more aligned workflow between local development, staging, and production environments. This is evident with the introduction of the additional files and services that are now included within our `sites` folder, all aimed at allowing us to have more control during development.

For example, while creating a theme, we will often find ourselves having to clear Drupal's cache to see any changes that we applied. This includes render cache, page cache, and Twig cache. Having to go constantly through the process of clearing cache not only takes up time but also becomes an unnecessary step.

Let's discuss the setup and configuration of our local environment to use a local settings file that will allow us to disable CSS/JS aggregation, disable render and page cache, and enable Twig debugging.

## Managing sites/default folder permissions

The first step in configuring our local development environment requires making changes to various files that will live within our `sites/default` folder or need to be placed within it. By default, Drupal protects the `sites/default` folder and any files within it from being written to. We will need to modify the permissions to make sure that the owner of the folder has read, write, and execute permissions while everyone else has only read and execute.

These steps assume that we are familiar with managing permissions, but for further reference, we can take a look at <http://www.wikihow.com/Change-File-Properties>.

Once we have made the required permission changes, we can proceed to creating and configuring our local settings file.

## Configuring settings.local.php

We are all familiar with Drupal's `settings.php` file. However, in Drupal 8, we can now have different configurations per environment by creating a `settings.local.php` file that the default `settings.php` file can reference.

We can follow these simple steps to create and enable the new file:

1. First, we will need to copy and rename `example.settings.local.php` located in the `sites` folder to `settings.local.php` within the `sites/default` folder.
2. Next, we need to open `settings.php` located in our `sites/default` folder and uncomment the following lines:

```
if (file_exists(__DIR__ . '/settings.local.php')) {
 include __DIR__ . '/settings.local.php';
}
```
3. Save the changes to our `settings.php` file.

Uncommenting the lines allows `settings.php` to include our new `settings.local.php` file within our default settings while allowing us to manage different environment configurations.

## Disabling CSS and JS aggregation

As part of the performance settings, Drupal will aggregate both CSS and JS to optimize bandwidth. During development, we are not concerned with bandwidth as we are developing locally. Using a `settings.local.php` file, CSS and JS aggregation are disabled for us. However, if for some reason we want to re-enable aggregation, we would simply change the TRUE values to FALSE as follows:

```
/**
 * Disable CSS and JS aggregation.
 */
$config['system.performance']['css']['preprocess'] = FALSE;
$config['system.performance']['js']['preprocess'] = FALSE;
```

## Disabling render and page cache

Another configuration option we can address while having the `settings.local.php` file open is render and page cache. This setting allows us to avoid having to clear Drupal's cache constantly when we make a file change.

Locate and uncomment the following lines:

```
$settings['cache']['bins']['render'] = 'cache.backend.null';
$settings['cache']['bins']['dynamic_page_cache'] = 'cache.backend.
null';
```

## Disabling test modules and themes

One last configuration we will want to make to our `settings.local.php` file has to do with test modules and themes. By default, our local settings file enables the display of various modules and themes meant for testing purposes only. We can disable them by changing the following TRUE value to FALSE:

```
$settings['extension_discovery_scan_tests'] = FALSE;
```

With all of these changes made, we will want to make sure that we save our `settings.local.php` file. Now, each time we refresh our browser, we will get a new copy of all files without the need to clear Drupal's cache to see any changes.

In some instances, we may need to rebuild Drupal's cache before the above settings will work. If that is the case, we can navigate to `/core/rebuild.php`, which will fix any issues.

Now that we have our local development environment configured its time we took a closer look at default versus custom themes.

## Default themes versus custom themes

We have a couple of options when it comes to what themes we want to use in Drupal 8; that is default themes, such as Bartik and Seven that ship with Drupal, or custom themes that a designer creates and which then get converted into themes. Both of these are similar in structure and configuration, which we will look closer at in a minute, but the main separation begins with the folder structure.

## Folder structure and naming conventions

In Drupal 8, the folder structure is changed to make it more logical. Everything that ships with Drupal now resides in a `core` folder including the default themes, which are now contained within the `core/themes` folder. However, any themes that we download or develop ourselves now reside within the `themes` folder.

The folder structure comprises the following:

- **Default themes:** These themes reside in the `core/themes` directory and include Bartik, classy, seven, stable, and stark.
- **Custom themes:** These themes reside in the `themes` directory at the root level of our Drupal installation and will contain any contributed themes or custom themes.

Before we can begin creating our own custom themes, we need to have a better understanding of how themes are configured and exactly how they let Drupal know where to display content and how the content should look.

## Managing configuration in Drupal 8

Theme configuration in Drupal 8 has now adopted YAML. YAML is a human-friendly data serialization standard used by many programming languages, including Symfony, which Drupal 8 is now built on. With this adoption, the syntax to create an info file has now changed as well. One important concept when creating or editing any \*.yml file is that proper indentation is required. Failure to properly indent configuration can lead to errors or to the configuration not loading at all. We can dive deeper into the specifics of YAML and find out more detailed information at the Symfony website ([http://symfony.com/doc/current/components/yaml/yaml\\_format.html](http://symfony.com/doc/current/components/yaml/yaml_format.html)).

## Reviewing the new info.yml file

The `Info.yml` file is required when creating any theme. It helps notify Drupal that a theme exists and provides information to the Appearance interface that a theme is available to install. We will be working with `*.info.yml` files when creating our first theme, so let's take a look at the makeup of a basic `example.info.yml` file:

```
name: Example
description: 'An Example theme.'
type: theme
package: Custom
base theme: classy
core: 8.x

libraries:
 - example/global-styling

regions:
 header: Header
 primary_menu: 'Primary menu'
 secondary_menu: 'Secondary menu'
 page_top: 'Page top'
 page_bottom: 'Page bottom'
 highlighted: Highlighted
 breadcrumb: Breadcrumb
 content: Content
 sidebar_first: 'Sidebar first'
 sidebar_second: 'Sidebar second'
 footer: 'Footer'
```

At first glance, the `example.info.yml` file is logical in structure and syntax. Starting from the top and moving our way down, the file is broken down by different sections of metadata containing general information, libraries, and regions. This information is described using a key: value format. We should begin with understanding how basic metadata works.

## Metadata

The metadata contained within any themes `*.info.yml` file helps to describe what type of document it is. In our case, it begins to describe a theme, including the name, description, and the version of Drupal the theme works with. Some metadata is required for the theme to function properly, so let's explore the keys in more detail as follows:

- **name** (*required*): This is the name of our theme.
- **type** (*required*): This is the type of extension (theme, module, or profile).
- **base theme** (*required*): This is the theme that the current theme is inheriting. In most cases, it is recommended we reference either `classy` or `stable` as our base theme. If we choose not to reference a based theme, then we will need to set the value to false (`base_theme: false`).
- **description** (*required*): This is the description of our theme.
- **package** (*optional*): This is used to group similar files when creating modules.
- **version** (*optional*): This is created by packaging script.
- **core** (*required*): This specifies the version of Drupal that a theme is compatible with.

One of the most common mistakes when first creating a `*.info.yml` file is forgetting to change the `core` value to `8.x`. Failure to set this value will result in the theme not being displayed within the Appearance interface in the admin.

The next section of a `*.info.yml` file allows us to manage assets (CSS or JS) using the new concept of libraries.

## Libraries

Drupal 8 introduced a new, high-level principle of managing assets using a libraries configuration file that can be loaded globally or on a per page basis. This concept helps to improve frontend performance as well as ensure that any dependencies that a particular asset needs is loaded properly. One advantage of this is that jQuery no longer loads on every page as it did in the previous versions of Drupal.

The concept of a `*.libraries.yml` configuration file also means that the style sheets and scripts properties that we may have been familiar with in Drupal 7 no longer exist. Instead, the process to manage assets includes saving any CSS or JS files to our theme's `css` or `js` folder and then defining a library file that references the files we want to use in our theme.

## Defining a library

When defining a `*.libraries.yml` file for a theme, each library will reference the location of individual CSS or JS files and be organized using the SMACSS (<https://smacss.com/>) style categorization.

- **Base:** This defines CSS reset/normalize plus HTML element styling
- **Layout:** This defines the macro arrangement of a web page, including any grid system
- **Component:** This defines the discrete, reusable UI elements
- **State:** This defines the styles that deal with client-side changes to components
- **Theme:** This is purely visual styling for a component

In most cases, a simple library reference will follow the theme categorization. For example, if we wanted to create an `example.libraries.yml` file that included assets for CSS and JS, we would create a library that pointed to our assets, as shown here:

```
libraryname:
 css:
 theme:
 css/style.css: {}
 css/print.css: { media: print }
 js:
 js/scripts.js
```

We would then reference the library within our `example.info.yml` configuration simply by adding the following:

```
libraries:
 - example/libraryname
```

This would result in Drupal adding to every page both CSS and JS files contained in our library. Where this becomes powerful is in the management of assets, as we would only ever need to make modifications to our `example.libraries.yml` file if we ever needed to add or remove assets.

## Overriding libraries

Libraries can also be overridden to modify assets declared by other libraries, possibly added by a base theme, by a module, or even the Drupal core. The ability to override libraries includes removing as well as replacing assets altogether. The same way we reference a library from our `*.info.yml` file, we can override libraries by adding the following:

```
libraries-override:

 # Replace an entire library.
 core/drupal.vertical-tabs: example/vertical-tabs

 # Replace an asset with another.
 core/drupal.vertical-tabs:
 css:
 component:
 misc/vertical-tabs.css: css/vertical-tabs.css

 # Remove an asset.
 core/drupal.vertical-tabs:
 css:
 component:
 misc/vertical-tabs.css: false

 # Remove an entire library.
 core/modernizr: false
```

In this case, the `libraries-override` configuration achieves something different for each line. Whether it is replacing an entire library or removing an asset, we now have the flexibility to control assets like never before.

## Extending libraries

Libraries can also be extended to allow overriding CSS added by another library without modifying the original files. This can be done by adding the following to our `*.info.yml` configuration as follows:

```
libraries-extend:
 core/drupal.vertical-tabs:
 - example/tabs
```

In this case, the `libraries-extend` configuration is extending Drupal's own `core.libraries.yml` file and extending the `drupal.vertical-tabs` library with additional styling.

While we now have a general understanding of how libraries are defined, overridden, and extended, we have only dealt with libraries globally loaded into our Drupal instance using our configuration file. However, there are two more methods to include assets within a page directly, without the need to add it to every page.

## Attaching a library

In many cases, we may be developing some CSS or JS functionality that is specific to an individual page. When we are presented with this requirement, we have the ability to attach a library to a page using two different methods.

### Using Twig to attach a library

While we will be learning all about Twig a little later in the chapter, we need to pause for a moment to reference a Twig function named `{{ attach_library() }}`. This function allows us to add to any Twig template a library that may include CSS or JS that will load on that page only.

For example, if we wanted to add the Slick Carousel (<http://kenwheeler.github.io/slick/>) to our page, we may define the library within our `example.libraries.yml` file as follows:

```
Slick
slick:
 version: VERSION
 css:
 theme:
 vendor/slick/slick.css: {}
 js:
 vendor/slick/slick.min.js: {}
 dependencies:
 - core/jquery
```

We could then turn around and add the following to our Twig template:

```
{{ attach_library('example/slick') }}
```

This provides us with some nice functionality to define individual libraries for various user functions and also to have those assets used wherever we choose to attach them.

## Using the preprocess functions to attach a library

Another method to attach a library to an individual page depends on creating a \*.theme file, which allows us to use preprocess functions to manipulate page variables. We will learn a lot more about creating a \*.theme file a little later in the chapter, but it's important to note that we could attach the same Slick Carousel to our homepage without globally calling it by using a preprocess function, as shown in the following example:

```
function example_preprocess_page(&$variables) {
 if ($variables['is_front']) {
 $variables['#attached']['library'][] = 'example/slick';
 }
}
```

Here, we are checking to see whether we are on the homepage of our website and attaching our Slick library using the #attached library array. Again, this may seem a little bit advanced at this point but does merit mentioning.

The last section we will want to cover when working with any \*.info.yml file is about regions that can be defined for the layout of our theme.

## Regions

Regions play a critical part in theming, as Drupal needs to know exactly where content can be displayed. This has an impact on what regions are visible to the Block layout for both system blocks and custom blocks that we may want to use. If we do not specify any regions within our \*.info.yml file, then Drupal will provide us with regions by default.

If we decide to add additional regions to our theme, we must also add the defaults or else we will not have access to them. Let's take a look at how this is implemented:

```
regions:
 header: Header
 primary_menu: 'Primary menu'
 secondary_menu: 'Secondary menu'
 page_top: 'Page top'
 page_bottom: 'Page bottom'
 highlighted: Highlighted
 breadcrumb: Breadcrumb
 content: Content
 sidebar_first: 'Sidebar first'
 sidebar_second: 'Sidebar second'
 footer: 'Footer'
```

The value for each key is what is displayed in the Block layout within the Drupal UI and can be named whatever we want to name it. We can add additional regions based on our theme as needed. We will look at this in more detail in *Chapter 4, Getting Started – Creating Themes*.

Now that we have covered the basics of theme configuration, it's time for us to set up a local development environment that will enable us to work with files and templates without worrying about having to clear the Drupal cache or guess what Twig templates are being used.

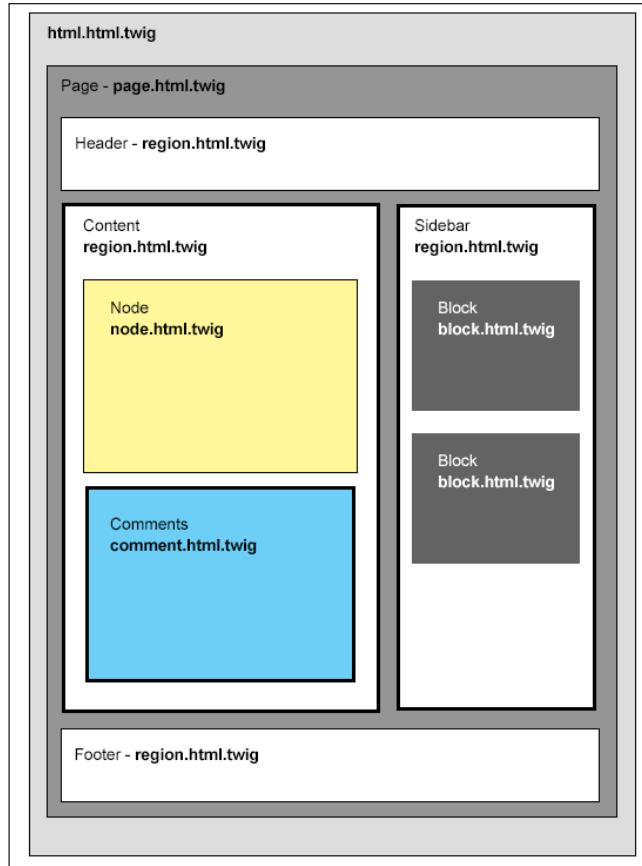
## **The role of templates in Drupal**

We may have heard the term "template" before when talking to someone about theming and Drupal. But what exactly is a template? We can think of a template as a text file no different from any HTML document that provides a method for separating the presentation layer from the business logic. In traditional PHP websites, we have the ability to mix PHP with HTML and CSS, which makes managing web pages both difficult and dangerous. Drupal provides us with the ability to use templating engines to enforce the separation of the two, so we can begin to focus more on the HTML and CSS and worry less about the PHP.

## **How templates work**

In general, templates can contain HTML markup and PHP variables that output content contained within a Drupal database. Templates can be as small as a few lines of HTML that hold the presentational layer for a block that is displayed in a region on the page, or the actual page itself, with containers defined for header, content, and so on.

To get a better idea of what this looks like, let's take a look at the following image:



If we break down the image into logical sections of a website, we can begin to get an idea of what constitutes a template. A template can be any of the following:

- **HTML wrapper:** This contains the top-level HTML markup, including title, metadata, style sheets, and scripts, and it is commonly referred to as `html.html.twig`.
- **Page wrapper:** This contains the content generally found between the body tags of an HTML document, and it is commonly referred to as `page.html.twig`.

- **Header:** This is also known as a region, generally containing the header content of our web page. This can be part of the `page.html.twig` template or may reside in a region specified within our configuration file. This is commonly referred to as `region.html.twig`.
- **Content:** This is also considered a region, generally containing our main content. This can consist of multiple subcontent regions, such as nodes and comments. Nodes and comments each have their own respective templates referred to as `node.html.twig` and `comment.html.twig`.
- **Sidebar:** This is also considered a region. This can contain blocks of content. Blocks are either created by the end user or by Drupal itself. The content within these blocks generally resides within `block.html.twig`.
- **Footer:** This is another region containing HTML content as well as blocks of content.

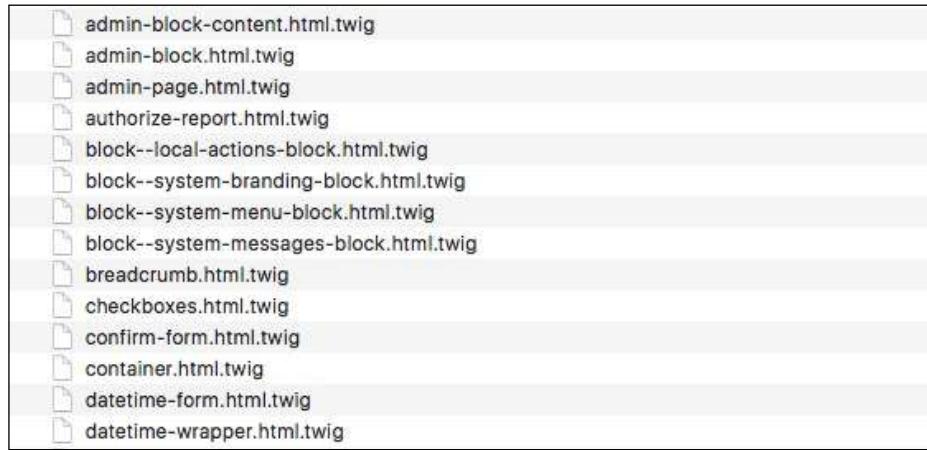
Drupal and the theme engine it uses to convert the markup and variables into HTML interpret each individual template or series of templates. We have full control over what is output using the new Twig templating engine.

Once we begin theming, we will start to see a pattern of how templates are used, and as we gain more experience, we will find ourselves using less and less templates. However, to begin with, we will build examples of each to help clarify their functionality within Drupal.

## Where to find templates

The nice thing about Drupal is that, by default, the core system provides us with all the templates we need to use. So, knowing where to find the core templates is important because it will allow us to copy them into our own theme folder to override with our own markup.

Let's begin by opening up our Drupal instance in MAC Finder or Windows Explorer and browsing to the `core/modules` folder. Contained within this folder are the core modules that make up Drupal, along with their respective templates. Most of the core templates will be located in the `core/modules/system/templates` folder, as shown in the following image:



If we browse the contents of the `templates` folder, we will see some of the most common templates we will be using including the following:

- `html.html.twig`: HTML wrapper
- `page.html.twig`: Page wrapper
- `region.html.twig`: Region wrapper

Three more template folders that we need to be aware of are:

- `core/modules/node/templates`: This contains the templates for nodes
- `core/modules/comment/templates`: This contains the comment templates
- `core/modules/block/templates`: This contains the templates for blocks

We will find ourselves frequently overriding templates, so we need to make sure that we know where to find any Twig template that we will be theming.

Most of us have done some PHP development or are at least familiar enough with it to work with the variables that Drupal outputs. So, as we look at the templates, we should be noticing that the files don't end with a file extension of .php but instead end with a file extension of .twig. In fact, if we were to look at the `html.html.twig` template located in the `core/modules/system/templates` folder, we won't even find PHP syntax inside it:

```
<!DOCTYPE html>
<html{{ html_attributes }}>
 <head>
 <head-placeholder token="{{ placeholder_token|raw }}>
 <title>{{ head_title|safe_join(' | ') }}</title>
 <css-placeholder token="{{ placeholder_token|raw }}>
 <js-placeholder token="{{ placeholder_token|raw }}>
 </head>
 <body{{ attributes }}>

 {{ 'Skip to main content'|t }}

 {{ page_top }}
 {{ page }}
 {{ page_bottom }}
 <js-bottom-placeholder token="{{ placeholder_token|raw }}>
 </body>
 </html>
```

Instead, we will see general HTML markup along with the Twig syntax that will output content within its place. We will take a closer look at Twig in a moment. First, we will try our hand at creating a basic theme.

## Creating our first basic theme

Now that we have reviewed the basics of how a theme is constructed, there is no better time than the present to create our first basic theme. We will begin by creating a theme named `twig` that we will use to work with exploring how Twig and the Theme system works in Drupal 8.

In order to make sure that we all are working from the same baseline, let's open up the `Chapter03/start` folder located in the exercise files and select the `drupal8.sql` database file. We will use this database snapshot to restore our current database instance. Refer to *Chapter 1, Setting Up Our Development Environment* for instructions on how to perform a database restore.

Now that we all have the same baseline Drupal instance, we can navigate to our Drupal 8 folder using MAC Finder or Windows Explorer and follow these next six steps to create a theme.

## Step One – creating a new folder

Create a new folder under our themes folder and call it twig, as shown in the following image:



## Step two – create an info file

Create a new \*.info.yml file named twig.info.yml and add the following configuration information to the file:

```
name: Twig
type: theme
description: 'A Twig theme for demonstrating TWIG syntax'
core: 8.x
base theme: false
```

## Step three – copy core templates

Copy the html.html.twig and page.html.twig templates from the core/modules/system/templates folder and paste it into our themes/twig folder. Open up page.html.twig in our editor and replace the HTML structure below the comments with the following code:

```
<h1>Welcome to Twig</h1>
{{ page.content }}
```

## Step four – include a screenshot

Not always a required step but one that will definitely help is including a screenshot that displays or represents our theme within the Appearance admin. In general, we would generate a screenshot based on the finished theme, but because we are just starting out, we can copy an existing one from our exercise files.

Begin by navigating to the Chapter03/end folder and copy the screenshot.png file to our newly created themes/twig folder.

## Step five – installing our theme

Next, we will need to install our new theme by navigating to /admin/appearance and locating our new theme named **Twig** under the **Uninstalled themes** section. Click on the **Install and set as default** link to install our new theme, as shown in the following image:

| Uninstalled themes                                                                                                                                                                                                                                                      |                                                                                                                                                |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Stark 8.0.1</b><br/>An intentionally plain theme with no styling to demonstrate default Drupal's HTML and CSS. Learn how to build a custom theme from Stark in the Theming Guide.</p> <p><a href="#">Install</a>   <a href="#">Install and set as default</a></p> | <p><b>Twig</b><br/>A Twig theme for demonstrating Twig syntax.</p> <p><a href="#">Install</a>   <a href="#">Install and set as default</a></p> |

## Step six – Welcome to Twig

We have successfully created our first theme. Although there is not much to it, we can preview what our website looks like by browsing back to the home page of our Drupal instance. We should see our new theme displaying a message of **Welcome to Twig**, as shown in the following image:

With our new theme in place, we can begin taking a deeper look into Twig and all of the great features that Drupal 8 introduces to us with this new templating engine.

## Introducing Twig

Twig (<http://twig.sensiolabs.org>) is the new template engine introduced to Drupal 8 and is a companion to Symfony, the new PHP framework that Drupal 8 is built on. Twig provides us with a fast and secure way to separate content from PHP logic in a manner that makes it easier for non-developers to work with templates. To help us get a better feel in order to work with Twig, let's first dive into the steps involved in enabling Twig debugging.

## Enabling Twig debug

When Twig debugging is turned on within Drupal, we are able to trace which template is being used, where a template is located, and a list of suggested file names to override a template. This functionality is very advantageous and actually quite simple to set up by following these steps:

1. Open the `development.services.yml` file located in the `sites` folder.
2. Add the following lines to the bottom of the file:

```
parameters:
 twig.config:
 debug : true
 auto_reload: true
 cache: false
```
3. Save the file.
4. Clear Drupal's cache.

If we navigate back to the homepage and inspect the markup using Google Chrome's Developer Tools, we can now see Twig debug outputting information, as shown in the following image:



There are a couple of items we should make note of when Twig debugging is enabled:

- **FILE NAME SUGGESTIONS:** This displays suggestions to name Twig HTML templates and displays in the order of precedence in which Drupal folders would look for templates.
- **OUTPUT:** This displays the location of the template currently being displayed, which, in our case, is `themes/twig/page.html.twig`.

Remember that we will only see the debug output when we have the Twig debugging enabled as part of our local development environment. It is best to remember to disable debugging before moving a Drupal site to production. So now that we have an understanding of what Twig debug provides us with, let's begin writing some of our own Twig syntax, beginning with comments.

## Twig fundamentals

A Twig template outputs PHP with a template-oriented syntax using opening and closing curly brackets `{ { . . . } }`. This syntax interprets the variable between the brackets and outputs HTML in its place. The following are three kinds of delimiters in Twig that trigger an evaluation to take place:

- The first is Twig commenting, which uses the comment tag `{# . . . #}` to provide comments inline or around a section of HTML.
- Next is the print tag `{ { . . . } }`, which is used to print the result of an expression or variable. The print tag can be used by itself or within a section of HTML.
- The third tag is to execute a statement such as conditional logic, looping structures, or the assignment of values to variables and is expressed by using `{% . . . %}`.

Each of the three delimiters will be used when we do any type of theming projects within Drupal 8. We will find that they are just as simple as using any regular HTML element, and we will quickly be writing these tags.

## Commenting variables

We are familiar with HTML commenting such as `<!-- This is a comment -->`, which allows us to add descriptive text to our markup. We saw an example of this in the Twig debug output once we enabled it. Twig provides us with the ability to add comments as well using the `{# comment #}` syntax.

If we open `page.html.twig` within our editor, we can add a Twig comment by adding the following:

```
{# This is a comment in Twig #}
<h1>Welcome to Twig!</h1>
```

Once we save our template, refresh the browser and inspect the heading. We will note that we don't actually see the comment being displayed. Unlike HTML comments, Twig comments are meant to be hidden from browser output and are meant only for the developer.

## Setting variables

Twig can also assign values to variables using a technique named Assignment. Assignment uses the `set` tag to place a value into a variable, which can then be used later within a template to output the value.

Open `page.html.twig` and add the following above our heading:

```
{# Setting a variable #}
{%
 set name = 'Drupal'
}

{# This is a comment in Twig #}
<h1>Welcome to Twig!</h1>
```

If we save our template and refresh the browser, we will not see any changes to our HTML as we are only setting a variable but not using it anywhere in our document. So how do we then use a variable?

## Printing variables

Twig allows us to print variables by simply referencing them within our document using the `{{ variable }}` syntax to trigger the interpreter to replace the variable name with the value stored in it. We can try this by replacing the word `Twig` in our heading with the `name` variable.

Open `page.html.twig` and add the following:

```
{# Setting a variable #}
{%
 set name = 'Drupal'
}

{# This is a comment in Twig #}
<h1>Welcome to {{ name }}</h1>
```

If we save our template and refresh the browser, we will see that our heading now says **Welcome to Drupal**. The name variable we set has output the word **Drupal** in its place. This is the same technique that we will be using to output variables in our Twig templates to display content from Drupal. In fact, if we sneak a peek at our `html.html.twig` template, we will see a variety of twig variables being used to output content.

## Dumping variables

While theming in Drupal, we will be working with both simple and complex variables consisting of PHP arrays that contain multiple values. Knowing that there can be multiple values, it is sometimes useful to dump the contents of the variable to know exactly what we are working with. The `{{ dump() }}` function allows us to view information about a template variable and is only available to us when Twig debugging is turned on. Let's take our `name` variable for instance and dump the contents to see what it contains.

Open `page.html.twig` and add the following to the bottom of the template:

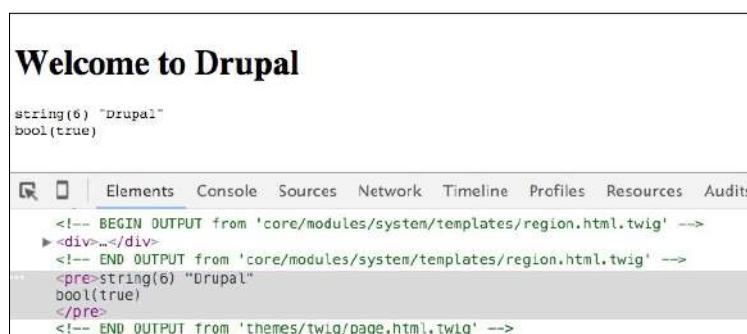
```
{# Dumping variables #}
{{ dump(name) }}
```

If we save our template and refresh the browser, we will now see the `name` variable being dumped to the page displaying some additional info about our variable.

Using the `dump()` function, we can introspect more than one variable at a time by passing multiple arguments. Let's try this by adding an additional Drupal variable named `is_front`, as shown in the following code sample:

```
{# Dumping variables #}
<pre>{{ dump(name, is_front) }}</pre>
```

If we save our template and refresh the browser, we will now see the `is_front` variable being dumped to the page as well as displaying some more information, as shown in the following image:



By now, we should be comfortable working with a Twig template and variables. However, we can do much more with Twig than just print variables though. We can also apply filters to variables to achieve different functionality.

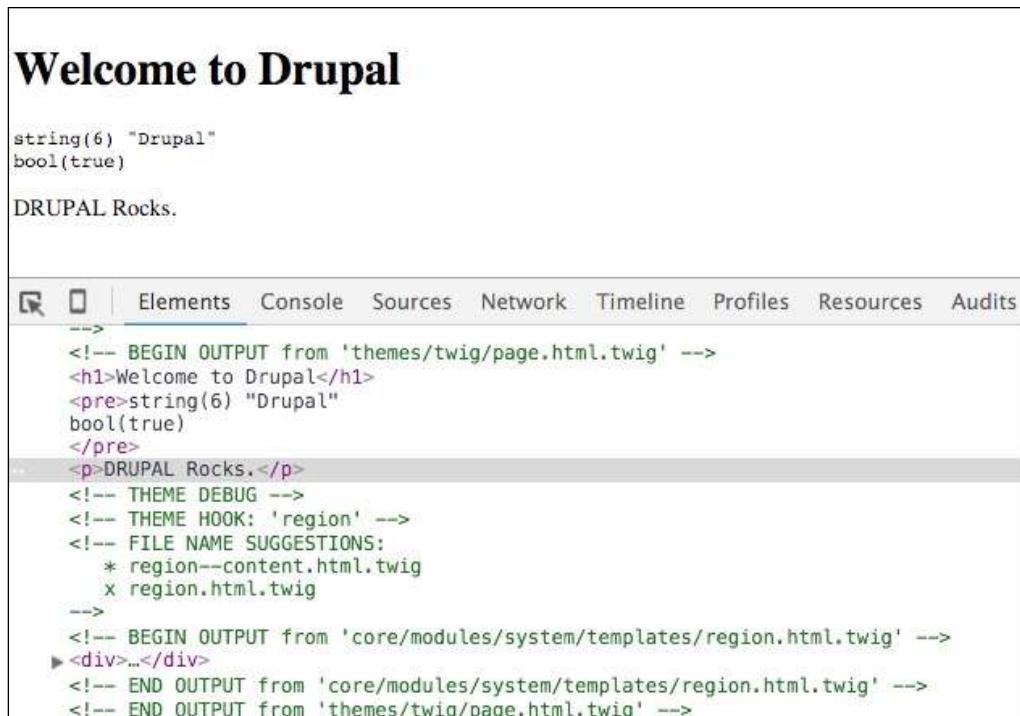
## Filters

Filters provide us with a way to modify variables. The filters are generally separated by a pipe character (|) and may accept arguments depending on the filter's purpose. Twig provides us with currently 30+ filters that we can apply to variables. Let's try out filters now by applying an uppercase filter on our name variable.

Open `page.html.twig` and add the following:

```
{# Apply filter to name variable #}
<p>{{ name|upper }} Rocks.</p>
```

If we save our template and refresh the browser, we will now see that the `name` variable is converted to uppercase inside our paragraph tag, as shown in the following image:



The screenshot shows the Chrome DevTools Elements tab. At the top, there is a rendered **Welcome to Drupal** page with the text "DRUPAL Rocks." in a paragraph. Below the rendered page, the DevTools interface shows the raw Twig template code. The line `<p>DRUPAL Rocks.</p>` is highlighted with a gray background, indicating it is the current selection in the code editor. The rest of the template code is visible above and below this line.

```
string(6) "Drupal"
bool(true)

DRUPAL Rocks.

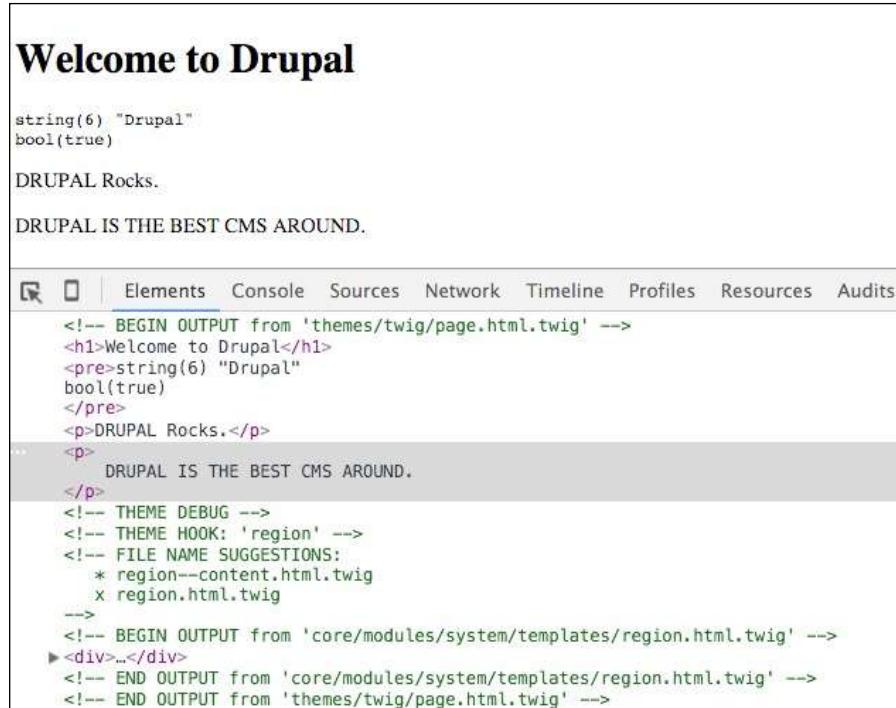
<!-- BEGIN OUTPUT from 'themes/twig/page.html.twig' -->
<h1>Welcome to Drupal</h1>
<pre>string(6) "Drupal"
bool(true)
</pre>
<p>DRUPAL Rocks.</p>
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'region' -->
<!-- FILE NAME SUGGESTIONS:
 * region--content.html.twig
 * region.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/system/templates/region.html.twig' -->
▶ <div>...</div>
<!-- END OUTPUT from 'core/modules/system/templates/region.html.twig' -->
<!-- END OUTPUT from 'themes/twig/page.html.twig' -->
```

We can also use filters to wrap sections of HTML and variables, which apply the filter to more than one item at a time. An example of this would be if we wanted to uppercase a whole paragraph versus just the name variable.

Open `page.html.twig` and add the following:

```
{% filter upper %}
<p>{{ name }} is the best cms around.</p>
{% endfilter %}
```

If we save our template and refresh the browser, we will now see that the entire paragraph including the `name` variable is converted to uppercase, as shown in the following image:



The screenshot shows a browser's developer tools with the "Elements" tab selected. The main pane displays the rendered HTML output, which includes an 

# element, a pre element containing the string "Drupal", and a p element containing the text "DRUPAL Rocks.". Below this, a smaller p element contains the text "DRUPAL IS THE BEST CMS AROUND.". The bottom pane shows the raw Twig template code. The `{% filter upper %}` and `{% endfilter %}` blocks are used to uppercase the entire content of the paragraph. The rendered output in the main pane reflects this, with the entire paragraph being uppercase.

```
string(6) "Drupal"
bool(true)

DRUPAL Rocks.

DRUPAL IS THE BEST CMS AROUND.

<!-- BEGIN OUTPUT from 'themes/twig/page.html.twig' -->
<h1>Welcome to Drupal</h1>
<pre>string(6) "Drupal"
bool(true)
</pre>
<p>DRUPAL Rocks.</p>
<p>
DRUPAL IS THE BEST CMS AROUND.
</p>
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'region' -->
<!-- FILE NAME SUGGESTIONS:
 * region--content.html.twig
 * region.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/system/templates/region.html.twig' -->
▶ <div>...</div>
<!-- END OUTPUT from 'core/modules/system/templates/region.html.twig' -->
<!-- END OUTPUT from 'themes/twig/page.html.twig' -->
```

This is just an example of one of the many filters that can be applied to variables within Twig. For a detailed list of filters, we can refer to <http://twig.sensiolabs.org/doc/filters/index.html>.

## Control structures

There will be situations while theming with Twig where we will need to check whether a variable is `True` or `False` or need to loop through a variable to output multiple values contained in an array.

Control structures in Twig allow us to account for these types of functions using `{% ... %}` blocks to test for expressions and traverse through variables that contain arrays. Each control structure contains an opening and closing tag similar to PHP logic. Let's take a look at a couple of the most commonly used control structures starting with the `if` tag used to test an expression.

Open `page.html.twig` and add the following:

```
{# Conditional logic #}
{% set offline = false %}

{% if offline == true %}
 <p>Website is in maintenance mode.</p>
{% endif %}
```

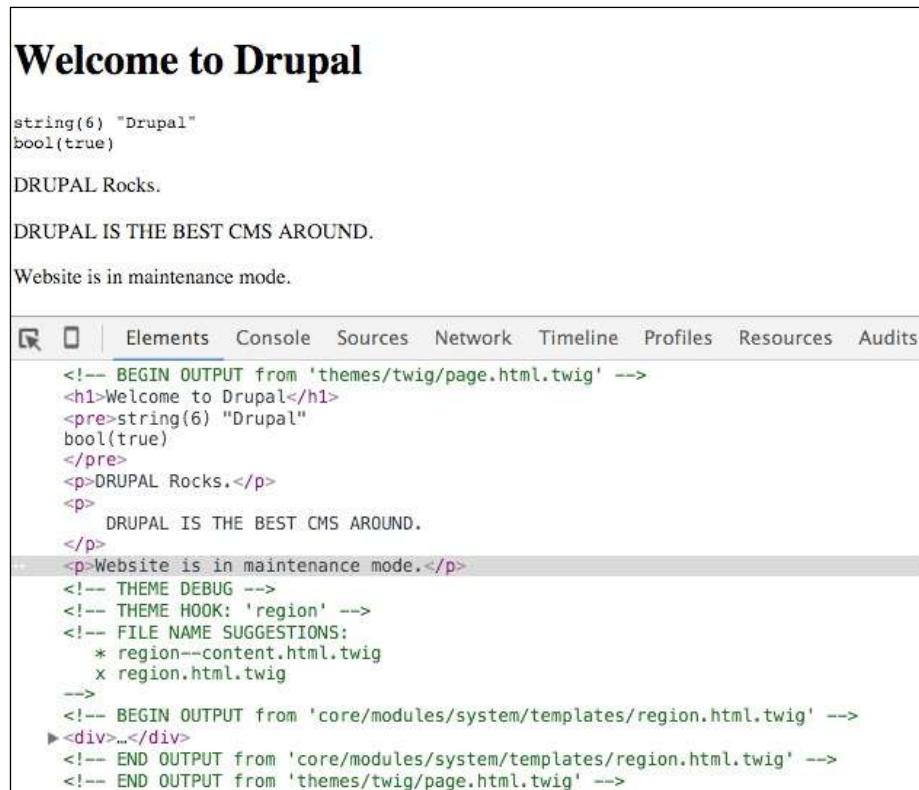
If we save our template and refresh the browser, we will not see anything actually displaying yet. The reason is that the `offline` variable is currently set to `false` and we are checking to see whether it is `true`.

Open `page.html.twig` and edit the `offline` variable changing its value to `true`:

```
{# Conditional logic #}
{% set offline = true %}

{% if offline == true %}
 <p>Website is in maintenance mode.</p>
{% endif %}
```

Now resave our template and view the page in the browser. This time, we will see our paragraph displayed, as shown in the following image:



The screenshot shows a browser window with developer tools open. The main content area displays the text "Welcome to Drupal" followed by several paragraphs of rendered content: "string(6) "Drupal\"", "bool(true)", "DRUPAL Rocks.", "DRUPAL IS THE BEST CMS AROUND.", and "Website is in maintenance mode.". Below this, the developer tools' Elements tab is active, showing the raw Twig template code. The rendered content is highlighted in grey in the DOM tree. The template code includes comments for theme hooks and file suggestions, as well as the BEGIN and END OUTPUT blocks from both the custom template and the core system template.

```

Welcome to Drupal

string(6) "Drupal"
bool(true)

DRUPAL Rocks.

DRUPAL IS THE BEST CMS AROUND.

Website is in maintenance mode.

<!-- BEGIN OUTPUT from 'themes/twig/page.html.twig' -->
<h1>Welcome to Drupal</h1>
<pre>string(6) "Drupal"
bool(true)
</pre>
<p>DRUPAL Rocks.</p>
<p>
 DRUPAL IS THE BEST CMS AROUND.
</p>
<p>Website is in maintenance mode.</p>
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'region' -->
<!-- FILE NAME SUGGESTIONS:
 * region--content.html.twig
 x region.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/system/templates/region.html.twig' -->
▶ <div>...</div>
<!-- END OUTPUT from 'core/modules/system/templates/region.html.twig' -->
<!-- END OUTPUT from 'themes/twig/page.html.twig' -->

```

By now, we are starting to see how control structures within Twig can come in handy to hide or show certain markup within our template based on the value of a variable. This will come in handy when we have certain Drupal regions that we want to display when a block is placed into a region.

The other commonly used control structure in Twig is looping. The `for` tag is used to loop over each item in a sequence. For our example, let's try looping based on a number of items and outputting the count.

Open `page.html.twig` and add the following:

```

{# Looping #}
{% for i in 0 ..10 %}
{{ i }}
{% endfor %}

```

If we save our template and view the page in the browser, we will be presented with the count within our loop displaying on the page starting at 0 and going to 10, as shown in the following image:

## Welcome to Drupal

```
string(6) "Drupal"
bool(true)

DRUPAL Rocks.

DRUPAL IS THE BEST CMS AROUND.

Website is in maintenance mode.

0 1 2 3 4 5 6 7 8 9 10
```

This is a simple loop, and it only really demonstrates the use of the `for` tag. Once we start creating additional Twig templates, we can loop through more complex Drupal variables. More extensive documentation regarding the `for` tag can be found at <http://twig.sensiolabs.org/doc/tags/for.html>.

## Template variables

Drupal 8 uses variables to output data within Twig templates. We know that variables generally consist of anything from a simple string to a complex object containing an array of values. If we look at the `html.html.twig` template, we will see documentation that outlines the variables available to us along with the name of the variable and a description of what the variable contains:

```
Variables:
logged_in: A flag indicating if user is logged in.
root_path: The root path of the current page (e.g., node, admin,
user).
node_type: The content type for the current node, if the page is a
node.
head_title: List of text elements that make up the head_title
variable.
 May contain or more of the following:
 - title: The title of the page.
 - name: The name of the site.
 - slogan: The slogan of the site.
 - page_top: Initial rendered markup. This should be printed before
'page'.
 - page: The rendered page markup.
```

```
- page_bottom: Closing rendered markup. This variable should be
printed after 'page'.
- db_offline: A flag indicating if the database is offline.
- placeholder_token: The token for generating head, css, js and js-
bottom placeholders.
```

Each of the variables that our template has access to can be output using Twig syntax. For example, the `head_title` variable outputs the title of our page within the `<title>` element. Drupal also uses `{{ attributes }}` to print out additional information to our page, for example, the `<body>` element to output CSS classes needed by modules or themes.

Each template we will work with uses variables to output database content. What if we want to add additional variables to Drupal? This is where the role of the theme file comes into use.

## The role of the theme file in Drupal

Themes can be simple to compose, sometimes containing a single configuration file, a couple of Twig templates, and a few assets. However, there will be times when we need to intercept and override variables and data that Drupal outputs before them reaching our Twig templates. Drupal's API (<https://api.drupal.org/api/drupal/8>) allows us to create a `*.theme` file where we can add theme functions that can hook into the API using different types of function calls.

- **Preprocess:** This is a set of function calls specific to different templates that allow us to manipulate variables before they are output to the page.
- **Hooks:** This is a set of function calls to hook into the Drupal API that allows us to alter variables and override default implementations.

## Preprocessors and hooks

The main role of preprocessor functions is to prepare variables to be used within our Twig templates using `template_preprocess` functions. These functions reference the theme and template we want to intercept. We would write an example of intercepting the `html.html.twig` template variables used within our Twig theme as follows:

```
twig_preprocess_html(&$variables) {
}
```

With this simple function call, we can hook into the theme preprocessing to intercept the `$variables` argument and manipulate it as needed before our template receives the variables. In order for us to use this function, we need to do the following steps:

1. Create a `twig.theme` file within the `themes/twig` folder. The `twig.theme` file will contain all the PHP functions we will write to work with Drupal's API.
2. Add the following within our `twig.theme` file and then save the file as:

```
<?php

/**
 * Implements hook_preprocess_html().
 */
function twig_preprocess_html(&$variables) {
 // add to classes
 $variables['attributes']['class'][] = 'twig';
}
```

Whenever we add a file or template for the first time, we will need to clear the Drupal cache.

## Overriding variables

Now that we have created our `twig.theme` file and have the outline of our first preprocess hook, let's take a look at how to override a variable. Previously, we saw that Drupal was adding classes to our body tag using the `$variables` variable. But what if we want to add additional classes specific to our theme?

Open `twig.theme` and edit the preprocess function to include the following:

```
<?php

/**
 * Implements hook_preprocess_html().
 */
function twig_preprocess_html(&$variables) {
 // add to classes
 $variables['attributes']['class'][] = 'twig';
}
```

Now if we save our `twig.theme` file and refresh the browser, we will see that our class is added, as shown in the following image.

```
<!-- BEGIN OUTPUT from 'themes/twig/html.html.twig' -->
<!DOCTYPE html>
<html lang="en" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/content/ dc: http://purl.org/dc/terms/ foaf: http://xmlns.com/foaf/0.1/ og: http://ogp.me/ns# rdfs: http://www.w3.org/2000/01/rdf-schema# schema: http://schema.org/ sioc: http://rdfs.org/sioc/ns# sioct: http://rdfs.org/sioc/types# xsd: http://www.w3.org/2001/XMLSchema# .>
<head>...</head>
<body class="twig_toolbar-tray-open toolbar-fixed toolbar-horizontal" style="padding-top: 78.9861px;">
```

While we have only touched the surface of the functionality that we can use when theming, we are purposely not going into depth regarding all the API calls that we have access to with Drupal 8. If you are interested in taking a deeper look, you can find the reference at <https://api.drupal.org/api/drupal/8>.

One last thing to note is that we can reference the completed exercise files for *Chapter 3, Dissecting a Theme*, if we need to compare any of the work we just completed or perform a database restore.

## Summary

From core themes to custom themes, we covered a lot of information. Remember that it's ok to go back and review any section to ensure that everything is understood. As we continue working through creating themes, our skills will only increase, and hopefully, we will all become theming experts when we're all done.

- We reviewed the new `info.yml` file and how Drupal recognizes metadata, stylesheets, scripts, regions, and settings.
- We looked at the role of assets in Drupal and what has changed since Drupal 7 with the addition of new JavaScript libraries and CSS best practices.
- Templates play a large part in theming, and we covered the basics of how they function including setting up our first theme and local development environment.
- We answered what Twig is and how much it empowers themers to build templates without having to worry about the laborious knowledge of PHP.
- Finally, we took a brief look at the `*.theme` file and how simple it is to override Drupal variables for use within our templates.

In the next chapter, we will dive even deeper into theming by creating a subtheme using Classy. We will also look at how easy it is to create a responsive starter theme following best practice methods to add CSS and JavaScript frameworks, such as Twitter Bootstrap. This will be followed up with a more detailed look at the `*.theme` file while using the Devel module to output variables within our Twig templates.



# 4

## Getting Started – Creating Themes

Drupal developers and interface engineers do not always create custom themes from scratch. Sometimes, we are asked to create starter themes that we begin any project from or subthemes that extend the functionality of a base theme. Having the knowledge of how to handle each of these situations is important, and in this chapter, we will be learning it as we cover the following:

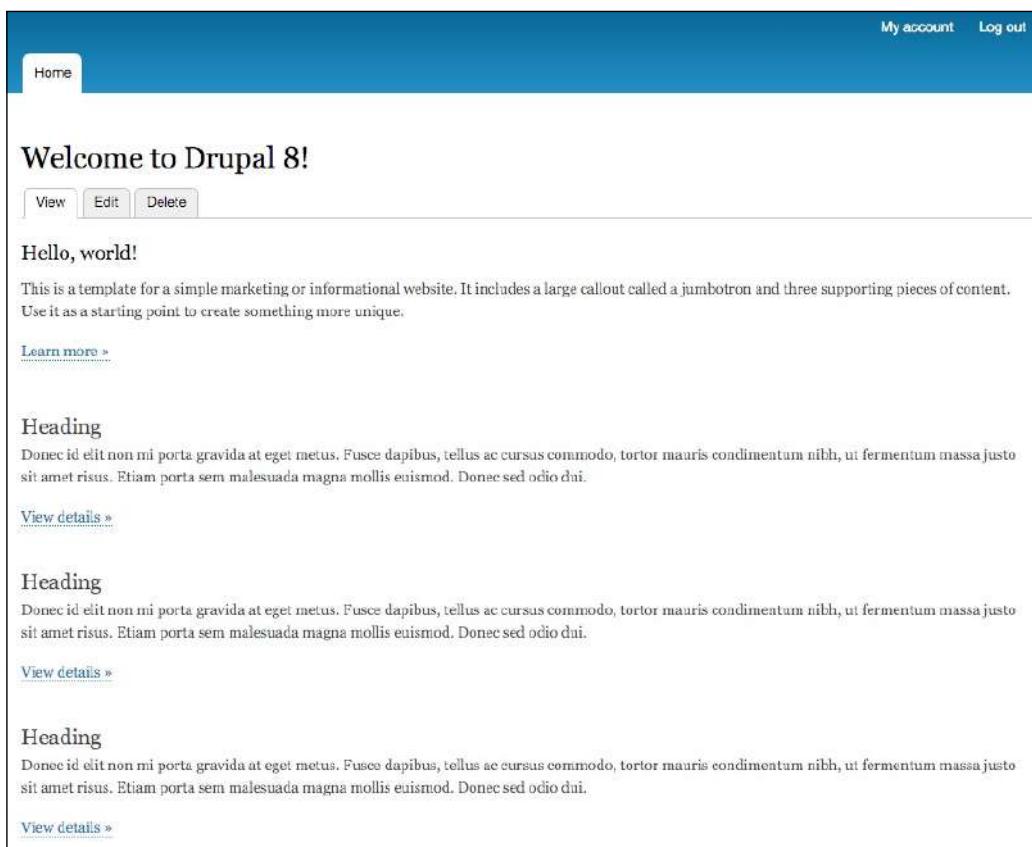
- First, we will create a starter theme that will walk us through managing folder and file structures, configuring a `*.info.yml` file, and allow us to work with a `*.libraries.yml` file to manage both CSS and JS assets. Our starter theme will involve multiple techniques that are common to any theme, including integrating a CSS framework such as Twitter Bootstrap.
- Next, we will rethink over layout strategies when creating a starter theme and discuss best practices to separate layout from presentation. This will include diving deeper into the Theme layer and how we can use contributed modules such as Devel to work with variables.
- Finally, we will create a subtheme that extends the base theme Classy. Having the ability to take advantage of a base theme's Twig templates and assets will allow us to focus on techniques to override CSS without changing the actual base themes files.

While we work through each section, we have the ability to refer back to the Chapter04 exercise files folder. Each folder contains a `start` folder and an `end` folder with files that we can use to compare our work when needed. This also includes database snapshots that allow us to start from the same point when working through various lessons. Information on how to restore database snapshots is covered in *Chapter 1, Setting Up Our Development Environment*.

## Starter themes

Whenever we begin developing in Drupal, it is preferable to have a collection of commonly used functions and libraries that we can reuse. Being able to have a consistent starting point when creating multiple themes means that we don't have to rethink much from design to design. This concept of a starter theme makes this possible, and we will walk through the steps involved in creating one.

Before we begin, take a moment to browse the `Chapter04/start` folder and use the `drupal8.sql` file to restore our current Drupal instance. This file will add additional content and configuration needed while creating a starter theme. Once the restore is complete, our homepage should look like the following image:

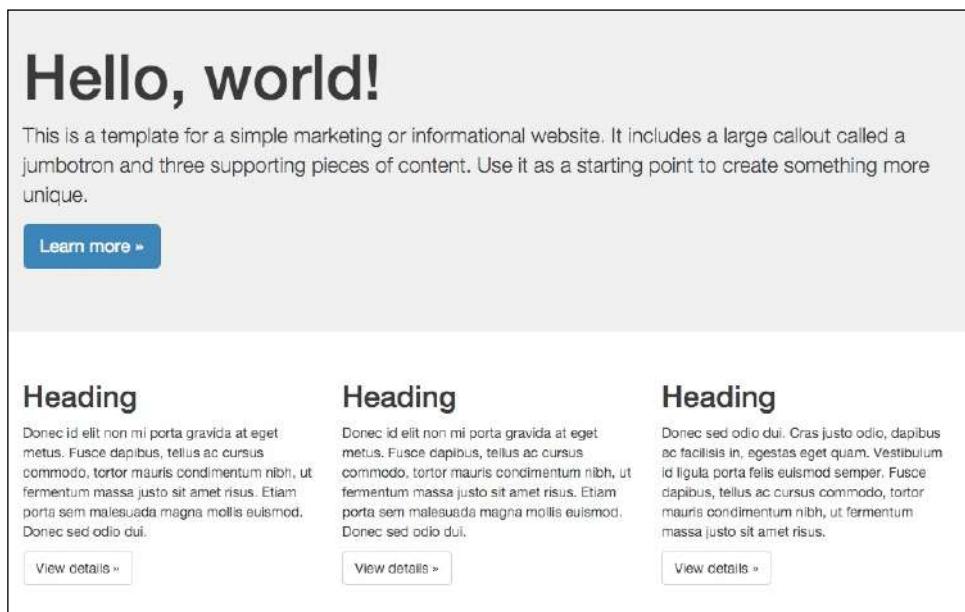


This is a pretty bland-looking homepage with no real styling or layout. So, one thing to keep in mind when first creating a starter theme is how do we want our content to look? Do we want our starter theme to include another CSS framework or do we want to create our own from scratch?

Since this is our first starter theme, we should not be worried about recreating the wheel but instead should leverage an existing CSS framework such as Twitter Bootstrap.

## Creating a Bootstrap starter

Having an example or mockup that we can refer to when creating a starter theme is always helpful. So, to get the most out of our Twitter Bootstrap starter, let's browse <http://getbootstrap.com/examples/jumbotron/> where we will see an example of a homepage layout:



As we take a look at the mockup, we can see that the layout consists of two rows of content with the first row containing a large callout known as a Jumbotron. The second row contains three featured blocks of content. The remaining typography and components are taking advantage of the Twitter Bootstrap CSS framework to display the content.

One advantage of integrating the Twitter Bootstrap framework into our starter theme is that our markup will be responsive in nature. It means that as the browser window is resized, the content will scale down accordingly. At smaller resolutions, the three columns will stack on top of one another enabling the user to view the content easier on smaller devices.

We will be recreating this homepage for our starter theme, so let's take a moment and familiarize ourselves with some basic Bootstrap layout terminology before creating our theme.

## Understanding grids and columns

Bootstrap uses a 12-column grid system to structure content using rows and columns. Page layout begins with a parent container that wraps all children elements and allows us to maintain a specific page width. Each row and column then have CSS classes identifying how the content should appear. So, for example, if we wanted to have a row with two equal width columns, we would build our page using the following markup:

```
<div class="container">
 <div class="row">
 <div class="col-md-6"></div>
 <div class="col-md-6"></div>
 </div>
</div>
```

The two columns within a row must combine to a value of 12 because Bootstrap uses a 12-column grid system. Using this simple math, we can have various size columns and multiple columns as long as their total is 12. We should also make a note of the column classes, as we have great flexibility in targeting different breakpoints:

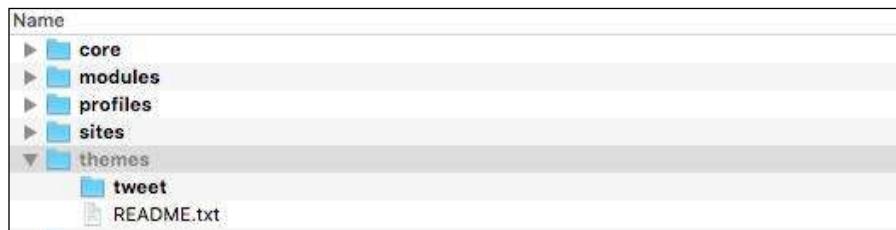
- Extra small (col-xs-x)
- Small (col-sm-x)
- Medium (col-md-x)
- Large (col-lg-x)

Each breakpoint references the various devices from smartphones all the way up to television-sized monitors. We can use multiple classes `class="col-sm-6 col-md-4"` to manipulate our layout, which gives us a 2-column row on small devices and a 3-column row on medium devices when certain breakpoints are reached.

To get a more detailed comprehension of the remaining Twitter Bootstrap documentation, we can browse <http://getbootstrap.com/getting-started/> any time. For now, it's time we begin creating our starter theme.

## Setting up a theme folder

The initial step in our process of creating a starter theme is simple. We need to open up MAC finder or Windows Explorer and navigate to the themes folder and create a folder for our theme. We will name our theme `tweet`, as shown in the following image:



## Adding a screenshot

Every theme deserves a screenshot, and in Drupal 8, all we need to do is simply have a file named `screenshot.png`, and the Appearance screen will use it to display an image above our theme.

Go ahead, copy `screenshot.png` from the `Chapter04/start/themes/tweet` folder, and place it within the `themes/tweet` folder.

## Configuring our theme

Next, we will need to create our themes configuration file, which will allow our theme to be discoverable. We will only worry about general configuration information to start and then add library and region information in the next couple of steps.

Begin by creating a new file in our `themes/tweet` folder named `tweet.info.yml` and add the following metadata to our file:

```
name: Tweet
type: theme
description: 'A Twitter Bootstrap starter theme'
core: 8.x
base theme: false
```

Note that we are setting the `base theme` configuration to `false`. Setting this value to `false` lets Drupal know that our theme will not rely on any other theme files. This allows us to have full control over our theme's assets and Twig templates.

We will save our changes here and clear the Drupal cache. Now we can take a look to check whether our theme is available to be installed.

## Installing our theme

Navigate to /admin/appearance within our browser, and we should see our new theme located in the **Uninstalled themes** section. Go ahead and install the theme by clicking on the **Install and set as default** link.

**Uninstalled themes**

|                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  <p>Stark 8.0.1<br/>An intentionally plain theme with no styling to demonstrate default Drupal's HTML and CSS. Learn how to build a custom theme from Stark in the <a href="#">Theming Guide</a>.</p> <p><a href="#">Install</a>   <a href="#">Install and set as default</a></p> |  <p>Tweet<br/>A Twitter Bootstrap starter theme</p> <p><a href="#">Install</a>   <a href="#">Install and set as default</a></p> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

If we navigate to the homepage, we should see an unstyled homepage:

Manage    Shortcuts    admin    Edit

- [Home](#)
- [My account](#)
- [Log out](#)

## Welcome to Drupal 8!

- [View](#)
- [Edit](#)
- [Delete](#)

Deep v butcher aesthetic taxidermy jean shorts, leggings tacos blue bottle street art actually gochujang pabst franzen hammock. Forage chia whatever, helvetica polaroid affogato venmo fingerstache etsy neutra dreamcatcher swag offal. Scenester church-key tumblr wolf, marfa slow-carb pinterest taxidermy green juice cronut offal truffaut godard lo-fi 8-bit. Kale chips put a bird on it cronut listicle bicycle rights iPhone. Swag tumblr kitsch, salvia mlkshk gastropub cold-pressed quinoa tacos etsy actually next level ennui gluten-free. Yuccie flexitarian 3 wolf moon, wayfarers drinking vinegar seitan synth bitters tumblr chia gentrify mustache green juice single-origin coffee. Polaroid hammock disrupt, selvage kogi green juice sriracha hella craft beer pour-over godard PBR&B.

### Custom Block One

Austin roof party bushwick art party quinoa brooklyn. 3 wolf moon beard gochujang wayfarers art party, 90's taxidermy gluten-free. Asymmetrical shoreditch try-hard keffiyeh farm-to-table DIY shabby chic, flannel pickled tattooed skateboard kombucha. Tote bag squid try-hard, schlitz food truck echo park master cleanse plaid leggings truffaut +1. Helvetica microdosing letterpress gastropub tattooed. Cliche gentrify authentic umami letterpress typewriter. Gentrify cold-pressed fixie, keffiyeh small batch literally chia pop-up tofu kombucha.

This clean palette is perfect when we are creating a starter theme as it allows us to begin theming without worrying about overriding any existing markup that a base theme may include.

## Working with libraries

While Drupal 8 ships with some improvements to its default CSS and JavaScript libraries, we will generally find ourselves wishing to add additional third-party libraries that can enhance the function and feel of our website. In our case, we have decided to add Twitter Bootstrap (<http://getbootstrap.com>), which provides us with a responsive CSS framework and JavaScript library that utilizes a component-based approach to theming.

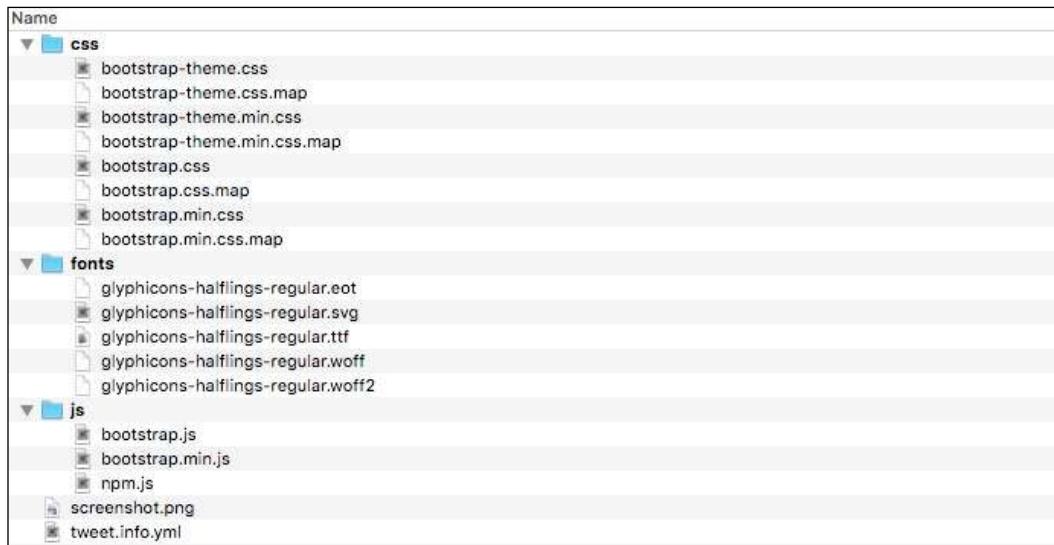
The process really involves three steps. First is downloading or installing the assets that make up the framework or library. Second is creating a `*.libraries.yml` file and adding library entries that point to our assets. Finally, we will need to add a library reference to our `*.info.yml` file.

### Adding assets

We can easily add the Twitter Bootstrap framework assets by following these steps:

1. Navigate to <http://getbootstrap.com/getting-started/#download>.
2. Click on the **Download Bootstrap** button.
3. Extract the `zip` file.
4. Copy the contents of the `bootstrap` folder to our `themes/tweet` folder.

5. Once we are done, our themes/tweet folder content should look like the following image:



Now that we have the Twitter Bootstrap assets added to our theme, we need to create a `*.libraries.yml` file that we can use to reference our assets.

## Creating a library reference

Anytime we want to add CSS or JS files to our theme, we will need to either create or modify an existing `*.libraries.yml` file that allows us to organize our assets. Each library entry can include one to multiple pointers to the file and location within our theme structure. Remember that the filename of our `*.libraries.yml` file should follow the same naming convention as our theme.

We can begin by following these steps:

1. Create a new file named `tweet.libraries.yml`.
2. Add a library entry named `bootstrap`.
3. Add a version that reflects the current version of Bootstrap that we are using.
4. Add the CSS entry for `bootstrap.min.css` and `bootstrap-theme.min.css`.
5. Add the JS entry for `bootstrap.min.js`.

6. Add a dependency to jQuery located in Drupal's core:

```
bootstrap:
 version: 3.3.6
 css:
 theme:
 css/bootstrap.min.css: {}
 css/bootstrap-theme.min.css: {}
 js:
 js/bootstrap.min.js : {}
 dependencies:
 - core/jquery
```

7. Save `tweet.libraries.yml`.

We have added a more complex library entry than we did in *Chapter 3, Dissecting a Theme*. However, in the previous library entry, we have added both CSS and JS files as well as introduced dependencies.

**Dependencies** allow any JS file that relies on a specific JS library to make sure that the file can include the library as a dependency, which makes sure that the library is loaded before our JS file. In the case of Twitter Bootstrap, it relies on jQuery and since Drupal 8 has it as part of its `core.libraries.yml` file, we can reference it by pointing to that library and its entry.

## Including our library

Just because we have added a library to our theme, it does not mean that it will automatically be added to our website. In order for us to add Bootstrap to our theme, we need to include it in our `tweet.info.yml` configuration file.

We can add Bootstrap by following these steps:

1. Open `tweet.info.yml`.
2. Add a `libraries` reference to `bootstrap` to the bottom of our configuration as follows:

```
libraries:
 - tweet/bootstrap
```

3. Save `tweet.info.yml`.

Make sure to clear Drupal's cache to allow our changes to be added to the Theme registry. Finally, navigate to our home page and refresh the browser so that we can preview our changes.

## Welcome to Drupal 8!

- View
- Edit
- Delete

### Hello, world!

This is a template for a simple marketing or informational website. It includes a large callout called a jumbotron and three supporting pieces a starting point to create something more unique.

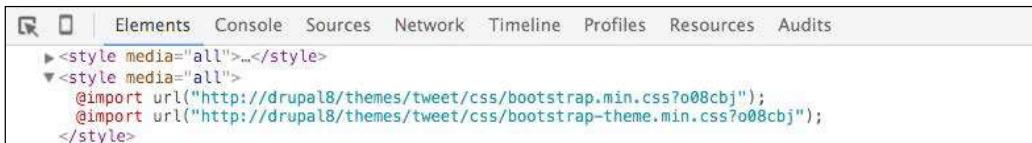
[Learn more »](#)

### Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

[View details »](#)

If we inspect HTML using Chrome's developer tools, we should see that the Twitter Bootstrap library is being included along with the rest of our files. Both the CSS and JS files are loaded into the proper flow of our document.



The screenshot shows the Chrome DevTools Elements tab. In the main pane, there is a single line of CSS code:

```
<style media="all">...</style>
<style media="all">
 @import url("http://drupal8/themes/tweet/css/bootstrap.min.css?o08cbj");
 @import url("http://drupal8/themes/tweet/css/bootstrap-theme.min.css?o08cbj");
</style>
```

## Creating a Jumbotron

Many times, a designer will create a section of content that they want to call the users' attention to. This is sometimes known as a Call to Action or a Hero. Bootstrap calls this visual treatment a Jumbotron and makes up the first part of our homepage mockup that we will be creating.

In order for us to implement the Jumbotron, we need to think about how our current homepage is laid out. We have a custom block called Jumbotron placed within the Content region. This means that potentially every page will have this block. Also, every page contains a page title block as well, and based on the mockup, we don't want that to display on our homepage. So, we need to address these two tasks while modifying our page to accommodate the Jumbotron.

First, we will take advantage of Drupal's new default WYSIWYG to directly edit the source HTML. This will allow us to add HTML markup directly into our custom block without worrying about creating a Twig template for it.

Second, we will need to hide the title block on the homepage using page restrictions on the Block layout admin.

## Step one – managing block content

We can manage a block content in multiple ways, but the easiest is by using the contextual links that Drupal provides. If we navigate to the homepage and locate the first block that displays "Hello, world!" we can hover over it to see the contextual links icon:



The contextual links menu will allow us to quickly get to the markup of the block by following these steps:

1. Click on the **Edit** link.
2. Locate the **Body** field.
3. Click on the **Source** button within the WYSIWYG bar.
4. Add the following markup including the jumbotron and container elements around the current markup:

```
<div class="jumbotron">
 <div class="container">
 existing markup...
 </div>
</div>
```

5. Click on the **Save** button.

This will accomplish part one of our steps. We should now see that our markup and content have been replaced and are now being styled according to the mockup:



## Step two – hiding the page title

The page title in Drupal 8 is now contained within a block. This allows us to easily place the page title wherever needed based on our design. It also allows us to manage it using the same visibility rules available in any block.

In our mockup, we need to suppress the page title from displaying on the homepage. We can accomplish this by using the contextual menu on the page title block to configure it, as follows:

1. Click on the **Configure block** contextual link.
2. Click on the **Pages** vertical tab on the **Configure block** screen.
3. Select the **Hide for the listed pages**.
4. Enter into the **Pages** text field the path `/home`.
5. Click on the **Save block** button.

The page title block is no longer displayed, and our Jumbotron looks pretty close to the mockup. While this was a pretty simple technique of adding HTML and Bootstrap classes directly to our content, this actually promotes some bad practices. Stop and think for a minute about what we just did.

We mixed layout and presentation markup together within a single field and stored that in the database. First, this is not very flexible. Second, we have no way to reuse the markup without continuing to add it to fields directly on a per need basis. The reason we approached the Jumbotron markup this way was to prove a point. There are always multiple ways to theme something, but often, we will need to rethink our layout.

## Rethinking our layout

Often, we will find ourselves having to rethink over the layout we are trying to accomplish while first creating a starter theme. In fact, creating a starter theme can actually be challenging at first with a lot of trial and error. Implementing our Jumbotron is quite a perfect example of trying to fit a square peg in a round hole. While Drupal will allow us to accomplish layouts in half a dozen different ways, we always want to follow the best practices.

After taking a look at the Jumbotron example again, we can actually break it down into more manageable and reusable components. To begin with, the Jumbotron example is to represent a homepage layout with one row for the Jumbotron and another row containing three blocks of content that float next to each other equally. When we started similarly with our Jumbotron block, we actually had all our blocks placed into our content region.

## Adding regions

Regions are key to any layout in Drupal, and the common rule is that anytime we look at a design or mockup, if we see multiple rows of content, we should equate each row to a Drupal region. In our case, we have identified a couple of different regions, but currently our starter theme has no defined regions at all. Therefore, it is using the default regions provided by Drupal.

What we really need is to add a Jumbotron region and the featured content region. These two regions will allow us to assign blocks of content to them apart from the main content region where we currently have them assigned.

One thing to note when adding regions to a theme is that we can't simply add regions to our configuration without also adding the default regions that Drupal provides. Failure to add the default regions will result in us only having the defined regions available to add content to, which is not ideal for a starter theme.

Let's begin by opening `tweet.info.yml` and adding the following regions to the bottom of our file:

```
regions:
 header: Header
 primary_menu: 'Primary menu'
 secondary_menu: 'Secondary menu'
 page_top: 'Page top'
 page_bottom: 'Page bottom'
 highlighted: Highlighted
 breadcrumb: Breadcrumb
```

## *Getting Started – Creating Themes*

---

```
content: Content
sidebar_first: 'Left sidebar'
sidebar_second: 'Right second'
footer: Footer
jumbotron: Jumbotron
featured: Featured
```

Make sure to save the configuration file, clear the Drupal cache, and visit the Block layout page to view our changes, as follows:

The screenshot shows the 'Block layout' configuration page in Drupal. It displays a list of regions and their current settings. Regions include 'Content', 'Left sidebar', 'Right second', 'Footer', 'jumbotron', and 'featured'. Each region has a 'Place block' button and a 'Configure' dropdown.

| Region       | Type                     | Status  | Action      |
|--------------|--------------------------|---------|-------------|
| Content      | Content                  | Content | Configure   |
| Left sidebar | No blocks in this region |         | Place block |
| Right second | No blocks in this region |         | Place block |
| Footer       | No blocks in this region |         | Place block |
| jumbotron    | No blocks in this region |         | Place block |
| featured     | No blocks in this region |         | Place block |

## Managing the block content

When we are on the Block layout screen, we will want to move our blocks into their respective regions. We can then take advantage of the different regions to address applying styling that will globally benefit our starter theme.

Begin by following these steps to move our four blocks into place:

1. Locate the **Hero** block within the **Content** region.
2. Select **Jumbotron** from the **Region** dropdown.
3. Locate the **Custom Block One** block within the **Content** region.
4. Select **Featured** from the **Region** dropdown.
5. Locate the **Custom Block Two** block within the **Content** region.
6. Select **Featured** from the **Region** dropdown.
7. Locate the **Custom Block Three** block within the **Content** region.
8. Select **Featured** from the **Region** dropdown.
9. Click on **Save blocks** button.
10. Our four blocks should now be placed within the regions, as shown in the following image:

The screenshot shows the 'Block layout' interface. It displays four blocks: 'Hero', 'Custom Block One', 'Custom Block Two', and 'Custom Block Three'. Each block has a 'Region' dropdown menu next to it. The 'Hero' block is set to 'Jumbotron', 'Custom Block One' is set to 'Featured', 'Custom Block Two' is set to 'Featured', and 'Custom Block Three' is also set to 'Featured'. Each block also has a 'Configure' button to its right.

If we were to navigate back to our homepage, we will no longer see any content being displayed. Once we have added two new regions and placed our blocks within those regions, the core Twig templates that Drupal is using to output our content have no idea that these regions exist.

## Using Twig templates

The easiest way to work with Twig templates is to allow Drupal and the Twig debug settings we enabled earlier to do the entire work for us. So what do I mean? Begin by navigating to the homepage and inspecting the markup using Chrome's developer tools. Locate the section of markup where we see the `div` element with a class of `layout-container`, as shown in the following image:

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'page' -->
<!-- FILE NAME SUGGESTIONS:
 * page--front.html.twig
 * page--.html.twig
 x page.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/system/templates/page.html.twig' -->
><div class="layout-container">...</div>
<!-- END OUTPUT from 'core/modules/system/templates/page.html.twig' -->
```

Twig debugging allows us to view all the information we need to identify which Twig template we can use for our homepage. If we look at the information provided, we can identify the following:

- Drupal is currently using `page.html.twig`
- The template is located at `core/modules/system/templates/page.html.twig`
- Drupal suggests that we can use `page--front.html.twig` to display the same content

With these three pieces of information, we can locate, copy, and modify any Twig template we may need in order to modify the layout and markup of the content coming from Drupal.

## Creating a homepage template

One rule that comes in handy while creating any Twig template is to be as specific as possible. There is generally multiple **FILE NAME SUGGESTIONS** that Drupal recommends and the more granular we are in choosing to name that template the less we will have to worry about overriding the content we didn't mean to overwrite.

Let's create our homepage template by following these steps:

1. Navigate to the `core/modules/system/templates` folder.
2. Copy `page.html.twig`.

3. Place the copy of `page.html.twig` into `themes/tweet` folder.
4. Rename `page.html.twig` to `page--front.html.twig`.

Clear Drupal's cache, browse the homepage, and use Chrome's developer tools to verify that we are using the `page--front.html.twig` template in the `themes/tweet` folder:

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'page' -->
<!-- FILE NAME SUGGESTIONS:
 x page--front.html.twig
 * page--.html.twig
 * page.html.twig
-->
<!-- BEGIN OUTPUT from 'themes/tweet/page--front.html.twig' -->
► <div class="layout-container">...</div>
<!-- END OUTPUT from 'themes/tweet/page--front.html.twig' -->
```

Since we have now created a `page--front.html.twig` template, any markup we add or modify within this template will only affect the homepage. Any interior pages that are added to our website will default to using `page.html.twig`.

The Jumbotron mockup only needs to display the header, primary menu, footer, Jumbotron, and featured regions. We can modify our `page--front.html.twig` template by replacing the current markup with the following code:

```
<div class="layout-container">
 <header role="banner">
 {{ page.header }}
 </header>

 {{ page.primary_menu }}
 {{ page.highlighted }}
 {{ page.jumbotron }}
 {{ page.featured }}

 <footer role="contentinfo">
 {{ page.footer }}
 </footer>
</div>
```

Now save the template and refresh the homepage in the browser. We should now see the regions we have defined being displayed along with any blocks that are assigned to them. Speaking of blocks, our Jumbotron block contains markup within the body field when it should really be moved to a region template.

## Creating region templates

Just like we were able to create a page-specific Twig template, we can also create region-specific Twig templates. If we inspect the Jumbotron region using Chrome's developer tools, we will see from the **FILE NAME SUGGESTIONS** that we can create a new Twig template named `region--jumbotron.html.twig`.

Create the region template by following these steps:

1. Navigate to the `core/modules/system/templates` folder.
2. Copy `region.html.twig`.
3. Place the copy of `region.html.twig` into `themes/tweet` folder.
4. Rename `region.html.twig` to `region--jumbotron.html.twig`.

Clear Drupal's cache, browse the homepage, and use Chrome's developer tools to verify that we are using the `region--jumbotron.html.twig` template.

Next, we will want to replace the markup within `region--jumbotron.html.twig` with the following markup:

```
{% if content %}
 <div class="jumbotron">
 <div class="container">
 {{ content }}
 </div>
 </div>
{% endif %}
```

Now save the template and refresh the homepage in the browser. If we inspect the Jumbotron region, we will see that our new markup has been added. All we have left to do is to edit the block and remove the layout markup that we added to the content previously.

Locate the Jumbotron block on the homepage, hover over it to reveal the context menu, and follow these steps:

1. Click on the **Edit** link.
2. Locate the **Body** field.
3. Click on the **Source** button within the WYSIWYG bar.

4. Replace the current markup with the following markup:

```
<h1>Hello, world!</h1>
<p>This is a template for a simple marketing or informational website. It includes a large callout called a jumbotron and three supporting pieces of content. Use it as a starting point to create something more unique.</p>
<p>Learn more ></p>
```

5. Click on the **Save** button.

We have now completed our Jumbotron region of the homepage by separating layout markup from presentational markup. This approach is now reusable and makes a great location in our starter theme to add Hero content. Let's replicate this process by adding a Twig template for our featured region as well, as follows:

1. Begin by creating a new file Twig template named `region--featured.html.twig` within our `themes/tweet` folder.
2. Replace the current markup with the following code:

```
{% if content %}
 <div class="container">
 {{ content }}
 </div>
{% endif %}
```

3. Now save the template and refresh the homepage in the browser. Our featured region now has the `container` class. In addition, the featured region is constraining the content to the same width as our Jumbotron region.

In order for us to complete the featured region, we need to know which blocks are within it and add CSS classes to them. Time to look toward the Drupal 8 Theme layer for help.

## Working with the Theme layer

Drupal 8 has an extensive API that includes the Theme layer, which gives us the ability to alter and preprocess variables before they are output by Drupal. The API is so extensive that we won't even scratch the surface of the functionality we can use. More detailed information can be found at <https://api.drupal.org/api/drupal/8>.

One such function we will be working with is `template_preprocess_block`, which prepares values passed to each block before them being output by `block.html.twig`. Before we can begin using preprocess functions, we will need to create a `*.theme` file.

Begin by creating a new file named `tweet.theme` within our `themes/tweet` folder. Once our theme file has been created, we can add the following preprocess function:

```
<?php

function tweet_preprocess_block(&$variables) {

}
```

Within our function, we will look for specific blocks based on their IDs and then apply a CSS class to them that allows the blocks to be displayed in three columns. One thing to note is that this is by no means the only way to accomplish this requirement, but to avoid getting too far into the Drupal API we will opt for a simple solution.

While working with the Theme layer, we need some way to print out the `$variables` array that is passed by reference to most functions. Although PHP provides us with the `var_dump()` function, this can be a tedious task of reading through all the information that is printed to the screen, especially since it is not formatted.

## Using Devel to print variables

The Drupal community has provided us with a better mechanism of working with variables using a third-party contributed module named Devel. The Devel module can be found at <https://drupal.org/project/devel> and is a set of helper functions to work with variables as well as a list of other functionality that we will not be using at this time.

Because this is our first time installing a contributed module for use with Drupal 8, we can follow these steps to download and install the module:

1. Navigate to the Devel project page <https://drupal.org/project/devel>.
2. Click on the TAR or ZIP download link for the latest Drupal 8.x version.
3. Create a folder named `contrib` within the `modules` folder of our Drupal 8 instance.

4. Extract the contents of the `devel` module to the `contrib` folder, as follows:



The `contrib` folder will hold any contributed modules that we install, including the `Devel` module. Now we need to install and configure the `Devel` module by following these steps:

1. Navigate to `/admin/modules` within the browser.
2. Locate the **Devel** module under the **DEVELOPMENT** section.
3. Click on **checkbox** next to **Devel** to install it.
4. Locate the **Devel Kint** module.
5. Click on **checkbox** next to **Devel Kint** to install it.
6. Click on the **Install** button at the bottom of the **Extend** page.



Now that we have `Devel` and `Devel Kint` installed, we can move on to using it to display `$variables` within our preprocess function to help identify information we will need to complete our function.

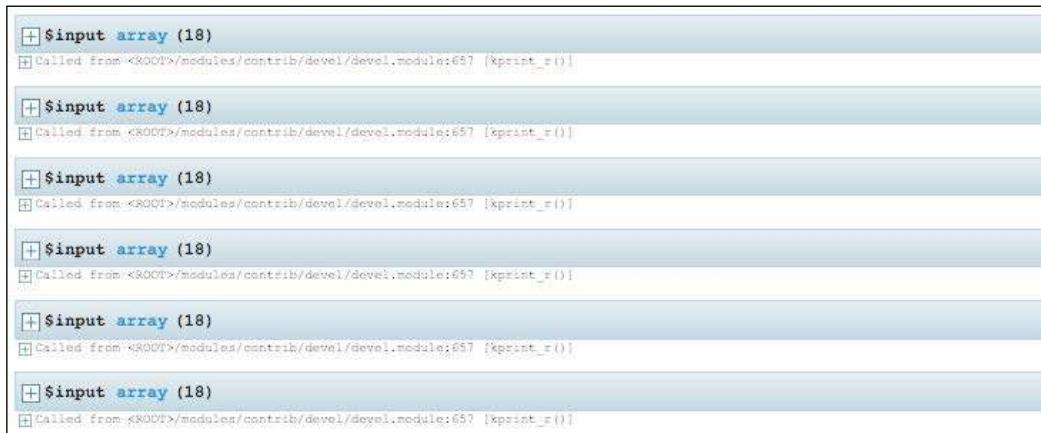
## Printing variables from a function

If we open back up our `tweet.theme` file, we currently have an empty preprocess function. This function accepts a parameter that is passed by reference that holds any \$variables available to be used by blocks. We can use the Devel module to now output \$variables to our page by adding the following line of code to our function:

```
<?php

function tweet_preprocess_block(&$variables) {
 dpm($variables);
}
```

The `dpm()` function will take whatever values that are passed to it and output the contents in a print friendly format. To see this in action, let's save our file, clear Drupal's cache, and browse to our homepage. If Devel is working properly, we should see six different sections of our site displaying a collapsed information box that contains the contents of the `$variables` array. Each instance represents the values for each block currently assigned to regions. This is due to the fact that our preprocess function runs once for each block, as follows:



The screenshot shows a browser developer tools console with six identical entries. Each entry consists of a blue header bar with a plus sign icon and the text '\$input array (18)' followed by a grey body bar with the text 'Called from <ROOT>/modules/contrib/devel/devel.module:657 [kprint\_r()]'. The console has a light grey background and a vertical scrollbar on the right side.

```
[+] $input array (18)
[+] Called from <ROOT>/modules/contrib/devel/devel.module:657 [kprint_r()]

[+] $input array (18)
[+] Called from <ROOT>/modules/contrib/devel/devel.module:657 [kprint_r()]

[+] $input array (18)
[+] Called from <ROOT>/modules/contrib/devel/devel.module:657 [kprint_r()]

[+] $input array (18)
[+] Called from <ROOT>/modules/contrib/devel/devel.module:657 [kprint_r()]

[+] $input array (18)
[+] Called from <ROOT>/modules/contrib/devel/devel.module:657 [kprint_r()]

[+] $input array (18)
[+] Called from <ROOT>/modules/contrib/devel/devel.module:657 [kprint_r()]
```

We are interested in the three custom blocks that appear in our featured region. Assuming the blocks load in the order that the regions are printed, we should be able to expand the fourth information box to see more information. In particular, we are interested in the block attributes that contain the ID of each block, as follows:

```

$input = array (18)
 [+] 'elements' => array (17)
 'theme_hook_original' => string (5) "block"
 [-] 'attributes' => array (3)
 'data-quickeedit-entity-id' => string (15) "block_content/1"
 'id' => string (26) "block-tweet-customblockone"
 [-] 'class' => array (1)
 string (17) "contextual-region"

```

The longer we look at the information being output, the more it makes sense how to traverse through the array to access information we can use. For instance, to grab the ID of each block, we could access it by writing `$variables['attributes']['id']` within our preprocess function. Now all we need to do is add some logic to our function that looks for the ID within a list of block IDs and add a CSS class to the block if found. We can accomplish this by adding the following to our preprocess function:

```

function tweet_preprocess_block(&$variables) {

 // Add layout class to Featured Blocks
 $featured = array('block-tweet-customblockone', 'block-tweet-
customblocktwo', 'block-tweet-customblockthree');

 $id = $variables['attributes']['id'];

 // If block id matches list - add class
 if(in_array($id, $featured)){
 $variables['attributes']['class'][] = 'col-md-4';
 }
}

```

Remember to remove the `dpm()` function we added previously. Next, we can clear Drupal's cache and then browse our homepage where we will see our three custom blocks aligned to our grid:

The screenshot shows a website layout with a header containing the text "Hello, world!". Below the header are three content blocks, each consisting of a heading, a paragraph of text, and a "View details >" button.

| Heading                                                                                                                                                                                                                                                                      | Heading                                                                                                                                                                                                                                                                      | Heading                                                                                                                                                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.<br><a href="#">View details &gt;</a> | Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.<br><a href="#">View details &gt;</a> | Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.<br><a href="#">View details &gt;</a> |

We have definitely mastered using the Twitter Bootstrap framework with our starter theme to recreate the mockup Jumbotron example. By modifying Twig templates, using theme suggestions, working with new regions, and diving deeper into the Theme layer, we were able to have Drupal output HTML markup exactly how we needed it.

While starter themes are very flexible, they do require a little more work than simply using an existing base theme. We can take a quick look at what this means by creating a subtheme next.

## Subthemes

One point of interest in Drupal 8 is that there is a new base theme named **Classy**, which both Bartik and Seven reference. This means that Bartik and Seven in reality are subthemes. So why not learn from the best, in this case, Morten Birch Heide-Jørgensen, otherwise known as "Div Killer." Morten has come through, as his nickname suggests, and created one hell of a base theme.

To become a little more intimate with this new base theme, we will create a subtheme of our own called **Sassy**. Since the steps involved to create and install a subtheme are similar to a starter theme, we will progress a little faster through this first part.

## Adding the theme folder

Begin by navigating to our themes folder and create a new folder inside named sassy.

## Including a screenshot

Go ahead and copy screenshot.png from the Chapter04/start/themes/sassy folder and place it within the themes/sassy folder.

## Configuring our theme

Begin by creating a new file in our themes/sassy folder named sassy.info.yml and add the following metadata to our file:

```
name: Sassy
type: theme
description: 'A Classy sub theme but a little more Sassy'
core: 8.x
base theme: classy
```

Note that we are setting the base theme configuration to classy this time. Setting this value to classy lets Drupal know that our subtheme will inherit all the configuration and files from the base theme.

We will save our changes at this time and clear the Drupal cache. Now we can take a look to see if our theme is available to install.

## Installing our theme

Navigate to /admin/appearance within our browser and we should see our new theme located in the **Uninstalled themes** section. Go ahead and install the theme by clicking on the **Install and set as default** link:



This time when we navigate to our homepage, we will see that our markup has changed, we are now inheriting the markup and libraries from Classy. We can verify this by using Chrome's developer tools to inspect the page. Looking at the **FILE NAME SUGGESTIONS** of any region, we will note that the Twig templates are coming from `core/themes/classy/templates/*`, as follows:

```
<!-- BEGIN OUTPUT from 'core/themes/classy/templates/layout/region.html.twig' -->

<!-- THEME DEBUG -->
<!-- THEME HOOK: 'block' -->
<!-- FILE NAME SUGGESTIONS:
 * block--sassy-customblockone.html.twig
 * block--block-content--8e1cf09a-2f14-40de-9100-a1cbae3ab6d4.html.twig
 * block--block-content.html.twig
 * block--block-content.html.twig
 x block.html.twig
-->
<!-- BEGIN OUTPUT from 'core/themes/classy/templates/block/block.html.twig' -->


```

## Touring Classy

We can take a closer look at Classy by navigating to the `core/themes/classy` folder of our Drupal instance. At first glance, the theme structure of Classy is quite organized. It is well-structured with folders for CSS, images, and a multitude of Twig templates. Each template has been organized based on its functionality as follows:



Everything so far screams best practices and is one of the major benefits of creating a subtheme that uses Classy as a base theme. However, we can still add our own regions, libraries, and Twig templates as we would for any other theme. However, in some cases, we may find ourselves also needing to override libraries with our own CSS or JS without modifying any assets directly located in the base theme.

## Overriding a library

So when we talk about overriding a library, we have options to replace the entire library, replace an asset with another asset, or remove an asset or entire library simply by using `libraries-override` within our theme's `*.info.yml` file.

Take the status message block as an example. Anytime we edit a block, a node, or anything in Drupal that has an edit form, once we click the save button, we will see a status message letting us know the outcome of our action.

 Basic page *Welcome to Drupal 8!* has been updated.

In this case, the default styling for the status message is coming from the base theme Classy. In fact, if we look at the `classy.libraries.yml` file located in the `core/themes/classy` folder, we can see an entry pointing to `css/components/messages.css`:

```
messages:
 version: VERSION
 css:
 component:
 css/components/messages.css: { weight: -10 }
```

What if our subtheme calls for different styling? If we want to override `messages.css` with our own version of the file, we can do so using `libraries-override`.

All we need to override this file is a `*.libraries.yml` file for our subtheme, a library entry pointing to our own `messages.css` file, and a reference within our `*.info.yml` file telling Drupal what we want to override.

Let's override Classy's messages styling by following these steps:

1. Copy the `css` folder from the `Chapter04/start/themes/sassy` folder and place it within the `themes/sassy` folder.
2. Create a new `sassy.libraries.yml` file within the `themes/sassy` folder.

3. Add the following library entry to `sassy.libraries.yml`:

```
messages:
 version: VERSION
 css:
 theme:
 css/messages.css: {}
```

4. Save `sassy.libraries.yml`.

5. Open `sassy.info.yml` and add the following configuration:

```
libraries-override:
 classy/messages: sassy/messages
```

6. Save `sassy.info.yml`.

Now we can clear Drupal's cache and then browse our homepage where we can edit the homepage by following these steps:

1. Click on the **Edit** tab to open the Welcome to Drupal 8 Edit form.
2. Click on the **Save and keep published** button.

Once back on the homepage, we should see that the status message is now picking up our `messages.css` rules and is now overriding the `messages.css` that was originally coming from the Classy base theme.

**Basic page Welcome to Drupal 8! has been updated.**

It is hoped that by now, we can see how `libraries-override` will come in handy whenever we want to replace assets injected from base themes, modules, or even Drupal core. For more information and examples, such as `libraries-override`, feel free to review the documentation at <https://www.drupal.org/theme-guide/8/assets>.

As we progress to creating custom themes, we will find we often need to add the JS functionality. Although we will not be covering it in this chapter, we will take a look at working with JS libraries in great detail in *Chapter 6, Theming Our Homepage*.

## Summary

The starter theme or subthemes are all just different variations on the same techniques. The level of effort to create each type of theme may vary, but as we saw there was a lot of repetition, and by now, we have already created a couple different themes. So, let's look back to what we covered in this chapter:

- We began with a discussion around starter themes and learned what steps were involved in integrating a CSS framework such as Twitter Bootstrap.
- We worked extensively with libraries and best practices for creating a homepage from a mockup. This included how to rethink layouts and how to avoid the pitfalls that we may come across when theming blocks and regions.
- Working with the Theme layer came in handy when needing to understand what was available to us when working with preprocess functions. From using contributed modules such as Devel to print variables to creating Twig templates, we learned how to separate layout from presentation.
- Finally, we took a quick look at subthemes and discussed the benefits of using them while still being able to override any assets they include without modifying the original assets.

In the next chapter, we will prepare ourselves for a large web-based project that will involve setting up our themes structure, using essential modules, and walking through the completed website that contains a home page, interior page, blog section, contact page, and search results.



# 5

## Prepping Our Project

One of the most important things that will help you learn how to become better frontend developers is taking a look at a design mockup and dissecting how you would implement it within Drupal. This would mean asking ourselves questions along the way, such as how the homepage is put together, how a user interacts with a webpage, and how we are going to implement a specific functionality. In this chapter, we will do exactly that as we begin with a fully working HTML mockup that we can review within the browser and then convert it into a Drupal 8 theme, piece by piece. To give us a better idea of what we will be covering, let's review the following:

- We will start by reviewing our completely designed mockup with a Homepage, Interior page, Landing page, Blog posts, a Contact Us page with a Google map and web form, and other user interactions that we will need to build.
- Once we have a better understanding of what we are building, we will take a backup of the database and restore it onto our Drupal 8 instance. This will allow us to have a baseline starting point from which to build a theme.
- Finally, we will finish up with creating our new theme structure, including defining the metadata, creating our regions, and implementing one of several CSS and JavaScript libraries.

While we work through each section, we have the ability to refer to the `Chapter05` exercise files folder. Each folder contains a `start` and `end` folder with files that we can use to compare our work when needed. This also includes database snapshots that will allow us to all start from the same point when working through various lessons.

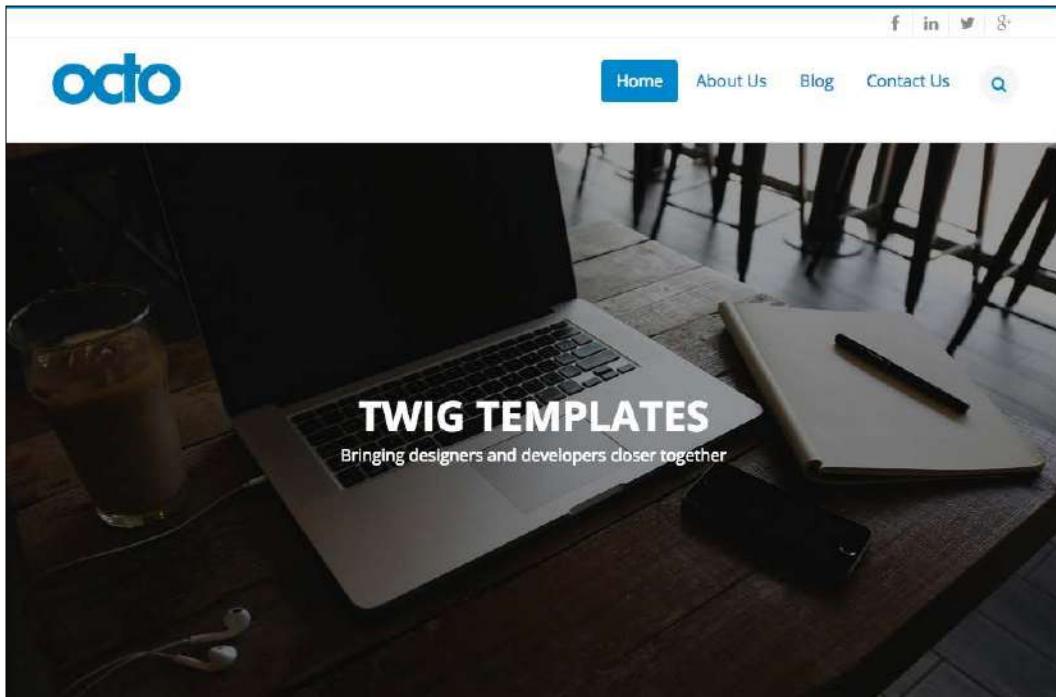
Before we get started, let's open up the `Mockup` folder located in our exercise files and the `index.html` page using Google Chrome web browser. The Mockup contains a fully functioning HTML website that we have been tasked with developing for our client. We will be reviewing this mockup throughout the remaining chapters to compare against our final Drupal 8 theme, so let's get started.

## **Walking through the design mockup**

Whether we are working for a digital agency or simply freelancing, in most cases, we will already have purchased a theme or designed one from scratch that has been built in pure HTML, CSS, and JavaScript. Having a theme already available to us makes our job as a frontend developer much easier in identifying what needs to be built. As we begin to review each page of our design, we will be taking notes about specific functionality that we will later revisit when creating our new theme. We will clearly point out such items as regions, page layouts, blocks of content, and how we would best implement CSS and JavaScript.

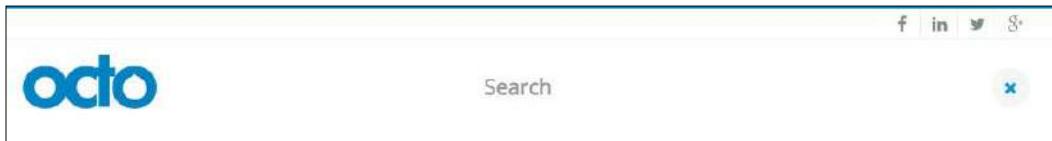
### **Homepage layout**

If we haven't already done so, let's open up the homepage of our mockup, as shown in the following image, and navigate as any other user visiting our site would.



At first glance, our mockup seems to contain some very standard components, such as a header with a logo, menu, full page slider, and some social network icons. However, there are several hidden characteristics that we may have missed unless we click around our homepage.

The first item to point out is the search icon in the main menu. Clicking on this icon reveals a hidden search input that allows the user to search content, as shown in the following image:



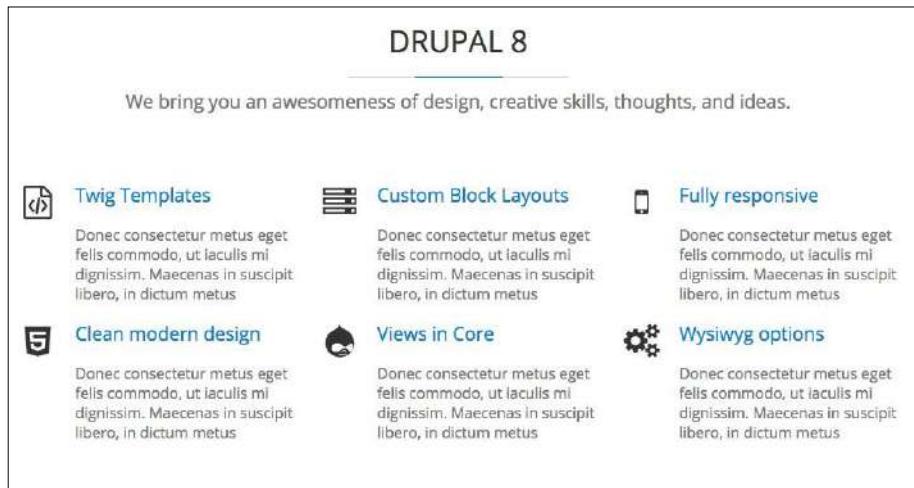
We know that Drupal 8 provides the user with the ability to search database content as well as providing us with a search block that will need to account for in our theme.

The second item is a Parallax function where the background image moves at a slower rate than the text overlaid on it as we scroll. If we happened to click on the arrow icon at the bottom of the page instead of scrolling, we also discover that we are automatically taken to another section of the homepage. One thing to note is that the scrolling effect is smooth and not sudden. This Parallax method, as well as the smooth scroll effect, will require us to implement some custom JavaScript or libraries that assist in providing this type of user interaction.

Our third item is the fixed header containing the logo, main menu, and search element. When a user begins to scroll down the page, the header becomes fixed to the top of the viewport. This feature allows the end user to navigate anywhere within the website without having to scroll back to the top of any long-form content pages.

## *Prepping Our Project*

As we continue further down the homepage layout, we come to another section of content, as shown in the following image:



This section of content should be of no concern as it contains some simple markup with headings and blocks of text, but we will need to make a note of the icons being used. We will look at implementing these icons using Font-Awesome, a CSS toolkit that allows for iconic fonts.

Finally, our homepage consists of a footer and subfooter with three small blocks of content containing various text blocks and a form element, as shown in the following image:



One thing to note is that our header and footer areas will be consistent throughout the mockup. Keeping this in mind, we will take a look at how to implement this in our theme without having to repeat the content from page to page.

## **Defining homepage regions and user interaction**

One last exercise we need to consider based on our homepage is what visible regions we have that can contain content. Starting from the top of our homepage and ending at the bottom of our document, we should be able to identify the following regions:

- Top Header with social network icons
- Header with logo, menu, and hidden search element
- Headline section with static background and vertically sliding text
- Before Content section to display the content before the main content
- Content section with various blocks of content
- Footer with three separate blocks of text and form elements
- Sub footer with the left and right sections containing content

We will need to define these regions within our theme, along with any others we discover as we review the internal pages of our mockup.

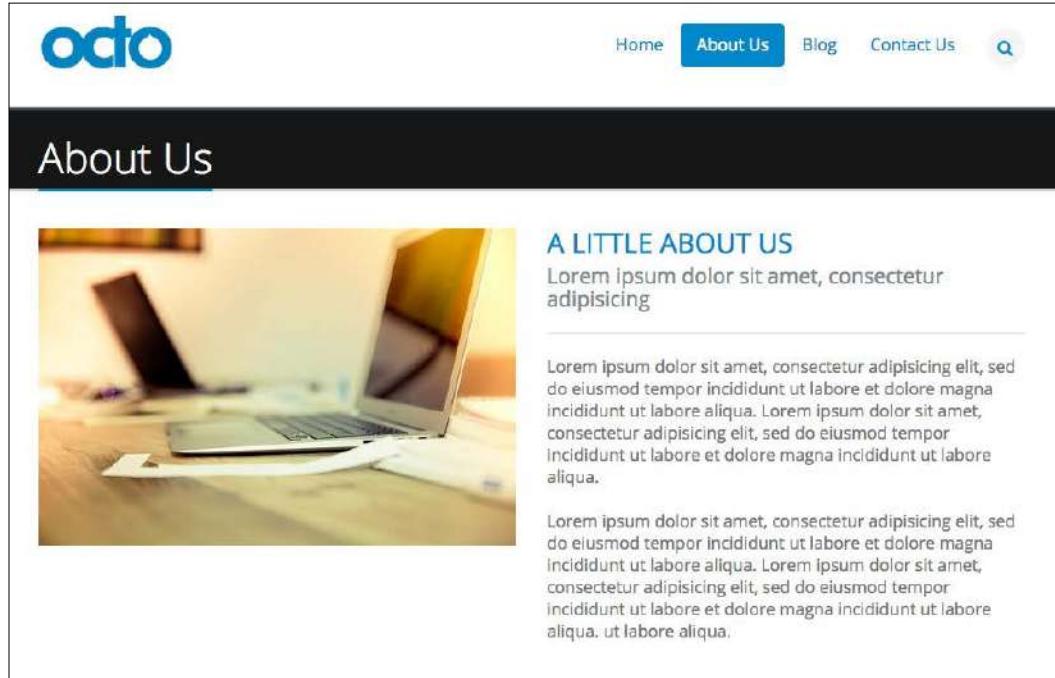
Finally, let's review the notes we took that pertain to user experience and functionality that may be new to us and that we will need to implement when building our homepage. Such items are:

- Search icon that when clicked shows and hides the search block to user
- Parallax background effect
- Slider text on the top of a fixed background image
- Smooth scrolling when the user clicks on the navigation link in slider
- Font icons

With our review of the complete homepage, it's now time to move on to the interior pages and investigate what else our mockup has in store for us.

## Basic page layout

Let's begin reviewing the **About Us** section of our mockup by clicking on the corresponding menu item in the header. As we begin to review our first interior page, we will note that our basic page layout includes a page title that spans across the width of our page, as shown in the following image:



This basic page differs from our homepage and is our first clue that we will need to consider one or more alternative page layouts for our interior pages.

After scrolling a little further down our page, we also find a section of content displaying team members that consists of a heading, subheading, and four blocks of identical content, highlighting each team member.

## MEET OUR AWESOME TEAM

*We bring you an awesomeness of design, creative skills, thoughts, and ideas.*



STEPHEN MATURIN  
*Technical Architect*

[f](#) [t](#) [g+](#)



PHOEBE CAULFIELD  
*Project Manager*

[f](#) [t](#) [g+](#)



NICK ADAMS  
*User Experience*

[f](#) [t](#) [g+](#)



MOLLY BLOOM  
*Graphic Designer*

[f](#) [t](#) [g+](#)

If we hover our mouse over each of the team member images, we will note a visual effect where the image fades from gray to full color. We will need to keep this in mind while identifying the fields that make up this piece of content and how we may need to manipulate the HTML markup to achieve this technique.

Our basic page is a simple one-column layout, which does not introduce any new layouts we may need to define as of yet, and which should not be too challenging to develop. It is also typical of most of the pages that a frontend developer or themer will face while creating themes.

## Defining interior regions

Starting from the top of our interior page and ending at the bottom, we should be able to identify the following new regions:

- A title bar with a page title
- The After Content section to display content blocks below main content

We will need to define these regions within our theme along with any others we discover as we continue to review pages of our mockup.

## Landing page layout

One of the more complex page layouts in our mockup is that of the Blog section. If we navigate to the Blog landing page by clicking on the **Blog** menu item in our header, we will be presented with a very rich looking page, as shown in the following screenshot:

The screenshot shows a blog landing page with a header featuring the 'octo' logo, social media links (Facebook, LinkedIn, Twitter, Google+), and a navigation bar with 'Home', 'About Us', 'Blog' (which is highlighted in blue), 'Contact Us', and a search icon. The main content area has a black header bar with the word 'Blog'. Below it is a large image of a man in a suit writing the word 'Idea' on a chalkboard with a lightbulb above it. To the right is a sidebar with a 'Categories' section listing 'Design', 'Lifestyle', 'News', 'Photos', and 'Videos'. At the bottom of the sidebar are tabs for 'Popular' and 'Recent', with 'Popular' being active. Below the tabs are two blog post teasers: 'Post Two' (May 23, 2015) and 'Post One' (May 23, 2015). The main content area below the image shows a post titled 'Post Three' from May 23, 2015, by admin, with 0 comments and a 'Read more...' button.

Landing pages often display a listing of content with related or highlighted information to accompany it. Our blog page is no different, and it consists of a teaser of content previewing each blog post and repeating down the page. We are also presented with a two-column layout with the content region to the left and a sidebar to the right. This page gives us our second layout to consider when creating our Twig templates.

Some other regions of content are the **Categories** listing and tabbed block of **Popular** and **Recent** blog posts that reside in the right sidebar. This content is just an extension of the blog post itself with a listing of taxonomy terms and a teaser. Drupal 8 will be able to handle vocabulary terms, View modes, and Views listing of content, all without having to worry about any contributed modules.

## Blog detail layout

Because landing pages only generally provide us with a listing of content, let's quickly review the blog content in more detail beginning with browsing an individual blog post. We can accomplish this by clicking on the **Post Two** blog post title, which should bring us to the Post content type detail page.

It looks like the two-column page layout is being continued from our landing page to our detail page. This will make it very easy for us to develop a Twig template that both our teaser and full content views can use. We also have repeating sidebar content on our detail page as well. This is a great indication of blocks of content that we will need to make sure is reusable, another great feature of Drupal 8 since blocks can now be reused.

Before we move on to the View of our contact page, we need to keep in mind two different features of the Blog detail page. The first is a slider present when a post has multiple images, which is indicated by the two blue navigational dots.



Upon closer inspection, we can view each image by clicking on either of the two dots. We will need to make another mental note when it comes to this image field and consider how we can determine if there are multiple images and how to apply the slideshow effect to them.

## *Prepping Our Project*

---

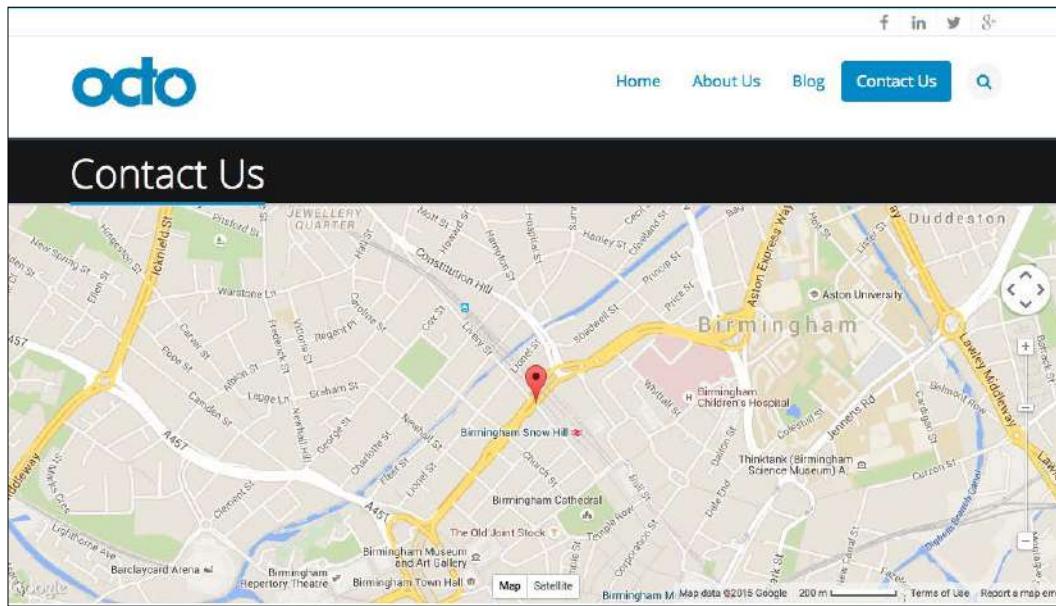
The other item of interest is towards the bottom of our Blog post, and it consists of a commenting feature, as shown in the following image:



This new commenting region allows users not only to leave feedback about a post, but also the ability to share a post using social networks, as is apparent with the **Share this post** heading being displayed above the comment region. We will look at how to theme comments in Drupal 8 in more detail later in *Chapter 8, Theming Our Blog Listing Page*.

## Contact page

Our second to last navigational page to review is the **Contact Us** page. Traditional contact pages consist of general business hours, e-mail addresses, and other methods to contact the user of the website. However, if we navigate to the contact page by clicking on the **Contact Us** link in the menu, we will be presented with the following image:



The **Contact Us** page shows a full-width one-column layout with a gorgeous Google Map highlighting the location of our office with a map marker. The map is fully functional, and it allows the end user to navigate wherever they would like in the world. The Google Maps API allows us to add this type of user interaction very easily to a page with very little JavaScript. Drupal 8 makes this even easier with their new way to handle JavaScript libraries. We will keep this in mind when developing this Twig template.

Adding a little more personalization to our contact page, this web form allows any user to contact us.

**GET IN TOUCH WITH US**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam Imperdiet ante in metus lobortis, vitae sagittis lorem viverra. In felis leo, posuere a lorem in, vestibulum hendrerit lectus. Vestibulum eget sapien dignissim, vestibulum lectus vitae. pretium mi.

**Your name \***

**Your email address \***

**Subject \***

**Message \***

Being fully responsive, our web form will function on all mobile devices and is a great example of customization versus just a typical e-mail address being displayed. We have various examples of form elements within our mockup that will be developed while creating our Drupal 8 theme.

However, what would a website be without giving our end user an ability to search for content?

## Search results

The final page of our mockup ties back into our search block in the header of our website, allowing us to display the content of our website based on a user query. Drupal 8 provides us with this mechanism to index content and allows us to then display the search results. We can find a mockup of the search results page located in the exercise files Mockup folder titled `search.html`, as shown in the following image:



One thing we will find though is that the search results are very limited without extending the functionality with other third-party search services such as Apache Solr, which provides for a much more robust search experience. Still, we will take a look at how to customize the search results page for a cleaner look and feel.

So enough review of our mockup, let's get busy creating a Drupal 8 theme based on the design we just previewed. This would be the time where we put on some good music and do the tedious work of installing Drupal 8, configuring content types, creating blocks and views, and populating our site with content so we have something to actually theme. However, let's skip all that tedious work and just start with a database snapshot.

## Restoring our database snapshot

Before we get started, let's open up the `Chapter05/start` folder located in our exercise files and restore the database snapshot by dropping the tables in our current Drupal 8 instance and importing the `drupal8.sql` file. Information on how to restore database snapshots was covered in *Chapter 1, Setting Up Our Development Environment*.

After restoring our database snapshot, we can browse our Drupal 8 instance by navigating to the homepage in our browser. We should see the typical **Bartik** theme being displayed with four pages containing content to match our mockup, including **Home**, **About Us**, **Blog**, and **Contact Us**.



At this point, we will need to log in to Drupal using the User login link, which has replaced the Drupal 7 login block. Once at the **Log in** screen, we enter admin's username and admin's password.

While we are using a very simple username/password combination to develop, I would advise you to use something stronger and more secure before moving any Drupal instance to a production web server. For more security features in Drupal 8, check out this article: <https://dev.acquia.com/blog/drupal-8/10-ways-drupal-8-will-be-more-secure/2015/08/27/6621>.

All of the content we need to recreate our mockup exists in the database, and as we begin to theme each section of our site, we will simply reference that content. If there is anything new or different with how content was created or configured, we will stop to briefly discuss it. For now, we will dive right into discussing the benefits of creating a custom theme and then proceed to setting up our theme folders.

## Creating a custom theme

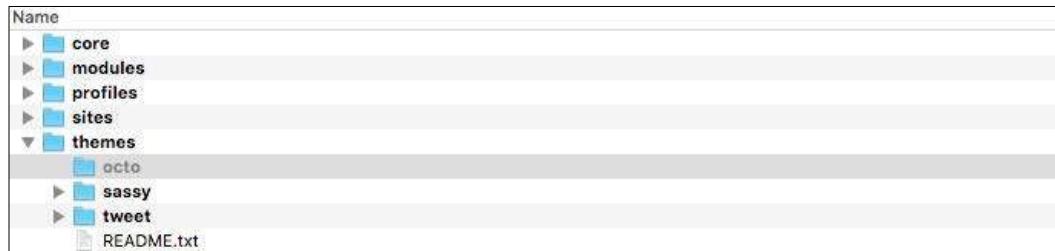
Previously, we looked at creating both a starter theme and a subtheme and, while each has its own benefit, we will often want to have the flexibility to develop on the fly. This means that we do not have to worry about managing a set of files already developed. This may sound contradictory to everything we have heard earlier, but taking an agile approach to theming allows for designers to create rich designs outside the boundaries of Drupal. With the introduction of Twig templates, we pretty much broke the mold on having to architect the layout of Drupal in a specific way. So, gone are the days of telling a designer that we can't implement their ideas.

As we create our custom theme, we will have the freedom to use whatever frontend tools are in the wild combined with the ability to implement both CSS and JS Frameworks using libraries, templates, and custom CSS/JS.

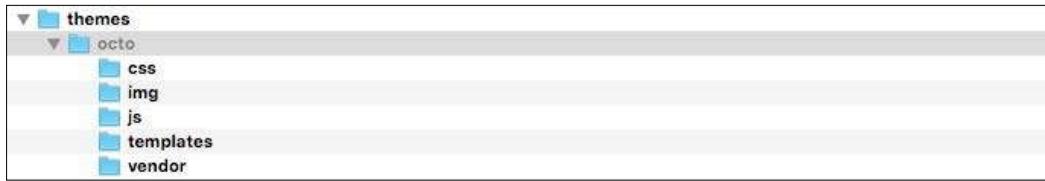
## Setting up theme folders

By now, setting up a theme should be second nature. We practiced this numerous times in *Chapter 4, Getting Started – Creating Themes*. But, in case we need a refresher, we can refer to that chapter for any outstanding questions.

We can begin by navigating to the `themes` folder and create a new folder named `octo`. This new folder will contain all of the files we will be using to develop our theme:



Next, we will create five additional subfolders within our main `themes` folder that will contain our CSS, Images, JavaScript, Twig templates, and any third-party vendor libraries such as Twitter Bootstrap. To ensure that we are all able to follow along without any naming conflicts, please make sure to name the five subfolders as follows:



We will be referencing these subfolders throughout the development of our theme with each folder containing the following files:

- **css:** This contains custom style sheets.
- **img:** This contains images used by the style sheets.
- **js:** This contains custom JavaScript.
- **templates:** This contains Twig templates.
- **vendor:** This contains JavaScript libraries.

## Adding a screenshot

Go ahead and copy `screenshot.png` from the `Chapter05/start/themes/octo` folder and place it within the `themes/octo` folder. Drupal will use this screenshot within the Appearance page to help visually identify our theme.

## Creating our configuration file

Any new theme must contain an `*.info.yml` file to define metadata, style sheets, libraries, and regions so that Drupal 8 recognizes that there is a new theme available to be installed. Let's begin by opening up our favorite text editor and creating a new file named `octo.info.yml`.

Our new configuration file will contain the following required metadata to start with:

```
name: Octo
type: theme
description: 'A responsive Drupal 8 theme.'
core: 8.x
base theme: false
```

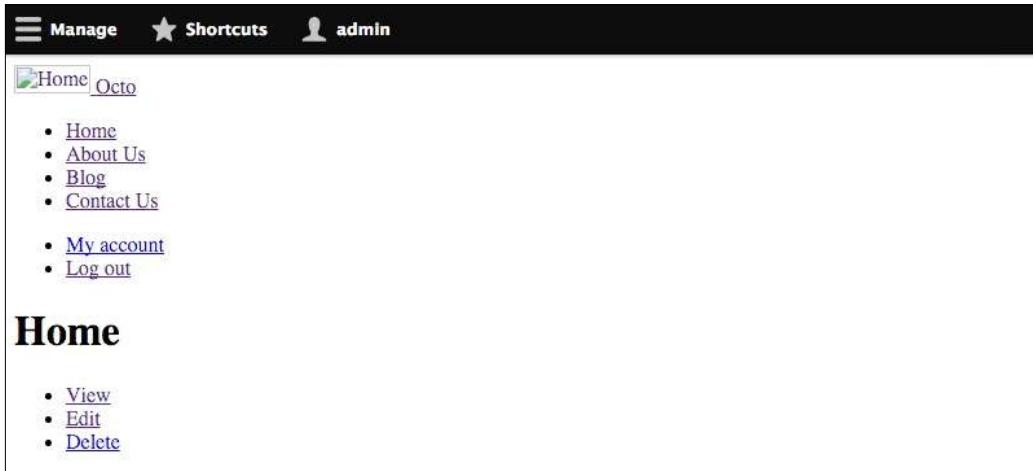
Make sure to save our changes and clear Drupal's cache. This will ensure that the theme registry picks up our changes.

## Installing our theme

Navigate to /admin/appearance and locate our new theme on the Appearance page. If we look within the **Uninstalled themes** section, we will see the **Octo** theme available to install, as shown in the following screenshot:



To install our theme, click on the **Install and set as default** link. With our new theme now installed, we can browse the homepage by clicking on the **Back to site** link in the **admin** menu. We will see that the original Bartik theme styling has been replaced with the non-styled look of our Octo theme, as shown in the following screenshot:



## Setting up our regions

Currently, our theme is using the default Drupal 8 regions as we have not specified any within our configuration file. This would be fine if the content in our Mockup worked nicely with those regions, but as we discovered earlier, there are definitely some regions outside of the default that will need to be defined.

For example, Drupal 8 provides only a single Footer region, and we will clearly need more than one region in our footer to accomplish the three blocks of content, as well as the two additional blocks of content that fall below our footer.

Let's edit our `octo.info.yml` file and define the specific regions we will need to place blocks of content needed by our homepage and interior pages. Begin by adding a new block of metadata to define regions with the following:

```
regions:
 header_top: 'Header Top'
 header: 'Header'
 primary_menu: 'Primary menu'
 secondary_menu: 'Secondary menu'
 page_top: 'Page top'
 page_bottom: 'Page bottom'
 highlighted: Highlighted
 headline: 'Headline'
 breadcrumb: Breadcrumb
 before_content: 'Before Content'
 content: Content
 sidebar_first: 'Sidebar first'
 sidebar_second: 'Sidebar second'
 after_content: 'After Content'
 footer_first: 'Footer first'
 footer_second: 'Footer second'
 footer_third: 'Footer third'
 footer_bottom_left: 'Footer Bottom Left'
 footer_bottom_right: 'Footer Bottom Right'
```

Make sure to save any changes to our `octo.info.yml` file and then clear Drupal's cache. Finally, we will want to confirm that our newly defined regions are available to be used by navigating to `/admin/structure/block` and looking at the Block layout page to verify that our regions are now available, as shown in the following image:

So far, we are progressing very nicely with adding our required metadata and regions that our theme will need, sort of a wash, rinse, and repeat pattern; with options like adding configuration information, clearing cache, and verifying changes being available. How about taking a look at adding some assets that we will need before we start tackling different page sections of our site?

## Setting up our assets

Drupal 8 definitely manages assets in a different way than we were used to in Drupal 7. With the introduction of Yaml configuration and asset libraries, we now have a separation of how CSS and JavaScript is referenced and used. Of course, we should be experts by now, but to recap, the process contains two steps.

1. First, we need to create a `*.libraries.yml` file, which will allow us to organize our theme's CSS, JavaScript, and dependencies.
2. Second, we will need to add the library reference to our theme's configuration file.

We will be using the Twitter Bootstrap library again with our theme. To ensure that we all use the same version of Bootstrap, we have provided a copy within the exercise files.

Begin by copying the Bootstrap folder contained in the `Chapter05/start/themes/octo/vendor` folder and place it within the `themes/octo/vendor` folder.

With Bootstrap accessible by our theme, we can create a new file named `octo.libraries.yml` and save it within the root of our theme. Next, we will want to add the following metadata to our `octo.libraries.yml` file:

```
bootstrap:
 version: 3.3.6
 css:
 theme:
 vendor/bootstrap/css/bootstrap.min.css: {}
 vendor/bootstrap/css/bootstrap-theme.min.css: {}
 js:
 vendor/bootstrap/js/bootstrap.min.js: {}
 dependencies:
 - core/jquery
```

The metadata we added basically starts with a variable name for how we want to access the library from our configuration file such as `octo/bootstrap`. Next, we need to make sure that we reference the version of the library we are adding followed by the path to the CSS, JS, and any dependencies. When dealing with YAML files, it is important to make sure that you have the proper indentations or else we may experience errors.

Now that we have our `octo.libraries.yml` file in place and have added a reference to Bootstrap, we need to open up our `octo.info.yml` file and add a pointer to our library in order for Drupal to recognize any assets that need to be loaded into our theme.

Open up the `octo.info.yml` file, and add the following prior to our regions section:

```
libraries:
 - octo/bootstrap
```

## *Prepping Our Project*

---

Make sure to save our changes and then clear Drupal's cache. If everything was successful, we should be able to navigate back to our homepage and note the Bootstrap CSS affecting our page elements. We can also confirm this by inspecting the page with Google Chrome and see in fact that Bootstrap is being loaded properly, as shown in the following screenshot:



## **Adding additional assets**

Now that we have a sense of how libraries are added to our theme, we will need to add additional assets including our images, CSS, and JavaScript files that we will be using to create our theme throughout the remaining lessons.

Begin by opening the `Chapter05/start/themes/octo` folder and copying the contents of the `css`, `img` and `js` folders and placing them into their respective folders inside the `themes/octo` folder, as shown in the following screenshot:



With our base assets in place, we will need to open back up our `octo.libraries.yml` file within our editor and add a global library entry that points to our new files.

Open up `octo.libraries.yml` and add the following entry directly below the bootstrap entry:

```
base:
 version: VERSION
 css:
 theme:
 css/styles.css: {}
 js:
 js/octo.js: {}
dependencies:
 - core/jquery
```

It's important to remember that the formatting and order of these entries is crucial in making sure that we don't experience any Drupal errors, so feel free to look at the completed `octo.libraries.yml` file located in the `Chapter05/end/themes/octo` folder as a reference. Just to clarify, we want to make sure that Twitter Bootstrap loads first, then our base styling and scripts.

Once we have completed adding the new entry, make sure to save the file, then open up `octo.info.yml`, and add the following reference to our `base` library so that it reads as follows:

```
libraries:
 - octo/bootstrap
 - octo/base
```

Save the file and don't forget to clear Drupal's cache, this will ensure that our changes take effect. Now if we browse back to our homepage and inspect the markup, we will see that the new references have been added to our page and our styling has changed again:



We will be adding additional assets to our `octo.libraries.yml` file as we address each page of our site that needs additional functionality, so it is important to be comfortable with this process.

## Handling default files

One last thing we need to make sure that we take care of before finishing this chapter is considering how Drupal handles default files. That is, files that we upload such as a logo, an image field on a content type, or any inline images that we would place directly in the content of our page.

Since we are using database snapshots to save time on having to recreate content, we want to make sure that any images that the database may be referencing are available to us and we don't encounter broken image paths.

We can take care of this by copying the contents of our `files` folder located within the `Chatper05/start/sites/default/files` folder and placing them into the `sites/default/files` folder of our Drupal 8 instance. If prompted to replace or overwrite files, go ahead and say yes to ensure that we have all the files needed.

Once we are done, our `sites/default/files` folder should contain subfolders with images organized based on the upload date, inline images, and styles for image derivatives, as shown in the following screenshot:



We have now completed adding all the assets and initial libraries that our new custom theme will need—a great start to strengthen our theming skills.

## Summary

The process of reviewing HTML mockups in preparation to convert it into a Drupal 8 theme takes time and patience. We need to make sure that we explore a website in great detail to spot possible layouts, regions, and user interactions that may require custom JavaScript and libraries. In this lesson, we accomplished the following:

- We broke down our HTML mockup page by page to enable us to better define what regions, layouts, and libraries we may need to create and configure for Drupal 8
- Having a clear starting point for everyone to begin theming from is important, and we used the database snapshots to ensure that we didn't have to work through the tedious process of entering content
- Finally, we began configuring our new theme by setting up our `octo.info.yml` file with regions and references to asset libraries that we set up within our `octo.libraries.yml` file

In the next chapter, we will dive into setting up our homepage layout to match the homepage of our mockup. We will begin creating Twig templates for our HTML wrapper and homepage as well as work further with libraries and assets.



# 6

## Theming Our Homepage

Any good design draws the user in with a visually exciting homepage, whether it is a clean, minimal navigation menu, great-looking photographs, or clear, concise information that keeps the user engaged. We are tasked with providing all of those features and more, though the thought of implementing a homepage with all these items may seem overwhelming at first. We will soon realize that they are just a series of steps that will become the norm for any Drupal 8 project. In this chapter, we will walk through implementing the following:

- We will start with the obvious task of applying our website logo and working with the new site branding block. This will be followed by creating our first Twig template to handle our HTML wrapper and any assets and functionality that should be globally applied.
- Next, we will address converting our mockup's homepage markup into a Twig template with various regions to hold content.
- We will start with static content and then slowly convert it into dynamic content with blocks for our search block, menu, and other regions.
- Because aggregating data is such an integral part of Drupal with Views now in core, we will discover how to replicate content to use with the theming of our homepage slider.

As we work through each section, we have the ability to refer back to the `Chapter06` exercise files folder. Each folder contains the `start` and `end` folders with files that we can use to compare our work when needed. This also includes database snapshots that will allow us to all start from the same point when working through various lessons.

## Creating our HTML wrapper

In order to start addressing the markup of our homepage, we need to look at creating our first Twig template. The `html.html.twig` template is a little different than most templates, as it contains the basic structure or wrapper for a Drupal page that the rest of our templates will inherit. This template contains your standard HTML5 markup containing `html`, `head`, `title`, and `body` elements along with any other variables that Drupal 8 needs to output content.

We can begin by navigating to `core/modules/system/templates` and copying the `html.html.twig` Twig template to our `themes/octo/templates` folder. One thing to keep in mind as we start working with the Twig templates is that we will always copy a template from core to our themes folder to ensure that we don't accidentally modify any core files.

Next, we can open `html.html.twig` and review the markup in our editor. We have the following code:

```
<!DOCTYPE html>
<html{{ html_attributes }}>
 <head>
 <head-placeholder token="{{ placeholder_token|raw }}>
 <title>{{ head_title|safe_join(' | ') }}</title>
 <css-placeholder token="{{ placeholder_token|raw }}>
 <js-placeholder token="{{ placeholder_token|raw }}>
 </head>
 <body{{ attributes }}>

 {{ 'Skip to main content'|t }}

 {{ page_top }}
 {{ page }}
 {{ page_bottom }}
 <js-bottom-placeholder token="{{ placeholder_token|raw }}>
 </body>
 </html>
```

The markup is similar to any other HTML document, with the addition of Twig variables and filters to output attributes, title, regions, and placeholders for CSS/JS. For example, `<css-placeholder token="{{ placeholder_token|raw }}>` outputs any CSS files that we added to our `*.libraries.yml` file and have referenced from within our themes configuration. Then, the `{{ page }}` variable will output the contents of any `page.html.twig` templates that it calls.

If we begin to compare the `html.html.twig` template to the markup of our homepage mockup, we can start to visualize how things come together.

## Introducing web fonts

Our mockup takes full advantage of Google Fonts by adding it to the head of our document. The external reference allows our CSS to render the typography on various pages. The only problem is that currently we are not including the web fonts in our Drupal theme. Because we cannot download Google Fonts and use them locally, they need to be externally hosted. But how do we add externally hosted files to a `*.libraries.yml` file?

The answer is actually quite simple. We need to specify the file type as external, and adding an external asset is something new we have yet to discuss. So, we can walk through the steps involved:

1. Open `octo.libraries.yml`.
2. Add the following entry:

```
webfonts:
 version: VERSION
 css:
 theme:
 //fonts.googleapis.com/css?family=Open+Sans:300,400,
 600,700,800|Roboto+Slab: { type: external }
```

3. Save `octo.libraries.yml`.
4. Open `octo.info.yml`.
5. Add the following library reference pointing to the entry of our new web fonts:

```
libraries:
 - octo/bootstrap
 - octo/webfonts
 - octo/base
```

6. Save `octo.info.yml`.

Make sure to clear Drupal's cache and refresh our homepage. If we inspect the page, we should see our external reference to Google Fonts being loaded directly after Twitter Bootstrap. Now that our HTML wrapper is complete, we can move on to creating our homepage template.

## Creating our homepage

The next item we will move on to is creating the main homepage template. By default, Drupal uses the `page.html.twig` template to render any regions we have defined within our configuration. Because we broke out our Mockup into functional areas, we have a sense of what each region will contain. Our job is to recreate the homepage, which will require us to follow these basic theming techniques.

1. First, we will take advantage of Drupal's file name suggestions to create our homepage template.
2. Then, we will replace the contents of our template with the contents of our Mockups homepage.
3. Finally, we will need to review the output of our new template.

## Using page templates

If we inspect the homepage, it is currently using the core `page.html.twig` template to output content. But if we take advantage of the FILE NAME SUGGESTIONS provided, we are presented with a couple of additional choices for displaying content.

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'page' -->
<!-- FILE NAME SUGGESTIONS:
 * page--front.html.twig
 * page--.html.twig
 x page.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/system/templates/page.html.twig' -->
►<div class="layout-container">...</div>
<!-- END OUTPUT from 'core/modules/system/templates/page.html.twig' -->
```

The reason we are interested in alternative templates for our homepage is due to the fact that as we navigate from page to page, we have clear layout changes. Our homepage has a completely different layout than our interior page, with the exception of any global elements. Knowing this, it would make sense to create a separate homepage template to manage our content.

We can begin by following these steps:

1. Navigate to `core/modules/system/templates` and copy `page.html.twig`.
2. Place the copy within our `themes/octo/templates` folder.
3. Rename `page.html.twig` to the suggested name of `page--front.html.twig`.

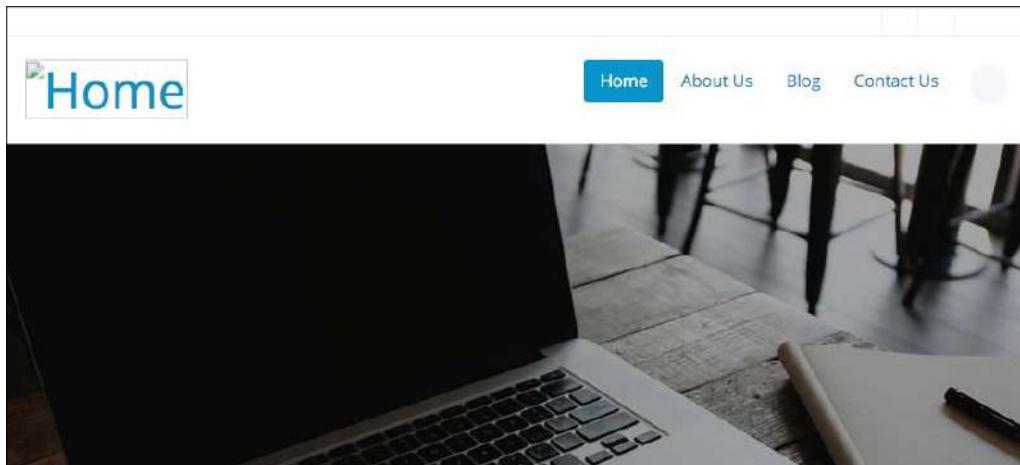
Make sure to clear Drupal's cache and refresh our homepage. If we inspect the page again, we will note that we have an indicator next to `page--front.html.twig` under the FILE NAME SUGGESTIONS, and the output is now pointing to our themes folder.

## Working with static content

When working with a mockup, the easiest way to start any theming project is by simply replacing the Twig templates contents with the static content from our design.

1. Open `page--front.html.twig` and delete the entire contents.
2. Navigate to `Mockup/index.html` and copy the markup between the opening and closing `body` element minus the JavaScript references and paste it into `page--front.html.twig`.
3. Save `page--front.html.twig`.

Make sure to clear Drupal's cache and refresh the homepage within our browser. We should now have a working copy of the homepage mockup; well, sort of.



If it was only that simple, we would all be Drupal rock stars. What we do have though is a good starting point for which we can begin to replace static content with dynamic content. Also, we can validate the following:

- Our `page--front.html.twig` template is only being used on the homepage, or else our Error page would look the same once we navigated to any other page.
- Our CSS, JavaScript, and assets are being included properly or else our homepage would look horrible.

## Implementing our Header Top region

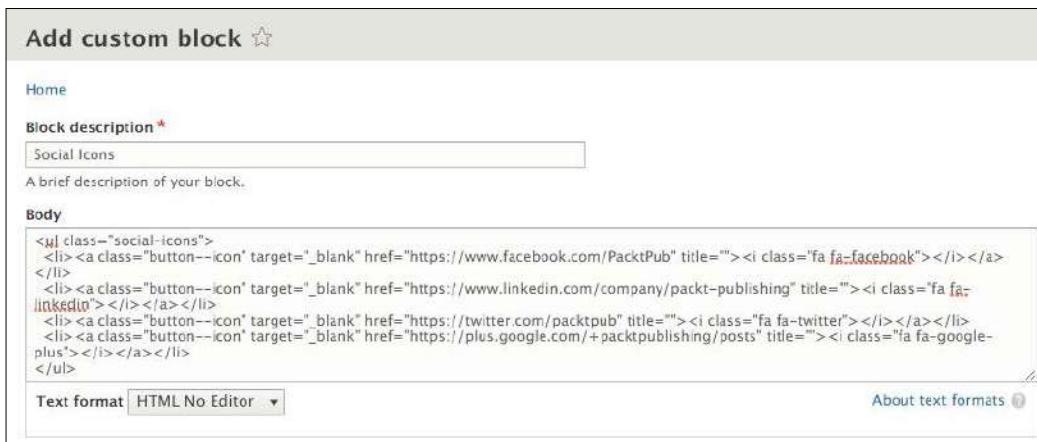
The first item we will need to address is replacing the static content in our Header Top region. Referring back to the Mockup, we have a menu of social icons that display within this region. Also, if we look closely, each social icon is using the Font Awesome library. Tackling the next section will require the following techniques:

1. First, we will create a custom block to display our Social Icons menu and then assign it to the Header Top region so that it is available to render.
2. Next, we will add the Font Awesome library for our social icon to use.
3. Finally, we will modify our static content to display our Header Top region.

## Creating our Social Icons block

Creating blocks of content is fairly simple, and we will be doing this often as we recreate each section of our theme. We will start by navigating to the **Block layout** page at /admin/structure/block and following these nine steps:

1. Click on the **Place block** button in the **Header Top** region.
2. Click on the **Add custom block** button.
3. Enter a **Block description** of Social Icons.
4. Select **HTML No Editor** from the **Text format** dropdown.
5. Add the markup located in the `Chapter06/start/content/SocialIcons.txt` file to the **Body** field, as shown in the following image:



6. Click on the **Save** button to proceed to the **Configure block** screen.
7. Uncheck the **Display title** checkbox.

8. Choose **Header Top** from the **Region** field.
9. Click on the **Save block** button, as shown in the following image:

The screenshot shows the 'Configure block' interface for a 'Social Icons' basic block. At the top, there's a success message: 'Basic block Social Icons has been created.' Below it, the 'Block description' is set to 'Social Icons'. The 'Title' is 'Social Icons' (Machine name: socialicons). The 'Display title' checkbox is unchecked. Under 'Visibility', the 'Content types' section shows 'Not restricted' and a dropdown menu for 'Pages' (selected) and 'Roles'. Another 'Content types' section on the right lists 'Headlines', 'Landing page', 'Post', and 'Team'. In the 'Region' section, 'Header Top' is selected. A note says 'Select the region where this block should be displayed.' At the bottom is a blue 'Save block' button.

We now have our Social Icons block created and assigned to our Header Top region. Next, we need to add the Font Awesome icon library.

## Installing Font Awesome library

Font Awesome is an icon font implementation that allows scalable vector icons to be referenced the same way you would a font family. Our social icons as well as other sections of our theme will take advantage of Font Awesome to display various icons. We can find detailed information regarding Font Awesome at <https://fontawesome.github.io/Font-Awesome>. To ensure that we all use the same version of Font Awesome, we will need to navigate to the Chapter06/start/themes/octo/vendor folder and copy the font-awesome folder to our themes/octo/vendor folder.

Once the files are accessible by Drupal, we can follow these remaining steps to add Font Awesome to our theme:

1. Open `octo.libraries.yml`.
2. Add the following entry:

```
font-awesome:
 version: 4.3.0
 css:
 theme:
 vendor/font-awesome/css/font-awesome.min.css: {}
```

3. Save `octo.libraries.yml`:
4. Open `octo.info.yml`:
5. Add the following library reference pointing to our new web fonts entry:

```
libraries:
 - octo/bootstrap
 - octo/webfonts
 - octo/font-awesome
 - octo/base
```

6. Save `octo.info.yml`:

Make sure to save our changes and clear Drupal's cache for our changes to take effect. Finally, we will need to add the Header Top region to our homepage before being able to see our Social Icons menu.

## Refactoring Header Top region

Currently, our `page--front.html.twig` template is not outputting our Header Top region. We will need to refactor our markup to replace the static content with the output of the `{{ page.header_top }}` region.

### Current markup

```
<div class="header-top">
 <div class="container">
 <div id="block-socialicons" class="block">
 <ul class="social-icons">
 <a class="button--icon" target="_blank"
 href="https://www.facebook.com/PacktPub" title="">
 <i class="fa fa-facebook"></i>
 <a class="button--icon" target="_blank"
 href="https://www.linkedin.com/company/
 packt-publishing" title="">
 <i class="fa fa-linkedin"></i>
```

```

<a class="button--icon" target="_blank"
 href="https://twitter.com/packtpub" title="">
 <i class="fa fa-twitter"></i>
<a class="button--icon" target="_blank"
 href="https://plus.google.com/+packtpublishing/posts"
 title=""><i class="fa fa-google-plus"></i>

</div>
</div>
</div>

```

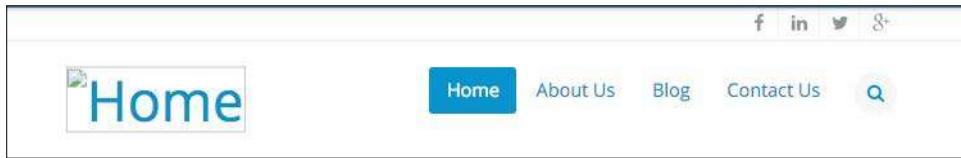
### New markup

```

<div class="header-top">
 <div class="container">
 {{ page.header_top }}
 </div>
</div>

```

Make sure to save our changes, clear Drupal's cache and then refresh the browser. If all was successful, our Header Top region should now be identical to the mockup:



## Implementing our Header region

The second item we will need to address is replacing the static content in our Header region. Referring back to the mockup, we have a logo, menu, and search form, each with their respective functionality. Tackling this next section will require quite a few more steps:

- First, we will address the logo that has been moved into the brand new site branding block. We will upload a new logo, assign the block to our header region, and work with block templates.
- Next, we will use Twig to print our Header region within our homepage to view any blocks assigned to it.
- Then, we will work with the search form block and create both the block template and the input template while introducing some new Twig techniques to work with variables. We will also add our first custom JavaScript to enable the toggle functionality.

- We will also take a look at our main menu and work with menu templates to modify the markup to match our design.
- Finally, we will add our custom script to make our header region sticky, so that as our user scrolls down the page, the header remains within our view.

## Adding a logo

Currently, our website is not displaying a site logo. This is in part due to the fact that we have yet to upload a logo for our theme. We can address this by navigating to /admin/appearance/settings in our browser, as shown in the following image:

The screenshot shows the 'Appearance settings' page. At the top, there are tabs for 'List', 'Update', and 'Settings'. Below the tabs, there are theme selection buttons for 'Global settings', 'Bartik', 'Seven', and 'Octo'. The 'Octo' theme is selected. The main content area shows the 'Appearance' path in the breadcrumb. A note states: 'These options control the default display settings for your entire site, across all themes. Unless they have been overridden by a specific theme, these settings will be used.' There are two expandable sections: 'TOGGLE DISPLAY' and 'LOGO IMAGE SETTINGS'. Under 'TOGGLE DISPLAY', there are checkboxes for 'User pictures in posts', 'User pictures in comments', 'User verification status in comments', and 'Shortcut icon', all of which are checked. Under 'LOGO IMAGE SETTINGS', there is a single checkbox for 'Use the default logo supplied by the theme', which is checked.

We can upload a new image by following these four steps:

1. Uncheck **Use the default logo supplied by the theme**.
2. Click on the **Choose File** button under the **Upload logo image** field.
3. Select the `logo.png` file located in the `Chapter06/start/themes/octo/img` folder.
4. Click on the **Save configuration**.

We should now see that the path to custom logo displays as `logo.png` with the path to our file being `public://logo.png` or `sites/default/files/logo.png`, as shown in the following image:

**▼ LOGO IMAGE SETTINGS**

Use the default logo supplied by the theme.

**Path to custom logo**  
logo.png  
Examples: `logo.png` (for a file in the public filesystem), `public://logo.png`, or `sites/default/files/logo.png`.

**Upload logo image**  
 No file chosen  
If you don't have direct file access to the server, use this field to upload your logo.

If for some reason the path to the image is different but the logo still displays properly, it may be due to us uploading a logo in the previous chapter.

## Enabling Site branding

In Drupal 8, the Site logo, Site name, and Site slogan have been moved into a brand new Site branding block. We will need to place this block into the Header region, so that it will display later once we add the region to our `page--front--html.twig` template.

Begin by navigating to `/admin/structure/block` and locate the **Disabled** region section. We will see the **Site branding** block currently disabled. We can place the **Site branding** block into the **Header** region by following these steps:

1. Select **Header** from the **Region** dropdown.
2. Click on the **Save blocks** button.

Now that the block is assigned to our Header region, we can continue with configuring it by clicking on the **Configure** button to the right of the **Site branding** block:

**TOGGLE BRANDING ELEMENTS**

Site logo  
Defined on the [Appearance Settings](#) or [Theme Settings](#) page.

Site name  
Defined on the [Site Information](#) page.

Site slogan  
Defined on the [Site Information](#) page.

Choose which branding elements you want to show in this block instance.

Located under the **TOGGLE BRANDING ELEMENTS**, we have the option of enabling or disabling specific page elements. In our case, we only want the Site logo to be displayed:

1. Uncheck **Site name**.
2. Uncheck **Site slogan**.
3. Click the **Save block** button.

Now that we have our logo uploaded and our site branding block assigned to a region, we need to add the Header region to our homepage template.

## Printing our Header region

Within our `page--front.html.twig` template, we want to be able to see any blocks of content that we assign to the Header region. In order to do this, we will need to add the `{{ page.header }}` variable.

1. Open `page--front.html.twig`.
2. Add the Twig variable `{{ page.header }}` directly below the `header-nav` container so that our markup looks like the following:

```
<div class="header-nav container">
{{ page.header }}</div>
```

3. Save `page--front.html.twig`.

Make sure to save our changes, clear Drupal's cache, and then refresh the browser. Currently, we are displaying two logos, which we will address next by moving our static markup in a new block template.

## Creating Block templates

In an attempt to clean up our page template, we will create a block template for our site branding block. However, we first need to know the location of the Twig template that Drupal is using to output our logo. When we inspect the page, we should see the following:

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'block' -->
<!-- FILE NAME SUGGESTIONS:
 * block--octo-branding.html.twig
 * block--system-branding-block.html.twig
 * block--system.html.twig
 * block.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/system/templates/block--system-branding-block.html.twig' -->
▶ <div id="block-octo-branding" class="contextual-region"></div>
<!-- END OUTPUT from 'core/modules/system/templates/block--system-branding-block.html.twig' -->
```

Now that we know the path to `block--system-branding-block.html.twig`, we can grab a copy and place it within our themes templates folder:

1. Navigate to `core/modules/system/templates` and copy the `block-system-branding-block.html.twig` template to our `themes/octo/templates` folder.
2. Open `block--system-branding-block.html.twig` and delete the current markup.
3. Replace the content with the following markup:

```
<div class="navbar-header">
 <button type="button" class="navbar-toggle"
 data-toggle="collapse" data-target=".navbar-main">
 <i class="fa fa-bars"></i>
 </button>

 {% if site_logo %}
 <h1 class="logo">
 <a href="{{ url('front') }}" title="{{ 'Home'|t }}"
 rel="home" id="logo">

 </h1>
 {% endif %}
</div>
```

4. Save `block--system-branding-block.html.twig`.
5. Open `page--front.html.twig`.
6. Delete the `navbar-header` section.
7. Save `page--front.html.twig`.

Make sure to save our changes, clear Drupal's cache, and then refresh the browser. Our header is coming along nicely, and should look like the following image:



There are a lot of different Twig variables, filters, and conditional logic happening with the markup for our `block--system-branding-block.html.twig` template, so let's take a moment to explain.

- First, we are using Twig conditional logic `{% ... %}` to test whether the site logo exists, and if so, print the markup between.
- Second, we are replacing the `href` value with `{{ url('front') }}`, which outputs the current URL path to our homepage.
- Third, we are using a Twig translation filter `{{ 'value|t' }}` to translate the values in the `title` and `alt` attributes.
- Finally, we are using a Twig variable `{{ site_logo }}` to grab the path to our logo.

If for some reason our header does not look like what we are expecting, we can always take a look at the `Chapter06/end/themes/octo/templates` folder and compare our Twig templates with the completed ones.

## Implementing our search form block

The next item we will move on to is replacing the search functionality within our page header. Referring back to the mockup, we have a hidden search form that can be toggled to display by the end user. Tackling this next section will require multiple theming techniques.

1. First, we will need to assign Drupal's search for block to the Header region.
2. Next, we will need to create a Twig template for the search block and move the markup out of our homepage and into the template.
3. Finally, we will need to add the proper JS to enable the toggling of the search field.

## Placing our search form block

Currently, we do not have a search form block available to us within our Block layout. We will need to locate the block and place it within our Header region.

1. Navigate to `/admin/structure/block`.
2. Locate the **Header** region.
3. Click on the **Place block** button.
4. Locate the **Search form** block from the **Place block** dialog.

5. Click on the **Place block** button.
6. Uncheck the **Display title** checkbox.
7. Click on the **Save block** button from the Block layout page.

Our Header region should now contain two blocks: one for site branding and the other for our search form.



If we navigate back to our homepage, we should see our search form block displayed to the right of our logo. We will want to refactor our navbar-search markup by adding it to the search form block being output by Drupal.

## Creating a search form block template

If we inspect the search form block using Chrome's developer tools, we will identify that Drupal is using the general `block.html.twig` template. We want to be more specific with our naming convention as we will be modifying the markup in this block. Based on the `FILE NAME SUGGESTIONS`, we can create a new block named `block--search-form-block.html.twig`. Follow these steps:

1. Navigate to `core/modules/block/templates` and copy the `block.html.twig` template to our `themes/octo/templates` folder.
2. Rename `block.html.twig` to `block--search-form-block.html.twig`.
3. Replace the content with the following markup:

```
<div class="navbar-search">
<div class="search-icons">
 <i class="fa fa-search"></i>
 <i class="fa fa-times"></i>
</div>
<div class="search-block-form">
 {{ content }}
</div>
</div>
```

4. Save `block--search-form-block.html.twig`.

Our search form block is now in place, but we still need to remove the navbar-search section from our `page--front.html.twig` template so that we don't have duplicate markup.

1. Open `page--front.html.twig`.
2. Delete the navbar-section of the markup:

```
<div class="navbar-search">
 ...content...
</div>
```

Make sure to save our changes, clear Drupal's cache and then refresh the browser. Our search form block is now in place and styled similarly to our mockup. However, if we click on the search icon, nothing will happen. We are currently missing the custom JavaScript to enable this functionality.

## Adding custom JavaScript

Initially, when we added our base styling to our `octo.libraries.yml` file, we also referenced a custom JavaScript file that is located in our `themes/octo/js` folder titled `octo.js`. If we open this file, we will see the shell to add jQuery that is initiated once the page has finished loading. We will be adding any custom script within this function:

```
! function($) {
 $(document).ready(function() {
 });
} (jQuery);
```

As this is not meant to be a JavaScript lesson, we will not be going into detail about any scripts we added to this function. We will, though briefly, discuss the intention of each script.

In order for our search form to be unhidden and hidden, it relies on the click event of the search icon being triggered. We can add the following script to our function to enable this interaction:

```
//-- Search icon
(function() {
 $(".open-form").click(function() {
 $(".open-form").hide();
 $(".close-form").css("display", "block");
 $(".search-block-form").show();
 $(".search-block-form input").focus();
```

```

 return false;
 });
 $(".close-form").click(function() {
 $(".close-form").hide();
 $(".open-form").css("display","block");
 $(".search-block-form").hide();
 return false;
 });
})();

```

We can also find the completed `octo.js` file within the `Chapter06/end/themes/octo/js` folder to compare with our file. Make sure to save the file, clear Drupal's cache, and then refresh our homepage. If we click on the search icon, we should see our search form being displayed, as shown in the following image:



We are not quite done; the input element is not displaying over the menu. We can clearly see the outline of the input and the placeholder attribute, which prompts the user what to enter into our input, is missing. We can fix this by adding one additional Twig template for the input element.

## Creating an input element template

If we inspect the markup for the search input, we can identify that the Twig template being used by Drupal is `input.html.twig`. As is the case with all input elements, we may find that this is not sufficient. Using the FILE NAME SUGGESTIONS, we can create a new input template titled `input--search.html.twig`:

1. Navigate to `core/modules/system/templates` and copy the `input.html.twig` template to our `themes/octo/templates` folder.
2. Rename `input.html.twig` to `input--search.html.twig`.
3. Replace the content with the following markup:

```

{% set classes = ['form-control',] %}

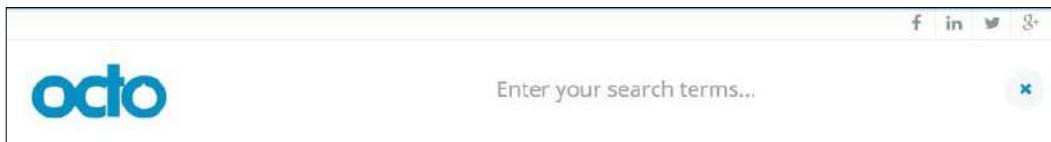
<input{{attributes.addClass(classes).setAttribute
('placeholder','Enter your search terms...') }} />
{{ children }}

```

### *Theming Our Homepage*

---

Make sure to save our changes, clear Drupal's cache, and then refresh the browser. Now if we click on the search icon, everything should look and function exactly like our mockup.



Therefore, we once again are introducing some new Twig functionality that should be explained in a little more detail.

First, we are using adding a new CSS class named `form-control` that is specific to Twitter Bootstrap. We are then setting a Twig variable named `classes` with that value, and then using the Twig function `attributes.addClass()` to pass the value to Drupal.

Second, we are using a second Twig function `setAttribute()`, which allows us to add the `placeholder` attribute with a value of **Enter your search terms**.

As we are starting to see, the new Twig functionality in Drupal is very powerful and allows us to achieve most theming requirements directly in a Twig template without the need to use the Theme layer.

## Working with menus

When dealing with our main menu, we are using Drupal's Main navigation menu block. This block is already assigned to our Primary menu region, which makes it very easy to access within our `page--front.html.twig` template:

1. Open `page--front.html.twig`.
2. Add the `{{ page.primary_menu }}` variable to our page directly below the `{{ page.header }}` variable.
3. Save `page--front.html.twig`.

Make sure to save our changes, clear Drupal's cache, and refresh the browser. We will now see the Main navigation block being displayed to the right of our logo, as shown in the following image:



Taking a closer look, it's clear that the unordered list for our menu is missing the following classes, `nav nav-pills nav-main`, which is causing the menu items to not display inline or show with the Bootstrap pill formatting that our static menu is displaying.

## Creating a menu template

If we inspect the markup for the main menu, we can identify the Twig template being used by Drupal is `menu.html.twig`. Using the `FILE NAME SUGGESTIONS`, we can create a more specific template titled `menu--main.html.twig`.

1. Navigate to `core/modules/system/template` and copy `menu.html.twig` to the `themes/octo/templates` folder.
2. Rename `menu.html.twig` to `menu--main.html.twig`.
3. Now we will need to open our new template, locate the first unordered list, and add the missing CSS classes by replacing the following markup:

### **Current markup**

```
<ul{{ attributes }}>
```

### **New markup**

```
<ul{{ attributes.addClass('nav nav-pills nav-main') }}>
```

Make sure to save our changes, clear Drupal's cache, and refresh the browser. We will now see the Main navigation block displaying inline similar to our static menu.

## Creating System Menu block template

Similar to how we handled the markup for our site branding, it would be much easier to manage the `navbar-main` wrapper within the block that outputs our menu. This will allow us to also remove the `navbar-main` markup completely from our homepage template.

If we inspect the markup for the main menu, we can identify the block template being used by Drupal is `block--system-menu-block.html.twig`. Knowing the fact that there will only ever be a single main menu, we can feel confident using this same template for our needs:

1. Navigate to `core/modules/system/template` and copy `block--system-menu-block.html.twig` to the `themes/octo/templates` folder.
2. Replace the content with the following markup:

```
<div class="navbar-main navbar-collapse collapse">
{{ content }}
</div>
```

3. Save `block--system-menu-block.html.twig`.
4. Open `page--front.html.twig`.
5. Modify the `header-nav` section to look like the following:

```
<div class="header-nav container">
{{ page.header}}
{{ page.primary_menu }}
</div>
```
6. Save `page--front.html.twig`.

Make sure to clear Drupal's cache and then refresh the homepage. Our menu is now complete and functional. Finally, we need to add our custom script that will turn our header into a sticky header.

## Creating a sticky header

One of the more common UI improvements seen around the Web is the implementation of sticky headers. Our mockup implements this with a little bit of CSS and some custom JavaScript.

First, the markup for our header region contain a class of `header` that will be used to add an additional class of `sticky` once the user has scrolled down the page a certain number of pixels.

Second, we can use the **Document Object Model (DOM)** with JavaScript to determine how far the user has scrolled past a specific element in our markup. We can use the reverse to then remove the `sticky` class once they have scrolled back to the top of the page.

1. Open `octo.js` located in the `themes/octo/js` folder.
2. Add the following script block:

```
//-- Sticky Header
(function() {
 var mainnav = $('.header');
 if (mainnav.length) {
 var elmHeight = $('.header-top').outerHeight(true);
 $(window).scroll(function() {
 var scrolltop = $(window).scrollTop();
 if (scrolltop > elmHeight) {

 if (!mainnav.hasClass('sticky')) {
 mainnav.addClass('sticky');
 }
 }
 });
 }
});
```

```

 } else {
 mainnav.removeClass('sticky');
 }
 })
})
})();

```

Make sure to save our changes, clear Drupal's cache and then refresh the browser. If we begin to scroll down the page, we will see the Header region stick to the top of our browser. Scrolling back up to the top, our Header region then returns to normal.

We have successfully completed the header of our homepage. Complete it with logo, dynamic main menu, and search functionality.

## Implementing our Headline Region

The third item we will need to address is replacing the static content in our Headline region. Referring back to the mockup, we have a responsive slider, parallax content, and a jump to the menu icon. Tackling the next section will introduce some new steps:

1. First, we will address the Headline slider, which will require us to build a view to aggregate Headline content using a block display.
2. Next, we will assign the new block to our Headline region and refactor the markup.
3. Finally, we will add a JS library for FlexSlider to enable the responsive slider.

## Creating our Headline View and Block

Drupal 8 has taken the popular Views module and integrated it into the core module system. We can take advantage of Views to aggregate the content that our Headline slider needs.

To get started, we will need to navigate to /admin/structure/views and click on the **Add new view** button from the **Views** Admin screen, where we will add the following information:

- **VIEW BASIC INFORMATION:**
  1. **View name:** Headlines.
  2. Check the **Description** box.
  3. **Description:** A listing of Headlines.

## *Theming Our Homepage*

---

- **VIEW SETTINGS:** Show: Content of type: Headlines sorted by: Newest first

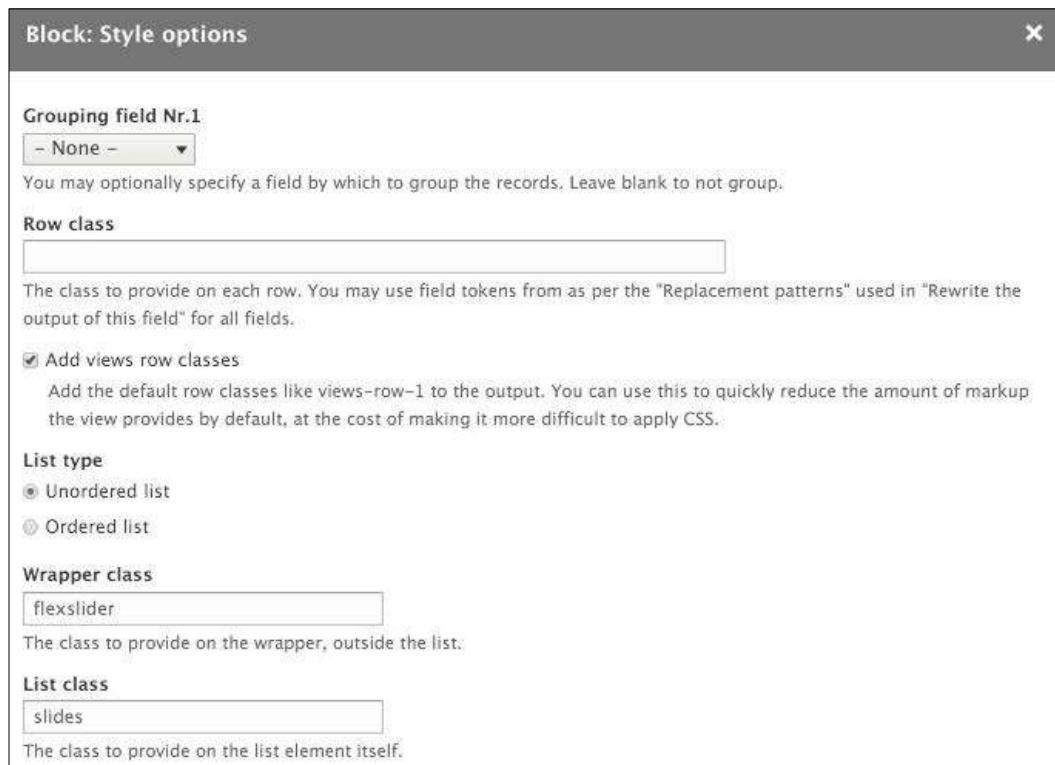
The screenshot shows the 'Add new view' page. At the top, it says 'Add new view ★'. Below that, the breadcrumb navigation is 'Home » Administration » Structure » Views'. The main area is divided into sections: 'VIEW BASIC INFORMATION' and 'VIEW SETTINGS'. In 'VIEW BASIC INFORMATION', there is a 'View name \*' field containing 'Headlines' with a note 'Machine name: headlines [Edit]' and a checked 'Description' checkbox with the text 'A listing of Headlines'. In 'VIEW SETTINGS', there are dropdown menus for 'Show: Content', 'of type: Headlines', and 'sorted by: Newest first'.

- **BLOCK SETTINGS:**
  1. Check the **Create a block** checkbox.
  2. **Block title:** Headlines.
- **BLOCK DISPLAY SETTINGS:**
  1. **Display format:** HTML List of: titles.
  2. **Items per block:** 3.
  3. Click on the **Save and edit** button.

The screenshot shows the 'BLOCK SETTINGS' and 'BLOCK DISPLAY SETTINGS' configuration dialog. Under 'BLOCK SETTINGS', the 'Create a block' checkbox is checked and 'Block title' is set to 'Headlines'. Under 'BLOCK DISPLAY SETTINGS', 'Display format' is set to 'HTML List' and 'of: titles'. 'Items per block' is set to '3'. There is also an unchecked 'Use a pager' checkbox. At the bottom are 'Save and edit' and 'Cancel' buttons.

Now that our Headlines view has been created, we will need to add an additional field and adjust the format settings before we can use our new block. With the view still open, we will need to make the following adjustments to the **Block: Style options**.

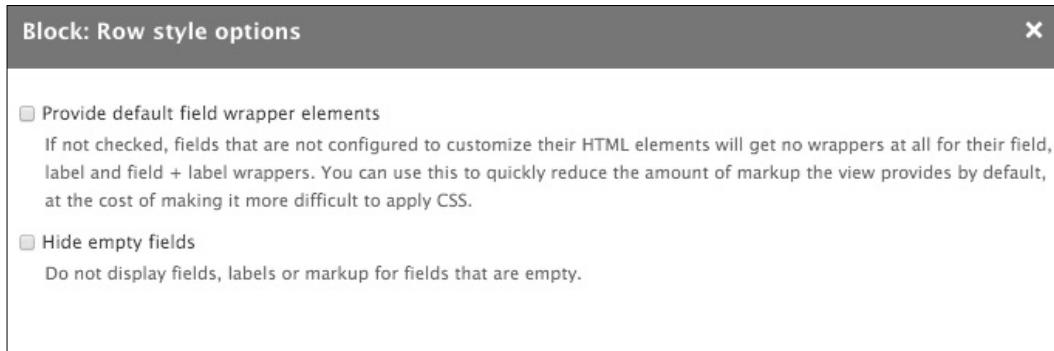
1. Click on the **Settings** link next to the **Format: HTML List** link under the **FORMAT** section.
2. Change the **Wrapper class** from **item-list** to **flexslider**.
3. Add a **List class** named **slides**.
4. Click on the **Apply** button.



Next, we will need to make adjustments to the **Block: Row style options**:

1. Click on the **Settings** link next to the **Show: Fields** link under the **FORMAT** section.
2. Uncheck the **Provide default field wrapper elements** checkbox.

3. Click on the **Apply** button.



We now need to add an additional field for our Headlines view that will display the subheading under the main heading of our content. We can accomplish this by performing the following steps:

1. Click on the **Add** button in the **Fields** section.
2. From the **Add fields** dialog, enter subheading in the **Search** field
3. Check the checkbox next to **Sub Heading**.
4. Click on the **Apply (all displays)** button.
5. From the **Configure field: Content: Sub Heading** dialog, expand the **STYLE SETTINGS** and uncheck **Add default classes** checkbox.
6. Click on the **Apply** button and then on the **Save** button to make sure that our changes to the Headlines view have been saved.

One last field configuration we will need to make is to the Title field. We need to have it displayed as an H2 heading for styling and SEO purposes.

1. Click on the **Content: Title** link in the **FIELDS** section.
2. Expand the **STYLE SETTINGS** section.
3. Check the **Customize field HTML** checkbox.
4. Select **H2** from the **HTML Element** field.
5. Uncheck the **Add default classes** checkbox.
6. Click on the **Apply (all displays)** button.

At this point, make sure to save the View, and then we can move on to the next step of placing our block.

## Adding our Headlines Block

We can start by navigating to /admin/structure/block and following these steps:

1. Click on the **Place block** button in the **Headline** region.
2. Locate the **Headlines** block.
3. Click on the **Place block** button.
4. Uncheck the **Display title** checkbox.
5. Click on the **Save block** button.

We now have our Headlines block placed into the Headline region for us to be able to output from our homepage template.

## Printing our Headline region

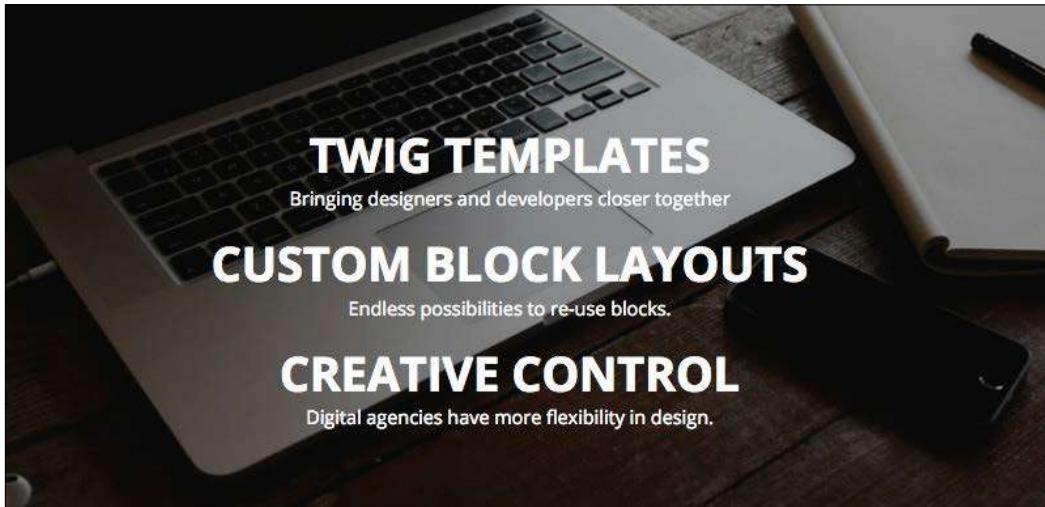
Within our `page--front.html.twig` template, we want to be able to display our Headlines block. In order to do this, we will need to add the `{{ page.headline }}` variable.

1. Open `page--front.html.twig`.
2. Add the Twig variable `{{ page.headline }}` directly below the `headline` container and delete the remaining markup between the opening and closing `headline` container so that our markup looks like the following:

```
<div class="headline">
 {{ page.headline }}
</div>
```

3. Save `page--front.html.twig`.

Make sure to save our changes, clear Drupal's cache, and refresh the browser. Feel free to review the `page--front.html.twig` template located in the `Chapter06/end/themes/octo/templates` folder to compare the markup if needed. If everything was done properly, our Headline region should be displaying three headlines, as shown in the following image:



## Configuring FlexSlider library

FlexSlider is a fully responsive jQuery slider developed by Woo Themes. The slider is very easy to implement and has numerous configuration options. We can find detailed information regarding FlexSlider at <https://www.woothemes.com/flexslider>.

To ensure that we use the same version of FlexSlider, we will need to navigate to the `Chapter06/start/themes/octo/vendor` folder and copy the `flexslider` folder to our `themes/octo/vendor` folder. Once the files are accessible by Drupal, we can follow these remaining steps to add FlexSlider to our theme:

Open `octo.libraries.yml` and add the following entry:

```
flexslider:
 version: 2.5.0
 css:
 theme:
 vendor/flexslider/flexslider.css: {}
 js:
 vendor/flexslider/jquery.flexslider-min.js: {}
 dependencies:
 - core/jquery
```

Make sure to save our changes and clear Drupal's cache for our changes to take effect. Next, we need to consider how we will be using the FlexSlider library. If we were going to use it globally, then we could add it to our `octo.info.yml` file. However, because we will only be using it on our homepage, we can take advantage of the `{{attach library()}}` function.

## Attaching a library using Twig

In order to attach a library using Twig, we will need to follow these steps:

1. Open `page--front.html.twig`.
2. Add the following Twig function to the top of our template:

```
{% attach_library('octo/flexslider') %}
```

Make sure to save our changes and clear Drupal's cache for our changes to take effect. If we now inspect our homepage, we will see the FlexSlider library loading. However, if we go to any of the interior pages and inspect the markup, we will see that the FlexSlider library is absent. Being able to attach libraries only where needed is helpful, as it makes sure unnecessary CSS or scripts are not loaded.

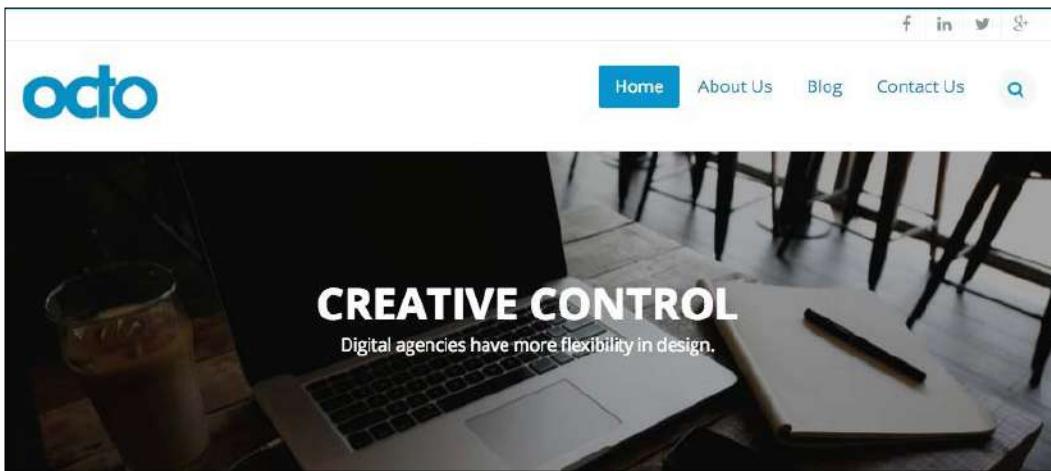
## Enable FlexSlider

In order for us to enable FlexSlider, we have one last step. We need to add the configuration for our slider to our `octo.js` so that it knows which markup to use for the slides:

Open `octo.js` and add the following script block:

```
//-- Flexslider
(function() {
 $('.flexslider').flexslider({
 direction: "vertical",
 controlNav: false,
 directionNav: false
 });
})();
```

Make sure to save our changes and clear Drupal's cache for our changes to take effect. If we browse our homepage, we will see our Headline slider is now fully functional, as shown in the following image:



We have almost completed our Headline region. There are two more pieces of UI to implement, one of these is what is known as the Parallax effect.

## Implementing Parallax

Parallax is a scrolling effect where the background image scrolls at a slower speed than the foreground of the page. The effect creates a subtle 3D effect. In order for us to implement this, we just need to add a small script block to our `octo.js` file, which targets our markup that has a class of `intro` and then uses data attributes on that section to manage the speed of the effect and how far to scroll.

Open `octo.js` and add the following script block:

```
//-- Parallax
(function() {
 $(window).scroll(function(e) {
 var bg = $('.intro');
 var yPos = -$(window).scrollTop() / bg.data('speed');
 var coords = '50% ' + yPos + 'px';
 bg.css({ backgroundPosition: coords });
 })
})();
```

Make sure to save our changes and clear Drupal's cache. If we browse our homepage, we will see the Parallax effect within the Headline region. As we begin to scroll down the page, we will see that our Headline background image scrolls at a slower speed.

## Adding a scroll effect

A subtle yet nice touch to our homepage is the addition of an animated scroll effect. At the bottom of our Headline region we have a link, which when clicked by a user smoothly scrolls the page to the next section. The markup is already in place and uses IDs for JavaScript to trigger the effect and to know where to scroll the page to. All we need to do is add the library and the script, and attach the library to our homepage.

Begin by navigating to the `Chapter06/start/themes/octo/vendor` folder and copy the `jquery-scrollTo` folder to our `themes/octo/vendor` folder. Once the files are accessible by Drupal, we can follow these remaining steps to add the library:

Open `octo.libraries.yml` and add the following entry:

```
scroll-to:
 version: 2.1.1
 js:
 vendor/jquery-scrollTo/jquery.scrollTo.min.js: {}
 dependencies:
 - core/jquery
```

Make sure to save our changes and clear Drupal's cache for our changes to take effect.

## Enabling the scroll script

In order for us to enable jQuery Scroller, we need to initialize it within our `octo.js` so that we know which element should be triggered when the user clicks on the link.

Open `octo.js` and add the following script block:

```
//-- Scroll to
(function() {
 $('#goto-section2').on('click', function(e) {
 e.preventDefault()
 $.scrollTo('#section2', 800, { offset: -220 });
 });
})();
```

Make sure to save our changes and clear Drupal's cache for our changes to take effect. Finally, all that is left is attaching the library to our homepage.

## Attaching ScrollTo library using Twig

In order to attach our library using Twig, we will need to follow these steps:

1. Open `page--front.html.twig`.
2. Add to the top of our template the following Twig function:

```
{% attach_library('octo/scroll-to') %}
```

Make sure to save our changes and clear Drupal's. If we browse our homepage and click on the link at the bottom of our Headline region, we will now see the animated smooth scrolling functioning perfectly.

We definitely covered a lot of different techniques when refactoring our Headline region, but we now need to move on to our Before Content region.

## Implementing our Before Content region

The fourth item we will need to address is replacing the static content in our Before Content region. Referring back to the mockup, we have two blocks of content that will require us to follow these steps to implement.

1. First, we will create the Our Services block and Our Features block and assign it to the Before Content region.
2. Finally, we will refactor the markup and print the Before Content region using Twig variables.

## Creating our Services block

Creating blocks of content is fairly simple and we should already be comfortable with the process.

We will start by navigating to the **Block layout** page at `/admin/structure/block` and following these steps:

1. Click on the **Place block** button in the **Before Content** region.
2. Click on the **Add custom block** button.
3. Enter a **Block description** of our services.
4. Select **HTML No Editor** from the **Text format** dropdown.
5. Add the markup located in the `Chapter06/start/content/OurServices.txt` file to the **Body** field.

6. Click on the **Save** button to proceed to the **Configure block** screen.
7. Uncheck the **Display title** checkbox.
8. Select the **Pages** vertical tab within the **Visibility** section.
9. Enter a value of <front> in the **Pages** text area.
10. Choose **Show for the listed pages** under **Negate the condition**.
11. Choose **Before Content** from the **Region** field.
12. Click on the **Save block** button.

We have one block completed, now let's add the second block.

## Creating our Features block

Creating blocks of content is fairly simple, and we should already be comfortable with the process.

We will start by navigating to the **Block layout** page at /admin/structure/block and following these steps:

1. Click on the **Place block** button in the **Before Content** region.
2. Click on the **Add custom block** button.
3. Enter a **Block description** of our features.
4. Select **HTML No Editor** from the **Text format** dropdown.
5. Add the markup located in the chapter06/start/content/  
OurFeatures.txt file to the **Body** field.
6. Click on the **Save** button to proceed to the **Configure block** screen.
7. Uncheck the **Display title** checkbox.
8. Select the **Pages** vertical tab within the **Visibility** section.
9. Enter a value of <front> in the **Pages** text area.
10. Choose **Show for the listed pages** under **Negate the condition**.
11. Choose **Before Content** from the **Region** field.
12. Click on the **Save block** button.

With our two blocks completed and assigned to the Before Content region, we need to make sure that they are ordered correctly. From the Block layout page, make sure that Our Services is followed by Our Features within the Before Content region. Now it's time to refactor our static markup and print the region to our `page--front.html.twig` template.

## Refactoring Before Content region

Currently, our `page--front.html.twig` template is not outputting our Before Content region. We will need to refactor our markup to add a new Twig variable that contains our two blocks `{{ page.before_content }}`.

Begin by opening `page--front.html.twig` and locating the section wrapper, and replace all the content between with our new Twig variable.

### Current markup

```
<section id="section2" class="section">
 <div class="container">
 ... content ...
 </div>
</section>
```

### New markup

```
<section id="section2" class="section">
 <div class="container">
 {{ page.before_content }}
 </div>
</section>
```

Make sure to save our changes, clear Drupal's cache and then refresh the browser. If all was successful, our Before Content region should now be identical to the Mockup.



Finally, we are at the end of our homepage, with the only piece left to refactor being our Footer region, which consists of several custom blocks.

## Implementing the footer

Our footer is a little different to what we have been implementing so far. The footer consists of multiple regions and custom blocks to easily match our mockup, and will mean us following these steps:

1. First, we will need to create five custom blocks for our Newsletter, About Us, Contact Us, Copyright, and Social Icons content. Once the blocks are created, they will need to be assigned to their respective regions.
2. Finally, we will need to refactor the markup in our footer to accommodate the various Twig variables to print out each region.

## Creating our custom blocks

We will begin with creating the five custom blocks as well as assigning them to the regions they need to be placed in. This will be somewhat repetitive, but is needed in order for us to complete our footer. We will start by navigating to the **Block layout** page at /admin/structure/block and following these steps.

### Newsletter block:

1. Click on the **Place block** button in the **Footer first** region.
2. Click on the **Add custom block** button.
3. Enter a **Block description** of Newsletter.
4. Select **HTML No Editor** from the **Text format** dropdown.
5. Add the markup located in the Chatper06/start/content/Newsletter.txt file to the **Body** field.
6. Click on the **Save** button to proceed to the **Configure block** screen.
7. Choose **Footer first** from the **Region** field.
8. Click on the **Save block** button.

### About Us block:

1. Click on the **Place block** button in the **Footer second** region.
2. Click on the **Add custom block** button.
3. Enter a **Block description** of About Us.
4. Select **HTML No Editor** from the **Text format** dropdown.
5. Add the markup located in the Chatper06/start/content/AboutUs.txt file to the **Body** field.
6. Click on the **Save** button to proceed to the **Configure block** screen.

7. Choose **Footer second** from the **Region** field.
8. Click on the **Save block** button.

**Footer Contact block:**

1. Click on the **Place block** button in the **Footer third** region.
2. Click on the **Add custom block** button.
3. Enter a **Block description** of Footer Contact.
4. Select **HTML No Editor** from the **Text format** dropdown.
5. Add the markup located in the chatper06/start/content/  
FooterContact.txt file to the **Body** field.
6. Click on the **Save** button to proceed to the **Configure block** screen.
7. Enter a title of Contact Us in the **Title** field.
8. Choose **Footer third** from the **Region** field.
9. Click on the **Save block** button.

**Copyright block:**

1. Click on the **Place block** button in the **Footer Bottom Left** region.
2. Click on the **Add custom block** button.
3. Enter a **Block description** of copyright.
4. Select **HTML No Editor** from the **Text format** dropdown.
5. Add the markup located in the chatper06/start/content/Copyright.txt  
file to the **Body** field.
6. Click on the **Save** button to proceed to the **Configure block** screen.
7. Uncheck the **Display title** checkbox.
8. Choose **Footer Bottom Left** from the **Region** field.
9. Click on the **Save block** button.

**Social Icons block:**

1. Click on the **Place block** button in the **Footer Bottom Right** region.
2. Click on the **Place block** button next to the Social Icons block. We don't  
need to recreate this block since blocks are now reusable.
3. Uncheck the **Display title** checkbox.
4. **Footer Bottom Right** should already be selected for us from the **Region** field.
5. Click on the **Save block** button.

We have successfully added all five blocks and assigned them to their respective regions. Now we just need to refactor the markup for each region, and we should be all set.

## Refactoring our main footer

We will be refactoring the markup for each section of our main footer by replacing the static markup with the Twig variable for each region. We should be comfortable with this process by now, so let's start.

Begin by opening `page--front.html.twig` and locating the `main-footer` section of our markup. Within the main footer, we will see individual sections of content for each of the blocks we just created.

### Footer first

Locate the following markup and replace the content between with the Twig variable that represents the page region.

#### Current markup

```
<div class="col-md-4">
 ...
</div>
```

#### New markup

```
<div class="col-md-4">
 {{ page.footer_first }}
</div>
```

### Footer second

Locate the following markup and replace the content between with the Twig variable that represents the page region.

#### Current markup

```
<div class="col-md-4">
 ...
</div>
```

#### New markup

```
<div class="col-md-4">
 {{ page.footer_second }}
</div>
```

## Footer third

Locate the following markup and replace the content between with the Twig variable that represents the page region.

### Current markup

```
<div class="col-md-4">
 ... content ...
</div>
```

### New markup

```
<div class="col-md-4">
 {{ page.footer_third }}
</div>
```

With our markup now refactored, if we look at the `main-footer` section of our homepage, the complete markup should look like the following:

```
<div class="container main-footer">
 <div class="row">
 <div class="col-md-4">
 {{ page.footer_first }}
 </div>

 <div class="col-md-4">
 {{ page.footer_second }}
 </div>

 <div class="col-md-4">
 {{ page.footer_third }}
 </div>

 </div>
</div>
```

Make sure to save our changes and then clear Drupal's cache. If we browse our homepage, we should see our static content has been replaced with the three custom blocks.

Now, let's refactor the two remaining blocks of content by locating the `footer-copyright` section of our markup. Within the footer copyright, we will see individual sections of content for the two blocks we just created.

## Footer bottom left

Locate the following markup and replace the content between with the Twig variable that represents the page region.

### Current markup

```
<div class="col-md-8">
 ... content ...
</div>
```

### New markup

```
<div class="col-md-8">
 {{ page.footer_bottom_left }}
</div>
```

## Footer bottom right

Locate the following markup and replace the content between with the Twig variable that represents the page region.

### Current markup

```
<div class="col-md-4">
 ... content ...
</div>
```

### New markup

```
<div class="col-md-4">
 {{ page.footer_bottom_right }}
</div>
```

With our markup now refactored, if we look at the footer-copyright section of our homepage, the complete markup should look like the following:

```
<div class="footer-copyright">
 <div class="container">
 <div class="row">
 <div class="col-md-8">
 {{ page.footer_bottom_left }}
 </div>
 <div class="col-md-4">
 {{ page.footer_bottom_right }}
 </div>
 </div>
 </div>
```

## *Theming Our Homepage*

---

Make sure to save our changes and then clear Drupal's cache. If we browse our homepage, we should see our static content has been replaced with the two custom blocks. Our footer is complete, and should look like the following image:



## **Summary**

When we first started our homepage, it seemed to be a daunting task filled with deciding how we should convert our mockup into a dynamic rendition of blocks, regions, Twig templates, and variables. We learned through repetition that such a daunting task is actually not that complicated after all. In this chapter, we accomplished the following:

- We worked with site branding by adding our logo, creating Twig templates for blocks, and refactoring markup.
- We learned best practices to add assets with libraries, custom scripts, and scripts to our website and individual pages.
- We discovered how to develop the aggregated content using Views, create Block layouts from Views, and format the output of Views content for use with JavaScript libraries.

In the next chapter, we will continue our theming by taking a look at creating an interior page template, carrying over any global regions such as our Header and Footer, and will continue working with various Twig templates.

# 7

## Theming Our Interior Page

One of the great features of Drupal 8 is the new Twig templating engine. Simply by using the recommended file name suggestions, we saw how easy it was to theme our homepage by creating a `page--front.html.twig` template. However, we are not able to use this same template for our interior pages as Drupal only renders our homepage template on the front page of our website.

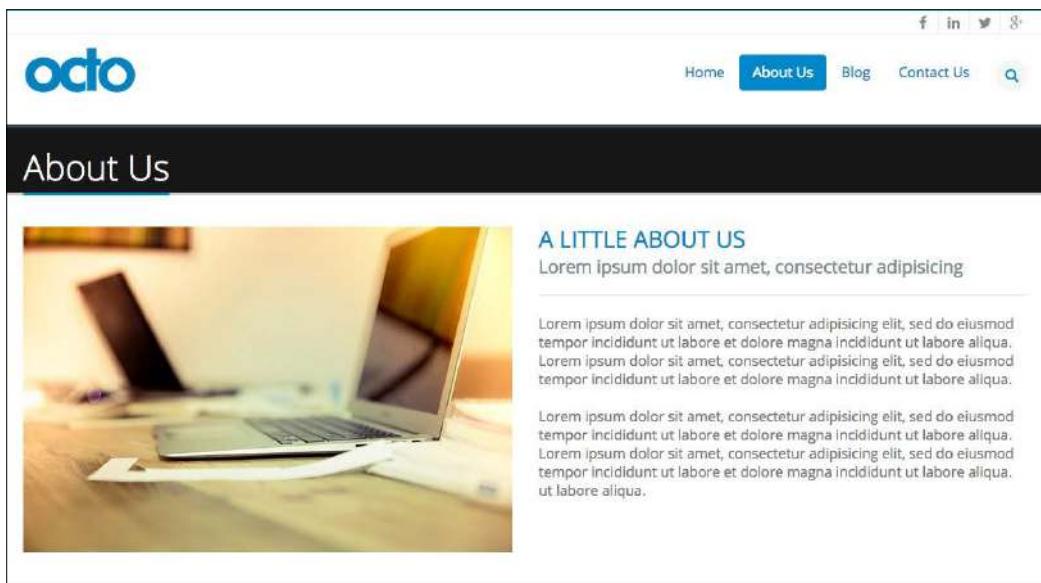
Instead, we will need to create a new Twig template that all of our interior pages can use when a user is navigating our website. By default, Drupal outputs content using the `page.html.twig` template. In this chapter, we will look at using the `page.html.twig` template, along with discussing strategies to address the following:

- We will begin with reviewing the About Us page mockup and identify any additional components that may require custom blocks, new regions, and Twig templates.
- Then, we will take a look at reusing regions such as our header and footer as they are considered global components that will be needed across all of our interior pages.
- Drupal 8 has moved the page title into a new Page title block. We will take a closer look at how we can use this block to recreate our Title Bar region.
- Using regions to control page flow is important in order to manage the content, and in Drupal we will look at printing out different regions while outputting any block content that has been assigned to them.
- Sometimes, we have the need to manipulate our main content's markup to add additional styling. You will learn how to not only print the content region but also take advantage of Node templates.
- Finally, Views play such an integrated part of any theme, so we will take another look at Drupal Views to display the content. You will learn how to use Twig variables to override fields as well as display the content by creating block displays.

Although we will work through each section, we can refer back to the Chapter07 exercise files folder. Each folder contains a start folder and an end folder with files that we can use to compare our work when needed. This also includes database snapshots that will allow us all to start from the same point when working through various lessons.

## Reviewing the About Us mockup

In order for us to identify page elements, we will be recreating them for the About Us page and need to take a closer look at our mockup. The About Us page can be found in the Mockup folder located in our exercise files. Begin by opening up the `about-us.html` file within the browser, as shown in the following image:

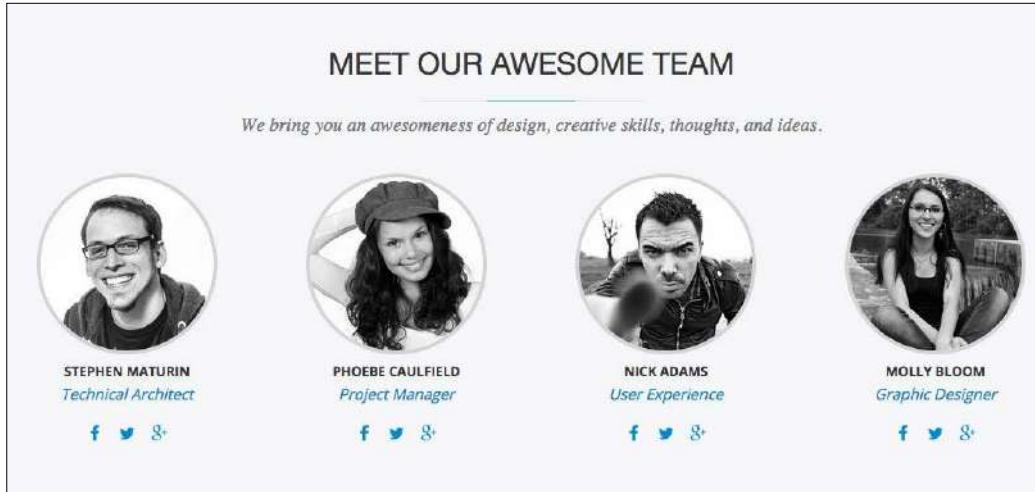


There are several page elements that we will need to recreate, and we can identify the following:

1. First is the header, which we created previously on our homepage. We will need to add this region to our interior pages as well to ensure that our users can navigate from page to page and use the global search functionality.
2. Second is the page title, which spans across the top of all our interior pages. This is a common element in Drupal that helps the user to identify which page they are currently on.

3. Third is the main content region. Any nodes or custom blocks can output content in this region. We will need to make sure that we account for the content assigned to this region and manage how it flows within our page.

As we continue to scroll down the page, we come across additional content, as shown in the following image:



4. The Team Member listing represents the content that is repeated, and anytime we see repeating content, we should consider building this using Drupal 8 views. This view will be specific to our About Us page, so we will look at creating a view block for this section and control the visibility accordingly.
5. Finally, we will need to include our page footer. Once again, we already created this region on our homepage, so we will need to make sure that we include it within our interior page template as well.

We identified five specific page sections that need to be developed and themed. Most of the sections are global to all pages, with the exception of our team member listing. Now that we have a plan for what we will be developing, let's get started by creating our interior page template.

## Creating our interior page template

Drupal provides us with multiple ways to address templating a specific page. As we saw when we created our homepage, we can take advantage of Twig debugging to identify which templates are being output. The same is true for our interior pages. If we navigate to the About Us page and inspect the markup we can identify that Drupal is using the default `page.html.twig` template, as shown in the following image:

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'page' -->
<!-- FILE NAME SUGGESTIONS:
 * page--node--2.html.twig
 * page--node--%.html.twig
 * page--node.html.twig
 x page.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/system/templates/page.html.twig' -->
▶ <div class="layout-container">...</div>
<!-- END OUTPUT from 'core/modules/system/templates/page.html.twig' -->
```

This template is clearly different than the `page--front.html.twig` template we created earlier, which explains why some of our page elements are missing. However, this is a good example of how we can use different Twig templates to control the markup.

Begin by following these steps:

1. Navigate to the `core/modules/system/templates` folder and copy `page.html.twig`.
2. Place the copy within our `themes/octo/templates` folder.

Make sure to clear Drupal's cache and refresh the About Us page. If we inspect the page again, we will note that our new template is being used. We can now begin to modify the markup safely.

## Adding our Global Header

Our website has several global components that were present on our home page that don't currently exist on our interior pages. One such item is the Global Header, which consists of the Utility menu, Logo, Main menu, and Search form block.

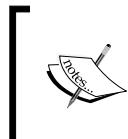
To add this section to our template, all we need to do is simply copy the header markup from the `page--front.html.twig` template.

1. Open `page.html.twig` and delete the entire contents.
2. Open `page--front.html.twig` and copy the following markup:

```
<header class="header" role="banner">
 <div class="header-top">
 <div class="container">
 {{ page.header_top }}
 </div>
 </div>

 <div class="header-nav container">
 {{ page.header}}
 {{ page.primary_menu }}
 </div>
</header>
```

3. Paste the markup into `page.html.twig`.



One more very important item to keep in mind is that we will need to also add the `page.content` region to our template. Failure to add the `{{ page.content }}` region will result in not being able to log in to the Drupal Admin or see any content that Drupal assigns to this region.

4. Add the following Twig variable to the bottom of our template:

```
 {{ page.content }}
```

5. Save `page.html.twig`.

Make sure to clear Drupal's cache and refresh the About Us page within our browser. Our header is now in place and functioning as we would expect.



## Implementing our page title

Every content type we develop in Drupal includes a title field, which is used to identify the node currently being displayed. In Drupal 8, the title field warrants its own block called Page title. This new block provides us with the flexibility to place the page title into any region and have it display wherever it is needed in our layout.

In order for us to implement the page title displayed within our mockup, we will need to complete a series of steps:

1. First, we will copy the static markup from our mockup into our interior page template and preview the results.
2. Second, we will configure our theme to add a new Title Bar region that we can use to assign content to.
3. Next, we will need to assign the Page title block to our new region and then output it within our `page.html.twig` template.
4. Finally, we will refactor the static Page title markup using Twig to create a new block template.

## Working with static HTML

When implementing any section of content within our theme, this will often begin with copying static HTML. Having actual working HTML within our template file ensures that our content displays the way we are expecting. The other advantage of copying static HTML into our template is that it allows us to easily replace the markup with dynamic content. Let's begin by following these steps:

1. Open `page.html.twig`.
2. Navigate to `Mockup/about-us.html` and copy the following markup. Don't forget to include the opening and closing main layout section, as shown here:

```
<main role="main" class="main">

 <section class="page-top">
 <div class="container">
 <h1>About Us</h1>
 </div>
 </section>

</main>
```

3. Paste the markup into the `page.html.twig` template.
4. Save `page.html.twig`.

Make sure to clear Drupal's cache and refresh the About Us page within our browser. We should now see the static page title being displayed.



We may also note the Page title block being displayed below our static title. By default, Drupal assigns the Page title block to the Content region. In order for us to replace our static title with Drupal's Page title block, we will need to add a new region.

## Adding new regions

When we first created our theme, we added regions for Drupal to use when assigning blocks of content. While we may have thought we accounted for all the regions our theme would need, we neglected to add one for our page title.

We can add new regions to our theme at any time by modifying our configuration file. In order to add a new Title Bar region, we will need to navigate to our themes/octo folder and follow these steps:

1. Open the `octo.info.yml` file.
2. Add the following to our regions section:

```
title_bar: 'Title Bar'
```
3. Save `octo.info.yml`.

Make sure to clear Drupal's cache and then navigate to the Block layout page located at `/admin/structure/block`. We should now see that the new region has been added.



## Reassigning the Page title block

With our new region added, we can focus on reassigning the Page title block by following these steps:

1. Locate the **Page title** block in the **Content** region.
2. Select **Title Bar** from the **Region** dropdown.
3. Click on the **Save blocks** button.

If for any reason the Page title block is missing from the Block layout screen, we can add it by using the **Place block** button next to the region we want to place it in. Now we need to print our new region so that we can view the Page title block within our template.

## Printing the Title Bar region

In order for our `page.html.twig` template to display the Page title block, we need to print the Title Bar region. The Twig variable that represents the region's name can always be found by looking in our themes `octo.info.yml` file.

1. Open `page.html.twig`.
2. Add the Twig variable `{{ page.title_bar }}` directly below the `main` element so that our markup looks like the following:

```
<main role="main" class="main">
{{ page.title_bar }}
```

3. Save `page.html.twig`.

Make sure to clear Drupal's cache and then refresh the browser. Our About Us page is now displaying two page titles – one dynamic and one static, as shown in the following image:



We can fix the duplicated page titles by moving our static markup into the Page title block template that Drupal is outputting. This will give us the freedom to then remove the static markup completely from our `page.html.twig` template while maintaining the styling that is currently being displayed.

## Creating a block template

If we inspect the page title on the About Us page, we will see that Drupal is using the default `block.html.twig` template. We can take advantage of FILE NAME SUGGESTIONS to create our own block template specific to the Page title and then refactor the markup into it by following these steps:

1. Navigate to `core/modules/block/templates` and copy the `block.html.twig` template to our `themes/octo/templates` folder.
2. Rename `block.html.twig` to `block--page-title-block.html.twig`.
3. Replace the content with the following markup:

```
<section class="page-top">
 <div class="container">
 {{ content }}
 </div>
</section>
```
4. Save `block--page-title-block.html.twig`.
5. Open `page.html.twig`.
6. Delete the `page-top` section.
7. Save both templates.

Make sure to clear Drupal's cache and then refresh the browser. As we navigate from page to page, we should see a single page title being displayed and updating to display the current page's title. Time to move on to our main content section.

## Implementing our main page structure

Our main page structure can be considered anything below the global header and page title and anything above the global footer. In our case, the main page structure for our About Us page consists of three regions – Before Content, Content, and After Content. Currently, we are already printing the main content region, but we have yet to add our structural layout or the other two regions.

---

## *Theming Our Interior Page*

---

Begin by opening `page.html.twig`, located in our `themes/octo` folder, and replace the following markup section with the new markup.

### **Current markup**

```
<main role="main" class="main">
 {{ page.title_bar }}
</main>

{{ page.content }}
```

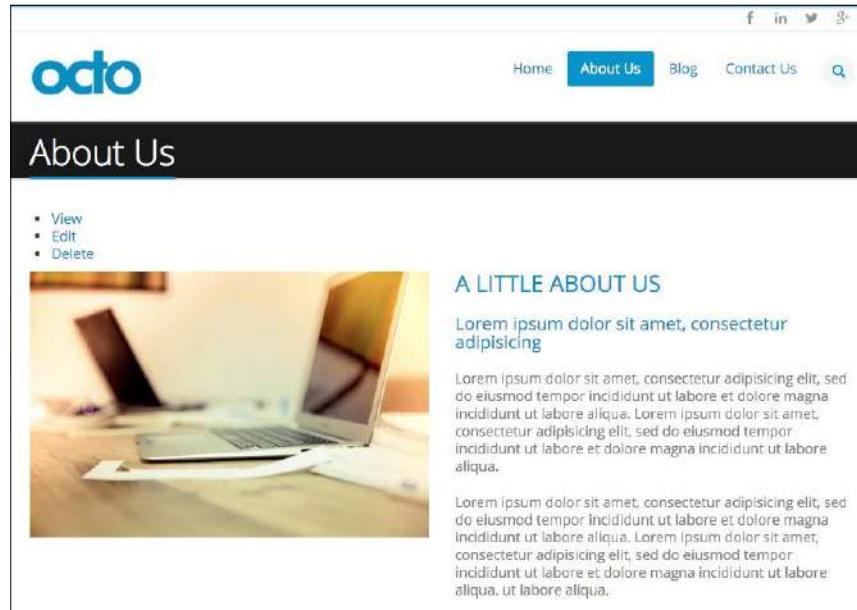
### **New markup**

```
<main role="main" class="main">
 {{ page.title_bar }}
 {{ page.before_content }}
 <div id="content" class="content full">
 <div class="container">
 <div class="row">
 <div class="col-md-12">
 {{ page.content }}
 </div>
 </div>
 </div>
 {{ page.after_content }}
 </main>
```

Make sure to save our changes and then clear Drupal's cache to ensure the theme registry has picked up our new layout.

While reviewing the markup earlier, we are adding the Before Content region followed by the structural markup for our main content and then our After Content region. These two new regions will allow our layout to be flexible enough to add block content above and below our main content area. We will be using both these regions as we continue to implement our theme.

In the meantime, let's preview our About Us page in the browser and compare it to our mockup. We want to make sure that we haven't lost any styling during our refactoring of markup.



In comparing our About Us page side by side with the mockup, we will note that our `h3` and `h4` elements are missing some CSS. Upon closer inspection, our design is expecting our Landing page content type to include a CSS class of `landing`. Including this CSS class would ensure that our `h3` has the adequate bottom margin and our `h4` would be colored gray and include the bottom border separating the Headings from the content.



So how would we go about adding a CSS class to our About Us page? Actually, we can create a Twig template for any specific node or content type using the same methods we have for any other page.

## Creating a Node template

If we inspect the About Us page, we can determine exactly what Twig template we should use. By default, Drupal will use the `node.html.twig` template. However, we can create our own copy of the template based on the multiple file name suggestions. This will result in all of our Landing Page content types using this new template:

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'node' -->
<!-- FILE NAME SUGGESTIONS:
 * node--2--full.html.twig
 * node--2.html.twig
 * node--landing-page--full.html.twig
 * node--landing-page.html.twig
 * node--full.html.twig
 x node.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/node/templates/node.html.twig' -->
▼<article data-history-node-id="2" data-quickeedit-entity-id="node/2" role="article" class="contextual-region"
```

Begin by navigating to the `core/modules/node/templates` folder and following these steps:

1. Copy the `node.html.twig` template to our `themes/octo/templates` folder.
2. Rename `node.html.twig` to `node--landing-page.html.twig`.
3. Replace the content with the following markup:

```
{% set classes = ['landing'] %}

<article{{ attributes.addClass(classes) }}>
 <div{{ content_attributes }}>
 {{ content }}
 </div>
</article>
```

4. Save `node--landing-page.html.twig`.

In the earlier-mentioned markup, we are adding the minimal structure that our landing page content type needs to output any fields that have been enabled. We also take advantage of Twig to create a variable named `classes` that allows us to add CSS class names to any existing classes that Drupal may be adding using the `attributes.addClass()` function. This is a simple technique, but one that will be used often to add CSS classes to our markup.

Make sure to clear Drupal's cache and then refresh the browser. If we take a look at the About Us page again, we will see that our `H3` and `H4` headings are styled to match our mockup.

## Implementing our Team members section

The next area of our About Us page we will need to create is the display of our Team members listing. We will be taking an existing Drupal content type named Team and using Views to display the four team members.

In order for us to implement the Team members section, we will need to complete a series of steps as follows:

1. First, we will review the structural markup for our team members section in preparation for creating a new Drupal View.
2. Next, we will look at various methods to theme View content as we complete our Team Members display.
3. Finally, we will assign our custom View block to a new region and limit the page visibility to our About Us page.

## Prepping our Team Member View

The Views module provides multiple ways of creating block displays, whether we are simply outputting the fields of a content type as an unordered list or relying on rendering a specific content type directly. In any case, it is best to start off by breaking down how the Team Members block is structured within our mockup and then creating our view based on those needs.

Let's start by reviewing the visual display and then break down the structural markup. Navigate to the exercise files and open up the `about-us.html` file found in the Mockup folder as shown here:

### MEET OUR AWESOME TEAM

---

*We bring you an awesomeness of design, creative skills, thoughts, and ideas.*



STEPHEN MATURIN  
*Technical Architect*

[f](#) [t](#) [g+](#)



PHOEBE CAULFIELD  
*Project Manager*

[f](#) [t](#) [g+](#)



NICK ADAMS  
*User Experience*

[f](#) [t](#) [g+](#)



MOLLY BLOOM  
*Graphic Designer*

[f](#) [t](#) [g+](#)

Visually, we can determine that our view will require the following content:

- Header
- Subheader
- Four-column layout, with each column representing a team member

However, if we inspect the markup of an individual Team member, we will get a better idea of what fields a team member will need to display.

### **Markup**

```
<div class="col-md-3 col-sm-6 views-row">

 <div class="img-round img-grayscale-hover">

 </div>

 <h6>Stephen Maturin</h6>

 <p>Technical Architect</p>

 <a class="button--icon" href="https://www.facebook.com/
PacktPub"><i class="fa fa-facebook"></i>

 <a class="button--icon" href="https://twitter.com/
packtpub"><i class="fa fa-twitter"></i>

 <a class="button--icon" href="https://plus.google.com/
+packtpublishing/posts"><i class="fa fa-google-plus"></i>

</div>
```

Based on the markup mentioned earlier, each team member content type will need to display the following fields:

- Team Photo
- Title
- Position
- Social icons

Now that we have identified both our visual and structural markup and fields, we can begin creating our new Team Member view.

## Creating our Team Member View

To get started, we will need to navigate to /admin/structure/views and click on the **Add new view** button. From the Views Admin screen, we will add the following information:

- **VIEW BASIC INFORMATION:**
  1. **View name:** Team Members
  2. Check the **Description** box
  3. **Description:** A listing of Team Members
- **VIEW SETTINGS:** Show: Content of type: Team sorted by: Newest first

The screenshot shows the 'Add new view' page. At the top, there is a title bar with the text 'Add new view ★'. Below it, a breadcrumb navigation shows 'Home » Administration » Structure » Views'. The main content area is divided into two sections: 'VIEW BASIC INFORMATION' and 'VIEW SETTINGS'. In the 'VIEW BASIC INFORMATION' section, there is a 'View name \*' field containing 'Team Members' and a 'Description' field containing 'A listing of Team Members'. A checked checkbox next to 'Description' indicates it is checked. In the 'VIEW SETTINGS' section, there is a dropdown menu for 'Show:' set to 'Content', a dropdown for 'of type:' set to 'Team', and a dropdown for 'sorted by:' set to 'Newest first'.

- **BLOCK SETTINGS:**
  1. Check the **Create a block** checkbox.
  2. **Block title:** Team Members Listing

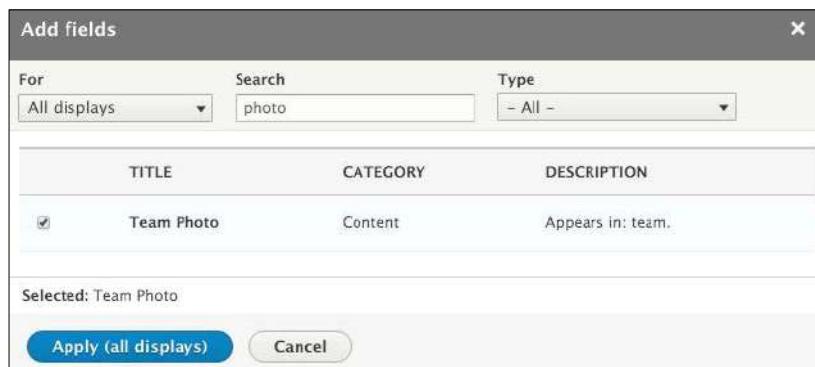
- **BLOCK DISPLAY SETTINGS:**

1. **Display format: Unformatted list of: fields.**
2. **Items per block: 4.**
3. Click on the **Save and edit** button.



Now that our Team Member view has been created, we will need to add additional fields and adjust the format settings before we can use our new block. With the view still open we will need to add the following fields:

1. Click on the **Add** button next to the **FIELDS** section.
2. Enter the term **photo** in the **Search** field to filter our choices.
3. Click on the **Team Photo** checkbox.
4. Click on the **Apply (all displays)** button.



Next, we will need to configure the Team Photo field.

1. Click on the **STYLE SETTINGS** section link.
2. Uncheck the **Add default classes** checkbox.
3. Click on the **Apply (all displays)** button.

**Configure field: Content: Team Photo**

**For**  
All displays ▾

Appears in: team;

Create a label  
 Exclude from display  
Enable to load this field as hidden. Often used to group fields, or to use as token in another field.

**Column used for click sorting**  
target\_id ▾  
Used by Style: Table to determine the actual column to click sort the field on. The default is usually fine.

**Formatter**  
Image ▾

**Image style**  
None (original image) ▾

[Configure Image Styles](#)

**Link image to**  
Nothing ▾

**▼ STYLE SETTINGS**

Customize field HTML  
 Customize label HTML  
 Customize field and label wrapper HTML  
 Add default classes  
Use default Views classes to identify the field, field label and field content.  
 Use field template  
If checked, field api classes will be added by field templates. This is not recommended unless your CSS depends upon these classes. If not checked, template will not be used.

With our Team Photo field added, we will need to repeat the steps mentioned earlier to add the remaining fields.

### **Job Position**

1. Click on the **Add** button next to the **FIELDS** section.
2. Enter the term **Job Position** in the **Search** field to filter our choices.
3. Click on the **Job Position** checkbox.
4. Click on the **Apply (all displays)** button.
5. Click on the **STYLE SETTINGS** section link.
6. Uncheck the **Add default classes** checkbox.
7. Click the **Apply (all displays)** button.

### **Facebook link**

1. Click on the **Add** button next to the **FIELDS** section.
2. Enter the term **Facebook link** in the **Search** field to filter our choices.
3. Click on the **Facebook link** checkbox.
4. Click on the **Apply (all displays)** button.
5. Click on the **STYLE SETTINGS** section link.
6. Uncheck the **Add default classes** checkbox.
7. Click on the **Apply (all displays)** button.

### **Twitter link**

1. Click on the **Add** button next to the **FIELDS** section.
2. Enter the term **Twitter link** in the **Search** field to filter our choices.
3. Click on the **Twitter link** checkbox.
4. Click on the **Apply (all displays)** button.
5. Click on the **STYLE SETTINGS** section link.
6. Uncheck the **Add default classes** checkbox.
7. Click on the **Apply (all displays)** button.

## Google Plus link

1. Click on the **Add** button next to the **FIELDS** section.
2. Enter the term **Google Plus link** in the **Search** field to filter our choices.
3. Click on the **Google Plus link** checkbox.
4. Click on the **Apply (all displays)** button.
5. Click on the **STYLE SETTINGS** section link.
6. Uncheck the **Add default classes** checkbox.
7. Click on the **Apply (all displays)** button.

With all of our fields now added to our view, we can click on the **Save** button to make sure that we don't lose any of our work. Our fields should now look like the following image:

The screenshot shows the 'Fields' and 'Block Settings' sections of a Drupal View configuration. The 'Fields' section lists several content fields: Title, Team Photo, Job Position, Facebook link, Twitter link, and Google Plus link. The 'Block Settings' section includes options for Block name, Category, Allow settings, Access, Header, Footer, No results behavior, and Pager.

| TITLE                                                    |  | BLOCK SETTINGS                              |  |
|----------------------------------------------------------|--|---------------------------------------------|--|
| Title: Team Members Listing                              |  | Block name: None                            |  |
| FORMAT                                                   |  | Block category: Lists (Views)               |  |
| Format: Unformatted list   Settings                      |  | Allow settings: Items per page              |  |
| Show: Fields   Settings                                  |  | Access: Permission   View published content |  |
| FIELDS                                                   |  | Add                                         |  |
| Content: Title                                           |  | Add                                         |  |
| Content: Team Photo                                      |  | Add                                         |  |
| Content: Job Position                                    |  | Add                                         |  |
| Content: Facebook link                                   |  | Add                                         |  |
| Content: Twitter link                                    |  | Add                                         |  |
| Content: Google Plus link                                |  | Add                                         |  |
| FILTER CRITERIA                                          |  | Add                                         |  |
| Content: Publishing status (= Yes)                       |  | Add                                         |  |
| Content: Type (= Team)                                   |  | Add                                         |  |
| SORT CRITERIA                                            |  | Add                                         |  |
| Content: Authored on (desc)                              |  | Add                                         |  |
| HEADER                                                   |  |                                             |  |
| FOOTER                                                   |  |                                             |  |
| NO RESULTS BEHAVIOR                                      |  |                                             |  |
| PAGER                                                    |  |                                             |  |
| Use pager: Display a specified number of items   5 items |  |                                             |  |
| More link: No                                            |  |                                             |  |
| Link display: None                                       |  |                                             |  |

We will be modifying our view quite frequently as we begin theming it, but in the meantime, let's assign our new block to the After Content region we created earlier. This will give us a better idea of what it currently looks like before proceeding.

## Managing our Team Members listing block

Any time we create a new block display using views, we can easily assign it to any region from the Block layout page. Begin by navigating to /admin/structure/block and follow these nine steps:

1. Locate the **After Content** region.
2. Click on the **Place block** button.
3. Locate the **Team Members** block.
4. Click on the **Place block** button.
5. Uncheck the **Display title** checkbox.
6. Select the **Pages** tab under **Visibility**.
7. Enter the path /about into the **Page** text field.
8. Make sure that the **Show for the listed pages** checkbox is selected.
9. Click on the **Save block** button.

With our Team Members block assigned to the After Content region, we can now navigate back to the About Us page and preview the display:



We clearly have to do something to match our Team Members block to our mockup. If we compare the mockup to our About Us page, we can see that it doesn't yet match. We can address this, starting with adding some CSS directly to our View.

## Formatting Views with CSS

The first noticeable thing is that our four team members are stacked on the page vertically versus displaying nicely in four columns across the page. In order to resolve this, we need to simply add the Twitter Bootstrap column classes to our view's rows. We can achieve this by navigating to /admin/structure/views/view/team\_members and following these four steps:

1. Click on the **Settings** link next to **Unformatted list** within the **FORMAT** section.
2. From the **Block: Style options** dialog, select the **Row class** text field and enter a value of col-md-3 col-sm-6.
3. Click on the **Apply** button.
4. Click on the **Save** button.

Navigate back to the About Us page, and our Team Members block should be displaying in their proper columns, as shown in the following image:



If we inspect the Team Members block, we can see that our view rows are now outputting the CSS class we added:

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'views_view_unformatted' -->
<!-- BEGIN OUTPUT from 'core/modules/views/templates/views-view-unformatted.html.twig' -->
► <div class="col-md-3 col-sm-6 views-row"></div>
<!-- END OUTPUT from 'core/modules/views/templates/views-view-unformatted.html.twig' -->
```

So far so good, but we still need to clean up how the Team Members block and fields are being output. Fortunately, we can add CSS classes to our overall view as well as rewrite the fields output to better match our mockup.

## Adding CSS classes to Views

So far we have been working with the basic settings of a View. We can actually achieve more complex settings, such as contextual filters, relationships, and other miscellaneous settings within the Views UI. In the case of our Team Members view, we need to globally add a single CSS class name. While you may think we would need to create a Twig template to achieve this, we can actually add a CSS class from the admin by navigating to /admin/structure/views/view/team\_members and following these steps:

1. Click on the **ADVANCED** field set to expand the options within it.
2. Locate the **OTHER** section.
3. Click on the **None** link next to **CSS class**:
4. Enter a value of team into the **CSS class name(s)** text field.
5. Click on the **Apply (all displays)** button.
6. Click on the **Save** button.

If we navigate back to the About Us page and inspect the Team Members block, we will see that our view now has the new class added to it.

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'views_view' -->
<!-- BEGIN OUTPUT from 'core/modules/views/templates/views-view.html.twig' -->
►<div class="team contextual-region js-view-dom-id-8206c0105827ad9ce1d4dd058e8e8c7550896824e2b22cac452a4d
<!-- END OUTPUT from 'core/modules/views/templates/views-view.html.twig' -->
```

## Using Twig variables to rewrite field content

Another feature widely used when theming Views is the ability to rewrite field content. Every field we add to a view can be rewritten to easily modify the markup, and with Twig now part of Drupal 8 we can use this syntax to achieve the markup that our view needs.

One important item to remember when using this technique is that by default Drupal will provide field wrapper elements to each field. This unfortunately adds additional and unneeded div elements around every field. We will want to turn this setting off by navigating to /admin/structure/views/view/team\_members and completing the following:

1. Locate the **FORMAT** section.
2. Click on the **Settings** link next to **Show: Fields**.
3. Uncheck the **Provide default field wrapper elements** checkbox.
4. Click on the **Apply** button.

Now that we have turned off the default field wrapper elements, we can proceed to rewriting each field starting with Team Photo.

### Content: Team Photo

1. Click on the **Content: Team Photo** link under the **FIELDS** section.
2. Choose the **Team (250x250)** image style from the **Image style** dropdown.
3. Expand the **REWRITE RESULTS** section.
4. Check the **Override the output of this field with custom text** checkbox.
5. Expand the **REPLACEMENT PATTERNS** section to view the replacement patterns we can use to rewrite our markup.
6. Enter the following markup in the **Text** field:

```
<div class="img-round img-grayscale-hover">

 {{ field_team_photo }}

</div>
```

7. Click on the **Apply (all displays)** button to complete the field rewrite.
8. Click on the **Save** button to save our view.

To validate that our field has been rewritten, navigate back to the About Us page and our Team Members' photographs should now be displaying, as shown in the following image:



If we inspect the markup of the page, we can see how simple rewriting the field output of views can be. Let's finish rewriting our fields by navigating to /admin/structure/views/view/team\_members and following these remaining steps.

#### **Content: Title field**

1. Click on the **Content: Title** link under the **FIELDS** section.
2. Uncheck the **Link to the Content** checkbox.
3. Expand the **REWRITE RESULTS** section.
4. Check the **Override the output of this field with custom text** checkbox.
5. Expand the **REPLACEMENT PATTERNS** section to view the replacement patterns we can use to rewrite our markup.
6. Enter the following markup in the **Text** field:  
`<h6><span>{{ title }}</span></h6>`
7. Click on the **Apply (all displays)** button.
8. Click on the **Save** button to save our view.

#### **Content: Job Position field**

1. Click on the **Content: Job Position** link under the **FIELDS** section.
2. Expand the **REWRITE RESULTS** section.
3. Check the **Override the output of this field with custom text** checkbox.
4. Expand the **REPLACEMENT PATTERNS** section to view the replacement patterns we can use to rewrite our markup.
5. Enter the following markup in the **Text** field:  
`<p>{{ field_job_position }}</p>`
6. Click on the **Apply (all displays)** button.
7. Click on the **Save** button to save our view.

#### **Content: Facebook link field**

1. Click on the **Content: Facebook link** link under the **FIELDS** section.
2. Expand the **REWRITE RESULTS** section.
3. Check the **Override the output of this field with custom text** checkbox.
4. Expand the **REPLACEMENT PATTERNS** section to view the replacement patterns we can use to rewrite our markup.

5. Enter the following markup in the **Text** field:

```
<a class="button--icon"
 href="{{ field_facebook_link_uri }}>
 <i class="fa fa-facebook"></i>
```

6. Click on the **Apply (all displays)** button to complete the field rewrite.
7. Click on the **Save** button to save our view.

### **Content: Twitter link field**

1. Click on the **Content: Twitter link** link under the **FIELDS** section.
2. Expand the **REWRITE RESULTS** section.
3. Check the **Override the output of this field with custom text** checkbox.
4. Expand the **REPLACEMENT PATTERNS** section to view the replacement patterns we can use to rewrite our markup.
5. Enter the following markup in the **Text** field:

```
<a class="button--icon"
 href="{{ field_twitter_link_uri }}>
 <i class="fa fa-twitter"></i>
```

6. Click on the **Apply (all displays)** button.
7. Click on the **Save** button to save our view.

### **Content: Google Plus link field**

1. Click on the **Content: Google Plus link** link under the **FIELDS** section.
  2. Expand the **REWRITE RESULTS** section.
  3. Check the **Override the output of this field with custom text** checkbox.
  4. Expand the **REPLACEMENT PATTERNS** section to view the replacement patterns we can use to rewrite our markup.
  5. Enter the following markup in the **Text** field:
- ```
<a class="button--icon"  
    href="{{ field_google_plus_link_uri }}>  
    <i class="fa fa-google-plus"></i></a>
```
6. Click on the **Apply (all displays)** button.
 7. Click on the **Save** button to save our view.

With the remaining fields rewritten using the proper Twig replacement patterns, we can navigate back to our About Us page and review the results, as shown in the following image:



Our Team Members block is coming together nicely. We do have one field that is out of order and that is our team members' names. Currently, they are appearing above our images, and we need to fix that by rearranging our View's fields.

Rearranging View fields

When we work with Views, we will often need to modify the fields that we have added, including rearranging them so that they display properly to match our design. We can easily accomplish this using the Views UI by navigating to `/admin/structure/views/view/team_members` and following these steps:

1. Click on the dropdown button next to the **FIELDS** section.
2. Choose **Rearrange** from the list.
3. Drag the **Content: Title** field below the **Content: Team Photo** field using the drag icons.
4. Click on the **Apply (all displays)** button.
5. Click on the **Save** button to save our view.

Now that we have our fields rearranged properly, we still need to add the header and subheader so that they appear above our block. Because we need the headings to be part of our view block, we can take a look at using the View header to add the markup and content required to match our mockup.

Adding a View header

Our best option to add this markup is to use the View header, which can be done by navigating to /admin/structure/views/view/team_members and following these steps:

1. Click on the **Add** button within the **HEADER** section of our views **BLOCK SETTINGS**.
2. Choose **Text area** from the **Add header** window, this will allow us to provide markup text for the area.
3. Click on the **Apply (all displays)** button.
4. Choose **Full HTML** from the **Text format** dropdown.
5. Enter the following markup in the **Content** field:


```
<div class="view-header">
  <h2 class="block-title">Meet our awesome team</h2>
  <hr>
  <p class="block-subtitle">We bring you an awesomeness of
    design, creative skills, thoughts, and ideas.</p>
</div>
```
6. Click on the **Apply** button.
7. Click on the **Save** button to save our view.

If we view our About Us page, we see that our header and tagline are now being displayed. We are almost finished with this section.

MEET OUR AWESOME TEAM

We bring you an awesomeness of design, creative skills, thoughts, and ideas.



NICK ADAMS
User Experience

f t g



MOLLY BLOOM
Graphic Designer

f t g



PHOEBE CAULFIELD
Project Manager

f t g



STEPHEN MATURIN
Technical Architect

f t g

The only visual element still missing is the gray background that helps separate our Team Members block from the rest of our content. Also, if we were to resize the page and check for any responsive qualities, we would note that something is just not quite right. In fact, upon further investigation, our After Content region does not include the container class that enables Twitter Bootstrap to apply its media queries properly.

Refactoring the After Content region

Adding markup to any regions requires us to either create or modify an existing Twig template. As we have seen before, all we need to do is inspect our page markup to determine where the template resides and what file name suggestion we should use:

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'region' -->
<!-- FILE NAME SUGGESTIONS:
  * region--after-content.html.twig
  x region.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/system/templates/region.html.twig' -->
▶ <div>...</div>
<!-- END OUTPUT from 'core/modules/system/templates/region.html.twig' -->
```

Begin by following these steps:

1. Navigate to `core/modules/system/templates` and copy the `region.html.twig` template to our `themes/octo/templates` folder.
2. Rename `region.html.twig` to `region--after-content.html.twig`.
3. Replace the content with the following markup:

```
{% set classes = ['region', 'region-' ~ region|clean_class,] %}

{% if content %}
  <div{{ attributes.addClass(classes) }}>
    <div class="container">
      {{ content }}
    </div>
  </div>
{% endif %}
```

4. Save `region--after-content.html.twig`.

Make sure you clear Drupal's cache and then refresh the browser. If we preview the About Us page, we will now see our completed Team Members section. By using Twig to create a template for our After Content region, we enabled the markup to be responsive with the `container` class as well as using the `classes` variable to display the name of the region for our global styling to take effect.

In order for us to complete the interior page structure, we need to add back our global footer. Let's take a look at doing that now.

Adding our global footer

Our website has several global components that were present on our homepage that don't currently exist on our interior pages. One such item is the global header, which consists of the utility menu, logo, main menu, and search form block.

To add this section to our template, we need to simply copy the `footer` markup from the `page--front.html.twig` template:

1. Open `page.html.twig` and delete the entire contents.
2. Open `page--front.html.twig` and copy the following markup:

```
<footer id="footer" role="contentinfo">

    <div class="container main-footer">
        <div class="row">
            <div class="col-md-4">
                {{ page.footer_first }}
            </div>

            <div class="col-md-4">
                {{ page.footer_second }}
            </div>

            <div class="col-md-4">
                {{ page.footer_third }}
            </div>

        </div>
    </div>

    <div class="footer-copyright">
        <div class="container">
            <div class="row">
```

Theming Our Interior Page

```
<div class="col-md-8">
    {{ page.footer_bottom_left }}
</div>

<div class="col-md-4">
    {{ page.footer_bottom_right }}
</div>

</div>
</div>
</div>

</footer>
```

3. Paste the markup into `page.html.twig`.

4. Save `page.html.twig`.

Make sure you clear Drupal's cache and refresh the About Us page within your browser. Our footer is now in place, and it is functioning as we would expect.

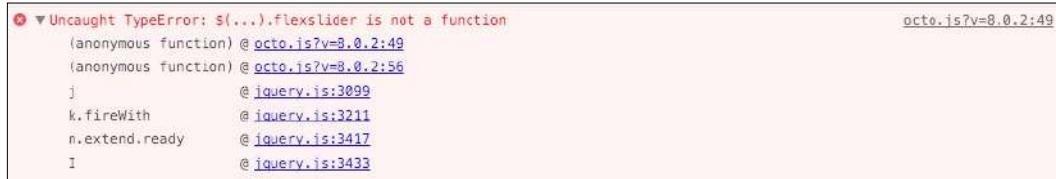


We have successfully completed theming our About Us page and in the process created our interior page template. The remaining pages we will be creating can take advantage of our new `page.html.twig` template, but before we wrap up this lesson I want to address one more item.

Fixing JavaScript errors

When we created our homepage, we attached a couple of JavaScript libraries directly to our `page--front.html.twig` template. However, since we are loading our theme's custom scripts file globally, this can sometimes create unnecessary JavaScript errors.

If we inspect our About Us page, we will see one such error caused by our script trying to configure the FlexSlider library, which only exists on our homepage.



```
① ▾ Uncaught TypeError: $(...).flexslider is not a function
  (anonymous function) @ octo.js?v=8.0.2:49
  (anonymous function) @ octo.js?v=8.0.2:56
    j          @ jquery.js:3099
    k.fireWith @ jquery.js:3211
    n.extend.ready @ jquery.js:3417
    I          @ jquery.js:3433
```

While we are not covering the fundamentals of JavaScript and how to write proper syntax, it is important to point out this issue. This can be a common error when using JavaScript with the different techniques used to theme Drupal 8. So, let's take a quick look at how to fix this to have this as part of our theming tools moving forward.

Begin by navigating to `themes/octo/js` and opening the `octo.js` file. From here, we can follow these steps to resolve the JavaScript error:

1. Locate the **Flexslider** function call.
2. Wrap the function in a conditional structure that will look to see if the `flexslider` library exists before configuring it. The revised structure should look like the following:

```
//-- Flexslider
(function() {

  if (typeof $.fn.flexslider === 'function') {

    $('.flexslider').flexslider({
      direction: "vertical",
      controlNav: false,
      directionNav: false
    });
  }

})();
```

3. Save `octo.js`.

The simple `typeof` operator can be used with any JavaScript library we may be referencing within our theme to ensure that we don't initialize a library unnecessarily. If we clear Drupal's cache and then reload our About Us page, we will no longer have any JavaScript function errors.

Summary

We covered a lot of different techniques while recreating our About Us page. From reviewing the mockup to working with various Twig templates, our theming skills have improved. Let's take a moment to recap what we accomplished in this chapter:

- We began with reviewing our About Us mockup to help identify the different page elements we would need to consider when creating our interior page template.
- Next, we added back any global regions to our template so that users would be able to navigate to the various pages of our website.
- The page title plays a very important part in identifying where a user is within our site, so you learned how to work with the Page title block, create a block template, and refactor markup to match our design.
- Our Team Members section required you to learn different techniques to create and format fields using Drupal's View module. We rewrote fields using Twig, added View headers to create introductory text, and followed up by adding CSS classes to various sections.
- Finally, we looked at a common JavaScript error and how to resolve loading libraries unnecessarily.

In the next chapter, we will continue our theming by taking a look at setting up our Blog Landing page, working with various display modes associated with content types, and using those display modes with Views.

8

Theming Our Blog Listing Page

The blog section of our website will be by far be the most complex to set up, as we will be taking advantage of the Twig template layer to modify the HTML markup from the Node level all the way down to the field level. What does this mean? It simply means we will be breaking our mockup down into small chunks, whether it be the blog teaser, a listing of blog categories, or even the simple blog image itself.

In this chapter, we will look at creating multiple Twig templates that our Blog listing page will use, as well as the following:

- We will begin by reviewing our Blog listing page as displayed in our mockup and identifying the areas we will need to theme.
- Next, we will create our Blog listing page, along with a teaser view of our content that will link to the Blog detail page.
- Our Blog listing also contains three custom blocks of content, which will require us to dive a little deeper into using Drupal views, custom blocks, and Twig templates to create categories, popular blogs, and a recent blogs list.
- We will also take a look at how we can deal with multiple field items to create a slideshow of images that will be used both on our Blog listing and Blog detail pages.
- Finally, we will focus on how to work with comments and the theme layer to display them properly.

While we work through each section, we have the ability to refer back to the Chapter08 exercise files folder. Each folder contains a start and end folder with files that we can use to compare our work when needed. This also includes database snapshots that will allow us all to start from the same point when working through various lessons.

Reviewing the Blog Listing mockup

In order to assist us in identifying page elements that we will be recreating for the blog page, it would make sense to open up our mockup and review the layout and structure. The Blog page can be found in the `Mockup` folder located in our exercise files. Begin by opening up the `blog.html` file within the browser, as shown in the following image:

The image shows a screenshot of a blog listing page. At the top, there's a header with the word "Blog". Below the header is a large image of a man in a suit writing the word "Idea" on a chalkboard. To the right of the image is a sidebar with a "Categories" section containing links for Design, Lifestyle, News, Photos, and Videos. Below that is a "Popular" section with two items: "Post Two" (May 23, 2015) and "Post One" (May 23, 2015). At the bottom of the sidebar is an "About Us" section with some placeholder text. The main content area features a post titled "Post Three" from May, with a preview of the post content and a "Read more..." button.

Blog

Post Three

May

Euismod atras vulputate iltricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae.

By admin Photos, Design 0 Comments Read more...

Categories

- Design
- Lifestyle
- News
- Photos
- Videos

Popular **Recent**

Post Two May 23, 2015

Post One May 23, 2015

About Us

Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Nulla nunc dui, tristique in semper vel. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero.

If we look at our mockup and break it down into manageable parts, we will notice several components that we will need to create using our existing post content type:

- The first is the blog teaser in the main content area, which consists of one or more images, post metadata, title, text, and some taxonomy terms that help identify the type of post we are viewing. Since this is repeated content, we will take advantage of Drupal views and custom view modes to recreate this section.
- The second is the **Categories** listing in the right-hand sidebar, which displays the category of posts available on our website. This is a simple HTML list that we can recreate using Drupal views.
- The third is a tabbed component to display **Popular** and **Recent** blog posts. While this looks a little more complex, we will use some advanced Drupal views techniques to recreate it.
- The fourth is a custom block with some **About Us** text and should be simple to develop.

So we have identified four specific page components for our blog listing page. We will concentrate on creating these various items of our site and, once complete, we should be able to compare our blog listing page with our mockup.

Creating our blog listing

Our blog listing is a shortened representation of our Blog detail page, with just enough information to tease our users into reading more. To help us identify what fields we will need to display for our blog listing, we should review each individual post on our mockup. We can visually identify that we will need to display the following fields:

- Image
- Post date
- Title
- Teaser
- Author
- Tags (taxonomy)
- Comments

While all of these fields make up our post content type, there are also additional fields such as Thumbnail and Full content that are not being displayed. So how would we go about presenting the same data to users differently, based on the layout? We could hide fields using CSS, or add PHP conditions, but there is a much easier method for creating different display modes.

Adding a new display mode

One feature of Drupal 8 that provides us with the flexibility to present content differently based on specific requirements is **display modes**. By default, Drupal provides each content type or node with a handful of displays for enabling fields based on a default view or teaser view. We can see a listing of all the current display modes associated with content by navigating to `/admin/structure/display-modes/view`, as shown in the following image:

| NAME | OPERATIONS |
|----------------------------------|------------|
| Full content | Edit |
| RSS | Edit |
| Search index. | Edit |
| Search result highlighting input | Edit |
| Teaser | Edit |

[Add new Content view mode](#)

However, we are not limited to just these displays. We can create different displays for both Node forms and Node views. In fact, let's create a new view mode for our blog listing by following these steps:

1. Click on the **Add new Content view mode** link.
2. Enter the value **Listing** in the **Name** text field.
3. Click on the **Save** button.

Add new **Content** view mode ★

Home » Administration » Structure » Display modes » View modes » Choose view mode entity type

Name Machine name: node.listing [Edit]

Save

Now that we have created our new View mode, we can use it with any content type that we choose. In the case of our post content type, we will utilize it to manage what fields will be displayed when using the listing display.

Managing the display

The ability to manage a content types display has actually been around since Drupal 6. Once known as build modes, we can now utilize our new display by navigating to /admin/structure/types/manage/post/display and enabling the custom display settings as shown:

▼ CUSTOM DISPLAY SETTINGS

Use custom display settings for the following modes

Full content
 Listing
 RSS
 Search index
 Search result highlighting input
 Teaser

1. Click on the **CUSTOM DISPLAY SETTINGS** fieldset to expand it.
2. Check the **Listing** checkbox under **Use custom display settings for the following modes**.
3. Click on the **Save** button.

Our new listing display should now be available to select along the top of our **Manage display** screen. Selecting **Listing** will display a new view mode that we can use to specify which fields will be enabled.

Enabling fields

Fields can be dragged, dropped, and rearranged by simply clicking on the drag icon next to each field name. Any fields located under the **Disabled** section will not be displayed when the Listing view mode is used. For our blog listing page, we only want to display the **Image**, **Teaser**, and **Tags** fields. Go ahead and drag these into place, as shown in the following image:

| FIELD | LABEL | FORMAT |
|-----------------|------------|------------|
| Image | - Hidden - | Image |
| Teaser | Above | Default |
| Tags | - Hidden - | Label |
| Disabled | | |
| Full content | - Hidden - | - Hidden - |
| Links | | - Hidden - |
| Thumbnail | Above | - Hidden - |
| Comments | Above | - Hidden - |

Once we have rearranged the fields as needed, we can click on the **Save** button to lock in our changes.

Managing the display of fields is not just about selecting which fields are enabled or disabled. We also have the ability to specify how the label of each field will be presented, as well as how each field will be formatted.

Field label visibility

The label for each field can have one of four states. The label can display above or inline with the field, as well as be hidden or visually hidden. Visually hiding a label does not prevent the label itself from being available for screen readers, and is great for accessibility requirements.

For our three fields, we will want them to all be hidden from our display, so we can follow these steps to hide each field:

1. Click on the **LABEL** dropdown next to the **Image** field and choose **Hidden**.
2. Click on the **LABEL** dropdown next to the **Teaser** field and choose **Hidden**.
3. Click on the **LABEL** dropdown next to the **Tags** field and choose **Hidden**.
4. Click on the **Save** button.

| Show row weights | | | | |
|------------------|------------|---------|-------------------------------|---|
| FIELD | LABEL | FORMAT | | |
| + Image | - Hidden - | Image | Original image | * |
| + Teaser | - Hidden - | Default | | |
| + Tags | - Hidden - | Label | Link to the referenced entity | * |

Formatting fields

Each field can also be formatted so that it is displayed differently, based on the formatting options. This allows us to have finer control over how text is displayed, how content is referenced, and what size an image is displayed at. Based on the formatting options, there may be additional settings that can be applied.

For example, if we click on the gear icon to the right of our image field, we will be presented with options to select various image styles and even enable the image to link to the node itself:

| FIELD | LABEL | FORMAT |
|---------|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| + Image | - Hidden - | <p>Format settings: Image</p> <p>Image style</p> <p>None (original image) <input type="button" value="Configure Image Styles"/></p> <p>Link image to</p> <p>Nothing <input type="button" value=""/></p> <p><input type="button" value="Update"/> <input type="button" value="Cancel"/></p> |

Currently, our fields do not require formatting, but it is important to know where to manage this functionality when the need arises.

Creating a Post Listing view

So now that we have a new listing display for our post content type, you may be asking how do we actually use it? Just like any content that we want to aggregate, we will start with creating a Drupal view. But unlike previous views that we created, which used fields, we will be creating a view that uses the Content Types display mode.

To get started, we will need to navigate to /admin/structure/views and click on the **Add new view** button. From the **Views** admin screen, we will add the following information:

- **VIEW BASIC INFORMATION:**
 1. **View name:** Post Listing.
 2. Check the **Description** box.
 3. **Description:** A listing of all Posts.
- **VIEW SETTINGS:** Show: Content of type: Post sorted by: Newest first
- **BLOCK SETTINGS:**
 1. Check the **Create a block**.
 2. **Block title:** Post Listing.
- **BLOCK DISPLAY SETTINGS:**
 1. **Display format:** Unformatted list of: teasers.
 2. **Items per block:** 3.
 3. Click on the **Save and edit** button.

All of these steps are similar to the Team Member view we created in *Chapter 7, Theming Our Interior Page*. Instead of displaying fields though, we are displaying the Teaser view mode to begin with and will modify our format to use the Listing display mode next.

Using Content Display modes with views

Our Post Listing view is currently using the Teaser view mode of our Post content type. This varies from the typical fields display that we have created so far. Using a view mode from a content type is more flexible because it allows us to manage the display of our fields from the content type itself, without the need to modify our view in the future.

If we preview our view, we will get a glimpse of what fields our post's teaser display has enabled:



In order for us to utilize the Listing view mode for our Post content type, we will need to modify the format currently being used. We can change this by following these steps:

1. Select the **Teaser** link under the **FORMAT** section.
2. Choose **Listing** from the **View mode** dropdown.
3. Click on the **Apply** button.
4. Click on the **Save** button to save the changes.

If we preview the results, we will see our display has changed and we are now showing only the fields we enabled previously on the Listing view mode.

Now that we have our Post Listing view created, we can manage the block display and place it on our Blog listing page.

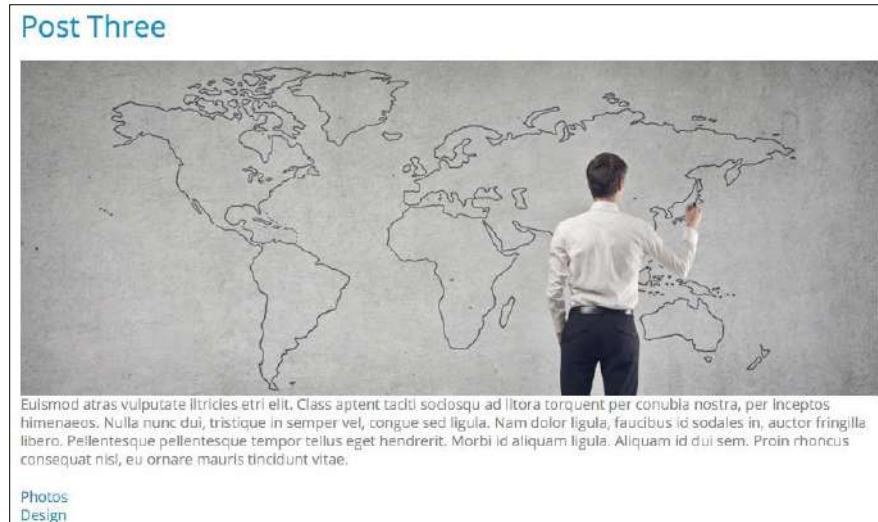
Managing our Post Listing block

Any time we create a new block display using views, we can easily assign it to any region from the Block layout page. Begin by navigating to /admin/structure/block and follow these steps:

1. Locate the **Content** region.
2. Click on the **Place block** button.
3. Locate the **Post Listing** block.
4. Click on the **Place block** button.
5. Uncheck the **Display title** checkbox.
6. Select the **Pages** tab under **Visibility**.
7. Enter the path /blog into the **Page** text field.
8. Make sure the **Show for the listed pages** checkbox is selected.
9. Click on the **Save block** button.
10. Click on the **Save blocks** button at the bottom of the **Block layout** page.

Make sure the Post Listing block is the last block displayed in our content region. If not, then reorder the blocks accordingly and click on the **Save blocks** button at the bottom of the Block layout screen.

Navigate back to the Blog listing page by browsing to /blog or by selecting the Blog link in the main menu. We can see that our Post listing block is now displaying on the page in a single column, as shown in the following image:



If we refer back to our mockup ,we can see that while most of the content we need is being displayed, we are still missing the post date, author, and comment count. Also our structure and styling still needs some work. This is where creating a Twig template specific to our nodes display mode will allow us to modify our markup and add any missing variables that our page may need.

Implementing our Node template

Creating a node template is not foreign to us; our process begins the same way we would create any Twig template. We begin by inspecting the markup of our page to identify which template is currently being used and the various file name suggestions available to us. Since our Post Listing block is referencing the listing view mode of our post content type, we can create a new Twig template called `node--post--listing.html.twig`.

Begin by grabbing a copy of the `node.html.twig` template from `core/modules/node/templates` and placing it into our `themes/octo/templates` folder. We will then rename the file to `node--post--listing.html.twig` so that we only affect the markup for this particular content types display. Next, we will need to replace the current markup within our template with the following new markup:

New markup

```
<article{{ attributes }}>
  {{ content }}
</article>
```

Make sure to save the template, clear Drupal's cache, and refresh the Blog page. At first glance, nothing has changed, but we are now using a custom Twig template to display our content. We will be walking through building up our template until it resembles our mockup by discussing the following techniques:

- First, we will use Twig to add additional CSS classes to our `article` element. This will allow us to apply formatted styling to each individual post that is being repeated on the page.
- Next we will work with the `{{ content }}` variable to print individual fields. This will give us the opportunity to add additional HTML markup for both formatting and structure to each post.
- We will learn how to work with field level templates to account for multiple content within a single field. This will enable us to create the image slider for each post when it has more than one image uploaded, as well as add additional markup to individual fields so that taxonomy tags display correctly.

- Twig filters play an important role in theming and we will learn how to use them to format dates. This will allow us to display the post date properly, based on the design.
- Finally, we will dive into the Theme layer to preprocess variables needed to help identify the path for each page and to retrieve the comment count if one exists for each post.

Adding CSS classes to Twig

Adding additional CSS classes to our markup is not new for us. We implemented this technique in the previous lesson when we developed our `node--landing-page.html.twig` template in *Chapter 7, Theming Our Interior Page*. In fact, Drupal 8 provides us with various ways to work with its attributes, with everything from adding classes to removing classes. We are only touching the surface of how we can use this technique. For even more information, we can refer to the documentation at [Drupal.org](https://www.drupal.org/node/2513632), located at <https://www.drupal.org/node/2513632>.

For our post listing, we need to add two CSS classes to our `article` element. We can accomplish this by setting a `classes` variable within our template and then using the `{{ attributes.addClass(classes) }}` function to inject the two new classes into the attributes.

Open up `node--post--listing.html.twig` and replace the markup with the following:

New markup

```
{% set classes = ['post', 'post--listing'] %}

<article{{ attributes.addClass(classes) }}>
  {{ content }}
</article>
```

Make sure to save the template, clear Drupal's cache, and refresh the blog page. We should see a slight change in our styling with the introduction of a bottom border separating each post.

Working with content variables

So far, our node template is just printing out the `{{ content }}` variable, which contains all the fields we have enabled for the listing display mode. In some cases this may suffice, but we can actually print the individual fields that the `{{ content }}` variable contains by referencing the field using dot syntax notation. Having this flexibility allows us to add structural markup for each field.

Let's give this a try by opening `node--post--listing.html.twig` and modifying our markup to print the image field for our post:

New markup

```
{% set classes = ['post', 'post--listing'] %}

<article{{ attributes.addClass(classes) }}>
  {{ content.field_image }}
  {{ content }}
</article>
```

Make sure to save the template, clear Drupal's cache, and refresh the blog page. We should see our post image is now duplicated for each post. Why is this?

Well, if we refer back to our markup, we are telling Drupal to output `{{ content.field_image }}`, which prints the image field. But we are also following that by printing all of the content using `{{ content }}`.

Using the without filter

Any time we are developing a Twig template and start printing individual fields for a content type, we will want to prevent the same field from being printed again when the `{{ content }}` variable is called. Twig provides us with the `without` filter to assist us in accomplishing this.

To test this, open up `node--post--listing.html.twig` and modify the markup to reflect the following:

New markup

```
{% set classes = ['post', 'post--listing'] %}

<article{{ attributes.addClass(classes) }}>
  {{ content.field_image }}
  {{ content|without('field_image') }}
</article>
```

Make sure to save the template, clear Drupal's cache, and refresh the Blog page. We now only have a single instance of our image field displayed. Using the `without` filter and passing the field name as a value has successfully prevented the field from printing a second time. We will continue using this technique, as well as printing the individual fields for our Post Listing block as we theme our node.

Creating our post image slider

If we refer back to our mockup of the blog page by opening the `blog.html` page located in our Mockup folder, we will notice that our post can contain one or more images. However, when more than one image exists, we present the image in a slider for the user to see all post images:



In order for us to implement this functionality, we will need to know exactly how many images our image field contains. While we can't access this information from the node level, we can implement this from the field level by creating a field template specifically for the field.

Working with field templates

If we browse back to our Drupal instance and navigate to the blog page, we can inspect the image field and locate which Twig template is currently being used by Drupal:

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'field' -->
<!-- FILE NAME SUGGESTIONS:
     * field--node--field-image--post.html.twig
     * field--node--field-image.html.twig
     * field--node--post.html.twig
     * field--field-image.html.twig
     * field--image.html.twig
     x field.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/system/templates/field.html.twig' -->
▶ <div data-quickeedit-field-id="node/11/field_image/en/listing">_</div>
<!-- END OUTPUT from 'core/modules/system/templates/field.html.twig' -->
```

Drupal provides multiple **FILE NAME SUGGESTIONS** for us to choose from, but since we don't know if we will reuse this field with another content type later, we will be as specific as possible and create `field--node--field-image--post.html.twig` by following these steps:

1. Navigate to the `core/modules/system/templates` folder and copy the `field.html.twig` template.
2. Place `field.html.twig` into our `themes/octo/templates` folder and rename it `field--node--field-image--post.html.twig`.
3. Open `field--node--field-image--post.html.twig` and replace the markup with the following:

```
{% if items|length > 1 %}
<div class="post-image">
  <div class="owl-carousel">
    {% for item in items %}
      <div class="img-thumbnail">
        {{ item.content }}
      </div>
    {% endfor %}
  </div>
  {% else %}
    <div class="single-post-image post-image">
      {% for item in items %}
        <div class="img-thumbnail">{{ item.content }}</div>
      {% endfor %}
    </div>
  {% endif %}
```

Make sure to save the template and clear Drupal's cache. Before we preview the blog page let's take a moment to discuss what exactly is happening in this new template.

1. First, we are using a Twig filter `{% if items|length > 1 %}` to test the length of the `items` variable to see if our image field contains multiple images. If it does, we will print the markup for that condition, but if not, we will print the markup contained in the `{% else %}` condition.
2. Second, since any field item can contain multiple items, we need to loop `{% for item in items %}` through the items and print `{{ item.content }}`.
3. Finally, the remaining markup adds the necessary structure and CSS classes for the next step of the process, which is adding the Owl Carousel library to our theme.

If we now browse to the blog page, we will see that our field images are styled to include a rounded border, a light box shadow, and some necessary padding.

Adding the Owl Carousel library

In order to add the slider functionality to multiple images, we will be implementing Owl Carousel, which can be found at <http://owlgraphic.com/owlcarousel/>. Owl Carousel provides us with a responsive touch enabled slider that we can apply to our post images. We will be grabbing a copy of the library from our exercise files, adding the library to our theme, configuring it within our `octo.libraries.yml` file, and then using Twig to attach the library to our node templates.

Begin by navigating to the `Chapter08/start/themes/octo/vendor` folder and copying the `owl-carousel` folder to our `themes/octo/vendor` folder. Once the files are accessible by Drupal, we can add the reference to our library by following these remaining steps:

1. Open `octo.libraries.yml`.
2. Add the following entry:

```
owl-carousel:  
  version: 1.3.3  
  css:  
    theme:  
      vendor/owl-carousel/owl.carousel.css: {}  
      vendor/owl-carousel/owl.theme.css: {}  
    js:  
      vendor/owl-carousel/owl.carousel.min.js: {}  
  dependencies:  
    - core/jquery
```

3. Save `octo.libraries.yml`.

Before we can preview the new functionality, we still have a few steps left to complete. Next, we will need to attach the library to our Blog listing page. As we will not want this library to load on every page, using the `{{ attach_library() }}` function will be our preferred method.

Begin by opening `node--post--listing.html.twig` and then adding the following Twig function to the top of our template:

```
{{ attach_library('octo/owl-carousel') }}
```

It is very important to make sure that we are using single quotes to surround the path to our library, otherwise the slightest typo will cause the library to not be loaded. Now make sure to save the template and then we can move on to initializing Owl Carousel.

Begin by opening `octo.js` and adding the following script to the bottom of our file directly below our `scrollTo` function:

```
//-- Owl Carousel
(function() {

    if (typeof $.fn.owlCarousel === 'function'){
        $('.owl-carousel').owlCarousel({
            slideSpeed : 300,
            paginationSpeed : 400,
            singleItem:true
        });
    }

})();
```

Make sure to save `octo.js` and then let's review exactly what our new function is doing:

1. First, we are conditionally checking if the Owl Carousel function exists. This ensures we avoid any JavaScript errors on other pages where our library is not loaded.
2. Next, we are initializing Owl Carousel and passing three parameters to it: one for how fast an image should slide, one for pagination, and then finally we tell it how many images we want to display.

With our library added, attached to our template, and initialized, we can now clear Drupal's cache and refresh our Blog listing page. We will now see that any posts that contain multiple images display as a slider. This provides some nice responsive functionality for our users.

Using Twig filters for dates

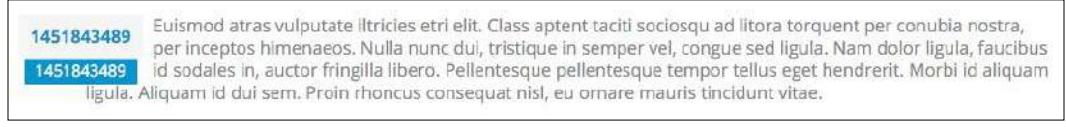
The next section of our post listing we need to address is the post date. By default, Drupal will print dates in the form of a Unix timestamp. In most cases, we will want to convert these dates into a more user friendly format. Luckily, we can take advantage of another Twig function, `date()`, to convert dates easily.

Begin by opening `node--post-listing.html.twig` and adding the following markup directly after where we are printing the post image:

New markup

```
<div class="post-date">
  <span class="day">{{ node.createdtime }}</span>
  <span class="month">{{ node.createdtime }}</span>
</div>
```

Make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. We will now see exactly what the Unix timestamp looks like when printed. While our theming is now being applied, we need to convert the timestamp into day and month:



1451843489 Euismod atras vulputate iitricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae.

1451843489

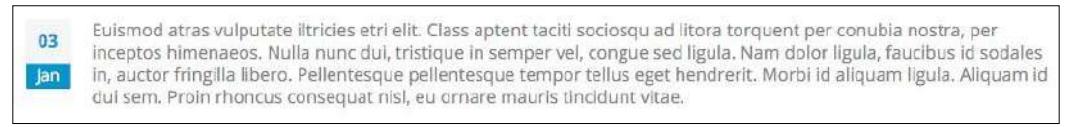
Let's now use the `date()` function and apply it to `node.createdtime` to display the day and month format of our timestamp.

Open `node--post-listing.html.twig` and modify our post-date markup to reflect the following:

New markup

```
<div class="post-date">
  <span class="day">{{ node.createdtime|date('d') }}</span>
  <span class="month">{{ node.createdtime|date('M') }}</span>
</div>
```

Make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. By adding the simple Twig function to our timestamp, we now have the correct formatting and styling:



03 Euismod atras vulputate iitricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae.

Jan

Our post listing is starting to come together. Next, we will add the title of our post along with the teaser field, before moving on to our metadata.

Printing title and teaser

Working with Node titles when theming can sometimes be a mystery, unless you have a good understanding of what Drupal is doing behind the scenes. The title field is one of the few required fields that each Content type must contain. The challenge is that while it's considered a field, it doesn't truly function like other fields. We do not have access to manage the Node title in the admin, as we can with the rest of our fields. While we could extend the functionality by using modules such as <https://www.drupal.org/project/title>, we will manage the Node title directly within our Twig template.

Open `node--post-listing.html.twig` and add the following markup directly after our post-date section:

New markup

```
<div class="post-content">

    {{ title_prefix }}
    <h2{{ title_attributes }}>
        <a href="{{ url }}" rel="bookmark">{{ label }}</a>
    </h2>
    {{ title_suffix }}

    {{ content.field_teaser }}

</div>

{{ content|without('field_image','field_teaser') }}
```

Make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. Our post title and teaser should now be displayed:

03

Jan

Post Three

Euismod atras vulputate iltricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc du, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pelientesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae.

We are familiar with printing individual fields and removing them from the main content flow. What is new though is how we print the title. Within a node template, the title is referred to as `{{ label }}` and the `{{ url }}` variable points to the path of the individual post. We utilize these two variables to create our title within a `h2` heading and a link. We also have some extraneous variables, such as `{{ title_prefix }}` and `{{ title_suffix }}`, which are utilized by contributed modules to inject markup before or after our title.

Creating our post metadata

We are getting close to completing the theming of the post listing. We are still missing the post metadata, which consists of the author, the categories that a post has been tagged with, and the comment count.

We will begin with adding the structural markup for our post-meta content, including printing the author associated with each post:

1. Open `node--post--listing.html.twig`.
2. Add the following markup directly below the `teaser` variable:

```
<div class="post-meta">

  <span class="post-meta-user">
    <i class="fa fa-user"></i> By {{ author_name }}
  </span>

</div>
```

Make sure to save the template, clear Drupal's cache, and then refresh the blog listing page. Our post author is now displayed, with a link to the user profile. Since we are using the default author information for the node, and not a specific field, we have the ability to print that information using the `{{ author_name }}` variable:

03 Post Three

Jan Euismod atras vulputate iltricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae.

By admin

Now we need to address the `tags` field, which displays any categories associated with a post. The `tags` field, as we will see in a moment, can contain one or more values, which will require us to modify the markup on the field level:

1. Open `node--post--listing.html.twig`.
2. Add the following markup within our `post-meta` section, directly below our `post-meta-user` markup:

```
<span class="post-meta-tag">
  <i class="fa fa-tag"></i> {{ content.field_tags }}
</span>
```
3. Make sure to exclude the `field_tags` variable from the main `content` variable.

Make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. We should now see our post categories displayed only once per post. However, our tags are displayed stacked on top of each other, instead of inline as our design requires. If we inspect the page markup for our tags, we will see that each category has a `div` element wrapped around it, causing them to be block level elements.

Field templates and taxonomy

In order for us to modify the markup for our taxonomy tags, we will need to create a field level template. Using the file name suggestions provided by Twig, we can create our own template by following these steps:

1. Navigate to the `core/modules/system/templates` folder and copy the `field.html.twig` template.
2. Place a copy of the template in our `themes/octo/templates` folder.
3. Rename the file `field--field-tags.html.twig`.
4. Replace the markup with the following:

```
<ul>
  {%
    for item in items %}
      <li{{ item.attributes.addClass('field-item') }}>
        {{ item.content }}
      </li>
    {% endfor %}
  </ul>
```

Make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. Based on the markup we added, we have replaced the block level elements with an unordered list, and each taxonomy tag is now a list item:

03
Post Three

Jan
Euismod atras vulputate iltricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae.

By admin
Photos, Design

Our post listing is starting to take shape, but you may have noticed that we are not returning the number of comments that each post contains. How do we remedy this when there is no comment count variable available to the Node template?

Handling comments in Drupal 8

One of the new things introduced in Drupal 8 is the comment field. Previously, when you created or edited a content type, you would simply enable Comments by turning them on or off. Well, comments are now their own field and must be added to a content type like any other field that you want to create.

If we navigate to `/admin/structure/comment`, we will get a glimpse of the **Default comments** configured by Drupal.



The screenshot shows the 'Comment types' page in the Drupal 8 administration interface. The title bar says 'Comment types'. Below it, a breadcrumb navigation shows 'Home » Administration » Structure'. A note states, 'This page provides a list of all comment types on the site and allows you to manage the fields, form and display settings for each.' A blue button labeled '+ Add comment type' is visible. The main content area has a table with three columns: 'COMMENT TYPE', 'DESCRIPTION', and 'OPERATIONS'. One row is shown for 'Default comments', which is described as 'Allows commenting on content'. To the right of the description is a 'Manage fields' button with a dropdown arrow.

Comment types are similar to Content types, as they are fieldable and multiple comment types can be created. Feel free to inspect the comment type in more detail, but be aware that this is the comment type we are using with our Post content type.

In fact, if we navigate to `/admin/structure/types/manage/post/fields`, we will see that we have a field called `field_comments` which is of **FIELD TYPE Comments**. The comment field was added to our Post content type and provided to us with the database snapshot.

If we were to add this field to our Listing display, it would not return a comment count, but instead display the Comment form. Based on our design, that does not help us. However, knowing the field name for our comments will allow us to do some advanced theming within the Drupal 8 theme layer to retrieve the comment count, place it within a variable and print it within our Twig template.

Creating a theme file

The `*.theme` file is a PHP file that contains theme hooks for preprocessing variables. We will create a theme file specific to our theme that we can use to grab the comment count, based on each individual post, and then return the count to our Twig template as a variable that can be printed.

Let's begin by creating a new file called `octo.theme` and saving it to our `themes/octo` folder.

Next, we will add the following PHP code:

```
<?php

function octo_preprocess_node(&$variables) {
    $node = $variables ['elements'] ['#node'];
    $id = $node->id();

    // Create comment count variable for template
    $count = _octo_comment_count($id);
    $variables['comment_count'] = _octo_plural($count, 'Comment',
    'Comments');
}
```

The `octo_preprocess_node(&$variables)` function is known as a theme hook and is an adaptation of `theme_preprocess_node`. Within this function, we are passing by reference any variables accessible to the Node using `&$variables`. Since everything in Drupal is an array, we can traverse the `$variables` array to retrieve the Node ID, which we use to pass to a custom function that returns the number of comments for each node.

Next, we will add the two custom functions directly below our preprocess function:

```
function _octo_comment_count($id) {
    $count = db_query("SELECT comment_count FROM
    comment_entity_statistics WHERE entity_id = :id",
    array(':id' => $id))->fetchField();

    return empty($count) ? '0' : $count;
}

function _octo_plural($count, $singular, $plural) {
    if ( $count == 1 )
        return $count . ' ' . $singular;
    else
        return $count . ' ' . $plural;
}
```

Our first custom function returns the comment count for a specific node ID, which is passed to the function from our preprocess function. This custom function uses `db_query` to select the count from the `comment_entity_statistics` table in Drupal.

Our second custom function allows us to pluralize the count and return a more formatted count to our preprocess function, which we in turn will assign to our `comment_count` variable for use in our Twig template.

Once finished, make sure to save our file and clear Drupal's cache.

Printing our comment count

Now that we have utilized the theme layer to create a new variable containing the comment count for each node, we can print the variable to our template by following these steps:

1. Open `node--post--listing.html.twig`.
2. Add the following markup directly after our `post-meta-tag` section:

```
<span class="post-meta-comments">
  <i class="fa fa-comments"></i>
  <a href="{{ url }}/#comments">{{ comment_count }}</a>
</span>
```

Make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. Based on the markup we added, we now have our comment count displayed for each post:

The screenshot shows a single blog post card. At the top left is a small blue square icon with the number '03'. To its right is the title 'Post Three'. Below the title is a small blue box containing the author's name 'Jan'. To the right of the author box is a large block of placeholder Latin text: 'Euismod atras vulputate iltricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae.' At the bottom of the card, there is a small footer area with icons for user, photos, design, and comments, followed by the text 'By admin Photos Design 0 Comments'.

Adding a read more link

We have almost completed the theming of our Post listing. We have one more component to add, and that is our read more link. We have all the elements we need to create this link, so let's start by following these steps:

1. Open `node--post--listing.html.twig`.
2. Add the following markup directly after our `post-meta-comments` section:

```
<a href="{{ url }}" class="button button--primary button--xs
pull-right">Read more...</a>
```

Make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. The main content area of our Post listing is now finished. It is time to move on to the sidebar and our three blocks of content, which contain Categories, Popular content, and our About block.

Creating a Categories block

The content of each of our posts has been assigned one or more tags to identify what category the post belongs to. This type of identification gives our end users another way to easily navigate content. On our mockup, the right-hand sidebar contains a custom block with an unordered list of categories. We will utilize views to create a block display of taxonomy terms by following these steps.

To get started, we will need to navigate to `/admin/structure/views` and click on the **Add new view** button. From the **Views** admin screen, we will add the following information:

- **VIEW BASIC INFORMATION:**
 1. **View name:** Categories.
 2. Check the **Description** box.
 3. **Description:** Post categories.
- **VIEW SETTINGS:** Show: Taxonomy terms of type: Tags sorted by: **Unsorted**.
- **BLOCK SETTINGS:**
 1. Check **Create a block**.
 2. **Block title:** Categories.
- **BLOCK DISPLAY SETTINGS:**
 1. **Display format:** Unformatted list of: Fields.
 2. **Items per block:** 5.
 3. Click on the **Save and edit** button.

Now that our Categories view has been created, we will need to adjust the format settings. With the view still open, we will need to adjust the following fields:

- **FORMAT SETTINGS:**

1. Click on the **Settings** link next to **Format: Unformatted list**.
2. Uncheck the **Add views row classes** checkbox from the **Block: Style options** window and click on the **Apply** button.
3. Click on the **Settings** link next to **Show: Fields**.
4. Uncheck the **Provide default field wrapper elements** checkbox from the **Block: Row style options** window and click on the **Apply** button.

We now need to adjust the settings of our Taxonomy term field to exclude any extraneous CSS. With the view still open, we will need to adjust the following field:

- **FIELDS:**

1. Select **Taxonomy term: Name**.
2. Expand **STYLE SETTINGS**.
3. Uncheck **Add default classes**.
4. Click on the **Apply (all displays)** button.

Make sure to click on the **Save** button in the main view window to save the changes we just made, and then look at the results in the **Preview** window.

Managing our Categories block

Any time we create a new block display using views, we can easily assign it to any region from the Block layout page. Begin by navigating to `/admin/structure/block` and then follow these nine steps:

1. Locate the **Sidebar second** region.
2. Click on the **Place block** button.
3. Locate the **Categories** block.
4. Click on the **Place block** button.
5. Select the **Pages** tab under **Visibility**.
6. Enter the path `/blog` into the **Page** text field.
7. On a second line, add another path to `/blog/*`.
8. Make sure the **Show for the listed pages** checkbox is selected.
9. Click on the **Save block** button.

With our Categories block assigned to the Sidebar second region, we will now need to add the Sidebar second region to our `page.html.twig` template before we can preview it.

Implementing responsive sidebars

So far, we have only been dealing with a one column layout. All of our blocks have been assigned to regions before, after, or within our main content. Now we are faced with our first block that is associated with a sidebar. The challenge is to make sure that when content is added to a sidebar, our main content region adjusts accordingly.

For this next section, we will be modifying our `page.html.twig` template to conditionally look for the existence of sidebars and alter the column classes of our content region.

Begin by opening `page.html.twig` and adding the logic and markup for the sidebar first region. This markup will be added directly below the `<div class="row">` section, but above the content wrapper:

New markup

```
{% if page.sidebar_first %}  
    <aside class="layout-sidebar-first" role="complementary">  
        <div class="col-md-3">  
            {{ page.sidebar_first }}  
        </div>  
    </aside>  
{% endif %}
```

The markup we added conditionally checks to see if any blocks are assigned to the Sidebar first region. If any blocks are present, it will then print the included markup and blocks within the region.

When a sidebar is available to print, we need to be able to adjust our main content region's grid measurements accordingly. We can use similar logic to test and then create a new column class that can be used for our content region.

Add the following markup directly after our Sidebar first region:

New markup

```
{% if page.sidebar_second and page.sidebar_first %}  
    {% set col_class = 'col-md-6' %}  
{% elseif page.sidebar_second or page.sidebar_first %}  
    {% set col_class = 'col-md-9' %}
```

Theming Our Blog Listing Page

```
{% else %}  
  {% set col_class = 'col-md-12' %}  
{% endif %}
```

The logic above checks for one or more sidebar regions and creates our new column class accordingly. We can then apply the new class to our content region by replacing the hardcoded column class with our new `col_class` variable:

```
<div class="{{ col_class }}>  
  {{ page.content }}  
</div>
```

Finally, we can add the conditional logic to print the Sidebar second region. This logic is similar to what we added for the Sidebar first region. Add the following markup directly below our main content region:

New markup

```
{% if page.sidebar_second %}  
  <aside class="layout-sidebar-second" role="complementary">  
    <div class="col-md-3">  
      {{ page.sidebar_second }}  
    </div>  
  </aside>  
{% endif %}
```

Make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. The main content area of our post listing is now adjusted to allow our Sidebar second region to display the Categories block:



If we look at our mockup, we can see that our Categories heading should be `<h4>` and the list of terms should be contained within an unordered list. How can we modify the markup for this block or view? Easy: we can create Twig templates for both the Block and the View to override the markup and add any classes that we need.

Theming a Block template

In the case of our Categories block, we can begin by navigating to the `core/modules/block/templates` folder and following these remaining steps:

1. Copy `block.html.twig` and place it into our `theme/octo/templates` folder.
2. Rename `block.html.twig` to `block--views-block--categories-block-1.html.twig` based on the template's suggestions.
3. Next, we will need to replace the current markup with the following new markup to convert the default `<h2>` to `<h4>`:

New markup

```
{% set classes = ['block'] %}

<div{{ attributes.addClass(classes) }}>
  {{ title_prefix }}

  {% if label %}
    <h4{{ title_attributes }}>{{ label }}</h4>
  {% endif %}

  {{ title_suffix }}

  {% block content %}
    {{ content }}
  {% endblock %}

</div>
```

Once finished, make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page in the browser. Our Categories block heading is now displayed properly. We have managed to alter the heading of our Categories block, but we still need to modify the output of our categories to display as an unordered list, along with any additional CSS classes we may need. Time to add a Views template.

Drupal Views and Twig templates

Unlike most of Drupal's templates, Views do not provide a file name suggestion for overriding Twig templates. So how do we know exactly what to name our template? View templates can be created in a variety of ways, but the easiest way to remember this is by following this rule:

```
[base template name] -- [view machine name].html.twig
```

So in the case of our Categories view, will want to create a new Twig template with the name `views-view-unformatted--categories.html.twig`.

Begin by navigating to the `core/modules/view/templates` folder and following these remaining steps:

1. Copy `views-view-unformatted.html.twig` and place it into our `theme/octo/templates` folder.
2. Rename `views-view-unformatted.html.twig` to `views-view-unformatted--categories.html.twig`.
3. Next, we will need to replace the current markup with the following new markup to convert the default `<div>` to ``:

New markup

```
{% if title %}
    <h3>{{ title }}</h3>
{% endif %}

<ul class="nav nav-list primary pull-bottom">
{% for row in rows %}
    {
        set row_classes = [
            default_row_class ? 'views-row',
        ]
    }
    <li{{ row.attributes.addClass(row_classes) }}>
        {{ row.content }}
    </li>
{% endfor %}
</ul>
```

Once finished, make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. Our Categories block is now styled correctly and matches our design:



Managing popular versus recent content

The second block of content that we need to create for our Blog listing page is a little more complex to build and will provide us with experience of building and combining multiple views into a single view.

Creating our recent posts block

Our recent posts block will be a view that contains a listing of three of the most recent posts added to our site. We will take advantage of the Teaser display mode of our Post content type to present our view block.

To get started, we will need to navigate to `/admin/structure/views` and click on the **Add new view** button. From the **Views** Admin screen, we will add the following information:

- **VIEW BASIC INFORMATION:**
 1. **View name:** Recent Posts.
 2. Check the **Description** box.
 3. **Description:** A listing of recent posts.
- **VIEW SETTINGS:** Show: Content of type: Post sorted by: Newest first.

- **BLOCK SETTINGS:**

1. Check **Create a block**.
2. **Block title:** Recent Posts.

- **BLOCK DISPLAY SETTINGS:**

1. **Display format:** Unformatted list of: Teasers.
2. **Items per block:** 3.
3. Click on the **Save and edit** button.

With our view now created, if we look at the **Preview** section, we will see our Post Teaser displayed with the title and thumbnail image. Since we are using the display mode of our post, we can manage the fields directly from the content type. Our Teaser display happens to be configured exactly how we will need it, so there is no need to change anything.

Make sure to click on the **Save** button to finalize our changes. We have the first part of our block created. Now we need to create our next view to display popular posts.

Creating our popular posts block

Popular posts, or anything popular for that matter, is all subjective. However, clients often want to see this type of information. We can accomplish this type of View block by utilizing the Comment statistics for each post. The number of comments each post has will determine which post will be displayed.

To get started, we will need to navigate to /admin/structure/views and click on the **Add new view** button. From the **Views Admin** screen, we will add the following information:

- **VIEW BASIC INFORMATION:**

1. **View name:** Popular Posts.
2. Check the **Description** box.
3. **Description:** A listing of popular posts.

- **VIEW SETTINGS:** Show: Content of type: Post sorted by: Newest first.

- **BLOCK SETTINGS:**

1. Check: **Create a block**.
2. **Block title:** Popular Posts.

- **BLOCK DISPLAY SETTINGS:**

1. **Display format:** Unformatted list of: Teasers.
2. **Items per block:** 3.
3. Click on **Save and edit** button.

With our view now created, if we look at the **Preview** section, we will see our Post Teaser displayed with the title and thumbnail image. However, we are only sorting the posts by the date they were authored versus the number of comments each post contains.

Sorting views by comment count

In order for us to determine the most popular posts, we will need to sort by the number of comments each post has. Begin by following these steps:

SORT CRITERIA:

1. Select the **Content: Authored on (desc)** link.
2. Click on the **Remove** link.
3. Click on the **Add** button.
4. Select **Comment count** from the **Add sort criteria** window.
5. Click on the **Apply (all displays)** button.
6. Choose **Sort descending** from the **Order** options.
7. Click on the **Apply (all displays)** button.

Make sure to click on the **Save** button in the main view window to save the changes. Now that we have our Popular Posts view complete, we need to combine it with our recent posts so that the two views act as one.

Attaching a view to the footer

One feature within views is the ability to create view footers. View footers can consist of custom text, other fields, or, as in our case, another view. We will use this feature to add our recent posts view by following these steps:

FOOTER:

1. Click on the **Add** button.
2. Scroll to bottom of the **Add footer** window and choose **View area**.
3. Click on the **Apply (all displays)** button.

4. Select **View: recent_posts - Display: block_1** from the **View to insert** dropdown.
5. Click on the **Apply (all displays)** button.

Make sure to click on the **Save** button in the main view window to save the changes. If we look in the preview window, we should now see both views being displayed. This is not so difficult once you understand how to use and manipulate Drupal views. Now that we have our two View blocks combined into a single Block, we can add it to our Blog listing page.

Managing our popular posts block

Any time we create a new block display using views, we can easily assign it to any region from the Block layout page. Let's begin by navigating to `/admin/structure/block` and following these steps:

1. Locate the **Sidebar second** region.
2. Click on the **Place block** button.
3. Locate the **Popular Posts** block.
4. Click on the **Place block** button.
5. Uncheck **Display title**.
6. Select the **Pages** tab under **Visibility**.
7. Enter the path `/blog` into the **Page** text field.
8. On a second line, add another path to `/blog/*`.
9. Make sure the **Show for the listed pages** checkbox is selected.
10. Click on the **Save block** button.

With our Popular Posts block assigned to the Sidebar second region, we will want to make sure it is second in the block order. Reorder the blocks if necessary and then click on the **Save blocks** button. If we navigate back to the Blog listing page, we will now see our new block displayed, but in desperate need of some styling.

Now for the fun part. We need to create a Twig template and modify the output of our View block so that we can place each view into its own tab. Let's take a look at how we can accomplish that.

Using Twig and Bootstrap tabs

The structure for Twitter Bootstrap tabs requires each block of content to be wrapped in a `<div>` element with a class of `tab-pane`. Also, each Tab pane must consist of an unordered list of items to display. We will start with converting both view blocks from an unformatted list to an unordered list, similar to what we did with our Categories block.

Recent Posts Twig template

Begin by navigating to the `core/modules/view/templates` folder and follow these remaining steps:

1. Copy `views-view-unformatted.html.twig` and place it in our `theme/octo/templates` folder.
2. Rename `views-view-unformatted.html.twig` to `views-view-unformatted--recent-posts.html.twig`.
3. Next, we will need to replace the current markup with the following:

New markup

```
{% if title %}  
    <h3>{{ title }}</h3>  
{% endif %}  
  
<ul class="simple-post-list">  
    {% for row in rows %}  
        {%  
            set row_classes = [  
                default_row_class ? 'views-row',  
            ]  
        %}  
        <li{{ row.attributes.addClass(row_classes) }}>  
            {{ row.content }}  
        </li>  
    {% endfor %}  
</ul>
```

Once finished, make sure to save the template, clear Drupal's cache, and then refresh the page in the browser to verify that our Recent Posts block is now displayed as an unordered list. We will now repeat this step for the Popular Posts view.

Popular Posts Twig template

Begin by navigating to the `core/modules/view/templates` folder and follow these remaining steps:

1. Copy `views-view-unformatted.html.twig` and place it in our theme/`octo/templates` folder.
2. Rename `views-view-unformatted.html.twig` to `views-view-unformatted--popular-posts.html.twig`.
3. Next, we will need to replace the current markup with the following:

New markup

```
{% if title %}
    <h3>{{ title }}</h3>
{% endif %}

<ul class="simple-post-list">
{% for row in rows %}
    {% set row_classes = [
        default_row_class ? 'views-row',
    ] %}
    <li{{ row.attributes.addClass(row_classes) }}>
        {{ row.content }}
    </li>
{% endfor %}
</ul>
```

Once finished, make sure to save the template, clear Drupal's cache, and then refresh the page in the browser to verify our Popular Posts block is now displayed as an unordered list. This next part will be a little trickier to accomplish, but will demonstrate that anything is possible with Twig templates.

Using Views-view templates

The main structure of views is contained within the `views-view.html.twig` template. We will need to modify this template to add some additional classes that will allow us to display each view within their own tab, as designed in the mockup. Like our previous view templates, the naming convention follows the same rules:

```
[base template name] -- [view machine name].html.twig
```

So in the case of our Popular Posts view, will want to create a new Twig template with the name of `views-view--popular-posts.html.twig` that we can then modify the markup to accomplish our tabbed design.

Begin by navigating to the `core/modules/view/templates` folder and follow these remaining steps:

1. Copy `views-view.html.twig` and place it in our `theme/octo/templates` folder.
2. Rename `views-view.html.twig` to `views-view--popular-posts.html.twig`.
3. Next, we will need to replace the current markup with the following new markup:

New markup

```
<div class="tabs">

    <ul class="nav nav-tabs">
        <li class="active">
            <a href="#popularPosts" data-toggle="tab">
                <i class="fa fa-star"></i> {{ 'Popular'|t }}
            </a>
        </li>
        <li>
            <a href="#recentPosts" data-toggle="tab">
                {{ 'Recent'|t }}
            </a>
        </li>
    </ul>

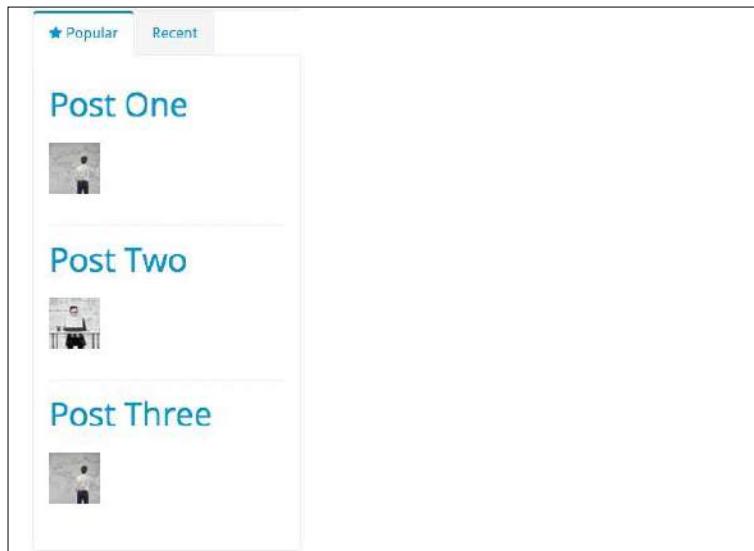
    <div class="tab-content">

        {%
            if rows %
                <div class="tab-pane active" id="popularPosts">
                    {{ rows }}
                </div>
            % elseif empty %
                <div class="view-empty">
                    {{ empty }}
                </div>
            % endif %
        %}
    
```

Theming Our Blog Listing Page

```
{% if footer %}  
  <div class="tab-pane" id="recentPosts">  
    {{ footer }}  
  </div>  
{% endif %}  
  
</div>  
  
</div>
```

Once finished, make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. Our Popular/Recent Posts block is now displaying in the tabbed interface, as shown in the following image:



By reviewing the markup, we can see that the way we constructed our view block allows for each independent view to be displayed in its own tab.

We are not quite done with our theming of this new block. As we can see from the page, our Post Teaser is missing some additional fields and formatting necessary to match our mockup. We will need to introduce another Twig template to handle the Teaser display mode and clean up our markup.

Creating a Post Teaser Twig template

Currently, the teaser display for our Post content type uses the default `node.html.twig` template. If we inspect the markup of our block, we can create a new Twig template with the recommended file name of `node--post--teaser.html.twig`.

Begin by navigating to the `core/modules/node/templates` folder and follow these remaining steps:

1. Copy `node.html.twig` and place it in our `theme/octo/templates` folder.
2. Rename `node.html.twig` to `node--post--teaser.html.twig`.
3. Next, we will need to replace the current markup with the following:

New markup

```
<div class="post-image">
  <div class="img-thumbnail">
    <a href="{{ url }}>
      {{ content.field_thumbnail }}
    </a>
  </div>
</div>

<div class="post-info">
  <a href="{{ url }}" class="tabbed-title">{{ label }}</a>
  <div class="post-meta">
    {{ node.createdtime|date('M d, Y') }}
  </div>
</div>

{{ content|without('field_thumbnail') }}
```

Theming Our Blog Listing Page

Once finished, make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page. As we can see from the following image, our tabbed interface is identical to our mockup:



The markup we added is pretty straightforward. We are utilizing the content variable that each node has available to print out the thumbnail image, title, and post created date. We used these same techniques when we created the Post Listing template earlier.

There are quite a few steps involved in creating the final tabbed interface, but the ease of being able to create Twig templates makes modifying the markup simple.

Adding the About Us block

After all the complex Views, Blocks, and Twig templates, adding the About Us block to our sidebar will seem quite simple. The last block to complete our Blog Listing page already exists, so adding it will just be an exercise in managing custom block layouts.

Begin by navigating to /admin/structure/block and follow these steps:

1. Locate the **Sidebar second** region.
2. Click on the **Place block** button.
3. Locate the **About Us** block.
4. Click on the **Place block** button.
5. Select the **Pages** tab under **Visibility**.

6. Enter the path `/blog` into the **Page** text field.
7. On a second line, add another path to `/blog/*`.
8. Make sure the **Show for the listed pages** checkbox is selected.
9. Click on the **Save block** button.

With our About Us block assigned to the Sidebar second region, we will want to make sure it is third in the block order. Reorder the blocks if necessary and then click on the **Save blocks** button. We have one final piece of theming before our About Us block is complete.

Implementing the About Us template

In the case of our About Us block, we need to adjust the heading to display it similar to our Categories block. This will require us to replace the current `<h2>` with a `<h4>` heading.

Begin by navigating to the `core/modules/block/templates` folder and follow these remaining steps:

1. Copy `block.html.twig` and place it in our `theme/octo/templates` folder.
2. Rename `block.html.twig` to `block--aboutus-2.html.twig`, based on the template suggestions.
3. Next, we will need to replace the current markup with the following new markup to convert the default `<h2>` to `<h4>`:

New markup

```
{% set classes = ['block'] %}

<div{{ attributes.addClass(classes) }}>
  {{ title_prefix }}

  {% if label %}
    <h4{{ title_attributes }}>{{ label }}</h4>
  {% endif %}

  {{ title_suffix }}

  {% block content %}
    {{ content }}
  {% endblock %}

</div>
```

Theming Our Blog Listing Page

Once finished, make sure to save the template, clear Drupal's cache, and then refresh the Blog listing page in the browser. Let's give ourselves a big pat on the back as our page is now complete:

The screenshot shows a Drupal blog listing page with a dark header containing the word "Blog". The main content area features a large image of a man in a white shirt and dark trousers standing and drawing a world map on a chalkboard. Below this image, a post card for "Post Three" by "Jan" is shown, with a preview of the post content: "Euismod atras vulputate iltricies etri elit. Class aptent taciti sodiosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisi, eu ornare mauris tincidunt vitae." Below the post card, there are links for "By admin", "Photos, Design", and "0 Comments", along with a "Read more" button.

Categories

- Design
- Lifestyle
- News
- Photos
- Videos

Recent

| Post | Date |
|------------|--------------|
| Post One | Jan 03, 2016 |
| Post Two | Jan 03, 2016 |
| Post Three | Jan 03, 2016 |

About Us

Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Nulla nunc dui, tristique in semper vel. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero.

Summary

Let's give ourselves a big pat on the back. We learned a lot of new techniques for theming Drupal 8 in this chapter and our Blog listing page looks great. Quite a few different techniques were covered in a short period of time. We adopted best practices for theming different sections of our page, which will be used in almost any theme we create. Let's take a moment to recap what we have accomplished in this chapter:

- We began by reviewing our Blog Listing mockup to identify the key areas of our website that we will need to recreate.
- We learned how to effectively use Display modes to manage our content types fields, including how to hide labels and use field formatters.
- Field level Twig templates came in handy for modifying individual field markup, adding classes, using filters, and checking for multiple field items.
- Twitter Bootstrap gave us the flexibility to add slideshows and tabbed interfaces to our page content.
- We took a deeper look at using preprocessing and creating a `*.theme` file to create new variables accessible by our Twig templates.

In the next chapter, we will continue with our Post content by theming our Blog detail page, which will include focusing on the Comment field in more detail, additional preprocessing functions, and additional Twig templates.

9

Theming Our Blog Detail Page

Having completed the Blog listing page, we now need to focus on the development and theming of our Blog detail page. While not quite as complex as creating a listing page, we will need to have a better understanding of how content types interact with comments. In this chapter, we will take a look at creating a single `node.html.twig` template that our Blog detail page will use. This template will be based on the Full Content display mode and the introduction of the revamped comment system in Drupal 8. Let's review what tasks we will be accomplishing:

- We will begin with reviewing our Blog detail page as displayed in our mockup, and identify how specific fields will need to be presented for display.
- Next, we will create our Blog detail template, which will focus on the default display mode for our Post content type.
- We will take a more detailed look at how comments work in Drupal 8 as we enable them the comment form, display them, and thread them for a post.
- Finally, we will take a look at how to use the profile images that users have attached to their accounts to display in our page.

While we work through each section, we have the ability to refer back to the Chapter09 exercise files folder. Each folder contains a `start` and `end` folder with files that we can use to compare our work when needed. This also includes database snapshots that will allow us all to start from the same point when working through various lessons.

Reviewing the Blog detail mockup

In order to assist us in identifying page elements we will be recreating for the Blog detail page, it would make sense to open up our mockup and review the layout and structure. The Blog page can be found in the `Mockup` folder located in our exercise files. Begin by opening up the `blog-detail.html` file within the browser, as shown in the following image:



The screenshot shows a blog detail page titled "Blog Detail". At the top, there's a large image of a man drawing a world map on a chalkboard. Below the title, the date "18 Oct" is displayed, followed by the title "Blog Detail" and author information "By admin News 0 Comments". The main content area contains a block of placeholder text (Lorem ipsum) and a "Read More" button. To the right, there's a sidebar with a "Categories" section listing "Design", "Lifestyle", "News", "Photos", and "Videos". Below that is a "Popular" posts section showing two entries: "Post Two" (May 23, 2015) and "Post One" (May 23, 2015), each with a small thumbnail image.

The Blog detail mockup looks very similar to the Blog listing page, with the exception of a few new areas that were not present before:

- First, we have replaced the teaser content with the full content of the post.
- Second, we now have a new section below our main content that lists any comment threads, with a photo of the comment's author.
- Third, we have a comment form that allows users to leave their name, a subject, and a comment for each post.

Having identified these three different components, we can now take a quick look at what our Blog detail page currently looks like and discuss the best way to tackle each of these requirements.

Previewing our Blog detail page

Navigate to one of the Blog detail pages by clicking on the Post title from the main blog page or simply entering /blog/post-one in the browser:

The mockup displays a blog detail page with the following components:

- Top Left:** A large image of a man in a white shirt and dark trousers standing in front of a chalkboard, drawing a world map.
- Bottom Left:** A smaller image of a man in a white shirt and tie writing the word "Idea" on a chalkboard, with a lit lightbulb above it.
- Top Right:** A sidebar titled "Categories" listing "Design", "Lifestyle", "News", "Photos", and "Videos".
- Bottom Right:** A sidebar titled "About Us" containing placeholder text: "Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Nulla nunc dui, tristique in semper vel. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero."
- Center:** A section showing three posts under the heading "Popular":
 - Post One (Jan 03, 2015)
 - Post Two (Jan 03, 2015)
 - Post Three (Jan 03, 2015)
- Bottom:** A footer note: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur lectus lacus, rutrum sit amet placerat et, bibendum nec mauris. Duis molestie, purus eget placerat viverra, nisi odio gravida sapien, congue tincidunt nisl ante nec tellus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Lorem"

We are in luck when it comes to our sidebar elements, as they are already positioned and themed the way they appear in the mockup. However, upon closer review we can see that we are missing some elements on our Blog detail page, or they are not themed the way we may have expected. These issues may include:

- Post date
- Post title
- Post tags properly themed
- Comment thread
- Comment form

Theming Our Blog Detail Page

The challenge for us is to think how Drupal outputs each of these sections and address them individually as we build our Twig templates. Since we have Twig debugging enabled, we can determine that we should start with creating a new `node--post--full.html.twig` template.

Creating a Post Full template

While we know what our new Twig template should be named, we should also consider just how similar the Blog detail page is to each Post displayed on our Blog listing page. In fact, the only real differences are that our Blog detail displays the full content of our Post along with the Comments.

So instead of creating a brand new template, we can begin by duplicating the `node--post--listing.html.twig` template located in our `themes/octo/templates` folder and rename it `node--post--full.html.twig`.

Make sure to clear Drupal's cache and refresh the Blog detail page.



A photograph of a man from behind, wearing a white shirt and dark trousers, standing at a chalkboard and drawing a world map with chalk. The chalkboard is light-colored and shows the outlines of all continents. Below the chalkboard, there is a screenshot of a blog post card. The card has a timestamp '03 Jan', the title 'Post One' in blue, and a date 'By admin'. It also shows 'Videos, Lifestyle' and '0 Comments' categories, and a 'Read more...' button. A small snippet of the post content is visible: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur lectus lacus, rutrum sit amet placerat et, bibendum nec mauris. Duis molestie, purus eget placerat viverra, nisi odio gravida sapien, congue'.

At first glance, we would think that most of our theming is completed for us. However, we actually have some components that need to be removed, such as the read more link and the teaser field. While some of these fields may not be displayed because they are being controlled from the Manager display admin, it is still good practice to remove these fields from our template.

Altering fields

We can begin by removing and replacing fields that we do not need. Begin by opening our Twig template, `node--post--full.html.twig`, and adjusting the markup in our post-content section:

New markup

```
<div class="post-content">
  {{ title_prefix }}
  <h2{{ title_attributes }}>
    <a href="{{ url }}" rel="bookmark">{{ label }}</a>
  </h2>
  {{ title_suffix }}

  <div class="post-meta">
    <span class="post-meta-user">
      <i class="fa fa-user"></i> By {{ author_name }}
    </span>

    <span class="post-meta-tag">
      <i class="fa fa-tag"></i> {{ content.field_tags }}
    </span>

    <span class="post-meta-comments">
      <i class="fa fa-comments"></i>
      <a href="{{ url }}/#comments">{{ comment_count }}</a>
    </span>
  </div>
</div>
```

Make sure to save the template, clear Drupal's cache, and refresh the Blog detail page. The read more link should now be gone and the page is starting to resemble our mockup:



One thing we may be asking ourselves is how is the full content of our Post being displayed if we are not printing it? We actually are printing it at the bottom of our template using the general `{{ content }}` variable. If we want to be more specific though, we can add the full content field directly to our template and exclude it from the general content variable by following these steps:

1. Open `node--post--full.html.twig`.
2. Modify the markup as shown here:

New markup

```
<div class="post-meta">
  ..existing markup..

  {{ content.field_full_content }}

</div>

{{ content|without('field_image', 'field_teaser', 'field_tags',
'field_full_content') }}

</article>
```

Make sure to save the template, clear Drupal's cache, and refresh the Blog detail page. We now have the first part of our blog detail page complete. What we are still missing though is the comment thread and comment form, which will allow users to interact with each post. Let's take a look at how to work with comments.

Working with comments

Drupal 8 introduced comments as a fieldable entity that can now be referenced by any other entity using the new comment field. So what exactly does that mean? This means you no longer manage comments as a configuration option from a content type. The benefit of moving comments into a fieldable entity is that it provides a wide range of flexibility. We can add additional fields if needed along with additional display modes to output comments.

In the case of our Post content type, we already created a relationship to comments to expedite our theming, but we should take a moment to review how that was done and then move on to printing comments in our Blog detail page.

Introducing Comment types

Comment types can be located by navigating to /admin/structure/comment and, as we can see by the interface in the following image, Comment types look very similar to Content types:

| COMMENT TYPE | DESCRIPTION | OPERATIONS |
|------------------|------------------------------|-------------------------------|
| Default comments | Allows commenting on content | Manage fields |

For our website, we are using the Default comments that Drupal creates as part of the default installation profile. It is just as simple to create additional Comment types the same way you would create any Content type. This comes in handy for providing multiple feedback mechanisms and provides us with various ways to display comments.

Reviewing Default Comment type fields and display

If we navigate to the **Manage fields** configuration by clicking on the **Manage fields** button, we will see that there is only a single text area field called Comment. This field allows the user to input the specific comment. We could easily add additional fields to capture more data, but this single field will suffice for our use.

| LABEL | MACHINE NAME | FIELD TYPE | OPERATIONS |
|---------|--------------|------------------------|----------------------|
| Comment | comment_body | Text (formatted, long) | Edit |

Theming Our Blog Detail Page

While this appears to be the only field, there are actually two additional default fields for author and subject that we may not be aware of until we navigate to the **Manage form display** screen by clicking on the **Manage form display** tab at the top of the page:

The screenshot shows the 'Manage form display' interface for a comment type. At the top, there are tabs: 'Edit', 'Manage fields', 'Manage form display' (which is selected), and 'Manage display'. Below the tabs, the breadcrumb navigation shows: Home » Administration » Structure » Comment types » Edit. There is also a link 'Show row weights'.

| FIELD | WIDGET | OPTIONS |
|---------|---------------------------|--------------------|
| Author | Visible | |
| Subject | Textfield | Textfield size: 60 |
| Comment | Text area (multiple rows) | Number of rows: 5 |

Below the table, there is a section labeled 'Disabled' with the note 'No field is hidden.' A blue 'Save' button is located at the bottom left.

One thing to note is that the **Manage form display** interface handles the display of the comment form and any of its fields that are displayed when a user looks to add a comment to a Post. These fields can be reordered, disabled, or modified as needed.

This differs from the **Manage display** interface, which controls the display of the comments thread that users see when viewing a post. We can navigate to the **Manage display** screen by clicking on its tab:

| FIELD | LABEL | FORMAT |
|---------|------------|---------|
| Comment | - Hidden - | Default |
| Links | | Visible |

No field is hidden.

CUSTOM DISPLAY SETTINGS

Save

A very important field to point out on the **Manage display** screen is the **Links** field. If this field is disabled for any Comment type, we will have no links attached to each comment thread that allow for replying, editing, or deleting a thread, based on a user's permissions. Now that we have oriented ourselves with the new Comment type, we will need to enable it for our Post type.

Enabling Post Type Comments field

Currently our Blog detail page is not displaying a comment form and therefore no way to display comment threads. If we navigate to /admin/structure/types/manage/post/display, we will be taken to the **Manage display** screen for our Post type. If we take a closer look, we can see that the **Comment** field is disabled for our **Default** display mode:

| FIELD | LABEL | FORMAT | |
|--------------|------------|------------|-------------------------------|
| Image | - Hidden - | Image | Original image |
| Tags | - Hidden - | Label | Link to the referenced entity |
| Full content | - Hidden - | Default | |
| Disabled | | | |
| Links | | - Hidden - | |
| Thumbnail | Above | - Hidden - | |
| Teaser | Above | - Hidden - | |
| Comments | Above | - Hidden - | |

We can remedy that by dragging the **Comments** field out of the **Disabled** section and placing it directly under the **Full content** field:

| FIELD | LABEL | FORMAT | |
|--------------|------------|--------------|-------------------------------|
| Image | - Hidden - | Image | Original image |
| Tags | - Hidden - | Label | Link to the referenced entity |
| Full content | - Hidden - | Default | |
| Comments | Above | Comment list | |

Once complete, click the **Save** button. If we navigate back to /blog/post-one, we will see the default appearance of our Comment form:

Add new comment

Subject

Comment

B I Format Source

Text format Basic HTML [About text formats](#)

- Allowed HTML tags: <a href hreflang> <cite> <blockquote cite> <code> <ul type> <ol start type> <dl> <dt> <dd> <h2 id> <h3 id> <h4 id> <h5 id> <h6 id>
- Lines and paragraphs break automatically.
- Web page addresses and email addresses turn into links automatically.

Save

One thing still missing is some actual comment threads, so let's take a moment and add a few comments by filling in the comment form for Post one. Make sure to fill in both the **Subject** and **Comment** fields. Once we hit the **Save** button, you should now see a new comment thread displayed directly above the comment form, as shown in the following screenshot:

Comments

new
Submitted by admin on Thu, 01/28/2016 - 19:01pm

[Permalink](#)

SUBJECT ONE

Comment one

- Delete
- Edit
- Reply

A general rule of theming comments is that when someone replies to an existing comment, the reply is displayed directly below the original comment and indented so that you have a visual clue as to the thread developing. So that we can see what an actual thread looks like, let's reply to our first comment by clicking on the **Reply** link, filling in the required fields, and hitting the **Save** button. Our **Comments** section should now contain a comment thread:

The screenshot shows a comment section with the following structure:

- Comments**
 - Submitted by admin on Thu, 01/28/2016 - 19:01pm
 - [Permalink](#)
 - SUBJECT ONE**
 - Comment one
 - Delete
 - Edit
 - Reply
 - new**
 - Submitted by admin on Thu, 01/28/2016 - 19:04pm
 - [Permalink](#)
 - REPLY TO SUBJECT ONE**
 - Reply message
 - Delete
 - Edit
 - Reply

Perfect, we now have almost everything we need in place to theme the Comment section of our Blog detail page. We have a **Comments** thread, nested comments, and an **Add new comment** form. We will address each of these components individually in order to implement our required markup to match our mockup.

Creating a Field Comments template

Just like any other field attached to a content type, this has a corresponding field template that Twig uses to output the content. If we inspect the markup of our Comment section, we can determine which template is being used and where it is located:

```
<!-- THEME DEBUG -->
<!-- THEME HOOK: 'field' -->
<!-- FILE NAME SUGGESTIONS:
  * field--node--field-comments--post.html.twig
  * field--node--field-comments.html.twig
  * field--node--post.html.twig
  * field--field--comments.html.twig
  * field--comment.html.twig
  * field.html.twig
-->
<!-- BEGIN OUTPUT from 'core/modules/comment/templates/field--comment.html.twig' -->
► <section data-quickeedit-field-id="node/11/field_comments/en/full">...</section>
<!-- END OUTPUT from 'core/modules/comment/templates/field--comment.html.twig' -->
```

Using **FILE NAME SUGGESTIONS**, we can navigate to the `core/modules/comment/templates` folder and copy the `field--comment.html.twig` template to our `themes/octo/templates` folder. Next, we will need to replace the markup within our template with the following new markup:

New markup

```
<section id="comments" class="post-block post-comments">

  {%- if comments and not label_hidden %}
    {{ title_prefix }}
    <h3{{ title_attributes }}>
      <i class="fa fa-comments"></i>{{ label }}
    </h3>
    {{ title_suffix }}
  {%- endif %}

  {{ comments }}

  {%- if comment_form %}
    <div class="post-block post-leave-comment">
      <h3{{ content_attributes }}>{{ 'Leave a comment'|t }}</h3>
      {{ comment_form }}
    </div>
  {%- endif %}

</section>
```

Make sure to save the template, clear Drupal's cache, and refresh the Blog detail page for Post One. The formatting we just applied adds a few classes to our comments wrapper, as well as adding a Font Awesome icon to the Comments heading. We should also notice that the `field--comment.html.twig` template really breaks the entire comment block into three distinct regions:

- The Comment heading, indicated by the `{{ label }}` variable
- The Comment thread, indicated by the `{{ comments }}` variable
- The Comment form, indicated by the `{{ comment_form }}` variable

Now that we have identified the three key pieces of a comment block, we need to focus on the `{{ comments }}` variable itself as it contains our comment thread. Currently, our thread is not displaying as we would like. Each thread is hard to differentiate where it begins and ends, and we are missing the styling that would help it match our mockup. To remedy this, we can take advantage of another Twig template.

Theming the Comment thread

Drupal only provides two Twig templates for outputting the Comment section of our page. We have already addressed the Comment field template, so all that is left for us to target is the `comment.html.twig` template. It is this template that contains the markup for the thread that displays, and we can modify the markup to display the content exactly how we need it by following these steps.

Using the **FILE NAME SUGGESTIONS**, we can navigate to the `core/modules/comment/templates` folder and copy the `comment.html.twig` template to our `themes/octo/templates` folder. Next, we will need to replace the markup with our own, making sure to print the existing variables:

New markup

```
<article{{ attributes }}>

  <div class="comment">
    {{ user_picture }}

    <div class="comment-block">
      <div class="comment-arrow"></div>

      <div class="comment-by">
        <strong><span>{{ author }}</span></strong>
        <span class="pull-right">
          {{ content.links }}
        </span>
      </div>

      <div class="comment-content">
        {{ content.comment_body }}
      </div>

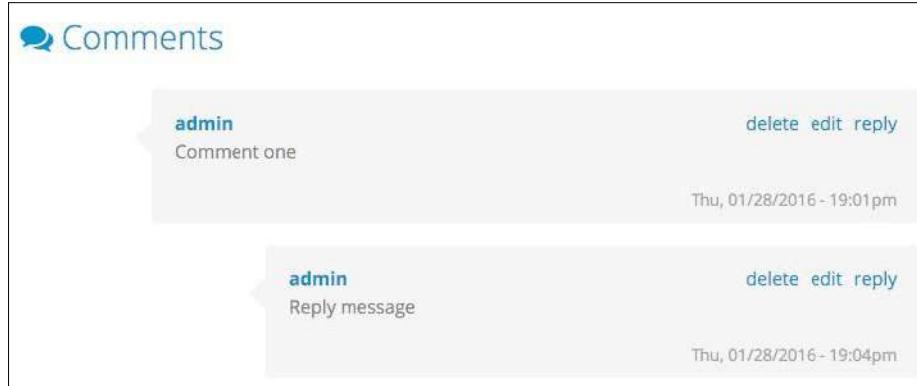
      <div class="comment-date">
        <span class="date pull-right">{{ created }}</span>
      </div>
    </div>

  </div>

  {%
    if parent %}
    <p class="visually-hidden">{{ parent }}</p>
  {%
    endif %}

</article>
```

Make sure to save the template, clear Drupal's cache, and then reload the Blog detail page in the browser. Our Comments thread should now be styled and indented as shown in the following image:



Obviously there is something still missing. The user image associated with the author of each comment is not being displayed. Also, our comment date is formatted using military time, which the majority of our users may not be familiar with. We can address both of these items fairly easily.

Enabling user photos for Comment threads

Right now, we have comment threads lacking a photo to identify the user that posted the comment. In most cases, this is due to a user not having uploaded a photo to their Drupal profile. In order for us to have an image for the {{ user_picture }} variable to print, we will need to upload our own photo.

Navigate to /user/1/edit, which will take us to our current profile page, keeping in mind that the user ID may be different, depending on how many users are in Drupal:

1. Locate the **Picture** field.
2. Click on the **Choose File** button.
3. Locate a photo or any image we want to represent ourselves, select it and then click on the **Open** button in the File dialog window.

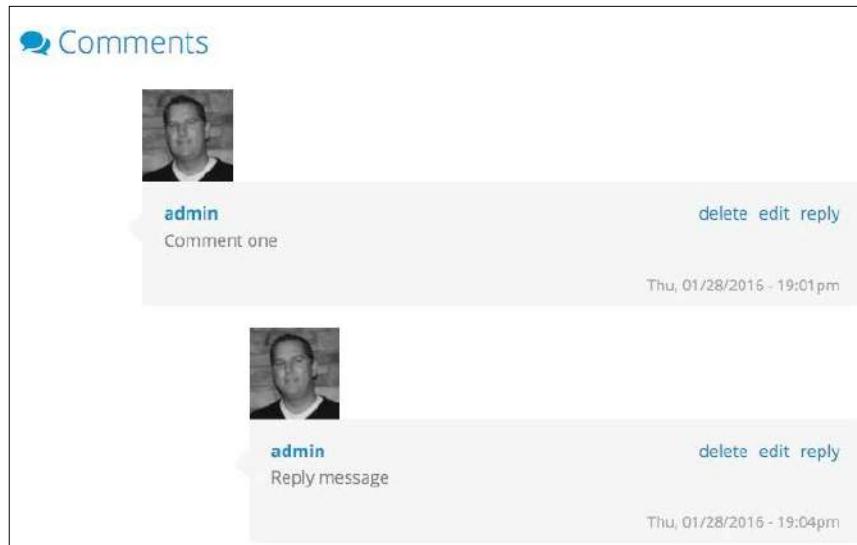
Theming Our Blog Detail Page

4. We should now see a thumbnail image:



5. Click the **Save** button to complete this.

If we navigate back to the Blog detail page located at /blog/post-one, we will see the user photo displayed next to each comment:



Now that we have the user image being displayed, we still need to clean up the markup by formatting the field template.

Cleaning up the User Picture field

If we inspect the markup for the user picture field, we will notice an `<article>` element wrapped around each image. Currently, the `<article>` element is respecting the block display and causing the user image to display on a separate line. While we could easily adjust the CSS to resolve this, we want to respect the HTML provided in the mockup.

To bring our User Picture back in line with how it displays in our mockup, we will need to first modify the `user.html.twig` template.

Begin by navigating to the `core/modules/user/templates` folder and copying the `user.html.twig` template to our `themes/octo/templates` folder. Next, we will replace the current markup with the following:

New markup

```
{% if content %}  
  {{- content -}}  
{% endif %}
```

Make sure to save the template, clear Drupal's cache, and then refresh the page in the browser. You will not see a visual change just yet, but the surrounding `article` element is now removed.

Creating the Field User Picture template

We will need to modify an additional Twig template that outputs the image itself. We will be borrowing the markup and styling from an earlier template so that our user picture is framed similar to our Post image.

1. Begin by navigating to the `core/modules/system/templates` folder and copying the `field.html.twig` template to our `themes/octo/templates` folder.
2. Next, we will rename `field.html.twig` to `field--user-picture.html.twig`, based on the recommended file name suggestions.
3. Finally, we will replace the markup with the following:

New markup

```
{% for item in items %}  
  <div class="img-thumbnail">  
    <div class="user-picture">  
      {{ item.content }}  
    </div>  
  </div>  
{% endfor %}
```

Theming Our Blog Detail Page

Make sure to save the template, clear Drupal's cache, and then refresh the page in the browser. Our comment threads are perfectly styled, with the user image for each thread aligned next to each other.



At this point, we only have one more item to adjust for our comment thread. Currently, the default date for each comment is displayed in a 24-hour format, often referred to as military time.

Date and time formats

There are multiple ways to address date and time formats within Drupal. In *Chapter 8, Theming Our Blog Listing Page*, we worked with Twig filters and field variables to format the date. However, the comment thread date is already formatted for us using the default medium date format. So where would we need to configure this to display a non-24-hour format?

If we navigate to `/admin/configuration/region/date-time`, we will get a glimpse of the date and time formats that Drupal configures for us:

| Date and time formats | | |
|---------------------------------------------------------------|--------------------------------------|----------------------|
| Home » Administration » Configuration » Regional and language | | |
| + Add format | | |
| NAME | PATTERN | OPERATIONS |
| Default long date | Thursday, February 4, 2016 – 18:41pm | Edit |
| Default medium date | Thu, 02/04/2016 – 18:41pm | Edit |
| Default short date | 02/04/2016 | Edit |

Each of the date and time formats that are shown can be used within Drupal by either managing a Content types field format or referencing the name with a Twig template or preprocessing function. As we can see, some of the formats can be edited, while others cannot. We can also add additional formats.

In the case of our comment thread created date, we can assume by reviewing the **PATTERN** of **Default medium date** that this is the format being used. Luckily for us, we can easily modify the pattern to change the time from 24-hour to 12-hour format by following these steps:

1. Click on the **Edit button** for the **Default medium date** format.
2. Replace **Format string** with the following pattern:
D, m/d/Y - h:ia
3. Click on the **Save format** button.

We have successfully changed the default medium date format. One thing to note is that the date and time formats in Drupal take advantage of the `Date` object in PHP. We can get additional information regarding various formats by reviewing the parameters available in the manual at <http://php.net/manual/en/function.date.php>.

If we now navigate back to our Blog detail page for Post One, we will see that our comment threads now use the modified default medium date format:



Our Blog detail page is just about complete. While users can now read our Posts and comment on them, we are still missing the ability for users to share a post on their favorite social networks.

Implementing social sharing capabilities

Social networks such as Facebook, Twitter, and Pinterest provide another medium for content to be shared with family, friends, and coworkers. Most websites provide a mechanism for sharing content, and our Blog detail page is no different.

Based on our mockup, we allow users to share a post as well as see the number of likes, tweets, or pins. In fact, there are a number of different third-party libraries or APIs that make this functionality easy to implement. Services such as Share This, <http://www.sharethis.com/>, or even Add This, <https://www.addthis.com/>, provide either a library or contributed modules to implement this functionality within Drupal.

The Add This buttons

For our particular page, we will be using the Add This service. There are various button options and configurations that can be created, so to avoid any confusion with adding this service to our template, we will be using the standard buttons. The implementation of the Add This button requires each of us to have created a free account. However, for demonstration purposes, we will be using my account. Please remember to replace the `pubid` with yours once an account has been created.

The process of adding the Add This library to our Twig template requires us to configure the type of social sharing buttons we want to use, copy the JavaScript to our page, and then add specific markup that will enable the display of the buttons:

Basic Code

Copy and paste the following code before the `</body>` tag in the HTML of your website on every page you want AddThis to work.

```
<!-- Go to www.addthis.com/dashboard to customize your tools -->
<script type="text/javascript"
src="//s7.addthis.com/js/300/addthis_widget.js#pubid=chazchumley" async="async">
</script>
```

Code for Activated Tools

Some of the tools you have activated need additional code to show within the body of your page. Copy and paste the corresponding code where you want the tools to show.

Original Sharing Buttons



```
<!-- Go to www.addthis.com/dashboard to customize your tools -->
<div class="addthis_native_toolbox"></div>
```

[928]

The basic code displayed above is a simple JavaScript block that needs to be placed within our webpage. We will be using our themes `octo.libraries.yml` file to configure this block and then using the `{{ attach_library }}` function to add it to our Blog detail page.

Creating a library entry

Begin by opening `octo.libraries.yml` located in our `themes/octo` folder. We will then add the following metadata to the bottom of our file:

```
add-this:  
  version: VERSION  
  js:  
    //s7.addthis.com/js/300/addthis_widget.js#pubid=chazchumley: {  
  type: external, asynch: asynch }
```

Note that the JavaScript path is all on a single line. In the above metadata, we are pointing to an external script and are also adding a new parameter for calling the script asynchronously. Also, please remember to replace the `pubid` value with your Add This username.

Once we have added the metadata to our file, we can save `octo.libraries.yml` and then clear Drupal's cache.

Attaching the library to our Blog detail page

Now that Drupal has knowledge of our new library, we can attach it to our Blog detail page by following these steps:

1. Open `node--post--full.html.twig`.
2. Add the following Twig function directly below where we are referencing the owl-carousel:
 `{{ attach_library('octo/add-this') }}`
3. Save `node--post--full.html.twig`.

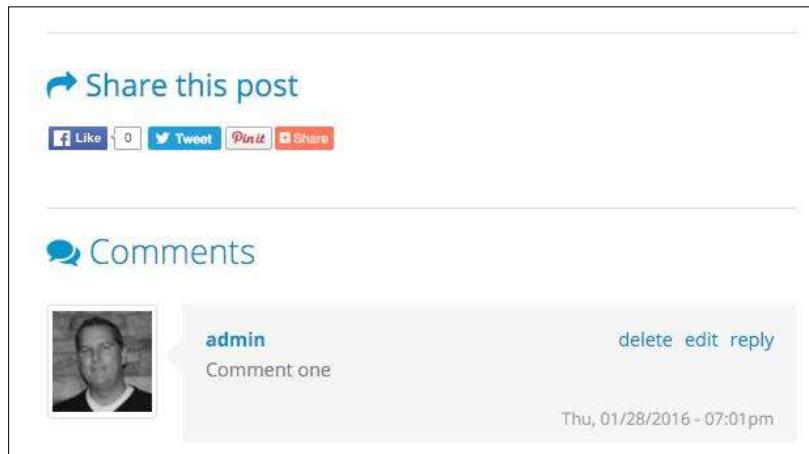
Make sure to clear Drupal's cache and then refresh the Blog detail page. We now need to add the markup required to display the sharing buttons on our page.

Displaying buttons

We will be adding some additional markup around the required div so that our "Share this post" section matches our mockup. With `node--post--full.html.twig` still open, we will add the following markup directly after the `{{ content.field_full_content }}` variable:

```
<div class="post-share">
  <h3><i class="fa fa-share"></i> Share this post</h3>
  <div class="addthis_native_toolbox"></div>
</div>
```

Make sure to clear Drupal's cache and then refresh the Blog detail page. If all the steps were completed successfully, we should see our new **Share this post** section displayed:



Congratulations, we have now completed the theming of our Blog detail page. Everything has been configured, styled, and modified to match our mockup.

Summary

While we revisited some common theming techniques, we also learned a few new ones. Slowly and methodically dissecting each section of our mockup, we walked through creating Twig templates, and worked with the new Comment field and a lot more to create a fully functional Blog detail page. We definitely covered a lot of material, so let's review everything we covered before moving on to the next chapter:

- We started by reviewing the Blog Detail mockup to identify key areas of our website that we would need to recreate.
- We familiarized ourselves with additional node templates and learned how to theme the default display of our Post content type.
- We dug a little deeper into the new Comment type and learned how to best manage the various Twig templates it provides. This included managing user profile pictures for each comment thread and configuring date and time formats for the comment created date.
- Finally, we implemented social sharing buttons using our themes `octo.libraries.yml` file, attaching the Add This library to our Blog detail page, and adding the required markup for our buttons to display properly.

In the next chapter, we will take a look at the new contact forms that are in the Drupal 8 core. We will add a default form to our page that users can interact with, and learn how to add a custom block to a Drupal-generated page. Even more excitingly, we will take a look at adding a Google Maps block to our page that provides a dynamic map with a map marker.

10

Theming Our Contact Page

Almost every website provides a mechanism for users to contact the individual, business, or association that owns the site, whether that be in the form of a simple e-mail link or something more advanced using a web form. Often, a contact page is part of the main menu hierarchy, as is evident in our mockup.

In this chapter, we will take a look at creating a contact page that uses the new contact forms that are part of Drupal 8 core. We will not be using any contributed modules, as core provides us with the configuration and templates needed to create most forms. We will also not be covering the extensive Form API, as it is beyond the scope of this module.

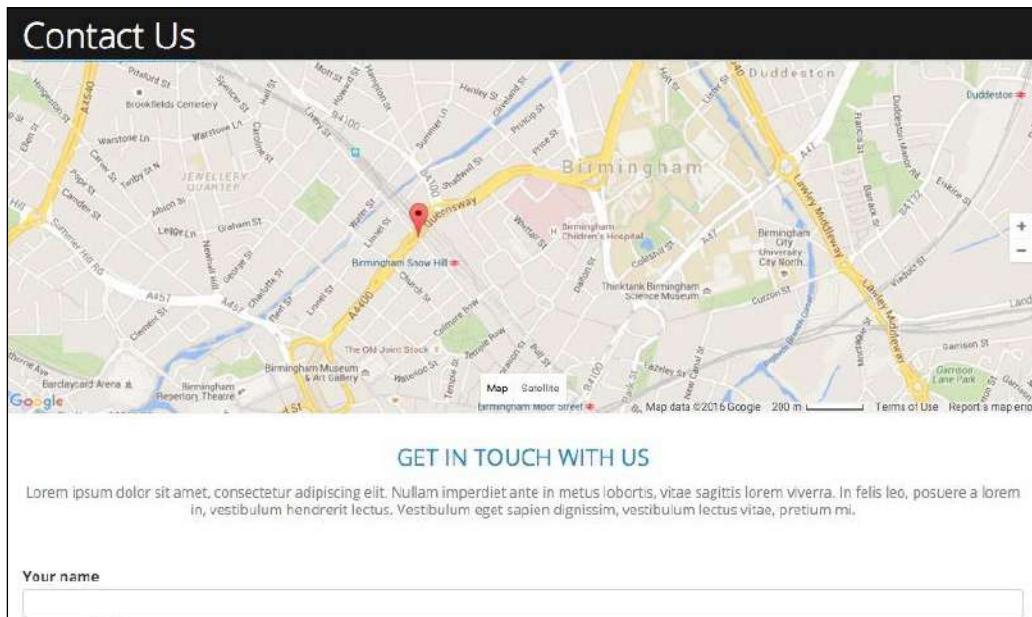
However, we will be learning the following theming techniques that will allow us to create a well-structured contact page:

- We will begin with reviewing the mockup of our contact page and identify how specific blocks or fields will need to be presented for display.
- Next, we will take a closer look at contact forms in Drupal and how to modify an existing form that we can use for our contact page.
- As Drupal creates the page that all contact forms utilize, we will make use of alternative regions within our theme to add additional content to our contact page.
- Finally, we will see how simple it is to add a Google map while working some more with libraries and Twig.

While we work through each section, we have the ability to refer back to the Chapter10 exercise files folder. Each folder contains start and end folders with files that we can use to compare our work when needed. This also includes database snapshots that will allow us to all start from the same point when working through various lessons.

Reviewing the contact page mockup

Like previous sections of our website, having a mockup to review makes planning how to develop a page much easier. Page structure, blocks, web forms, and other functionality we will need to consider can easily be discovered by looking at the contact page in the Mockup folder located in our exercise files. Begin by opening up the `contact.html` file within the browser.



The contact page mockup has some fairly standard components that most websites seem to use today. Starting at the top of our layout and working our way down, we can identify three different sections that we will need to develop and theme for our Drupal site.

1. First, we have a Google map displaying the current address using a map marker. We will revisit building this section of the site after we have created our contact form.
2. Second, we have a simple block of information or callout telling users how they can get in touch with us.
3. The last section is the web form itself and is by far the most important component of the contact page.

Having identified these three different components, we will focus on the most important and most detailed piece of functionality first—the contact form.

Introducing contact forms

Contact forms in Drupal 8 have taken some of the characteristics of previously used contributed modules such as WebForm and placed the most common functionality into core. This new core feature allows us to create any type of form needed for users to be able to contact us. We can see the basic implementation of a contact form by navigating to /contact within our site.

Your name admin
 Your email address cbchumley@gmail.com
Subject
 Please enter Subject
Message
 Please enter message

Send message | Preview

By default, Drupal creates a contact form that contains fields for name, e-mail, subject, and message. As part of the database snapshot, we have a contact page already created for us. However, we can easily add additional fields and manage the display and format of fields just like we can with content types.

To get a better understanding of how contact forms work, we can navigate to the **Contact forms** admin by entering /admin/structure/contact within our browser.

| FORM | RECIPIENTS | SELECTED | OPERATIONS |
|-----------------------|---------------------|----------|------------------------------------------------------|
| Contact Us | cbchumley@gmail.com | Yes | Edit Delete |
| Personal contact form | Selected user | No | Manage fields Delete |

We can see that two forms have already been configured:

- **Personal contact form** that each user of our site will receive.
- **Contact Us** (renamed from the default of **Website feedback**) that our global contact page will be using.

Editing a contact form

Contact forms can be managed similar to how we work with content types and blocks. Contact forms are also fieldable and can have additional fields added to them to capture a variety of information. We can get a closer look at how our form can be configured by clicking on the **Edit** button to the right of the **Contact Us** form.

The screenshot shows the 'Edit contact form' page for the 'Contact Us' form. At the top, there are four tabs: 'Edit' (selected), 'Manage fields', 'Manage form display', and 'Manage display'. Below the tabs, the breadcrumb navigation shows 'Home > Administration > Structure > Contact forms'. The main configuration area includes fields for 'Label' (set to 'Contact Us'), 'Machine name' (set to 'feedback'), 'Recipients' (set to 'chaz@packtpub.com'), and an optional 'Auto-reply' message. There is also a 'Weight' dropdown set to '0'. A note states: 'When listing forms, those with lighter (smaller) weights get listed before forms with heavier (larger) weights. Forms with equal weights are sorted alphabetically.' A checkbox labeled 'Make this the default form' is checked. At the bottom, there are 'Save' and 'Delete' buttons.

The Edit screen consists of several fields vital to a working contact form:

- A **Label** that identifies which form we are creating
- **Recipients**, which contains a required list of e-mail addresses that we would like the web form and its content to be sent to
- **Auto-reply**, which contains an optional message that we want to send users after they have submitted the form
- The **Weight** field, used to simply sort multiple forms on the contact form's admin page
- The **Make this the default form** checkbox, which designates which form to use as the default contact form

One thing to note is that the default **Website feedback** form has been renamed to **Contact Us** using the **Label** field. If for some reason the database snapshot has not been imported at this time, you will see the default form instead.

Whenever we create a new contact form, the machine name provided by the **Label** field is used to generate the predefined URL of /contact/machine-name. In the case of our Contact Us form, we were able to navigate to the form by entering /contact or /contact/feedback. Keep in mind that we cannot modify the machine name once we have entered a label and saved the form.

Managing form fields

Our Contact Us form is not using any additional fields. However, if we wanted to add any, the functionality is identical to how fields are added to content types or blocks using the Field UI.

| LABEL | MACHINE NAME | FIELD TYPE | OPERATIONS |
|----------------------------|--------------|------------|------------|
| No fields are present yet. | | | |

Managing form display

Any time we create a new contact form, there are five fields that are created by default that Drupal requires to handle functionality behind the scenes. Those five fields can be seen on the **Manage form display** screen.

The screenshot shows the 'Manage form display' interface for a contact form. At the top, there are tabs: 'Edit', 'Manage fields', 'Manage form display' (which is selected), and 'Manage display'. Below the tabs, the breadcrumb navigation shows: Home > Administration > Structure > Contact forms > Edit contact form. There is also a link 'Show row weights' on the right. The main area is titled 'FIELD' and 'WIDGET'. It lists five fields: 'Sender name' (Visible dropdown), 'Sender email' (Visible dropdown), 'Subject' (Textfield dropdown with placeholder 'Please enter Subject'), 'Message' (Text area (multiple rows) dropdown with placeholder 'Please enter message'), and 'Send copy to sender' (Hidden dropdown). On the right side of the 'Subject' and 'Message' rows are gear icons for configuration. A 'Disabled' section is shown below, containing the 'Send copy to sender' field. At the bottom left is a 'Save' button.

All contact forms consist of the following fields:

- **Sender name** – an input to collect the user's name
- **Sender email** – an input to collect the user's e-mail
- **Subject** – an input for the subject line of the form
- **Message** – a text area to collect the message or content of the form
- **Send copy to sender** – a checkbox to allow the user to receive a copy of the submitted form

For our contact form, we have chosen to disable the **Send copy to sender** control. This means that when a user submits the form, they will not receive a copy of their submission.

As we can see from the **Manage form display** screen, we have all the flexibility to enable, disable, and format our fields as needed.

Now that we have a better understanding of contact forms, let's navigate back to our default contact page located at /contact and discuss how we will begin to lay out the remaining components.

Contact page layout

So far, we have been working with mainly content types and blocks. Content created using any of our content types generates a Node and a Twig template with it. However, contact forms generate a page for us that is not quite like what we are used to working with. The only way for us to add additional content such as our Callout block or Google map is by using blocks. This requires us to rethink the layout of the Contact page a little.

We can begin by inspecting the markup to see what Twig templates Drupal is providing us.

```
<!-- BEGIN OUTPUT from 'core/modules/block/templates/block.html.twig' -->
▼ <div id="block-octo-content">
  <!-- THEME DEBUG -->
  <!-- THEME HOOK: 'form' -->
  <!-- BEGIN OUTPUT from 'core/modules/system/templates/form.html.twig' -->
  ▼ <form class="contact-message-feedback-form contact-message-form contact-form" data-drupal-selector="contact-message">
    <!-- THEME DEBUG -->
    <!-- THEME HOOK: 'form_element' -->
    <!-- BEGIN OUTPUT from 'core/modules/system/templates/form-element.html.twig' -->
    ▶ <div id="edit-name" class="js-form-item form-item js-form-type-item form-item-name js-form-item-name">...</div>
    <!-- END OUTPUT from 'core/modules/system/templates/form-element.html.twig' -->
```

It appears that the Contact Us form is output as a form element and is assigned to our Main content region. This means that we can add additional content both above and below the form using the Before Content and After Content regions. In fact, this is a perfect example of why creating regions in our design that can appear above or below the main content flow provides flexibility.

Adding a Callout block

We will take advantage of the Before Content region we created in our theme's configuration file to add our next component. The Callout block we identified in our mockup earlier allows us to add additional information that helps introduce our Contact form.

If we quickly review the `contact.html` page from the `Mockup` folder, we can identify that we will need to create a custom block that consists of a heading and a paragraph.



Theming Our Contact Page

This is a pretty simple block to create, so let's get started by navigating to /admin/structure/block, which will take us to the **Block layout** admin.

Next, we will follow these steps:

1. Click on the **Place block** button in the **Before Content** region.
2. Click on the **Add custom block** button.
3. Enter a **Block description** of Contact Callout.
4. Select **HTML No Editor** from the **Text format** dropdown.
5. Add the markup located in the Chatper10/start/content/ContactCallout.txt file to the **Body** field, as shown in the following image:



The screenshot shows the 'Add custom block' interface. At the top, there is a 'Block description' field containing 'Contact Callout'. Below it is a 'Body' field containing the following HTML code:

```
<h3>Get in touch with us</h3>
<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam imperdiet ante in metus lobortis, vitae sagittis lorem viverra. In felis leo, posuere a lorem in, vestibulum hendrerit lectus. Vestibulum eget sapien dignissim, vestibulum lectus vitae, pretium mi.</p>
```

At the bottom left, the 'Text format' dropdown is set to 'HTML No Editor'. On the right, there is a link 'About text formats'.

6. Click on the **Save** button to proceed to the **Configure block** screen.
7. Uncheck the **Display title** checkbox.
8. Select the **Pages** vertical tab within the **Visibility** section.
9. Enter a value of /contact in the **Pages** text area.
10. Choose **Show for the listed pages** under **Negate the condition**.
11. Choose **Before Content** from the **Region** field.
12. Click on the **Save block** button.

We now have our Contact Callout block created and assigned to our Before Content region. Let's make sure that our new block is displaying correctly by navigating back to our Contact page.

GET IN TOUCH WITH US

Lore ipsum dolor sit amet, consectetur adipiscing elit. Nullam imperdiet ante in metus lobortis, vitae sagittis lorem viverra. In felis leo, posuere a lorem in, vestibulum hendrerit lectus. Vestibulum eget sapien dignissim, vestibulum lectus vitae, pretium mi.

Therefore, our content is displaying in the correct region but not quite visually what we were expecting. However, like any other block, we can remedy this by creating a Twig template and apply any additional markup or classes that may be needed.

Creating the Callout block template

Using the **FILE NAME SUGGESTIONS**, we can navigate to the `core/modules/block/templates` folder and copy the `block.html.twig` template to our `themes/octo/templates` folder. Next, we will need to rename the template to `block--contactcallout.html.twig` and then replace the markup with the following:

New markup

```
{% set classes = ['block', 'contact-intro'] %}

<div{{ attributes.addClass(classes) }}>
  <div class="container">
    {{ title_prefix }}
    {% if label %}
      <h4{{ title_attributes }}>{{ label }}</h4>
    {% endif %}
    {{ title_suffix }}
    {% block content %}
      {{ content }}
    {% endblock %}
  </div>
</div>
```

Make sure to save the template, clear Drupal's cache, and refresh the Contact page in the browser. Our block should now look exactly like our mockup. Now we will move on to our next component, which involves integrating Google Maps.

Integrating Google Maps into our contact page

The Google Maps API provides developers with the flexibility to add interactive mapping functionality to any website. With our Contact page, we are implementing a map that provides a map marker pointing to a specific location based on the latitude and longitude that we will provide. As we implement this functionality, it is important to note that we will not be covering the in-depth details required to create a Google map or work with the developer API found at <https://developers.google.com/maps/tutorials/fundamentals/adding-a-google-map>.

Instead, we will take advantage of one of the many libraries that simplify the JavaScript knowledge required. For our next lesson, we have chosen to work with the `jQuery-gMap` plugin, which can be found at <https://github.com/marioestrada/jquery-gMap>.

In order to implement our map, we will need to follow a series of steps that involve making sure that Drupal can locate the `jquery-gMap` library, create a library entry with any dependencies, create our custom block, and finally attach the library to our template.

Because the JavaScript to initialize Google Maps is quite long, we have broken the script into its own file. This allows us to keep specific functionality organized better for implementation. Let's get started by configuring the necessary files.

Configure Google Maps

In order to configure the Google Maps library, we will be adding two different library entries to our `octo.libraries.yml` file. The first will be pointing to our custom map script. The second will be pointing to the `jquery-gmap` library, which includes an external reference of the Google Maps API.

Before we get started, let's ensure that we have the proper files copied to our theme. Begin by navigating to the `Chapter10/start/themes/octo/vendor` folder and copy the `jquery-gmap` folder to our `themes/octo/vendor` folder. Next, copy the `map.js` file from the `Chapter10/start/themes/octo/js` folder to our `themes/octo/js` folder.

With the two files now accessible by Drupal, we can add the library entries:

1. Open `octo.libraries.yml`.
2. Add the following entry:

```
map:  
  version: VERSION  
  js:  
    js/map: {}  
  dependencies:  
    - octo/jquery-gmap
```

Note the dependency to `jquery-gmap`; we will add that entry directly below the `jquery-gmap` entry.

3. Add the following entry:

```
jquery-gmap:  
  version: 2.1.5  
  js:  
    vendor/jquery-gmap/jquery.gmap.min.js: {}  
    //maps.google.com/maps/api/js?sensor=true: { type: external }  
  dependencies:  
    - core/jquery
```

Make sure to save `octo.libraries.yml` and clear Drupal's cache to ensure that our new library entries are added to the theme registry. To recap, we added two library entries that will allow us to enable Google Maps.

The first entry points to our custom JavaScript that initializes the map and then looks for any markup within our page that has an ID of `#map`, and renders a map. The first entry also has a dependency of `jquery-gmap`, which we have added.

The second entry points to our vendor library, which simplifies the creation of Google Maps, and because it obviously requires both jQuery and the Google Maps API, we add those to our entry.

With Google Maps now configured, we will need to create a new block and add the required markup that will render our map.

Creating our Google Maps block

We will take advantage of the Before Content region to add our Contact Map block. Being by navigating to /admin/structure/block and follow these steps.

1. Click on the **Place block** button in the **Before Content** region:
2. Click on the **Add custom block** button.
3. Enter a **Block description** of Contact Map.
4. Select **HTML No Editor** from the **Text format** dropdown.
5. Add the markup located in the chapter10/start/content/ContactMap.txt file to the **Body** field, as shown in the following image:



The markup we are adding introduces data attributes, which allow us to better describe the element being displayed while storing extra information. For example, we are adding data attributes for latitude and longitude that our custom script uses to render our map.

1. Click on the **Save** button to proceed to the **Configure block** screen.
2. Uncheck the **Display title** checkbox.
3. Select the **Pages** vertical tab within the **Visibility** section.
4. Enter a value of `/contact` in the **Pages** text area.
5. Choose **Show for the listed pages** under **Negate the condition**.
6. Choose **Before Content** from the **Region** field.
7. Click on the **Save block** button.

Once the block has been saved, make sure that the order of our Blocks with the Before Content region has the Contact Map displaying before our Contact Callout block.



Now that we have our Contact Map block created and assigned to the correct region, we will need to create a Twig template that will allow us to attach the library entry to it.

Creating the Callout Map template

If we navigate back to our Contact page, we will see the outline for our map represented by a gray box. Our markup is actually being output correctly, but we do not yet have a reference to the Google Maps script that renders the map. We can remedy this by using the **FILE NAME SUGGESTIONS** to create a Twig template for our block.

Navigate to the `core/modules/block/templates` folder and copy the `block.html.twig` template to our `themes/octo/templates` folder. Next, we will need to rename the template to `block--contactmap.html.twig` and then replace the markup with the following:

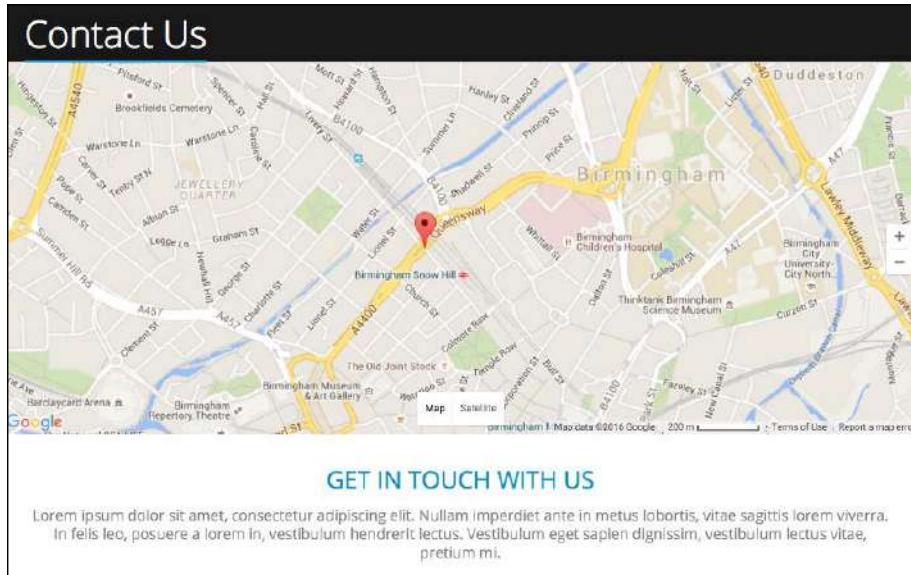
New markup

```

{{ attach_library('octo/map') }}

<div{{ attributes.addClass(classes) }}>
  {{ title_prefix }}
  {% if label %}
    <h4{{ title_attributes }}>{{ label }}</h4>
  {% endif %}
  {{ title_suffix }}
  {% block content %}
    {{ content }}
  {% endblock %}
</div>
```

Make sure to save the template, clear Drupal's cache, and refresh the Contact page in the browser then. Now we have our map rendering properly.



Summary

While our Contact Us page may have seemed at first to be a little more complex, it actually turned out to be quite simple as we were able to harness the power of Drupal 8's core functionality when it comes to both blocks and the new contact forms. In review, we covered the following:

- We began by reviewing the Contact Us page mockup to identify specific components and functionality that we would need to build.
- Next, we took a look at how Drupal implements contact forms for general website feedback and how to configure the fields and display of forms for use on our Contact Us page.
- Then, we used the Block layout admin to create two blocks for use on our Contact Us page—one to implement a callout and the second to render a Google map.
- Finally, we used libraries, scripts, and Twig to attach our jquery-gmap functionality to our Contact Map block.

In the next chapter, we will move on to Drupal's core search functionality as we tie back in our search block. We will also work with global search, use Twig to theme the results, and discuss how to best handle the search in Drupal 8.

11

Theming Our Search Results

Providing users the capability to search content within Drupal will help you ensure that the various content types are easily discoverable. Whenever a user cannot find content they are looking for, they will generally default to using some sort of global search. Earlier, we developed a Search form block that we placed within the main menu to globally search our site. In this chapter, we will circle around this block and focus on the Search results page that is displayed.

- We will begin with reviewing the mockup of our search page and identify how our search form input and any search results will need to be presented for display
- Next, we will take a closer look at search pages in core to learn how to configure what will be displayed in our results
- Finally, we will extend upon search by working with the Search API module to provide flexibility regarding which content types and fields can be added to search and how to use views to display our results

While we work through each section, we have the ability to refer back to the Chapter11 exercise files folder. Each folder contains a start and end folder with files that we can use to compare our work when needed. This also includes database snapshots that will allow us to all to start from the same point when working through various lessons.

Reviewing the Search Results mockup

Like previous sections of our website, having a mockup already provided to us to review makes planning how to develop a page much easier. Page structure, blocks, web forms, and other functionality, which we will need to consider, can easily be discovered by looking at the Search results page in the Mockup folder located in our exercise files.

Begin by opening up the `search.html` file within the browser.

DISPLAYING 1 - 1 OF 1

Post One

Euismod atras vulputate iltricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dul, tristique in semper vel, congue sed ligula. Nam dolor ligula, faudibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae.

The search page mockup provides us with a look at how the search term `lorem` would look when Drupal has returned results. The nice thing is that Drupal already provides us with a Search results page regardless of whether there are any results, which we will see later when we test different search terms. Because we do not need to provide any additional content, we will have no additional blocks of content to worry about. In fact, the only item we will need to focus on to recreate the themed mockup is the following:

1. First, we will need to inspect the markup provided by the Search input including any form of controls displayed on our page and determine if we need to alter the HTML.
2. Second, we will want to take a look at what Twig variables the Search results page is displaying and determine whether we need to suppress anything from being output.

Having identified these two different components, we need to keep one thing in mind. Our mockup takes into account what our search results will look like once we have extended Drupal's default search using the Search API module, https://www.drupal.org/project/search_api, which you will learn about later in the lesson. For now, let's take a look at what a default search results page looks like, so we can determine what work we have in store for us.

Looking at default Search results

The easiest way for us to take a look at what Drupal will return is by navigating to the homepage of our site and clicking on the search icon in the main menu.

We can now enter the keyword or term of `lorem`, as shown in the following image:



Once we have entered a keyword, we can hit *Enter* on our keyboard, which will take us to the Search results page located at `/search/node?keys=lorem`. We now have our first glance at the markup that Drupal displays by default.

Search for lorem

Content
Users

Enter your keywords

lorem

Search

Search results

ABOUT US

... A little about us. **lorem** ipsum dolor sit amet, consectetur adipisicing **lorem** ipsum dolor sit amet, consectetur adipisicing elit, ... ut labore et dolore magna incidunt ut labore aliqua. **lorem** ipsum dolor sit amet, consectetur adipisicing elit, ...

POST TWO

... News **lorem** ipsum dolor sit amet, consectetur adipisicing elit. ... in faucibus orci luctus et ultrices posuere cubilia Curae; **lorem** ipsum dolor sit amet, consectetur adipisicing elit. ... elit. Aliquam non ipsum id leo eleifend sagittis id a **lorem**. Cum sociis natoque penatibus et magnis dis parturient ...

Comparing the results to our Mockup, we can visually see that each individual result is displaying as an ordered list. Within each result, there is also additional information such as comments, which we will need to suppress. Search provides us with a couple of Twig templates we can use to clean up our markup. But before we move on to theming, it would help to have a better understanding of the options we have within core search and how to configure it for our needs.

Introducing core search

The admin for search pages can be located by navigating to /admin/config/search/pages. Search pages are part of the core search and metadata that allow both users and content to be indexed and searched based on different factors. If we take a more detailed look, we will note that we can index content, configure minimum word length, and specify which search page to use for our results.

The screenshot shows the 'Search pages' configuration page. At the top, there is a breadcrumb navigation: Home > Administration > Configuration > Search and metadata. Below the breadcrumb, there is a sidebar with the following sections: INDEXING PROGRESS, INDEXING THROTTLE, DEFAULT INDEXING SETTINGS, LOGGING, and SEARCH PAGES. A blue 'Save configuration' button is located at the bottom of the sidebar. The main content area is currently collapsed under the 'INDEXING PROGRESS' section.

Indexing content

The most important aspect of search pages is the **INDEXING PROGRESS** status. Indexing is the process of crawling the site or database, which, in turn, stores a result set that allows content to be found when a user enters a keyword or term in the search form. Unless content has been indexed, we will not have any results to display.

In order to index our site, we need to complete two steps.

1. First, we must click on the **Re-index site** button, as shown in the following image:

The screenshot shows the 'INDEXING PROGRESS' configuration sub-page. It displays the message: 'Only items in the index will appear in search results. To build and maintain the index, a correctly configured cron maintenance task is required.' Below this, it states '100% of the site has been indexed. There are 0 items left to index.' A blue 'Re-index site' button is located at the bottom of the sub-page.

Clicking on the **Re-index site** button does not perform the actual indexing but merely triggers the indexing to occur.

2. Second, we must run a cron maintenance task, which can be located at /admin/config/system/cron by clicking on the **Run cron** button.

Cron allows specific tasks to run based on a set interval. The interval can be modified by changing **CRON SETTINGS**. By default, Drupal will run Cron every 3 hours. Cron is triggered when a user first visits the site after the 3-hour period has expired. Cron can also be triggered from a URL outside the site, which allows the manager of the Cron task to be run from the web server itself.

We will not be covering the configuration of Cron from systems administration level. Just know that we can manually run Cron when needed by visiting this page.

After completing the two steps required to index our site, if we navigate back to the search page's admin, we should now see that our **INDEXING PROGRESS** reports that **100% of the site has been indexed**.

Editing search pages

Another configuration within the search pages interface allows us to modify or configure additional settings for the label, URL, and content ranking.

Theming Our Search Results

In order to configure our **Content** search page, we will need to expand the **SEARCH PAGES** section and click on the **Edit** button, as shown in the following image:

The screenshot shows a table with columns: LABEL, URL, TYPE, STATUS, INDEXING PROGRESS, and OPERATIONS. The 'Content' page has a URL of 'search/node', is of type 'Content', is in 'Default' status, and is 12 of 13 indexed. The 'Users' page has a URL of 'search/user', is of type 'Users', is in 'Enabled' status, and does not use indexing.

| LABEL | URL | TYPE | STATUS | INDEXING PROGRESS | OPERATIONS |
|---------|-------------|---------|---------|--------------------|-----------------------|
| Content | search/node | Content | Default | 12 of 13 indexed | <button>Edit</button> |
| Users | search/user | Users | Enabled | Does not use index | <button>Edit</button> |

The most important configuration option is the **CONTENT RANKING** section, which allows us to influence certain factors that search uses, including but not limited to **Keyword relevance** and **Number of comments**. Content ranking, as shown in the following image, can be modified by changing the value next to each factor.

The screenshot shows the 'Edit Content search page' form. Under 'CONTENT RANKING', it says: 'Influence is a numeric multiplier used in ordering search results. A higher number means the corresponding factor has more influence on search results; zero means the factor is ignored. Changing these numbers does not require the search index to be rebuilt. Changes take effect immediately.' Below this, there is a table with 'FACTOR' and 'INFLUENCE' columns for five different factors, each with a dropdown menu set to 0.

| FACTOR | INFLUENCE |
|---------------------------------------|-----------|
| Number of comments | 0 |
| Keyword relevance | 0 |
| Content is sticky at top of lists | 0 |
| Content is promoted to the front page | 0 |
| Recently created | 0 |

Save search page

A higher **INFLUENCE** value determines the order of the search results, and modifying this value does not require reindexing the site. For demonstration purposes, we will not change the values on this page.

One important thing to note is that Drupal not only indexes content but also any users that have accounts within the website. Although this is great in order to create member directories, in our case, we only want users to be able to search on content. We can remedy this by disabling users' search pages.

Disabling search pages

From the **Search pages** admin, if we look under the **SEARCH PAGES** section, we will see **Users** currently enabled:

1. Click on the **Edit** button for Users.
2. Click on the **Delete** link.
3. Click on the **Delete** button for complete removal.

We can always create a new Users page at a later time by returning to the Search pages admin. One last step, make sure to reindex the site now that we have removed users and then it's time to move on to review the markup of our Search results page.

Working with Search Results templates

If we navigate back to our Search results page, we can inspect the markup to help locate which Twig templates Drupal uses to output the content. If we break the page into sections, we will be left with two different sections:

1. First is the search results list, which is currently being output as an ordered list. We can address this by modifying the `item-list.html.twig` template.
2. Second is the Search results itself, which contains the title and snippet with a highlighted keyword. We will address this by modifying the `search-result.html.twig` template.

Modifying the item list template

Using the **FILE NAME SUGGESTIONS**, we can navigate to the `core/modules/system/templates` folder and copy the `item-list.html.twig` template to our `themes/octo/templates` folder. Next, we will need to rename the template to `item-list--search-results.html.twig` and then add the following markup:

New markup

```
{% set classes = ['list-unstyled'] %}

{% if context.list_style %}
  {- set attributes = attributes.addClass('item-list__' ~
```

Theming Our Search Results

```
    context.list_style) %}
{%- endif %}

{%- if items or empty %}

{%- if title is not empty -%}
    <h3>{{ title }}</h3>
{%- endif -%}

{%- if items -%}
    <{{ list_type }}{{ attributes.addClass(classes) }}>
        {%- for item in items -%}
            <li{{ item.attributes }}>{{ item.value }}</li>
        {%- endfor -%}
    </{{ list_type }}>
{%- else -%}
    {{ empty -}}
{%- endif -%}

{%- endif %}
```

Make sure to save the template, clear Drupal's cache, and then refresh the Search results page in the browser.

Search results

ABOUT US

... A little about us **lorem** ipsum dolor sit amet, consectetur adipisicing **lorem** ipsum dolor sit amet, consectetur adipisicing elit, ... ut labore et dolore magna incididunt ut labore aliqua. **lorem** ipsum dolor sit amet, consectetur adipisicing elit, ...

POST TWO

... News **lorem** ipsum dolor sit amet, consectetur adipisicing elit. ... in faucibus orci luctus et ultrices posuere cubilia Curae; **lorem** ipsum dolor sit amet, consectetur adipisicing elit. ... elit. Aliquam non ipsum id leo eleifend sagittis id a **lorem**. Cum sociis natoque penatibus et magnis dis parturient ...

0 comments

Our ordered list should now be displaying exactly like the Mockup. All we had to do to accomplish this was add a new CSS class to our template using Twig. We then added the new class to our markup using the Twig function `attributes.addClass()`.

Cleaning up each result

Now that our list is styled accordingly, we can focus on each individual search result. By default, each result returns the title, snippet, and additional information such as the number of comments if the result contains a Post. As each result is consistently styled, we will be removing the `{{ info }}` variable from the Twig template.

Using the **FILE NAME SUGGESTIONS**, we can navigate to the `core/modules/search/templates` folder and copy the `search-result.html.twig` template to our `themes/octo/templates` folder. Next, we will need to replace the markup by adding the following:

New markup

```

{{ title_prefix }}
<h3{{ title_attributes }>
  <a href="{{ url }}>{{ title }}</a>
</h3>
{{ title_suffix }}
{% if snippet %}
  <p{{ content_attributes }}>{{ snippet }}</p>
{% endif %}

```

Make sure to save the template, clear Drupal's cache, and then refresh the Search results page in the browser. Each result is now consistent in the information it displays.

Search results

ABOUT US

... A little about us **lorem** ipsum dolor sit amet, consectetur adipiscing **lorem** ipsum dolor sit amet, consectetur adipiscing elit; ... ut labore et dolore magna incididunt ut labore aliqua. **lorem** ipsum dolor sit amet, consectetur adipiscing elit; ...

POST TWO

... News **lorem** ipsum dolor sit amet, consectetur adipiscing elit. ... in faucibus orci luctus et ultrices posuere cubilia Curae; **lorem** ipsum dolor sit amet, consectetur adipiscing elit. ... elit. Aliquam non ipsum id leo eleifend sagittis id a **lorem**. Cum sociis natoque penatibus et magnis dis parturient ...

POST THREE

... Design **lorem** ipsum dolor sit amet, consectetur adipiscing elit. ... in faucibus orci luctus et ultrices posuere cubilia Curae; **lorem** ipsum dolor sit amet, consectetur adipiscing elit. ... elit. Aliquam non ipsum id leo eleifend sagittis id a **lorem**. Cum sociis natoque penatibus et magnis dis parturient ...

Search alternatives

Although working with the core search functionality in Drupal 8 can feel somewhat limited, it is not the only solution. There are alternatives to search that can be implemented to provide for more robust options. Two such alternatives are: Search API and Search API Solr Search. We will not be discussing Apache Solr as it is a little more complex to install and configure. However, the Search API will allow us to extend the default database search and is a perfect solution for our needs.

Search API

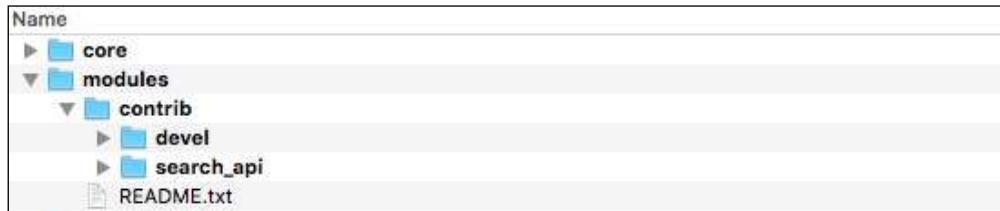
The Search API module at https://www.drupal.org/project/search_api provides a framework in order to extend core search. Multiple Search indexes can be created that then allow the use of Views to list search content. Each index can also enhance the user interaction by creating a series of Facets that allow results to be filtered down to a granular level. Finally, each index based on the content type can be configured to let Drupal know exactly which fields should be included and the importance or weight of each field.

The advantage of using this great module is the flexibility it provides to display the search results. Instead of having to manage the display using a single Twig template as we did previously, we can use display modes for each content type. This, in turn, allows us to also have additional Twig templates for each result if needed.

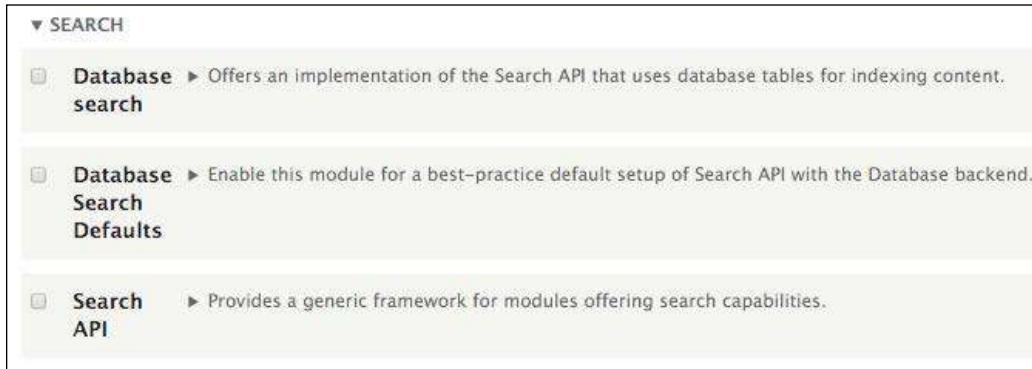
Let's take a deeper look at enhancing our site's search functionality by installing and configuring the Search API module.

Installing the Search API

Begin by browsing the Search API project page located at https://www.drupal.org/project/search_api and extract the contents to our modules directory, as shown in the following image:



With the module in place, we can now navigate back to our Drupal instance and finish installing the Search API. Navigate to /admin/modules and locate the **Database Search** and **Search API** modules located in the **SEARCH** section.



1. Select the checkbox for **Database Search**.
2. Select the checkbox for **Search API**.
3. Click on the **Install** button.

One important thing to note is that we will also want to uninstall the default Drupal Search module because we are replacing it with the Search API. We can do this by following these steps:

1. Select the **Uninstall** tab.
2. Select the checkbox for **Search**.
3. Click on the **Uninstall** button.
4. Click on the **Uninstall** button again from the **Confirm uninstall** page.

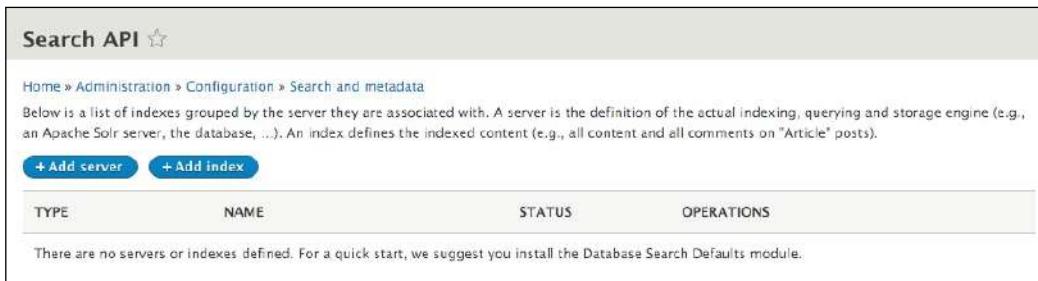
Now that we have replaced the default Drupal search with the Search API module, it is time to do some simple configuration.

Adding a server

The first step in configuring the Search API is to add a server definition for our index to use. In our case, we will be pointing to the default Drupal database.

Theming Our Search Results

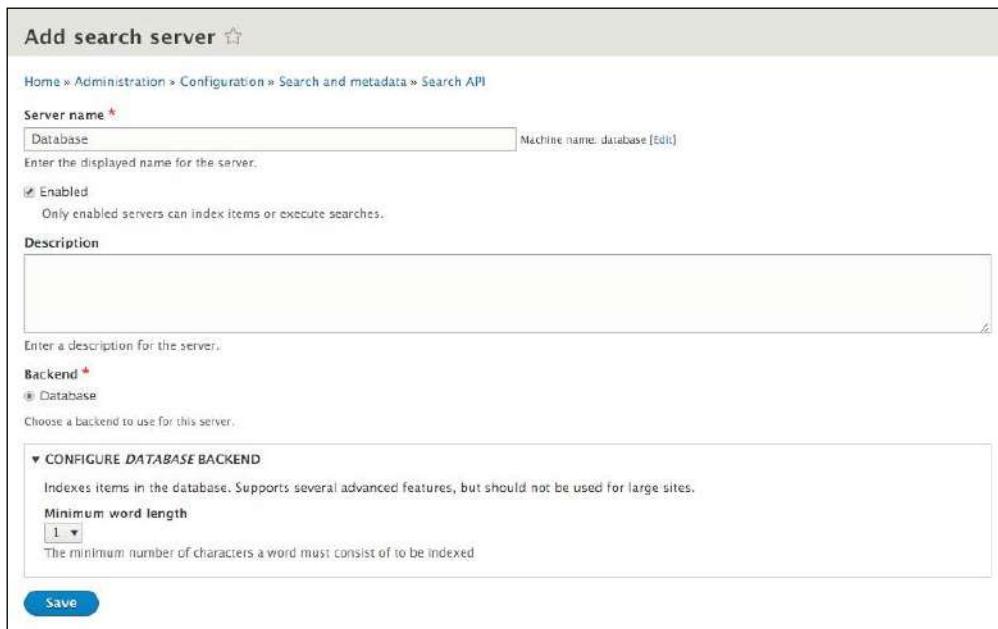
To get started, we will need to navigate to /admin/config/search/search-api, which will bring us to the Search API admin page.



The screenshot shows the 'Search API' configuration page. At the top, there are two buttons: '+ Add server' and '+ Add index'. Below them is a table with columns 'TYPE', 'NAME', 'STATUS', and 'OPERATIONS'. A message at the bottom states: 'There are no servers or indexes defined. For a quick start, we suggest you install the Database Search Defaults module.'

From here, we can follow these steps to add our server:

1. Click on the **Add server** button.
2. Enter a value of database in the **Server name** field.
3. Leave the remaining defaults, as shown in the following image:



The screenshot shows the 'Add search server' configuration form. It includes fields for 'Server name' (set to 'Database'), 'Enabled' status (checked), 'Description' (empty), 'Backend' (set to 'Database'), and a 'CONFIGURE DATABASE BACKEND' section with a minimum word length of '1'. A 'Save' button is at the bottom.

4. Click on the **Save** button.

We have now successfully added a search server that uses our Drupal database. We will point to this search server in our next step when we create an index.

Adding an index

Adding a search index allows us to be more explicit with the type of content we want Drupal to use when someone searches for a keyword or term. Having an index available allows us to use Views to create our Search results page and utilize our content's various display modes.

From the Search API admin page, we can follow these steps to add our index:

1. Click on the **Add index** button.
2. Enter a value of Content in the **Index name** field.
3. Choose **Content** from the **Data sources** field.
4. Choose **Database** from the **Server** checkbox.
5. Leave the remaining defaults, as shown in the following image:

Add search index

Home » Administration » Configuration » Search and metadata » Search API

Index name *
Content Machine name: content [Edit]
Enter the displayed name for the index.

Data sources *
Comment
Contact message
Content
Custom block
Select one or more data sources of items that will be stored in this index.

⚠ Please configure the used datasources.

Server
 - No server -
 Database
Select the server this index should use. Indexes cannot be enabled without a connection to a valid, enabled server.

Enabled
Only enabled indexes can be used for indexing and searching. This setting will only take effect if the selected server is also enabled.

Description
Enter a description for the index.

INDEX OPTIONS

Save **Save and edit**

6. Click on the **Save** button.

At this point, we have created an index but we have not defined which content types or fields our index will use. By default, all content types will be searched. While this may be okay in most cases, we only want our Post content to be searchable. Let's take a look at how to restrict which content types are indexed.

Configuring bundles

Content in Drupal is referred to as an entity, and the type of entity is called a bundle. We can configure which bundles are indexed by selecting them from our Edit page within the **CONFIGURE THE CONTENT DATASOURCE** section, as shown in the following image:



By default, all bundles will be indexed. However, we only want the post bundle to be searched, hence we make sure that we select the following:

1. Select **All except those selected** under **What should be indexed?**
2. Select **Headlines**, **Landing page**, and **Team** from **Bundles**.
3. Click on the **Save** button.

Now that we have configured our bundles, we will move on to choosing which fields will be searched on within our Post content type.

Adding fields to our index

The Search API module provides great flexibility when it comes to exactly what fields will be indexed. In the case of our Post content type, we only want to search on the **Title** and **Teaser** fields. We can start by selecting the **Fields** tab from our index and following these steps:

1. Click on the **Add fields** button.
2. Click on the **plus sign** next to **Content** to expand the field.

3. Click on the **Add** button next to **Title**.
4. Click on the **Add** button next to **Teaser**.
5. Click on the **Done** button.

We should now have our two fields added to the **CONTENT** section.

| FIELD | MACHINE NAME | TYPE | BOOST | REMOVE |
|--------|--------------|----------|-------|--------|
| Title | title | String | 1.0 | Remove |
| Teaser | field_teaser | Fulltext | 1.0 | Remove |

Although we are still within the **Fields** tab, we will want to take a quick look at some additional settings we have available to us.

- The first setting is the **TYPE** dropdown, which allows us to specify how a field will be treated when searched. In the case of our **Title** field, which is currently set to **String**, we will want to be able to do a **Fulltext** search on this field. We can select **Fulltext** from the dropdown to change the value. This will ensure that when a user enters a keyword, it searches all characters within the Title. If we left the default set to **String**, we would not get results returned properly.
- The second setting is the **BOOST** field, which allows us to give a field a higher level of importance when searched on. We will want our **Title** field to be more important than our **Teaser** field, so we can increase the **BOOST** value so that it has a higher number.

Make sure to click on the **Save changes** button to finish configuring our fields. Because we have made changes to both our bundles and fields, we will need to reindex our content. If we select the **View** tab, we can then click on the **Index now** button.

Now that our index is ready to use, we can create our Search results view.

Creating a Search Results View

We should be very comfortable working with Views, but this will be the first time we have used a search index as the source. To get started, we will need to navigate to `/admin/structure/views` and click on the **Add new view** button.

From the **Views** admin screen, we will add the following information:

- **VIEW BASIC INFORMATION:**
 1. **View name:** Search.
 2. Check the **Description** box.
 3. **Description:** Search results.
- **VIEW SETTINGS:** Show: Index Content sorted by: Unsorted.
- **PAGE SETTINGS:**
 1. Check: **Create a page**.
 2. **Page title:** Search.
 3. **Path:** search.
- **PAGE DISPLAY SETTINGS:**
 1. **Display format:** Unformatted list of: Rendered entity.
 2. **Items to display:** 10.
 3. Click on the **Save and edit** button.

From the **Search (Index Content)** admin, we will need to add some additional settings to our page. This includes selecting the display mode for our post to use, a filter criteria so that we can expose a form to replace our Global Search form, and a header to display the number of results.

Using the Search index view mode

We are familiar with using view modes to display content within a view. Our Search view will be using the Search index view mode, which we will need to add to our Post content type. Navigate to /admin/structure/types/manage/post/display and expand the **CUSTOM DISPLAY SETTINGS** field.

1. Select **Search index** from the **Use custom display settings for the following modes**.
2. Click on the **Save** button.

Next, we will want to enable only our **Teaser** field to display when the Search index view mode is used. We can accomplish this by selecting the **Search index** tab and adding our **Teaser** field, as shown in the following image:

| FIELD | LABEL | FORMAT |
|----------|------------|---------|
| # Teaser | - Hidden - | Default |
| Disabled | | |

With the field now enabled, make sure to click on the **Save** button to complete our changes. We can now navigate back to our Search view and select our new view mode.

Begin by navigating to `/admin/structure/views/view/search` and follow these steps:

1. Select the **Settings** link next to **Rendered entity** within the **FORMAT** section.
2. Choose **Search index** from the **View mode for datasource Content, bundle Post** dropdown.
3. Click on the **Apply** button.
4. Click on the **Save** button to finalize our changes.

If we preview our results, we should see only the title and teaser fields being returned. Next, we need to add a Fulltext search filter that we can expose to our end users.

Adding filter criteria

Within the **FILTER CRITERIA** section, we can add various filters and expose them for use within our page. The form we will be adding will also be replacing our Global search form within the header:

1. Click on the **Add** button under **FILTER CRITERIA**.
2. Choose **Fulltext search**.
3. Click on the **Apply (all displays)** button.
4. Enter a value of 3 within the **Minimum keyword length** field.
5. Click on the **Apply (all displays)** button.
6. Click on the **Save** button to finalize our changes.

We now have a Fulltext search field added to our view. However, we still need to expose the form to end users. We can enable this by selecting the **Search: Fulltext search** link and following these steps:

1. Click on the **Expose this filter to visitors, to allow them to change it** checkbox.
2. Delete the **Label** value.
3. Replace the **Filter identifier** field value with the value term.
4. Click on the **Apply (all displays)** button.
5. Click on the **Save** button to finalize our changes.

Now that we have exposed our form, we will want to make it available within the Block layout page. We can do this by expanding the **ADVANCED** section of our view and following these steps:

- **EXPOSED FORM:**
 1. Select the link next to **Expose form in block**.
 2. Select **Yes**.
 3. Click on the **Apply** button.
- **OTHER:**
 1. Select the **Machine Name** link and change the value to search.
 2. Click on the **Apply** button.
 3. Select the **CSS class** link and change the value to search-index.
 4. Click on the **Apply (all displays)** button.
 5. Click on the **Save** button to finalize our changes.

With our exposed form now available to our Block layout, let's add it back to our Header region so that we can test our Search view.

Placing our exposed search form

If we navigate to `/admin/structure/block`, we will now be able to add our exposed form to the Header region. This new exposed form will replace the global search form.

1. Click on the **Place block** button next to the **Header** region.
2. Click on the **Place block** button next to **Exposed form: search-search**.
3. Uncheck **Display title**.
4. Add a value of `search_form_block` in the **Machine-readable name** field.
5. Click on the **Save block** button.

If we navigate to our home page, we can now test our exposed search form by clicking on the search icon. At first glance, we may not notice any difference from our original search form. However, we are missing the placeholder that once prompted our users to enter their search terms. Let's remedy this by adding some custom JavaScript to our theme that will add back this attribute.



Adding our placeholder attribute

So far we have only worked with Twig templates to add additional attributes to our markup. However, there may be times where simple JavaScript makes more sense to add to our project to add functionality. In the case of our Search form input, we would like to prompt our users what to enter, but the input is currently empty. Using some simple jQuery, we can target the element and add our placeholder attribute.

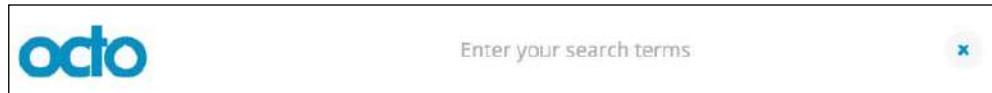
If we open up `octo.js`, located in our `themes/octo/js` folder, we can add the following snippet directly after the last function to accomplish this:

JavaScript

```
// Add Placeholder attribute
$(".search-block-form #edit-term").attr("placeholder", "Enter your
search terms");
```

Theming Our Search Results

Make sure to save the file, clear Drupal's cache and then refresh the homepage. If we now click on the search icon, we will see that we now have a placeholder prompting our users what to enter.



The JavaScript that we added uses jQuery to point to the element and then adds an attribute to the element with the specified value. This is an example of how easy it is to use custom script to add simple functionality.

Using our search form

Let's now add a search term to our input. We can enter the term *Post* and hit *Enter*, which will take us to the Search results page generated by our view.

A screenshot of a search results page. At the top is a dark header bar with the word 'Search' in white. Below it is a white content area. The first result is a card titled 'Post One' in blue. The card contains a short snippet of text: 'Euismod atras vulputate iltricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae.' The second result is a card titled 'Post Three' in blue. It also contains a similar snippet of text.

The search results are returning all content that we included in our search index. The term we entered is used to look at both the Title and Teaser fields for a match to the value of Post. One thing we are missing that would enhance our page is the number of search results being displayed.

Displaying the number of search results

Because our search results page is created by our view, we have the option to add additional information to our page using the Views header. We are familiar with this capability as we used it in earlier views. This time around, we will use it to display the number of records being returned by our view.

Begin by navigating to `/admin/structure/views/view/search`, which will allow us to modify our Search view. We will be adding Result summary to our page by following these steps:

1. Click on the **Add** button next to the **HEADER** section.
2. Select the checkbox next to **Result summary** within the **Add header** modal.
3. Click on the **Apply (all displays)** button.
4. Within the **Display field**, we will wrap the content with an `h3` heading, as shown here:



5. Click on the **Apply (all displays)** button.
6. Click on the **Save** button to finalize our Views configuration.

Theming Our Search Results

If we now navigate back to our Search results page by navigating to /search, we will see our newly added Result summary heading along with all results displayed.

The screenshot shows a search results page with a dark header bar containing the word "Search". Below the header, the text "DISPLAYING 1 - 3 OF 3" is displayed in blue. Underneath this, there is a section titled "Post One" which contains some placeholder Latin text: "Euismod atras vulputate iltricies etri elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla nunc dui, tristique in semper vel, congue sed ligula. Nam dolor ligula, faucibus id sodales in, auctor fringilla libero. Pellentesque pellentesque tempor tellus eget hendrerit. Morbi id aliquam ligula. Aliquam id dui sem. Proin rhoncus consequat nisl, eu ornare mauris tincidunt vitae."

We can use the global search form to add different terms, and based on the term we enter, the Results summary will change to reflect the number of items in our results.

One last thing to test is what happens when we enter a term that has no results. Let's enter a term of Drupal and hit *Enter*. Our page returns no results, but we have not displayed that to the user. This can sometimes lead our user to believe that search is broken. It is always a best practice to alert the user that no results were found instead of just a blank page.

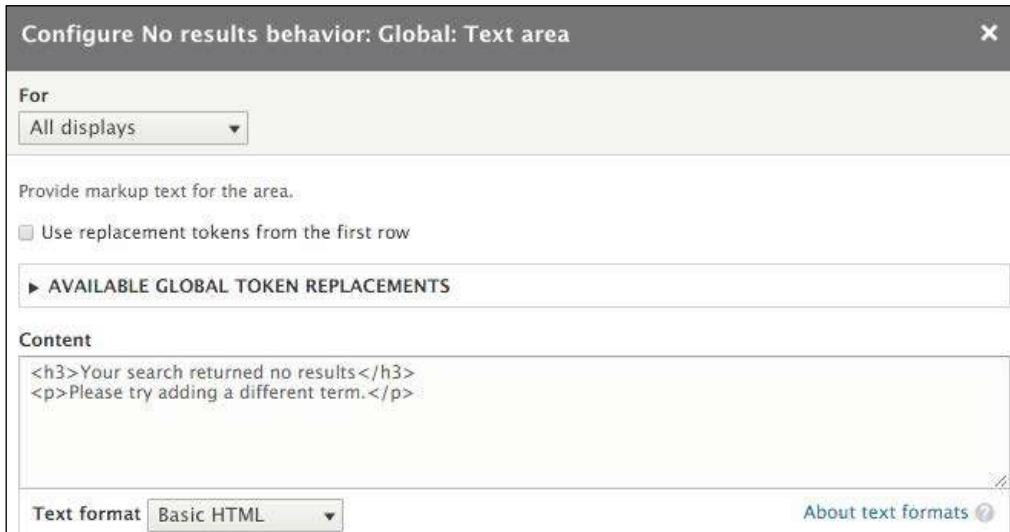
Adding a No Results message

If we navigate back to our Search view located at /admin/structure/views/view/search, we can easily add a message to our page when the search index doesn't return results.

We will be adding a Text area field to our page by following these steps:

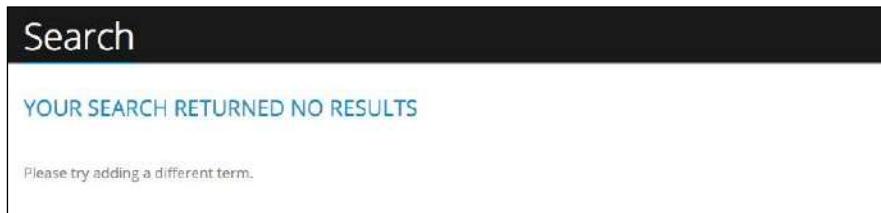
1. Click on the **Add** button next to the **NO RESULTS BEHAVIOR** section.
2. Select the checkbox next to **Text area**.
3. Click on the **Apply (all displays)** button.

4. Within the **Content** field, we will add the following markup as shown here:



5. Click on the **Apply (all displays)** button.
6. Click on the **Save** button to finalize our Views configuration.

If we now navigate back to our Search results page and enter the term Drupal again, our page will now display the **No Results Behavior** section that we just added.



Summary

Congratulations! We have completed the theming of our Search results page. We began with the default core search and worked our way through replacing core search with the more robust Search API. The ability to extend Drupal using contributed modules is one of the primary reasons that makes it such a widely used platform to develop websites.

Let's review what we covered in this lesson:

- We started with our Search results page mockup and identified form elements and markup that would need to be modified in Drupal.
- Using core Drupal search, you learned the important aspects of configuration and how to control keyword factors. Once search was configured, we used Twig templates to override the Search results templates to match our mockup with minimal changes.
- Realizing that we needed to extend search to make it more flexible for our needs, we explored using the Search API module. Being able to create a search server, search index, and use Views to build our Search results page provided us with the ability to better customize the search experience.

At this point, we have themed every section of our website and successfully recreated our Mockup using Drupal 8 and Twig. So where do we go from here? Well, there are still a few tricks for us to learn in the final chapter. These are: theming a few Admin components, such as the local tasks menu and status messages, reusing Twig templates, and finally getting involved in the community.

12

Tips, Tricks, and Where to Go from Here

Now that we have followed the frontend developer's path of taking a design or mockup and converting it into a working Drupal 8 theme, we have to ask that burning question, what next? The answer to that question depends on the problems we need to solve.

For example, what about theming some more common, but often forgotten, admin sections, such as the local tasks menu or status messages block? How about extending Twig templates to reduce having to manage markup in multiple places? However, the most common question is what about contributed modules that can help us with our theming?

In this final chapter, we will take a look at answering these last-minute questions, as we cover the following:

- We will begin with cleaning up our theme by adding some additional Twig templates for both the local tasks menu and the status messages block.
- Next, we will take a look at extending Twig templates by using template inheritance to reduce the amount of markup we have to manage in our theme.
- Finally, we will take a look at the state of some common Drupal contributed modules, such as Display Suite.

While we work through each section, we have the ability to refer back to the Chapter12 exercise files folder. Each folder contains `start` and `end` folders with files that we can use to compare our work when needed. These also include database snapshots that will allow us all to start from the same point when working through various lessons.

Working with Local Tasks

One of the most common content blocks within Drupal 8 that is often forgotten about when creating a theme is Local Tasks, often referred to as Tabs. We can see an example of the Tabs block whenever a user needs to perform some sort of action, such as viewing and editing a Node, or even when logging in to Drupal. If we make sure that we are logged out of Drupal Admin and then navigate to /user/login, we will see the **Log in** and **Reset your password** links that make up the tabs on the user login page:



The screenshot shows the 'Log in' page of a Drupal site. At the top, there is a dark header bar with the word 'Log in' in white. Below the header, there is a list of links under the heading 'Local tasks': 'Log in' and 'Reset your password'. There are two input fields: 'Username' and 'Password'. Below each field is a placeholder text: 'Enter your Octo username.' and 'Enter the password that accompanies your username.'. At the bottom left of the form is a 'Log in' button.

If we input our admin credentials and log in to our Drupal instance we will see that the local tasks menu changes to display **View**, **Shortcuts**, and **Edit** links. The local tasks menu will change, based on the type of page we are on and the permissions that each user has been assigned.

If we navigate to the About Us page located at /about, we will see that our local tasks menu now provides us with the ability to **View**, **Edit**, or **Delete** the current Node.



The screenshot shows the 'About Us' page of a Drupal site. At the top, there is a dark header bar with the word 'About Us' in white. Below the header, there is a list of links under the heading 'Local tasks': 'View', 'Edit', and 'Delete'. To the right of these links is a small image of a computer monitor displaying a colorful abstract background. To the right of the image is the text 'A LITTLE ABOUT US' in blue and 'Lorem ipsum dolor sit amet, consectetur adipisicing' in smaller black text.

Now that we have a better understanding of how the Local Tasks menu or tab changes, lets dive into how we would go about theming it.

Theming local tasks

Local tasks, or the Tabs menu, is quite simple to theme. Drupal provides us with two Twig templates, `menu-local-tasks.html.twig` and `menu-local-task.html.twig`, which we can modify using techniques we are familiar with. For this exercise, we will simply add some classes to the template so that it styles our links as the pill buttons provided by the Twitter Bootstrap framework.

Begin by creating a copy of `menu-local-tasks.html.twig` located in the `core/modules/system/templates` folder and add it to our `themes/octo/templates` folder. Next, we will open the template and replace the markup with the following:

New markup

```
{% if primary %}
  <h2 class="visually-hidden">{{ 'Primary tabs' | t }}</h2>
  <ul class="nav nav-pills primary">{{ primary }}</ul>
{% endif %}
{% if secondary %}
  <h2 class="visually-hidden">{{ 'Secondary tabs' | t }}</h2>
  <ul class="nav nav-pills secondary">{{ secondary }}</ul>
{% endif %}
```

Make sure to save our changes, clear Drupal's cache and then refresh the browser. Our local tasks are now displayed inline and if we hover over each item, we will see the outline of the pill formatting. Next, we need to add the `active` class to each list item to help differentiate which action is currently being displayed.

Begin by creating a copy of `menu-local-task.html.twig`, located in the `core/modules/system/templates` folder, and add it to our `themes/octo/templates` folder. Next, we will open the template and replace the markup with the following:

New markup

```
<li{{ attributes.addClass(is_active ? 'active') }}>{{ link }}</li>
```

Make sure to save our changes, clear Drupal's cache, and then refresh the browser. The markup we added looks to see whether the list item is active and if it is, we add the `active` class to it. This then displays as a blue pill button within our webpage:



Great, we have now themed our first Admin component. The next item we will look at implementing is our Status messages block.

Working with Status messages

The Status messages block is what Drupal uses to inform users of specific actions they have completed, as well as to display any PHP warnings or errors. If we navigate to the Block layout admin, we can see that the Status messages block is currently assigned to the Highlighted region of our theme.



However, we are currently not outputting this region within our page template, which means that any messages Drupal is trying to display will not be seen. Let's remedy this by printing the region to our `page.html.twig` template.

Adding the Highlighted region

Begin by opening `page.html.twig`, located in our `themes/octo/templates` folder. We will want to modify our markup to add the `page.highlighted` region variable to our template. We can add it between our `title_bar` and `before_content` regions as shown here:

New markup

```
{<code>{{ page.title_bar }}</code>
{<code>{{ page.highlighted }}</code>
{<code>{{ page.before_content }}</code>
```

Make sure to save our changes, clear Drupal's cache, and then refresh the browser. If we are on the **About Us** page, we simply need to click on the **Edit** button to modify the page content and then click on the **Save and keep published** button to trigger our status message.

A screenshot of the 'About Us' page in Drupal. The page title is 'About Us'. Below the title, a status message reads 'Landing page About Us has been updated.' At the bottom of the page, there is a horizontal bar with three buttons: 'View' (highlighted in blue), 'Edit', and 'Delete'.

It is easy to see how a user visiting our site could miss any status messages that we may want them to see, simply because we forgot to theme this component.

Theming our Status message block

It is important to ensure that any messages that we display to the end user are visible and invoke the proper message, based on whether the action was successful or whether the action returned an error. For this purpose, we can borrow from the Twitter Bootstrap Alert component.

If we inspect the page, we can see that Drupal is using the `status-messages.html.twig` template. Begin by creating a copy of `status-messages.html.twig`, located in the `core/modules/system/templates` folder, and add it to our `themes/octo/templates` folder. Next, we will open the template and replace the markup with the following:

New markup

```
<div class="container">
  {# for type, messages in message_list #-}

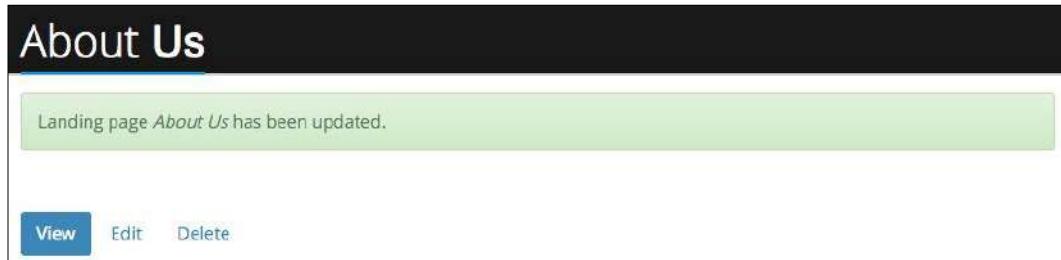
  {# if type == 'error' #-}
    {# set classes = ['alert', 'alert-danger', 'alert--' ~
      type] #-}
  {# else #}
    {# set classes = ['alert', 'alert-success', 'alert--' ~
      type] #-}
  {# endif #-}

  <div {{ attributes.addClass(classes) }} role="alert">
    {# if messages|length > 1 %}
      <ul>
        {# for message in messages %}
          <li>{{ message }}</li>
        {# endfor %}
      </ul>
    {# else %}
      {{ messages|first }}
    {# endif %}
  </div>

  {# Remove type specific classes #}
  {{ attributes.removeClass(classes) }}

  {# endfor %}
</div>
```

Make sure to save our changes, clear Drupal's cache, and then refresh the browser. If we click on the **Edit** button and then click on the **Save and keep published** button, we will trigger our status message to display. However, this time it is displayed with the proper styling:



A screenshot of a web page titled "About Us". At the top, there is a dark header bar. Below it, a green rectangular box contains the text "Landing page About Us has been updated.". At the bottom of the page, there is a navigation bar with three buttons: "View" (highlighted in blue), "Edit", and "Delete".

There are a couple of things to point out regarding our modification of the `status-messages.html.twig` template:

1. First, we added a block element to wrap our status message block with a class attribute of `container` to ensure we had proper margin and constraints on the content.
2. Next, we tested for the type of message Drupal was outputting and added the proper alert type to the `classes` variable. We then appended our `classes` variable to any existing classes on our alert element, using the `attributes.addClass()` function.
3. Finally, since we may have multiple messages of varying types being displayed, we removed the `classes` variable using the Twig function `attributes.removeClass()` through each iteration, so that we display the proper alerts.

We have now successfully themed both our tabs and status messages. This provides us with a cleaner user interface and allows for messages to be clearly displayed.

Reusing Twig templates

If we look at our `themes` folder, we can see that we have created quite a few Twig templates. While having an abundance of Twig templates is not necessarily a bad thing, it does become a little bit harder to manage our code.

One powerful feature of Twig templates is that they allow us to address this by extending or sharing markup between each template. This is great for global sections of markup. For example, the header and footer of our page never change, so why have the same markup in both page templates?

Using `extends` to share layouts

Twig `{% extends %}` allows us to share markup between templates by extending a template from another one. In our case, we could use this Twig function by extending the `page.html.twig` template from our `page--front.html.twig` template.

Begin by opening up the `page--front.html.twig` template in our favorite editor and add the following markup to the very top of our page:

New markup

```
{% extends 'themes/octo/templates/page.html.twig' %}
```

Now, if we were to save our template, clear Drupal's cache, and browse to our homepage, we would see Drupal complaining about how a template that extends another cannot have a body. The reason is that, while we are extending the `page.html.twig` template, we are not exposing any Twig blocks, which is a requirement for reusing sections of markup.

Working with Twig blocks

Twig blocks, not to be confused with Drupal Blocks, are just containers that we can place around a section of content which allow another template to be able to replace the content within it. So how does this work exactly?

First, a Twig block is referenced using the following syntax:

```
{% block content %}  
some content or markup  
{% endblock %}
```

In this example, some content or markup could be replaced from another template extending the parent template. It's not important what you call your `{% block %}`, just as long as you're consistent.

Continuing with extending the `page.html.twig` template, begin by adding the following Twig block around our `<main>` content section:

New markup

```
{% block content %}  
<main>...</main>  
{% endblock %}
```

Now we have a Twig block that can be referenced from within `page--front.html.twig` so that any content within the `<main>` content element can potentially be replaced.

Next, we will want to edit `page--front.html.twig` and first remove the `<header>` and `<footer>` blocks, as we will inherit those from the `page.html.twig` template:

New markup

```
{% extends 'themes/octo/templates/page.html.twig' %}  
{% block content %}  
{{ attach_library('octo/flexslider') }}  
{{ attach_library('octo/scroll-to') }}  
  
<section class="intro" id="section1" data-speed="5" data-type="background" style="background-position: 50% 0px;">  
    <div class="overlay">  
        <div class="headline">  
            {{ page.headline }}  
        </div>  
    </div>  
    <a href="#" id="goto-section2" class="arrow-down">Get  
        started</a>  
</section>  
  
<main role="main" class="main">  
    <div class="layout-content">  
        <section id="section2" class="section">  
            <div class="container">  
                {{ page.before_content }}  
            </div>  
        </section>  
    </div>  
</main>  
  
{% endblock %}
```

Now, if we save both templates, clear Drupal's cache, and refresh our homepage, we should see that our markup is being output correctly. So here's a quick explanation of what's happening here:

- First, our homepage is now inheriting all markup from `page.html.twig` including the header, footer, and Twig block. This is why we still see the header and footer of our website being displayed. However, we now only have to worry about managing that markup from a single template.

- Second, since we are including our own markup within the Twig block from our `page--front.html.twig` template, it is overriding the markup in `page.html.twig` and displaying the proper content for our homepage.

This is very powerful, as we can start to look at any content that is repeated across our templates and, with some proper planning, use `{% extends %}` and `{% block %}` to manage our markup.

This is just the surface of how Twig can be extended; for more information and more details of all the possibilities, take a look at the documentation at Sensio Labs: <http://twig.sensiolabs.org/>.

Where do we go from here?

We have covered a lot of information on how to use Drupal 8 and the new Twig template architecture to produce a great looking theme. So where do we go from here? One of the challenges of working with Drupal 8 in its infancy is the lack of contributed modules that have been ported over from Drupal 7. Just know that module maintainers are working hard to bring new and familiar ways to extend Drupal 8.

Some great modules worth taking a look at that have made great progress include:

- Panels: <https://www.drupal.org/project/panels>
- Page Manager: https://www.drupal.org/project/page_manager
- Display Suite: <https://www.drupal.org/project/ds>

Each module by itself is great, but together these modules will allow you to achieve a lot of different layouts quickly and easily, without the need to always create a Twig template.

As always, keep an eye on the Drupal 8 theming documentation located at <https://www.drupal.org/theme-guide/8> for detailed information on how to accomplish basic concepts.

Drupal also has a great community of people with various levels of knowledge and expertise, located at <https://www.drupal.org/>. While visiting Drupal.org, please become a member of the Drupal Association at <https://assoc.drupal.org/support-project-you-love> as well as creating a Drupal profile.

Finally, get involved and attend a local Drupal camp or DrupalCon itself for great sessions, training and fun. More information can be found at <https://www.drupal.org/drupalcon>.

Summary

Wow! Time to pat ourselves on the back; we shared a lot of information that has taken us from zero to hero with Drupal 8 theming with Twig. We learned basic site architecture, how to navigate all that is new with the exciting changes in the Drupal 8 Admin, Custom Blocks, Views, Twig templates and so much more. Before we know it, we will be challenging ourselves with the next great design and creating themes with ease.

So until next time, keep coding...

Bibliography

This Learning Path is a blend of content, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Learning Drupal 8, Nick Abbott, Richard Jones*
- *Drupal 8 Development Cookbook, Matt Glaman*
- *Drupal 8 Theming with Twig, Chaz Chumley*



Thank you for buying
**Drupal 8: Enterprise Web
Development**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Please check www.PacktPub.com for information on our titles

