



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Programming Drupal 7 Entities

Expose local or remote data as Drupal 7 entities and
build custom solutions

Sammy Spets

[PACKT] open source*
PUBLISHING community experience distilled

www.it-ebooks.info

Programming Drupal 7 Entities

Expose local or remote data as Drupal 7 entities
and build custom solutions

Sammy Spets



BIRMINGHAM - MUMBAI

Programming Drupal 7 Entities

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book 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 book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, 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 book.

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

First published: June 2013

Production Reference: 1190613

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.
ISBN 978-1-78216-652-8

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Author

Sammy Spets

Project Coordinator

Suraj Bist

Reviewers

James Roughton

Michelle Williamson

Proofreader

Dirk Manuel

Acquisition Editor

James Jones

Indexer

Priya Subramani

Lead Technical Editor

Harsha Bharwani

Graphics

Ronak Dhruv

Technical Editors

Sanhita Sawant

Dennis John

Production Coordinator

Kirtee Shingan

Cover Work

Kirtee Shingan

About the Author

Since 2004, **Sammy Spets** has been finding pleasure in his life making Drupal do wild things. During that time, Sammy volunteered to be a core maintainer for Drupal 6 and a maintainer of the *ecommerce* module, which was the commerce module of choice way back when. For the *ecommerce* module, Sammy made design changes to the payment system, built a few modules to support payment gateways, and added PostgreSQL support, among other things.

In 2008, IDG Australia contracted Sammy to design and lead the development of a hybrid Drupal/legacy platform. The platform allowed IDG developers to gradually migrate their websites and web applications over to Drupal 6, which was still in beta. In addition to the platform, Sammy was tasked with creating a module suite for the IDG staff to create surveys and reports on them. This module suite was built prior to webform, and leveraged the power of the Drupal 6 Form API in all its glory. Sammy also trained IDG developers to develop modules and themes in Drupal 6.

Early in 2009, a short contract with Demonz Media in Sydney, Australia brought about some patches to Ubercart, which Demonz gladly contributed back to the community.

Following that, Sammy travelled to Louisville, Kentucky, USA where he contributed code to improve the developer experience for developers extending Ubercart by using its API. Ryan Szrama introduced Sammy to *Chick-fil-A* and Lyle Mantooth introduced Sammy to Korean food and some amazing fried chicken.

In 2011, Sammy joined the Magicspark team, building Drupal sites and maintaining servers. During this time, Sammy built a services platform to feed webform data to Marketo and LoopFuse from client Drupal sites via Magicspark's servers. In addition to this, Sammy redeveloped the UI on the *Where to Buy* page of the Redwood Systems website using OpenLayers mapping.

Aside from the geeky stuff, Sammy loves to cook, fine-tune recipes, play pool, carve turns on a snowboard, hit the gym, ride motorcycles, dine fine, and drink champagne.

Programming Drupal 7 Entities, Packt Publishing, is the first book Sammy has authored. Sammy was the technical reviewer for *Migrating to Drupal 7*, Packt Publishing.

Sammy can be contacted by e-mail at sammys@sammypets.com.

Acknowledgement

I would like to thank Jason Chinn from Magicspark for his understanding, his belief in me, and giving me spare time to write this book. Thank you to my Mum, Anja Spets, for her unconditional support over the years. To my Dad, Raimo Spets; I know you would have been proud to see this book published; may you rest in peace. Thank you to Raija and Markku Tujula for taking care of my Mum.

Thank you to Arphaphorn Phrompt (Waew) for filling my life with peace, fun, and companionship. Last, but not least, I thank my great friends, Martijn Blankers and Job de Graaff, for pretending to be interested when listening to me rant about this book. You both are awesome! Beer time!

About the Reviewers

James Roughton, received his Bachelor of Science degree in Business Administration from Christopher Newport College and his Masters degree in Safety Science from Indiana University of Pennsylvania (IUP). In addition, he is a Certified Safety Professional (CSP), a Registered Canadian Safety Professional (R-CRSP), and a Certified Hazard Material Management (M-CHMM). He also holds several training certifications: Certified Environmental Trainer (CET) and a Certified Instructional Technologist (CIT) with a certification in Six Sigma Black Belt. He recently became certified as an InBound Marketer in Social media.

He is an accomplished author and manages his own websites, www.safetycultureplus.com; and www.jamesroughton.com. He has received awards for his efforts in safety, and was named the Project Safe Georgia Safety Professional in 2008 and the Georgia ASSE Chapter Safety Professional of the Year (SPY) 1998-1999. James is an active member of the Safety Advisory Board of the Departments of Labor/Insurance of Georgia, and has been an adjunct instructor for several universities.

James has been very active in developing expertise in social media productivity and its use in communication of safety culture and safety management system concepts and information. In his latest project, he as just co-authored a new book entitled *Safety Culture: An Innovated Leadership Approach*, Butterworth-Heinemann.

You can use the following links to connect with him:

- YouTube: http://www.youtube.com/subscription_center?add_user=mrjamesroughton
- Twitter: <http://twitter.com/jamesroughton>
- LinkedIn: <http://www.linkedin.com/in/jamesroughtoncsp>
- Google +: <https://plus.google.com/u/0/102851102730471202754>

James is an independent consultant on safety and social media productivity. He has previously reviewed another book on Drupal.

Michelle Williamson began her journey with computers in 1994 as the result of a traumatizing mishap involving a 15-page graduate class paper and an unformatted floppy disk. She spent 5 years as a staunch Luddite before becoming obsessed with web development and technology in general. She has been a freelance web developer since 2000, starting out on Microsoft platforms, then drinking the open source Kool-Aid in 2008, and since then has devoted her time primarily to Drupal development. She's an incessant learner and is addicted to head-scratching challenges, and looks forward to experiencing the continued evolution of mobile technology.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Understanding Entities	7
Introducing entities	8
Entity and solution modules	9
Introducing entity types, bundles, and fields	9
Types	10
Bundles	10
Fields	11
Drupal core entity structure	11
Powerful entity use cases	12
User profiles	12
Internationalization	13
Commerce products	13
Our use case	13
Summary	14
Chapter 2: Developing with Entity Metadata Wrappers	15
Introducing entity metadata wrappers	15
Creating an entity metadata wrapper object	16
Standard entity properties	17
Entity introspection	18
Using an entity metadata wrapper	18
Create	18
Drush commands	18
Code snippet	19
Retrieve	19
Drush commands	19
Code snippet	19

Update	21
Drush commands	21
Code snippet	21
Delete	21
Drush commands	21
Code snippet	21
Safely using text property values	22
Self-imposed limitation of entity programming	22
References	22
A note about EntityFieldQuery	23
Summary	23
Chapter 3: Developing with Non-fieldable Entities	25
What are non-fieldable entities?	25
File entities	26
Vocabulary entities	30
Recipe site vocabularies	31
Summary	33
Chapter 4: Developing with Fieldable Entities	35
What are fieldable entities?	35
Node entities	36
Comment entities	39
Term entities	40
Summary	46
Chapter 5: Developing with Fields	47
Field types	47
Single-value and multi-value fields	48
Structure fields	50
Field type-specific code	51
File and image fields	52
Link fields	52
Datetime fields	53
Putting it all together	54
Converting the recipe content type to use fields	55
Creating fields	56
Exporting fields to a feature	58
Copying the code to the recipe module	60
Tweaking recipe.module and recipe.info	61
Upgrading recipe module	63
Summary	68

Chapter 6: Developing with Field Collections	69
Before Drupal 7	69
Creating a field collection field	71
Field collection entities	74
Adding a field collection to a node	76
Attaching a field collection to a content type	77
Exporting field collection and fields	78
Copying the code to the recipe module	80
Tweaking recipe.module	80
Updating code is unnecessary	82
Summary	82
Chapter 7: Expose Local Entities	83
Motivation for exposing entities	83
Fast track your data exposure	84
Allow fields on your entity	85
Give it multiple bundles	86
Administration interface and exportability	87
Storing bundle information	87
Exposing bundle information and handling access rights	91
Adding the support code	94
Summary	100
Chapter 8: Expose Remote Entities	101
Introducing the Remote Entity API	101
Requirements for exposing remote entities	102
Implementing remote entity exposure	103
Database schema	104
Connection code	104
Remote query code	105
Entity exposure code	105
Entity metadata API integration	107
Import and administration code	108
Running	108
Adding write support	109
Customization for your use case	110
Summary	110
Index	111

Preface

Drupal 7 brought about many innovations for developers, themers, and site builders. Entities are, without a doubt, the most fundamental innovation, and their birth produced the biggest impact in the way in which Drupal sites are built and modules are developed. The entity paradigm made available a powerful and unified API, making it easy to build solutions with minimal code catering for specific data structures.

This book peels the onion layers away, showing you how to Create, Retrieve, Update, and Delete (CRUD) entities in general; how to use entity metadata wrappers; how to utilize common entity types such as Nodes, Comments and Field Collections; and how to expose local or remote data to Drupal and contributed modules. Each chapter offers, you some code examples showing you how to do things with each of the entity types. All that without making your eyes water.

What this book covers

Chapter 1, Understanding Entities, differentiates entity and solution modules, and introduces entity types, bundles, and fields, followed by entity structures and some use cases where the entity paradigm is powerful.

Chapter 2, Developing with Entity Metadata Wrappers, delves into development using entity metadata wrappers for safe CRUD operations and entity introspection.

Chapter 3, Developing with Non-fieldable Entities, unveils non-fieldable entities and how they can be manipulated in code. File and Vocabulary entity types implemented in core are dissected and used as examples.

Chapter 4, Developing with Fieldable Entities, covers fieldable entities and how they can be manipulated in code. Core-implemented Node, Comment, and Term entity types are pulled apart and used as examples.

Chapter 5, Developing with Fields, discusses the differences between single-value and multi-value fields, explains structure fields, and then uncloaks the properties of common field types: date, file, image, link, number, text, and term reference. Practical examples also covered are: how to access fields of an entity, how to add fields to an entity, and how to migrate data into fields.

Chapter 6, Developing with Field Collections, introduces field collections and how they are programmatically manipulated, declared, and created.

Chapter 7, Expose Local Entities, discloses how easy it is to expose a database table as either a non-fieldable or fieldable entity, and then explains how to enable exporting, importing, and cloning of bundle configurations.

Chapter 8, Expose Remote Entities, covers the requirements of exposing remote data as entities. It also describes how to expose batch-imported remote data as entities in our example site.

What you need for this book

To complete the practical exercises in this book, you will need to have the following in your environment:

- A Web server capable of running Drupal 7, with PHP 5.2.4 or higher installed
- A MySQL database server that is accessible from the web server
- System-wide installation of Drush 5.x

You can avoid tweaking the Drupal settings if you use a MySQL server on your web server (`localhost`) and have a MySQL user account with the following credentials:

- Username: `drupal_entities`
- Password: `W43wSu4Ym44K`

Who this book is for

This book is aimed at readers with PHP development experience, along with some experience installing websites on a server. Familiarity with Drush and GIT is also recommended.

Using the example code

Example code for this book contains a GIT repository, and chapters 2 to 8 each have a branch. The branches are named `chapter_02` through to `chapter_08`. There is a branch named `complete`, which contains the completed code that you would achieve after finishing all practical examples.

At the beginning of each chapter you need to check out a branch by using the following command:

```
$ git checkout chapter_02
```

It's a good idea to configure a different site for each chapter on your web server. Otherwise, you can commit changes you make to the code before checking out another chapter branch.

Refer to the readme file supplied with the code for information about installing the example code.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
$options = array('sanitize' => TRUE);
$output1 = $entity->myproperty->value($options);
$options = array('decode' => TRUE);
$output2 = $entity->myproperty->value($options);
```


When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:


```
$output = 'First property: ';
$output .= $wrapper->property[0]->value();
foreach ($wrapper->property as $vwrapper) {
    $output .= $vwrapper->value();
}
```


Any command-line input or output is written as follows:

```
$ drush eu ingredient 1 Salt
$ drush help eu
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "You'll also see that all properties are read-only on the wrapper. This is denoted by the **R** in the **Type** column".

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via 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 book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, 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 all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

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 books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright 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

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Understanding Entities

Developing Drupal code has always been interesting and fun because the APIs change a lot between Drupal releases. Other CMS platforms have adopted a more static API approach, resulting in a much slower innovations. One such Drupal innovation has been the entity paradigm that simplified data manipulation. This enabled developers to build more powerful solutions and liberate their brains to remember more important things such as anniversaries and birthdays. Exposing custom data as entities can be done with simpler code and less repetition. Those entities can then be utilized by all modules with very little developer effort. The result: "Good Codes!"

In this chapter we will cover the following topics:

- Introduce entities and how the entity paradigm makes code more flexible, useful, simple, and robust
- Understand the difference between entity modules and solution modules
- Introduce entity types, bundles, and fields
- Learn the structure of entities exposed by Drupal core
- Highlight some use cases where the entity paradigm is powerful
- Introduce the use case that will be implemented in this book

Introducing entities

Let's start off by clearing up something very confusing. The word **entity** can be used to describe different perspectives of Drupal data and code. For clarity, we will use the following perspectives for this introduction:

- **Structure:** This is the description of the names, types, and sizes of data inside a container
- **Interface:** This is the channel through which code interacts with data inside a container
- **Box:** This is the structure and interface wrapped together so they can be manipulated, stored, or transferred as a single unit

Plastic food containers in our homes come in all shapes and sizes; with or without lids, clear or opaque. This is the container's **interface** perspective. Through this interface, we can take content out, put content in, and know what type of content it has.

Food containers can contain anything, and you can use spacers to separate the content in interesting ways. This is the **structure** perspective.

In our homes, these containers are all dealt with in a similar way or; in other words, a unified way. We can take them off the shelf, put them on the shelf, and even cook the content in a microwave (with the lid ajar of course). This is the **box** perspective.

Some of Drupal's boxes were, and still are, nodes, comments, users, terms, and vocabularies. These boxes were difficult to deal with collectively in a unified way, because their properties and methods differed a lot. In much the same way, food containers are not interchangeable. For example, some are microwave safe, while others are not. There aren't enough variations in food containers to cause difficulties, but there are in Drupal boxes. In the past, Drupal boxes had varying structures and many interfaces, making interchangeability impossible. The result: *spaghetti code*. The good news is that Drupal developers could do something about this.

These clever folks realized that making the interface the same, regardless of the structure, means that every box can be treated the same. They could end the spaghetti madness by creating an abstraction, offering a unified interface for Drupal data! This revelation resulted in an explosion of innovations, and Drupal entities were born.

Some of these clever folks had even predicted the power of such an abstraction. A unified data interface simplifies code and makes more data available for manipulation. In addition to this, new data structures exposed to Drupal could be manipulated by existing code with little or no additional code. In other words, developers can create a new entity and the many Drupal features leveraging entities will access the new entity in full, with very little effort.

Last, but definitely not least, a unified interface reduces bugs and improves maintainability because less specialized code is used.

Entity and solution modules

The Drupal community uses the following two categories for modules dealing with entities:

- **Entity modules:** These expose and manage the structure and interface by supplying any classes needed above and beyond the mechanisms provided by Drupal core in order to store and manipulate the entity. For example, comment, file, node, taxonomy, and user modules.
- **Solution modules:** They implement functionality and site features using entities (boxes) as their data source. For example, rules, search, token, and views modules.



A module can be both an entity module and a solution module at the same time!



Introducing entity types, bundles, and fields

The three conceptual components of an entity are as follows:

- Types
- Bundles
- Fields

Let's look at these components in detail, from a solution module developer's perspective.

Types

Semantically, an **entity type** defines the name, base structure, and interface of an entity. The entity type is tied to a table of data from which fields automatically become the entity properties.

In code, an entity type simply consists of metadata and classes. Drupal core uses its classes and the entity type metadata to expose entity data to code that uses a well-known structure-independent interface. This enables modules to **Create Retrieve Update Delete (CRUD)** and query different entity structures by using the same code.

Because the interface is consistent between entity types, it's quite safe to say that only the structure varies between them.

Bundles

The next rung up the conceptual ladder is a bundle, which is simply a name. A **bundle** can be considered an entity subtype and, when paired with the entity type, becomes an entity instance. It is possible for an entity type to only have one bundle, and this is used when a developer does not need more than one instance.

One real-world analogy would be to use *vegetable* as the entity type and then define *aboveground* and *underground* as the subtypes. Both subtypes have *dimensions* and other common properties, and those are defined in the *vegetable* entity type.

You might be wondering why the subtypes chosen are weird and not something like broccoli and spinach. The reason is because subtypes must be structurally different in order to warrant the division. Structurally similar things don't need a subtype. We had to recognize a distinguishing characteristic that makes the *properties of each entity different*. Underground vegetables have roots coming out of them, so only they will have properties related to roots.

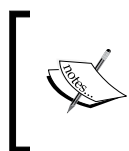
Similarly, Drupal has an entity type named **node** and two example subtypes are *blog posts* and *events*. Both bundles have an author and a creation date, but we probably want an image attached to the blog post and a start timestamp attached to the event. The common properties are part of the entity type; the uncommon properties are attached to bundles.

Fields

Drupal 7 fields came from Drupal 6 **Content Construction Kit (CCK)** fields. Even though CCK fields did cause a Darwin, inspired module extermination, they benefited the Drupal landscape by paving the way to entities. CCK fields made it possible to attach use case-specific fields to content types (nodes) without writing a single line of code. A site builder could attach a field by using the site's administration UI. In Drupal 7, field implementation was moved to Drupal core, and fields now attach to bundles of all entity types not just nodes.

As already mentioned, an entity's structure is based on the properties of the entity type. The structure is then extended by its fields. Fields are attached to an entity bundle by a developer in code or by a duly authorized user using the **Manage fields** user interface.

Entities can either accept fields or not accept them. This is known, in Drupal parlance, as **fieldability** or, in other words, the ability to attach fields. Entities are considered as either fieldable or non-fieldable. An entity's fieldability is defined in the entity type declaration.



Fields are attached to bundles and not to entity types. This is an important distinction that may save design or debugging time. Another important note is that fieldability can not be different for two bundles of the same entity type.

More details about fields is given in *Chapter 5, Developing with Fields*.

Drupal core entity structure

Entities exposed by Drupal core are comment, file, node, term, user, and vocabulary. Their structure is shown in the following table. Although in these entities fieldability does correlate with multiple bundle support, it is possible to have a fieldable single-bundle entity.

The following table shows Drupal core entity types, their fieldability, and whether they support multiple bundles:

Entity type	Fieldability	Multiple bundles
Comment	Y	Y
File	N	N
Node	Y	Y
Term	Y	Y
User	Y	Y
Vocabulary	N	N

Powerful entity use cases

The entity paradigm has allowed developers to expose their custom data to Drupal and utilize the full power of many Drupal solution modules, with minimal effort. The following are some use cases that have transformed immensely since entities were born:

- User profiles
- Internationalization
- Commerce products

User profiles


Drupal core, before Drupal 7, did not associate users with nodes. Site builders had to use either the core `profile` module or a contributed "profile-as-a-node" module, which tied users to nodes. All of these user profile modules had idiosyncrasies, limiting their scope or flexibility. To make any "profile-as-a-node" module useful to site builders, each had to implement their own rules, search, token, and views (solution module) integration, resulting in too much repetition.

Thanks to the fieldability of Drupal 7 user entities, a user account can be tied to any entity by a site builder (no code!) using an entity reference field. Through this reference field, solution modules will automatically be able to traverse from the user entity to the other entity regardless of the entity types. Because a single entity reference field can have multiple values, the site builder can create multiple profile types for different user facets. Solution modules will handle them all automatically! Developers exposing custom data to Drupal using entities will get all of this for free or with very little code.

Internationalization

At the time of writing, it would be terribly naïve for anyone to say that Drupal's internationalization (i18n) features are mature. The lack of innovation in this area can be largely attributed to the lack of a unified interface. The birth of entities opened the floodgates, and the development of i18n flourished in the Drupal meadows.

Adding an i18n support to a module is now quite trivial. These translation features are supported by the mainstream solution modules and multilingual Drupal sites have become much simpler to build.


 Internationalization of Drupal 7 entities requires the `entity_translation` module. At the time of writing, this module is in beta. All `entity_translation` features are slated to become part of core in Drupal 8!

Commerce products

Drupal Commerce was custom built for Drupal 7 using entities. Older code combined the visual representation of a product (description, images, and so on) and the product details such as **Stock-Keeping Unit (SKU)** and price. This made it difficult to support product combinations (multiple products per line item) and product variations (for example, size and color). The cart and checkout modules were custom forms, and customizing them required many lines of hook implementations and theme overrides. Other difficult features were taxes, discounts, and currencies. Along came bucket loads of contributed modules trying to support every possible feature combination, plus custom glue code to fill the gaps. The result: spaghetti!

Developers introduced new entities (products, orders, line items, and payments) along with new fields (price, product reference, and line item reference). Doing so exposed all data to solution modules and eliminated the many contribution modules that were previously needed for a basic e-commerce website. Code became simpler yet it was more flexible!

Our use case

Through out the course of this book, we will gradually update a recipe website, starting with a basic installation of Drupal 7 preconfigured with modules and content. For the recipe features, we will use a contribution module named `recipe`. The module is written as a node module – the design pattern in use before CCK even existed! In each chapter, we will gradually bring it closer to being a fully-fledged Drupal 7 module while we learn about programming Drupal entities using "Good Codes!"

Summary

In this chapter, we were introduced to entities, entity types, bundles, and fields with entity structure dissected. The entity and solution module categories were described, and we discovered how the entity paradigm makes code more flexible, useful, simple, and robust. Some powerful use cases were examined in before-and-after styles, to emphasize how powerful the entity paradigm is. Finally, you were introduced to the use case we will build as you progress through this book: a recipe website.

Next up, we will cook our first Drupal entity dish without burning it, because we have a super special spatula: entity metadata wrappers.

2

Developing with Entity Metadata Wrappers

Now that you've read the previous chapter, you know everything about entities, right? Absolutely! Now it's time to play with them by using some well designed object classes: entity metadata wrappers.

In this chapter we will cover the following topics:

- What entity metadata wrappers are
- Instantiate an entity metadata wrapper for an entity
- CRUD an entity
- Entity introspection
- Commonly used wrapper methods
- Safely using text property values

Introducing entity metadata wrappers

Entity metadata wrappers, or wrappers for brevity, are PHP wrapper classes for simplifying code that deals with entities. They abstract structure so that a developer can write code in a generic way when accessing entities and their properties. Wrappers also implement PHP iterator interfaces, making it easy to loop through all properties of an entity or all values of a multiple value property.

The magic of wrappers is in their use of the following three classes:

- `EntityStructureWrapper`
- `EntityListWrapper`
- `EntityValueWrapper`

The first has a subclass, `EntityDrupalWrapper`, and is the entity structure object that you'll deal with the most. Entity property values are either data, an array of values, or an array of entities. The `EntityListWrapper` class wraps an array of values or entities. As a result, generic code must inspect the value type before doing anything with a value, in order to prevent exceptions from being thrown.

Creating an entity metadata wrapper object

Let's take a look at two hypothetical entities that expose data from the following two database tables:

- `ingredient`
- `recipe_ingredient`

The `ingredient` table has two fields: `iid` and `name`. The `recipe_ingredient` table has four fields: `riid`, `iid`, `qty`, and `qty_unit`. The schema would be as follows:

ingredient		recipe_ingredient	
iid	bigint	riid	bigint
name	varchar	<i>rid</i>	bigint
		<i>iid</i>	bigint
		qty	decimal
		qty_unit	varchar

Schema for `ingredient` and `recipe_ingredient` tables

To load and wrap an `ingredient` entity with an `iid` of 1 and, we would use the following line of code:

```
$wrapper = entity_metadata_wrapper('ingredient', 1);
```

To load and wrap a `recipe_ingredient` entity with an `riid` of 1, we would use this line of code:

```
$wrapper = entity_metadata_wrapper('recipe_ingredient', 1);
```

Now that we have a wrapper, we can access the standard entity properties.

Standard entity properties

The first argument of the `entity_metadata_wrapper` function is the entity type, and the second argument is the entity identifier, which is the value of the entity's **identifying property**. Note, that it is not necessary to supply the bundle, as identifiers are properties of the entity type.

When an entity is exposed to Drupal, the developer selects one of the database fields to be the entity's identifying property and another field to be the entity's **label property**. In our previous hypothetical example, a developer would declare `iid` as the identifying property and `name` as the label property of the `ingredient` entity. These two abstract properties, combined with the `type` property, are essential for making our code apply to multiple data structures that have different identifier fields.

Notice how the phrase "type property" does not format the word "property"? That is not a typographical error. It is indicating to you that `type` is in fact the name of the property storing the entity's type. The other two, identifying property and label property are metadata in the entity declaration. The metadata is used by code to get the correct name for the properties on each entity in which the identifier and label are stored. To illustrate this, consider the following code snippet:

```
$info = entity_get_info($entity_type);
$key = isset($info['entity keys']['name'])
  ? $info['entity keys']['name'] : $info['entity keys']['id'];
return isset($entity->$key) ? $entity->$key : NULL;
```

Shown here is a snippet of the `entity_id()` function in the `entity` module. As you can see, the entity information is retrieved at the first highlight, then the identifying property name is retrieved from that information at the second highlight. That name is then used to retrieve the identifier from the entity.



Note that it's possible to use a non-integer identifier, so remember to take that into account for any generic code.

The label property can either be a database field name or a hook. The entity exposing developer can declare a hook that generates a label for their entity when the label is more complicated, such as what we would need for `recipe_ingredient`. For that, we would need to combine the `qty`, `qty_unit`, and the `name` properties of the referenced ingredient.

Entity introspection

In order to see the properties that an entity has, you can call the `getPropertyInfo()` method on the entity wrapper. This may save you time when debugging. You can have a look by sending it to `devel` module's `dpm()` function or `var_dump()`:

```
dpm($wrapper->getPropertyInfo());  
var_dump($wrapper->getPropertyInfo());
```

Using an entity metadata wrapper

The standard operations for entities are CRUD: create, retrieve, update, and delete. Let's look at each of these operations in some example code. The code is part of the `pde` module's Drush file: `sites/all/modules/pde/pde.drush.inc`.

Downloading the example code



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

Each CRUD operation is implemented in a Drush command, and the relevant code is given in the following subsections. Before each code example, there are two example command lines. The first shows you how to execute the Drush command for the operation.; the second is the `help` command.

Create

Creation of entities is implemented in the `drush_pde_entity_create` function.

Drush commands

The following examples show the usage of the `entity-create (ec)` Drush command and how to obtain help documentation for the command:

```
$ drush ec ingredient '{"name": "Salt, pickling"}'  
$ drush help ec
```

Code snippet

```
$entity = entity_create($type, $data);
// Can call $entity->save() here or wrap to play and save
$wrapper = entity_metadata_wrapper($type, $entity);
$wrapper->save();
```

In the highlighted lines we create an entity, wrap it, and then save it. The first line uses `entity_create`, to which we pass the entity type and an associative array having property names as keys and their values. The function returns an object that has `Entity` as its base class. The `save()` method does all the hard work of storing our entity in the database. No more calls to `db_insert` are needed!

Whether you use the `save()` method on the wrapper or on the `Entity` object really depends on what you need to do before and after the `save()` method call. For example, if you need to plug values into fields before you save the entity, it's handy to use a wrapper.

Retrieve

The retrieving (reading) of entities is implemented in the `drush_pde_print_entity()` function.

Drush commands

The following examples show the usage of the `entity-read (er)` Drush command and how to obtain help documentation for the command.

```
$ drush er ingredient 1
$ drush help er
```

Code snippet

```
$header = ' Entity (' . $wrapper->type();
$header .= ') - ID# ' . $wrapper->getIdentifier() . ':';
// equivalents: $wrapper->value()->entityType()
//               $wrapper->value()->identifier()

$rows = array();
foreach ($wrapper as $pkey => $property) {
    // $wrapper->$pkey == $property
    if (!(($property instanceof EntityValueWrapper)) {
        $rows[$pkey] = $property->raw()
```



```
        . ' (' . $property->label() . ' )';  
    }  
    else {  
        $rows[$pkey] = $property->value();  
    }  
}
```

On the first highlighted line, we call the `type()` method of the wrapper, which returns the wrapped entity's type. The wrapped `Entity` object is returned by the `value()` method of the wrapper. Using wrappers gives us the wrapper benefits, and we can use the `entity` object directly!

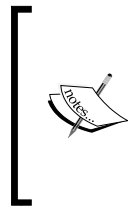
The second highlighted line calls the `getIdentifier()` method of the wrapper. This is the way in which you retrieve the entity's ID without knowing the identifying property name. We'll discuss more about the identifying property of an entity in a moment.

Thanks to our wrapper object implementing the `IteratorAggregate` interface, we are able to use a `foreach` statement to iterate through all of the entity properties. Of course, it is also possible to access a single property by using its key. For example, to access the `name` property of our hypothetical `ingredient` entity, we would use `$wrapper->name`.

The last three highlights are the `raw()`, `label()`, and `value()` method calls. The distinction between these is very important, and is as follows:

- `raw()`: This returns the property's value straight from the database.
- `label()`: This returns value of an entity's label property. For example, `name`.
- `value()`: This returns a property's wrapped data: either a value or another wrapper.

Finally, the highlighted `raw()` and `value()` methods retrieve the property values for us. These methods are interchangeable when simple entities are used, as there's no difference between the storage value and property value. However, for complex properties such as dates, there is a difference. Therefore, as a rule of thumb, always use the `value()` method unless you absolutely need to retrieve the storage value. The example code is using the `raw()` method only so we that can explore it, and all remaining examples in this book will stick to the rule of thumb. I promise!



- **Storage value:** This is the value of a property in the underlying storage media. for example, database.
- **Property value:** This is the value of a property at the entity level after the value is converted from its storage value to something more pleasing. For example, date formatting of a Unix timestamp.

Multi-valued properties need a quick mention here. Reading these is quite straightforward, as they are accessible as an array. You can use Array notation to get an element, and use a `foreach` to loop through them! The following is a hypothetical code snippet to illustrate this:

```
$output = 'First property: ' ;
$output .= $wrapper->property[0]->value();
foreach ($wrapper->property as $vwrapper) {
    $output .= $vwrapper->value();
}
```

Update

The updating of entities is implemented in the `drush_pde_entity_update()` function.

Drush commands

The following examples show the usage of the `entity-update (eu)` Drush command and how to obtain help documentation for the command:

```
$ drush eu ingredient 1 Salt
$ drush help eu
```

Code snippet

```
$wrapper->$pname = $pval;
$wrapper->save();
```

Updating an entity is very easy for simple properties, as can be seen in the preceding two highlighted lines—an assignment followed by a call to the wrapper's `save()` method. Complex properties can be trickier, and these are covered in the relevant chapters later.

Delete

The deletion of entities is implemented in the `drush_pde_entity_delete()` function.

Drush commands

The following examples show the usage of the `entity-delete (ed)` Drush command and how to obtain help documentation for the command:

```
$ drush ed ingredient 1
$ drush help ed
```

Code snippet

```
$wrapper->delete();
```

A single call to the wrapper's `delete()` method is all that's needed to zap an entity away.

Safely using text property values

Calls to the `value()` method on the wrapped text property values can pass an array of options for processing the value before it's returned.

```
$options = array('sanitize' => TRUE);  
$output1 = $entity->myproperty->value($options);  
$options = array('decode' => TRUE);  
$output2 = $entity->myproperty->value($options);
```

In the preceding example, `myproperty` is a text property. Only one of the two options, `sanitize` and `decode`, should be used at a time. If you set both to `TRUE`, the value will be `sanitize`. Both are `FALSE` by default.

When `sanitize` is set to `TRUE`, the text is passed through `check_plain`, which ensures that the text is ready to be displayed in HTML. It does so by converting characters such as angled brackets to HTML entities (nothing to do with Drupal entities). So, a less than symbol `>` becomes `<`. Use the `sanitize` option if the value will be fed directly into HTML output.

Setting the `decode` option is necessary when the value is HTML or PHP. This option will decode the string, by removing all HTML and PHP tags, and will then convert all HTML entities into their plaintext equivalents.

Self-imposed limitation of entity programming

There is one very important thing to mentally note down about entity programming. It is intended only for low-level data manipulation. Your humble author recommends that you do not use entity code for rendering content to a browser, because your code will not play well with modules or themes. If you wish to modify the rendering of some content, use Drupal's theme layer. The use of entity code within the theme layer is safe, that provided you put the code within the theme hooks.

References

- The `entity_metadata_wrapper()` function at http://drupalcontrib.org/api/drupal/contributions!entity!entity.module/function/entity_metadata_wrapper/7.
- The `Entity` class API at <http://drupalcontrib.org/api/drupal/contributions!entity!includes!entity.inc/class/Entity/7>.
- The `EntityMetadataWrapper` class API at <http://drupalcontrib.org/api/drupal/contributions!entity!includes!entity.wrapper.inc/class/EntityMetadataWrapper/7>.
- The `Entity` metadata wrappers from the Drupal handbook at <http://drupal.org/node/1021556>.

A note about EntityFieldQuery

The `EntityFieldQuery` class comes bundled with `entity` module and is a wonderful method of searching for queries. Unfortunately, your humble author accidentally omitted it from this book. To get started, take a look at the Drupal Handbook page at <https://drupal.org/node/1343708>.

Summary

In this chapter we discovered how easy `entity` coding can be, thanks to all of the cleverly designed wrapper classes. We now know what `entity` metadata wrappers are, how to CRUD entities by using the wrappers, and how to ensure that the data coming from entities is safe to use. Next, we will look at and play with non-fieldable entities in Drupal core!

3

Developing with Non-fieldable Entities

Up to this point, we have spent time learning about entities in general by using hypothetical entities. It is now time to play with some of the entity structures exposed by Drupal core. In this chapter we will cover the following:

- What non-fieldable entities are
- File entities
- Vocabulary entities
- Programmatically creating a file and a vocabulary
- Programmatically modifying a file and a vocabulary

What are non-fieldable entities?

So far in this book, we have only brushed on non-fieldable entities, so a little more detail is needed before we can appreciate them for what they are: structurally restricted data containers.

Wait a second! Aren't entities meant to be flexible so that we can extend them to our will?! Aren't fields one of the three necessary ingredients of a delicious entity sandwich?! Well, yes and no. Yes, we *do* want entities to be flexible and allow us to bend them to suit different use cases. However, some entities have no known use cases in which they should have fields, and some entities are better left non-fieldable until their full scope and supporting code are finalized. In Drupal 7 core, there are two such entities: vocabulary and file.

While you may be able to think of a use case in which, say, vocabulary entities would benefit from having fields, there just wasn't enough need to include support for that in Drupal 7. Keeping fields off these entities simplified the transition to the entity paradigm and reduced the time needed to make it all happen. That said, there is a contributed module named `file_entity`, working towards making file entities fieldable, among other features. This functionality, or part of it, will no doubt work its way into Drupal core eventually. At the time of writing, the earliest will be Drupal 9 as no developers have taken on the project.

For now, let's take a look at these two non-fieldable entity types in detail.



To get a quick look at an entity type's property information, you can use the `dump-entity-properties (dep)` Drush command in the `pde` module from the downloadable code.

File entities

Drupal 7 introduced some rather important changes for files. First and foremost, they became entities, although they are not fully-fledged entities. This is most likely due to how different they are to content. For example, at the time of writing, file entities don't have any write support through wrappers. All changes must be done using the `entity` object itself or the File API. The latter is the best option, as the File API is quite simple to use. The wrapper `save()` method still works if you truly wish to bypass the File API.

In your applications, you will more often read values from file entities than you will write them. For reading, we should use wrappers or entity properties. The following table shows the wrapper and entity properties of file entities in Drupal 7:

Wrapper property	Type (Read/Write)	Description	Entity property
fid	integer (R)	File ID	
name	string (R)	Name of the file	filename
mime	string (R)	MIME type of the file	filemime
size	integer (R)	Size of the file in kilobytes	filesize
url	string (R)	Web-accessible URL of the file	uri
timestamp	date/integer (R**)	Time of the most recent file update	
owner	user/integer (R)	User who originally uploaded the file	uid
	integer (R)	Status: temporary (0) or permanent (1)	status



The `timestamp` property can only be set by code directly changing the database record

Note that some of the wrapper property names differ from the entity property names (database field names). For those that differ, the entity property name has been placed in the right-hand side column. You will also notice that the values can be different between the wrapper property and the entity property; so can the type. For example, the `owner` wrapper property will be a user entity wrapper whereas the `uid` entity property is just a numeric user ID.

You'll also see that all properties are read-only on the wrapper. This is denoted by the **R** in the **Type** column. The `timestamp` field is also a special field because none of the API functions or wrapper code can alter the value that is stored in it. A developer must directly change the database record to make any change to this field. Finally, you'll notice the `status` property is totally missing from the wrapper.

Another concept introduced in Drupal 7 is the distinction between managed and unmanaged files. **Managed files** are known by Drupal and these are the entities that are most often used. **Unmanaged files** are useful in use cases where you want to do something outside of Drupal's API reach.

Creating a file entity by using the File API is typically done by using the `file_save_data()` function. The following code snippet comes from the `system_retrieve_file()` API function.

```
$local = $managed
? file_save_data($result->data, $path, $replace)
: file_unmanaged_save_data($result->data, $path, $replace);
```

Notice here that this function supports the creation of both managed and unmanaged files. Let's take a look inside `file_save_data`.

```
if ($uri = file_unmanaged_save_data($data, $destination, $replace)) {
  // Create a file object.
  $file = new stdClass();
  $file->fid = NULL;
  $file->uri = $uri;
  $file->filename = drupal_basename($uri);
  $file->filemime = file_get_mimetype($file->uri);
  $file->uid = $user->uid;
  $file->status = FILE_STATUS_PERMANENT;
  // If we are replacing an existing file re-use its database
  record.
  if ($replace == FILE_EXISTS_REPLACE) {
    $existing_files = file_load_multiple(array(), array('uri' =>
    $uri));
    if (count($existing_files)) {
      $existing = reset($existing_files);
      $file->fid = $existing->fid;
      $file->filename = $existing->filename;
    }
  }
  // If we are renaming around an existing file (rather than a
  directory),
  // use its basename for the filename.
  elseif ($replace == FILE_EXISTS_RENAME && is_file($destination)) {
    $file->filename = drupal_basename($destination);
  }

  return file_save($file);
}
```

The first highlight through to the second highlight shows the manual way of creating file entities. There is no use of `entity_create`. To create a file named `helloworld.txt` by using the entity, use the following code:

```
global $user;

$filename = 'helloworld.txt';
$uri = 'public://'.$filename;
$content = "Hello, world!\nI am Programming Drupal Entities!\n";

$uri = file_unmanaged_save_data($content, $uri, FILE_EXISTS_REPLACE);
$data = array(
  'fid' => NULL,
  'uri' => $uri,
  'filename' => drupal_basename($uri),
  'filemime' => file_get_mimetype($uri),
  'uid' => $user->uid,
  'status' => FILE_STATUS_PERMANENT,
);
$entity = entity_create('file', $data);
$wrapper = entity_metadata_wrapper('file', $entity);
$wrapper->save();
```

The preceding code provides a useful illustration but it is not really worth using anymore, as you can achieve much more with less code. In practice you would do the following instead:

```
$filename = 'helloworld.txt';
$uri = 'public://'.$filename;
$content = "Hello, world!\nI am Programming Drupal Entities!\n";
$file = file_save_data($content, $uri, FILE_EXISTS_REPLACE);
$wrapper = entity_metadata_wrapper('file', $file);
```

As you can see, we get a wrapper object in less lines by using the File API.

Seeing that updating status is impossible using wrappers, we must directly use the entity. Let's pretend that we want to flag the file for removal on a later cron run. This is done by setting the file entity's status to `FILE_STATUS_TEMPORARY`. The code would be as follows:

```
$wrapper = entity_metadata_wrapper('file', $fid);
$wrapper->value()->status = FILE_STATUS_TEMPORARY;
$wrapper->save();
```

Alternatively, we can use the entity object itself:

```
$entity = entity_load_unchanged('file', $fid);
$entity->status = FILE_STATUS_TEMPORARY;
$entity->save();
```

As a quick reminder, to delete an entity using a wrapper, you can use the following code:

```
$wrapper->delete();
```

Calling this on a file entity wrapper also deletes the file on disk.

Vocabulary entities

Vocabulary entities are a little strange. Developers opted to use the type name `taxonomy_vocabulary` instead of just `vocabulary` when they implemented the entity type in core. This is worth remembering so you don't get tripped up. The following table shows the properties of vocabulary entities in Drupal 7:

Wrapper property	Type (Read/Write)	Description	Entity property
vid	integer (R)	Vocabulary ID	
name	string (R/W)	Name	
machine_name	string (R/W)	Machine name	
description	string (R/W)	Description	
term_count	integer (R)	Number of terms in the vocabulary	N/A
N/A	integer	Hierarchy type: disabled (0), single (1), or multiple (2)	hierarchy
N/A	string	Module that created the vocabulary	module
N/A	integer	Weight of this vocabulary entity versus other vocabulary entities	weight

For most cases, the `hierarchy` property is best left untouched. The `taxonomy` module will automatically adjust this property to match how you organize terms using the UI.

To create a vocabulary named `cuisine` we would use the following code:

```
$data = array(
    'name' => 'Cuisine',
    'machine_name' => 'cuisine',
    'description' => 'Contains terms representing different cuisines.',
);
$entity = entity_create('taxonomy_vocabulary', $data);
$wrapper = entity_metadata_wrapper('taxonomy_vocabulary', $entity);
$wrapper->save();
```

The last three lines are quite ok if you plan on using the wrapper for other things. If you don't need to use a wrapper, these lines can be shrunk to the following line:

```
$status = taxonomy_vocabulary_save((object) $data);
```

API wins again! Let's put this into action in our recipe module overhaul.

Recipe site vocabularies

In its 7.0-1.x implementation, the `recipe` contribution module has no vocabularies by default. Site builders can easily add their own vocabularies to match their use case. However, it would be quite cool if the recipe module created two vocabularies out of the box: `cuisine` and `difficulty`.

We will now add code to the `recipe` module's install file. The code will create these vocabularies for new and existing sites. Open the `recipe.install` file (`sites/all/modules/customized/recipe/recipe.install`) and add the following code to the bottom of the file:

```
/**
 * Implements hook_install().
 */
function recipe_install() {
    recipe_install_vocabularies();
}

/**
 * Install default vocabularies introduced in 7.x-2.x.
 *
 * @return
 *   FALSE if the operation was successful otherwise the vocabulary
 *   machine_name that failed.
 */
```

```
function recipe_install_vocabularies() {
  $vocabularies = array(
    array(
      'name' => 'Cuisine',
      'machine_name' => 'cuisine',
      'description' =>
        'Contains terms representing different cuisines.',
    ),
    array(
      'name' => 'Difficulty',
      'machine_name' => 'difficulty',
      'description' =>
        'Contains terms representing difficulty levels.',
    ),
  );
  foreach ($vocabularies as $vdata) {
    // Make sure we're not overwriting existing vocabularies
    $v = taxonomy_vocabulary_machine_name_load($vdata['machine_
name']);
    if (!$v
      && taxonomy_vocabulary_save((object) $vdata) === FALSE) {
      // We got a problem
      return $vdata['machine_name'];
    }
  }



  return FALSE;
}

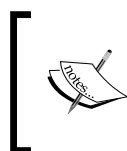
/**
 * Install default vocabularies introduced in 7.x-2.x.
 */
function recipe_update_7200(&$sandbox) {
  if ($machine_name = recipe_install_vocabularies()) {
    throw new DrupalUpdateException('Error occurred when attempting to
create vocabulary: '.$machine_name);
  }
}
```

The code we used was simply an install hook, a handler function, and an update hook. The highlighted line creates a vocabulary. Note that we used neither the entity code nor the wrapper code, just the good old core API. To see what you've done, scoot into your website document root using your shell and update the site database using Drush:

```
$ drush updatedb
```

Surf over to `admin/structure/taxonomy` and you'll see the new vocabularies. The properties of vocabulary entities in Drupal 7 are shown in the following screenshot:

Vocabulary name	Operations		
 Cuisine	edit vocabulary	list terms	add terms
 Difficulty	edit vocabulary	list terms	add terms



There may be times when you want to load a vocabulary by using its machine name. To do so, you can call the `taxonomy_vocabulary_machine_name_load()` function, and then feed the returned value into the `entity_metadata_wrapper()` function.

Updating a vocabulary can be done by using the standard CRUD techniques outlined in *Chapter 2, Developing with Entity Metadata Wrappers*. A quick reminder; for updates, use the following code:

```
$wrapper = entity_metadata_wrapper('taxonomy_vocabulary', $vid);
$wrapper->description = 'New description';
$wrapper->save();
```

For deletion, use the following code:

```
$wrapper = entity_metadata_wrapper('taxonomy_vocabulary', $vid);
$wrapper->delete();
```

That wraps up the core-exposed non-fieldable entities and how they can be used during development.

Summary

In this chapter, we took a look at non-fieldable entities exposed by Drupal core, learned that they are not fully-fledged entities, and also learned how to develop code using them. We then added some code to create vocabularies in our first change to the `recipe` contributed module.

Next, we will enjoy the aroma from a huge loaf of fieldable entities baking in Drupal core oven.

4

Developing with Fieldable Entities

Now we can sink our teeth into the main course: **fieldable entities**. In this chapter we'll cover the following:

- What are fieldable entities
- Node entities
- Comment entities
- Term entities
- Programmatically CRUD node, comment, and term entities

Let's get started!

What are fieldable entities?

By now you probably don't need an explanation, but in brief, fieldable entities are entities to which fields can be attached. In Drupal 7 core they are nodes, comments, and terms.

The cool thing about these fieldable entities is that they all have bundles, so you can have more flexibility. For example, consider the comments on different content types on the same site:

- Attach an image field to comments made on support issue posts
- Attach a rating field to comments on each product display page

As you can see, the content types are different, yet we are attaching a field to comments associated with those content types. Bundles! Very cool! Even more cool is the fact that all of this requires *zero* code!


Lucky for us we still have a job, because there's always some custom requirement. Let's break out the stand mixer, combine all of these fieldable entities, and bake some more "Good Codes!"

Node entities

Node entities are the bread and butter of Drupal – the content types around which sites are built. They are also the most complex core-exposed entities, because they have a large number of properties. Let's take a look at the wrapper and entity properties of node entities in Drupal 7:

Wrapper property	Type (Read/Write)	Description	Entity property
nid	integer (R)	Node ID	
vid	integer (R)	Revision ID	
is_new	string (R)	Whether or not the node is new	N/A
type	string (R/W)	Node type (bundle name)	
title	string (R/W)	Title	
language	token (R/W)	Language of the content	
url	string (R)	URL for viewing node	N/A
edit_url	string (R)	URL for editing node	N/A
status	integer (R/W)	Publishing status	
promote	Boolean (R/W)	TRUE to promote the node to the front page	
sticky	Boolean (R/W)	TRUE to display the node at the top of lists	
created	date/integer (R/W)	Date the node was created	
changed	date/integer (R)	Date the node was last changed	
author	user/integer (R/W)	Author	uid

Wrapper property	Type (Read/Write)	Description	Entity property
source	node/integer (R)	Original-language source node	N/A
log	text (R/W)	Log message for new revision	N/A
revision	Boolean (R/W)	TRUE if a new revision is to be created when the node is saved	N/A
comment	integer (R/W)	Comments allowed: no (0), closed/read-only (1), or open (2)	N/A
comment_count	integer (R)	Total number of comments posted	N/A
comment_count_new	integer (R)	Number of comments unseen by current the user	N/A
body	text_formatted (R/W)	Body of the node	

[ body is a special case because it is actually a field]

CRUD operations on node entities are the same as on other entities; only the `body` field needs thorough attention.

The value of `body` fields can be retrieved in the same way as for all other properties:

```
$body_value = $wrapper->body->value();
```

However, it won't return the content of the `body` field as we would expect. The `body` field has a **compound** type instead of a **scalar** type. A property or field with a scalar type will return a scalar value when you call the `value()` function. Scalar values are integers, strings, and things like that. A property or field with a compound type will return an array when you call the `value()` function.

Sending `$body_value` in the previous line through `print_r` would display the following:

```
Array
(
    [value] => Body of my node.
    [summary] => Summary of my node.
    [format] => plain_text
    [safe_value] => <p>Body of my node.</p>
    [safe_summary] => <p>Summary of my node.</p>
)
```

As you can see, there are sub-values within the body field. These are:

- `value`: Raw body content as saved in the database
- `summary`: Raw summary content as saved in the database
- `format`: Input format of the body
- `safe_value`: Computed body content that is HTML safe
- `safe_summary`: Computed summary content that is HTML safe

We could use these values directly by using code similar to this:

```
$wrapper->body->value->value()
```

Developers have a choice to use either a raw output or a sanitized (HTML safe) output. We will refer to this as the **processing type**. Unfortunately, these outputs are accessed inconsistently throughout the different field types. For example, one field type returns raw output for a call to `value()` whereas a different field type outputs sanitized output. Wipe the sweat off your brow and take a deep breath. There's a way around it, and those of you who guess it right get a candy bar.

We can get what we want by calling `value()` and giving it an option to specify the processing type. For each processing type the code is as follows:

- raw: `$wrapper->body->summary->value(array('decode' => TRUE));`
- sanitized: `$wrapper->body->summary->value(array('sanitize' => TRUE));`

So, that was easy right? Indeed it was. That's all of the tricks necessary if your code deals with a known entity type or field type. If you need to do anything generic, write code targeting the field type, as the field type must declare a fixed set of property names and types.

Don't forget though...programming at the entity-level should have nothing to do with output rendering. Entity-level code must only deal with low-level data. Rendering for the browser is always handled by the theme layer!

Comment entities

Drupal provides comment entities to allow users to attach comments to node entities. Let's take a look at the wrapper and entity properties of comment entities in Drupal 7:

Wrapper property	Type (Read/Write)	Description	Entity property
cid	integer (R)	Comment ID	
hostname	string (R)	IP address of the posting computer	
name	string (R/W)	Author's name	N/A
mail	string (R/W)	Author's e-mail address	
homepage	string (R/W)	Author's homepage	
subject	string (R/W)	Subject of the comment	
url	string (R)	URL for viewing comment	N/A
edit_url	string (R)	URL for editing comment	N/A
created	date/integer (R/W)	Date comment was created	
parent	comment/integer (R)	Parent comment's ID	pid
node	node/integer (R/W)	Node the comment was posted to	nid
author	user/integer (R/W)	Author	uid
status	integer (R/W)	Published status: no (0) or yes (1)	
comment_body	text_formatted (R/W)	Content of the comment	Comment body is a special case because it is actually a field

The `comment_body` field is similar to the `body` field in node entities. The only difference is that it doesn't have a summary. Refer to the previous section for information about that.

Now that we've covered both node and user entities, we can take a look at the entity reference properties of a comment entity: `author` and `node`. When using wrappers, the `author` property becomes a user wrapper and the `node` property becomes a node wrapper. Via these, you can easily access properties of the author's user account or the node to which the comment was posted. Some examples are as follows:

```
$mail = $wrapper->author->mail->value();
$title = $wrapper->node->title->value(array('decode' => TRUE));
```

The first line in the preceding code returns the author's e-mail address (if they are a registered user), and the second line returns the node title. Changing the author is simple:

```
$wrapper->author = $new_author_uid;
$wrapper->save();
```

Comments can also be posted by anonymous users. In this case, the value of the `uid` field is zero. To check that, do the following:

```
if ($wrapper->author->raw() === '0') {
    $mail = $wrapper->mail->value(array('decode' => TRUE));
    $name = $wrapper->name->value(array('decode' => TRUE));
}
```

An important thing to note here is that `raw` actually returns the user's ID as a string value. The underlying field is an integer, but the wrapper code turns this into a string. That is the reason for the quotes and the use of `===` for the equality comparison.

Anonymous users will have values in the `name` and `mail` properties, as they don't have a user account to refer to. The code that accesses these properties are highlighted in the previous code snippet.

Term entities

Similar to the vocabulary entities, terms have a special type name: `taxonomy_term`. The properties of `taxonomy_term` entities are shown in the following table:

Wrapper property	Type (Read/Write)	Description	Entity property
<code>tid</code>	integer (R)	Term ID	
<code>name</code>	string (R/W)	Name	
<code>description</code>	string (R/W)	Description	

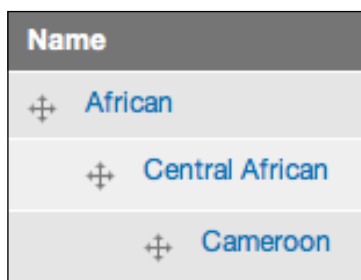
Wrapper property	Type (Read/Write)	Description	Entity property
weight	integer (R/W)	Weight; value for the order of terms	
node_count	integer (R)	Number of nodes tagged with the term	
url	uri/string (R)	URL	N/A
vocabulary	taxonomy_vocabulary/ integer (R/W)	Vocabulary	vid
parent	list<taxonomy_term> (R/W)	Parent terms	N/A
parents_all	list<taxonomy_term> (R/W)	Ancestor terms	N/A

Properties of `taxonomy_term` entities are quite straightforward. Notable properties are `vocabulary`, `parent`, and `parents_all`.

We can use `vocabulary` to chain through to the `taxonomy_vocabulary` entity containing this term.

```
$v_name = $wrapper->vocabulary->name->value(array('decode' => TRUE));
```

Code carefully when using the `parent` and `parents_all` properties. The following is some sample code that assumes a hierarchy of cuisines that looks like the following:



Example cuisine term hierarchy

The `parents` property only contains the direct parents of the term. You read that correctly—it is a plural. Terms in Drupal can have multiple parents, so a term can live in two parts of the hierarchy. On the other hand, the `parents_all` property contains the entire term path to the vocabulary root including the term itself. To illustrate this, take a look at the following code and output.

```
$wrapper = entity_metadata_wrapper('taxonomy_term', $cameroon_tid);
foreach ($wrapper->parent as $pwrapper) {
  print($pwrapper->name->value())."\n";
}
print "\n";
foreach ($wrapper->parents_all as $pwrapper) {
  print($pwrapper->name->value())."\n";
}
/* OUTPUT:
Central African

Cameroon
Central African
African
*/
```

Our recipe website needs to have some terms installed for the new vocabularies that we created in the previous chapter. To do so, we amend the code as follows:

```
/**
 * Implements hook_install().
 */
function recipe_install() {
  recipe_install_vocabularies();
  recipe_install_terms();
}
```

Add the highlighted line to the `recipe_install()` function. Next, paste the following code at the bottom of the file.

```
/**
 * Installs recipe module's default terms that are read from
 * text files in the module's includes folder.
 */
function recipe_install_terms() {
  foreach (array_keys(recipe_vocabularies()) as $machine_name) {
    $v = taxonomy_vocabulary_machine_name_load($machine_name);
    $wrapper = entity_metadata_wrapper('taxonomy_vocabulary', $v);
```

```

    if ($wrapper->term_count->value() == 0) {
        $path = drupal_get_path('module', 'recipe')
            . '/includes/terms_' . $v->machine_name . '.txt';
        $lines = file($path, FILE_SKIP_EMPTY_LINES);
        recipe_install_term_tree($wrapper, $lines);
    }
}
}

/**
 * Installs a term tree.
 * @param $vwrapper
 *   EntityMetadataWrapper of a taxonomy_vocabulary entity.
 * @param $lines
 *   Array of lines from the term text file. The iterator must be set
 *   to the line to parse.
 * @param $last
 *   Either NULL or the parent term ID.
 * @param $depth
 *   Current depth of the tree.
 */
function recipe_install_term_tree($vwrapper, &$lines,
                                $last = NULL, $depth = 0) {

    $wrapper = NULL;

    while ($line = current($lines)) {
        $name = trim($line);
        $line_depth = max(strlen($line) - strlen($name) - 1, 0);

        if ($line_depth < $depth) {
            return;
        }
        else if ($line_depth > $depth) {
            $tid = $wrapper ? $wrapper->tid->value() : NULL;
            recipe_install_term_tree($vwrapper, $lines, $tid, $depth+1);
        }
        else {
            $data = array(
                'name' => $name,
                'vid' => $vwrapper->vid->value(),
                'parent' => array($last ? $last : 0),
            );

```



```
        $term = entity_create('taxonomy_term', $data);
        $wrapper
            = entity_metadata_wrapper('taxonomy_term', $term);
        $wrapper->save();
        next($lines);
    }
}

/**
 * Installs terms into default vocabularies.
 */
function recipe_update_7201(&$sandbox) {
    recipe_install_terms();
}
```

In the preceding code, term names are read from text files that have tab indentation to symbolize the term hierarchy. A snippet of the `terms_cuisine.txt` file is shown as follows:

```
African
  Central African
    Cameroon
    Congo
  East African
    Burundi
    Kenya
    Maasai
    Tanzania
    Uganda
```

Each line of the file is analyzed and the creation of a term entity occurs in the highlighted lines. All that's needed for the correct creation of a term entity is to specify the name, vid, and parent properties. For terms without a hierarchy, you need to set the parent term ID to zero. If you don't provide the parent term like this, you will not get an error, but the term will not be saved properly. In addition, you can specify a description and a text format, as shown in the following code snippet:

```
$data = array(
  'name' => $name,
  'vid' => $vid,
  'parent' => array(0),
  'description' => $description,
  'format' => 'plain_text',
);
```

```
$term = entity_create('taxonomy_term', $data);
entity_save('taxonomy_term', $term);
```

You'll notice that `entity_save` has been used instead. This is an alternative if you don't need the wrapper.

Once you have added the previous code, open up your terminal and navigate to your website document root folder and update the site database by using Drush:

```
$ drush updatedb
```

Navigate your browser to `admin/structure/taxonomy/cuisine` and you'll see the new terms in the Cuisine vocabulary, as shown in the following screenshot:

Name	Operations
+ African	edit
+ Central African	edit
+ Cameroon	edit
+ Congo	edit
+ East African	edit
+ Burundi	edit
+ Kenya	edit
+ Maasai	edit
+ Tanzania	edit
+ Uganda	edit
+ Horn African	edit
+ Eritrea	edit
+ Ethiopia	edit
+ Somalia	edit
+ North African	edit
+ Algeria	edit
+ Egypt	edit
+ Libya	edit
+ Morocco	edit
+ Sudan	edit
+ Tunisia	edit

Cuisine terms after updating the database with the new recipe module updates

That wraps up everything for core-exposed fieldable entities.

Summary

This chapter delved into the fieldable entities exposed by Drupal 7 core: `comment`, `node`, and `term` entities. We also added some code to our recipe website that installs default terms into the `Cuisine` and `Difficulty` vocabularies.

In the next chapter, we will whizz up some flexibility with fields.

5

Developing with Fields

Now that we've covered the core-exposed entities, we can unveil the most powerful part of the Drupal entity paradigm: fields. In this chapter we'll look at the following:

- Different field types: date, file, image, link, number, text, and term reference
- The difference between multi-value and single-value fields
- Structure fields
- Programmatically accessing the fields of an entity
- Programmatically adding fields to an entity
- Programmatically migrating data into fields

Field types

We will look at the most commonly-used field types in Drupal 7: text, numeric, date, link, file, and image in the following table:

Field type	Type name	Core/Contributed	Module name
date	datetime	contributed	date
date (ISO format)	date	contributed	date
date (UNIX timestamp)	datestamp	contributed	date
decimal	number_decimal	core	number
file	file	core	file
float	number_float	core	number
image	image	core	image

Field type	Type name	Core/Contributed	Module name
Integer	number_integer	core	number
link	link	contributed	link
Long text	text_long	core	text
Long text and summary	text_with_summary	core	text
Text	text	core	text

The previous table displays field types, whether it's in core or in a contributed module, and the name of the Drupal 7 module in which the field is implemented.

You might be wondering why `options` and `list` modules are not included in the previous table. It is because they are not field types, and their responsibility is to provide widgets on the entity edit forms. Remember to differentiate the widget you see from the underlying field type when coding.

Single-value and multi-value fields

Fields can either be single-value or multi-value fields. This means that they either store one value or multiple values. Through wrappers, most single-value fields are easy to code using techniques already discussed for entity properties. On the other hand, multi-value fields are arrays and need to be dealt with using either array notation or through the use of iterators (for example, `foreach` loops).

When coding with any field, it's important to determine whether it is single-value or multi-value before proceeding. This could be done by using the following code snippet:

```
$field_name = 'field_blah';
if (is_array($wrapper->$field_name->value())) {
  // ... code for a multi-value field
}
```

However, there are fields, where the previous code will give you a false positive. For example, the body field of a node. As a result, a more robust approach is needed. To do so we can employ the Field API as follows, assuming `$field_name` is already valid:

```
$field_info = field_info_field($field_name);
if (isset($field_info['cardinality'])
    && $field_info['cardinality'] != 1) {
  // ... code for a multi-value field
}
```

Multi-value fields have a cardinality of either -1, for unlimited values, or the number of values stored. Thus, anything other than one will be a multi-value field. There is another, far easier way to check for multi-value fields: inspecting the field wrapper class. A field wrapper class is the wrapper of a field, and is accessed by using `$wrapper->field`. For multi-value fields, the class will be `EntityListWrapper`. To check for multi-value fields we use the `instanceof` operator. In the following snippet we assume again that `$field_name` is already valid.

```
if ($wrapper->$field_name instanceof EntityListWrapper) {
    // ... code for a multi-value field
}
```

A field wrapper class can either be an `EntityListWrapper`, an `EntityDrupalWrapper`, an `EntityStructureWrapper`, or an `EntityValueWrapper` object instance. `EntityListWrapper` is essentially an array of wrappers. `EntityDrupalWrapper` is an entity wrapped up. `EntityStructureWrapper` wraps anything that isn't an entity and has more than one property. `EntityValueWrapper` wraps a single value.

The first part of the code is as follows:

```
function pde_field_value($field_wrapper) {
    try {
        if ($field_wrapper instanceof EntityListWrapper) {
            // Handle EntityListWrapper multi-value fields
            $output = array();
            foreach ($field_wrapper as $value) {
                $output[] = pde_field_value($value);
            }
        }
        else if ($field_wrapper instanceof EntityDrupalWrapper) {
            $output = pde_entity_value($field_wrapper);
        }
        else if ($field_wrapper instanceof EntityStructureWrapper) {
            $output = pde_structure_value($field_wrapper);
        }
        else {
            $output = pde_output(
                $field_wrapper->value(array('decode' => TRUE)));
        }
    }
}
```

```
        catch (EntityMetadataWrapperException $e) {
            return '';
        }
        return $output;
    }

    function pde_output($msg) {
        return $msg . "\n";
    }
```

The highlighted code checks for an `EntityListWrapper` instance by using the `instanceof` operator. As we discovered earlier, this catches multi-value fields, and the code iterates through each of the field's values, calling itself with each one (`$value`).

After that, you'll notice two more checks for object instances, then an `else` clause. The `else` clause at the end prints the value of the `EntityValueWrapper` objects by calling the `value()` method. Before this `else` clause, the object instance check blocks call handler functions that print an entity or a structure, respectively.

The second highlighted line catches `EntityMetadataWrapperException` exceptions thrown when a field does not have a value. This allows us to gracefully handle the situation.

Structure fields

Now that we've successfully distinguished single-value and multi-value fields, we have to get around one last, very nasty trap before we can cook code with wrapped fields. The trap: some fields are neither lists of values nor just values. For these fields we can't use the `value()` method directly and have to resort to other means. We will call these types of fields **structure fields**.

Structure field wrappers are either of the `EntityDrupalWrapper` class or the `EntityStructureWrapper` class. The former class means that the field wrapper is wrapping an entity. In our example, code for printing `EntityDrupalWrapper` values would look as follows:

```
function pde_entity_value($entity_wrapper) {
    return pde_output($entity_wrapper->label());
}
```

We are relying on the entity's `label()` method to provide the appropriate output. All CRUD of the entity are the same as we covered in earlier chapters.

You'll notice in the code snippet of `pde_field_value` that the check for `EntityDrupalWrapper` is done before `EntityStructureWrapper`. This is important because `EntityDrupalWrapper` is a subclass of `EntityStructureWrapper` and a wrapped entity will be an instance of both the classes.

Field type-specific code

Remember at the beginning of your entity journey, you learned that entities reduce repetitive code to a minimum and that's what makes them so cool? Code snippets in this chapter have highlighted this characteristic quite well. Unfortunately, there is some unfinished business in Drupal's entity implementation and, as a result, we have to resort to some old school code to handle these special cases. Fortunately, it will be easy to write because there are so few!

Of all the fields that we are looking at in this chapter, only the following five underlying field types are in need of special handling:

- `datetime`
- `file`
- `link`
- `image`
- `text_with_summary`

Each of these is a wrapper of the `EntityStructureWrapper` class. All the other field types are a wrapper of the `EntityValueWrapper` class. The following table shows these field types and their associated **wrapper types**, which is returned by calling the `type()` method on the field's wrapper:

Field type	Wrapper type
<code>datetime</code>	<code>struct</code>
<code>file</code>	<code>field_item_file</code>
<code>link</code>	<code>field_item_link</code>
<code>image</code>	<code>field_item_image</code>
<code>text_with_summary</code>	<code>text_formatted</code>

We've already covered `text_formatted` fields in depth, when we looked at `node` body and comment body properties. Another cool thing is the handling for `file` and `link` fields is the same. Let's take a peek at the `file/image`, `link`, and `datetime` fields in some code snippets, followed by the whole `pde_structure_value()` function and its underlings.

File and image fields

File and image fields are only here because they missed the "implemented as EntityDrupalWrapper" boat and gained a level of indirection. If we were to call the `getPropertyInfo()` method on the field wrapper, we would see that there is a `file` property and the property type is `file`. When the property type matches an entity type name, we can start to get excited and try one more thing: call the `get_class` PHP function, passing it the property. Behold! It's an `EntityDrupalWrapper` class!

The `file` property of either `field_item_file` or `field_item_image` wrappers is a file entity wrapper so, in our example, we can call the previously mentioned `pde_entity_value()` function and pass the file property, as shown in the following line of code:

```
pde_entity_value($struct_wrapper->file);
```

We can instead throw it back to `pde_field_value`, as given in the following line of code:

```
pde_field_value($struct_wrapper->file);
```

Link fields

Next in order of complexity are link fields. Calling `getPropertyInfo()` on these will show you that they have `title` and `url` text properties. In our example, we will print a HTML link by using the following code:

```
$url = $struct_wrapper->url->value(array('decode' => TRUE));
$title = $struct_wrapper->title->value(array('decode' => TRUE));
if (empty($title)) {
    $title = $url;
}
return pde_output(l($title, $url));
```

Datetime fields

Fields of type `datetime` are quite complex and care must be taken to ensure that things work as expected. Inspecting the **wrapper properties** and **struct properties** shows us how to access all of the data. A wrapper property is a property of the wrapper and is accessed directly from the wrapper. A struct property is a property of the data returned by the `value()` method of the field wrapper.

Wrapper properties	Struct properties
value	value
value2	value2
duration	
	timezone
	timezone_db
	date_type

The following code shows us how to retrieve date and time:

```
$output = $date_wrapper->value->value(array('decode' => TRUE));
if ($date_wrapper->duration->value(array('decode' => TRUE))) {
    $output .= ' - '
        . $date_wrapper->value2->value(array('decode' => TRUE));
}

// $date_wrapper->value() returns the array of data available
$date_value = $date_wrapper->value();
$output .= ' ' . $date_value['timezone'];
```

The first line retrieves the start date, `value`; the second line consults the `duration` to see if it's necessary to print the end date, `value2`. These are all accessed by using the wrapper properties just named. The last line outputs the `timezone` string, which comes from a struct property. Notice that we have to retrieve the structure first, by using the `value()` method before this.

Putting it all together

Here's the `pde_structure_value()` function, and the rest of the code needed to finish off the example. The code can be found in the `sites/all/modules/pde/pde/pde.drush.inc` file.

```
function pde_structure_value($struct_wrapper) {
    $field_type = $struct_wrapper->type();

    switch ($field_type) {
        case 'field_item_link':
            $url = $struct_wrapper->url->value(array('decode' => TRUE));
            $title
                = $struct_wrapper->title->value(array('decode' => TRUE));
            if (empty($title)) {
                $title = $url;
            }
            return pde_output(1($title, $url));
        case 'field_item_image':
        case 'field_item_file':
            // Special case!
            // File entity: $field_wrapper->file
            return pde_field_value($struct_wrapper->file);
        case 'struct':
            return pde_struct_value($struct_wrapper);
        case 'text_formatted':
            return pde_output(
                $struct_wrapper->value->value(array('decode' => TRUE)));
        default:
            throw new Exception(
                'No idea how to handle structure type ' . $field_type);
    }
}

function pde_struct_value($struct_wrapper) {
    $struct_type = $struct_wrapper->value->type();
    switch ($struct_type) {
        case 'date':
            return pde_date_value($struct_wrapper);
        default:
            throw new Exception(
                'No idea how to handle struct type ' . $struct_type);
    }
}
```

```

function pde_date_value($date_wrapper) {
    $output = $date_wrapper->value->value(array('decode' => TRUE));
    if ($date_wrapper->duration->value(array('decode' => TRUE))) {
        $output .= ' - '
            . $date_wrapper->value2->value(array('decode' => TRUE));
    }

    // $date_wrapper->value() returns the array of data available
    $date_value = $date_wrapper->value();
    $output .= ' ' . $date_value['timezone'];

    return pde_output($output);
}

```

In the `pde_structure_value()` function, we switch on the field wrapper type and either funnel execution to a type handler or render the output in place. Links are output as HTML anchor tags. The `pde_struct_value()` function could be deemed as bloat, as date is the only struct type field we've dealt with. It's there for possible expansion.

This is a simple, yet illustrative example of the data structures in play throughout the common field implementations. There are differences that can make coding tedious at times. Hopefully, with the previous information, you will be able to bake your Drupal pie without it exploding in your oven— or in your face!

Converting the recipe content type to use fields

Our conversion of the recipe module to use fully-fledged entities can continue now, and we will convert all of the database fields in the recipe table to entity fields. Converting the structure is only a part of the task. There are many code changes necessary to ensure that all supporting code correctly manipulates the new structure. The steps we will take to upgrade the module code are as follows:

1. Create the fields using the UI.
2. Export them to code inside a feature.
3. Put the exported declarations into recipe module and convert field name prefixes to `recipe_`.

4. Tweak the recipe module code so that the old fields don't clobber the new ones in the field admin UI, and add `link`, `number`, `text`, and `token` modules as dependencies.
5. Add the update code in order to attach the new fields to recipe nodes, and then for any existing recipe nodes copy data from the recipe table into the new fields.

Creating fields

Create the fields in the same way as you would when building any content type. Surf to `admin/structure/types/manage/recipe/fields` and add each field. When creating the field type, the label and the field name will be the same for all fields except the `Yield units`, `Preparation time`, `Cooking time`, and `Additional notes` fields. Make sure that you change these to `field_yield_unit`, `field_preptime`, `field_cooktime`, and `field_notes`, respectively.

The following table lists the new field specifications for recipe module. You'll also notice that the field type in parentheses next to the field's machine name and under the label will be any non-default settings that you need to specify.

Old field name	New field specification
<code>recipe_description</code>	<code>field_description</code> (Long text) Label: Description
<code>recipe_yield</code>	<code>field_yield</code> (Decimal) Label: Yield Minimum: 0
<code>recipe_yield_unit</code>	<code>field_yield_unit</code> (Text) Label: Yield units
<code>recipe_notes</code>	<code>field_notes</code> (Long text) Label: Additional notes
<code>recipe_source</code>	<code>field_source</code> (Text) Label: Source
N/A	<code>field_source_link</code> (Link) Label: Source link Link Title: Static Title Static Title: <code>[node:field_source]</code>

Old field name	New field specification
recipe_instructions	field_instructions (Long text) Label: Instructions
recipe_preptime	field_preptime (Integer) Label: Preparation time Minimum: 0 Suffix: minutes
recipe_cooktime	field_cooktime (Integer) Label: Cooking time Minimum: 0 Suffix: minutes

We have added one extra field to ease linking to external web sources: `recipe_source_link`. This field uses a token to automatically take the link title from the `Source` field.

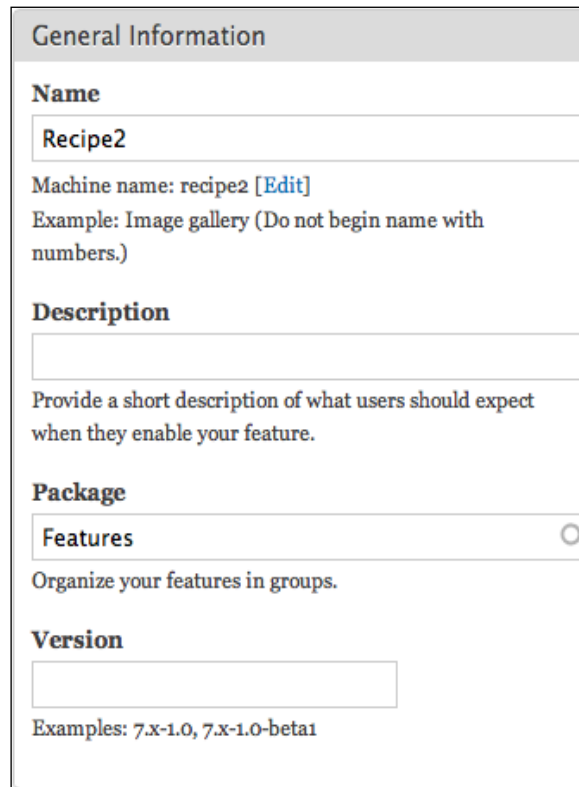
Once you're done adding the fields, you'll have a huge number of fields showing. These will look as shown in the following screenshot:

Label	Machine name	Field type	Widget	Operations	
+ Title	title	Node module element			
+ Description	field_description	Long text	Text area (multiple rows)	edit	delete
+ Yield	field_yield	Decimal	Text field	edit	delete
+ Yield units	field_yield_unit	Text	Text field	edit	delete
+ Additional notes	field_notes	Long text	Text area (multiple rows)	edit	delete
+ Source	field_source	Text	Text field	edit	delete
+ Source link	field_source_link	Link	Link	edit	delete
+ Instructions	field_instructions	Long text	Text area (multiple rows)	edit	delete
+ Preparation time	field_preptime	Integer	Text field	edit	delete
+ Cooking time	field_cooktime	Integer	Text field	edit	delete
+ Description	recipe_description	Recipe module element			
+ Yield	recipe_yield	Recipe module element			
+ Yield units	recipe_yield_unit	Recipe module element			
+ Ingredients	recipe_ingredients	Recipe module element			
+ Additional notes	recipe_notes	Recipe module element			
+ Source	recipe_source	Recipe module element			
+ Instructions	recipe_instructions	Recipe module element			
+ Preparation time	recipe_preptime	Recipe module element			
+ Cooking time	recipe_cooktime	Recipe module element			

Manage fields page of recipe content type after fields are added

Exporting fields to a feature

Surf over to `admin/structure/features/create` and enter information into the **General Information** fieldset at the top, so that it matches this screenshot:



The screenshot shows a form titled "General Information" with several sections:

- Name**: A text input field containing "Recipe2". Below it, the text "Machine name: recipe2 [Edit]" and "Example: Image gallery (Do not begin name with numbers.)" are displayed.
- Description**: A text input field. Below it, the text "Provide a short description of what users should expect when they enable your feature." is displayed.
- Package**: A dropdown menu with "Features" selected. Below it, the text "Organize your features in groups." is displayed.
- Version**: A text input field. Below it, the text "Examples: 7.x-1.0, 7.x-1.0-beta1" is displayed.

Top fieldset of the create feature page

We are calling the feature `Recipe2`, and it will automatically have a machine name of `recipe2`. Now you can move to the **Components** fieldset and expand **Field instances (field_instance)** by clicking on the text. Select the checkbox in front of each field beginning with `node-recipe-recipe_`. Each time that you select a checkbox, it is wise to wait a moment for the dependencies to be automatically selected for you and for the UI to be updated before selecting the next one. The UI moves around a little and clicking too fast may result in unwanted checkboxes being selected. Once you are done, it will look as shown in the following screenshot:

Components

Expand each component section and select which items should be included in this feature export.

Search Clear

☐ Select all

► Dependencies (dependencies)

☒ Features ☒ Link ☒ Number

► Field Bases (field_base)

☒ field_cooktime

☒ field_description

☒ field_instructions

☒ field_notes

☒ field_preptime

☒ field_source

☒ field_source_link

☒ field_yield

☒ field_yield_unit

▼ Field Instances (field_instance)

☒ node-recipe-field_cooktime

☒ node-recipe-field_description

☒ node-recipe-field_instructions

☒ node-recipe-field_notes

☒ node-recipe-field_preptime

☒ node-recipe-field_source

☒ node-recipe-field_source_link

☒ node-recipe-field_yield

☒ node-recipe-field_yield_unit

Component selection for the feature export

Click on the **Download feature** button and you will download a TAR file containing the feature. Don't download or extract the feature TAR file into the development site because it may cause havoc!

Copying the code to the recipe module

In your terminal, go to the folder into which the feature was downloaded and extract the feature file by using the following command:

```
$ tar xf recipe2.tar
```

Getting a directory listing of the feature's folder, you'll see files named `recipe2.features.field_base.inc` and `recipe2.features.field_instance.inc`. Copy `recipe2.features.field_base.inc` into the recipe module's folder and rename the copy to `recipe.field.inc`. The recipe module folder can be found at `sites/all/modules/customized/recipe` inside your development site's document root. Now copy the `recipe2_field_default_field_instances()` function in `recipe2.features.field_instance.inc` to the bottom of `recipe.field.inc`.

Open `recipe.field.inc` in your editor and you'll see the following code:

```
<?php
/**
 * @file
 * recipe2.features.field_base.inc
 */

/**
 * Implements hook_field_default_field_bases().
 */
function recipe2_field_default_field_bases() {
  $field_bases = array();
  ...
  // Exported field_base: 'field_cooktime'
  $field_bases['field_cooktime'] = array(
    'active' => 1,
    'cardinality' => 1,
    'deleted' => 0,
    'entity_types' => array(),
    'field_name' => 'field_cooktime',
    'foreign keys' => array(),
    'indexes' => array(),
    'locked' => 1,
    'module' => 'number',
    'settings' => array(
      'entity_translation_sync' => FALSE,
    ),
    'translatable' => 0,
    'type' => 'number_integer',
  );
  ...
}
```

Highlighted in the previous snippet, you'll see there are three instances of the cooking time field's machine name, `field_cooktime`. Machine names for all fields need to have their prefixes changed from `field_` to `recipe_` plus, we need to lock the fields so that they can't be deleted. This will require the following four search and replace operations:

- `-field_ to -recipe_`
- `> 'field_ to > 'recipe_`
- `:field_ to :recipe_`
- `'locked' => 0 to 'locked' => 1`

In the second search and replace directive be sure to put the space between `>` and `'`. Now remove the 2 from both function names, and then save the file and we're done!

Tweaking `recipe.module` and `recipe.info`

Add the following lines to `recipe.info` so that all dependencies are installed:

```
dependencies[] = link
dependencies[] = number
dependencies[] = text
dependencies[] = token
```

Tweaks to `recipe.module` are quite vast as there are many places where the old implementation used the fields directly. For now, we will just make two edits. The first is to place the following code at the top of `recipe_field_extra_fields()` function:

```
if (variable_get('recipe_fields_installed', FALSE)) {
  return array(
    'node' => array(
      'recipe' => array(
        'form' => array(
          'recipe_ingredients' => array(
            'label' => t('Ingredients'),
            'description' => t('Recipe module element'),
            'weight' => -3,
          ),
        ),
      ),
    'display' => array(
      'recipe_ingredients' => array(
        'label' => t('Ingredients'),
```

```
        'description' => t('Recipe module element'),
        'weight' => -3,
    ),
),
),
),
),
);
}
```

The `recipe_field_extra_field()` function notifies Drupal about custom lines, which the module wants in the manage field table or manage display UI. This allows an administrator to reorder any custom widgets on the node edit form and node display. Custom widgets can be injected into forms in a `hook_form_alter` implementation. What we've done previously is to still show the ingredients custom field on those pages since we are not overhauling that at this point. The first line checks the schema version of recipe module to see if the updates have been performed. We create the update code in the next step.

Our final edit to recipe module is to add the highlighted key/value pair to the `recipe_node_info()` function, so that it now looks as shown in the following code:

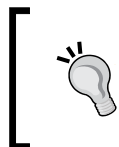
```
/**
 * Implementation of hook_node_info().
 */
function recipe_node_info() {
  return array(
    'recipe' => array(
      'name' => t('Recipe'),
      'base' => 'recipe',
      'description' => t('Share your favorite recipes with your fellow
        cooks.'),
      'locked' => TRUE,
    )
  );
}
```

The `locked` directive prevents an administrator from changing the content type's machine name or deleting the content type. At this point we won't audit the module to check the code can handle a machine name change. Locking the node type will keep our heads above water if the code is unsafe.

We also replace the `recipe_form` hook implementation to create the node edit form correctly and also add `recipe_form_recipe_node_form_alter`, which alters the node form by injecting the ingredients fieldset and widgets in much the same way as `recipe_form` did previously.

You can see the code changes in `recipe.module`, when you check out the `chapter_05` branch. It might be handy to make a copy of the file from `chapter_04` and run a diff to see the changes. In particular, notice the call to `node_content_form` in `recipe_form()` function.

We also need to amend the `recipe_load()` function to no longer load values into the node objects.



For a challenge, try to find the changes made to `recipe_load()` in the `chapter_06` branch version of `recipe.module`. Hint: It's a different hook name because the module will only use Drupal 7 fields for its content.

Upgrading recipe module

A site already using recipe module will need an upgrade path in order to create and attach Drupal 7 fields matching the names and types of the legacy recipe content. We can then copy the legacy data into the new fields. To create and attach the fields, we need to edit the `recipe.install` file, and add the following block of code to the `recipe_install_fields()` function skeleton:

```
// Get info we will need to put fields into the same place
// they were in the non-field implementation
$winfo = field_info_extra_fields('node', 'recipe', 'form');
$dinfo = field_info_extra_fields('node', 'recipe', 'display');

foreach (recipe_field_default_fields() as $info) {
  $field_name = $info['field_config']['field_name'];
  $instance = &$info['field_instance'];

  // Don't install this twice
  if (field_info_instance('node', $field_name, 'recipe')) {
    continue;
  }

  // Set the weight of the field in all display modes
  if (isset($dinfo[$field_name]['display'])) {
    foreach ($dinfo[$field_name]['display']
      as $view_mode => $settings) {
      $instance['display'][$view_mode]['weight']
        = $settings['weight'];
    }
  }
}
```

```
// Set the weight of the editing widget
if (isset($winfo[$field_name])) {
    $instance['widget']['weight'] = $winfo[$field_name]['weight'];
}

// Add the field to the recipe content type
field_create_field($info['field_config']);
field_create_instance($instance);
}
```

Non-highlighted lines of code copy legacy field **weights** (ordering) to the new field declarations that we implemented earlier in `recipe_field_default_fields()`. Field API function `field_info_extra_fields()` is used to retrieve weights (ordering) and display settings for the legacy data. Following that, we adjust the new field declaration weights for all display modes and for the edit form.

To complete the code block, we then call `field_create_field()` to save the field configuration, and make a call to `field_create_instance()` to attach the field to recipe entities.

We also add some term reference fields to recipe entities so that we can associate our new terms to recipe entities. Inside the `recipe_install_fields` functions and directly below the code that we just added, paste the following code block:

```
// term reference fields
foreach (recipe_vocabularies() as $machine_name => $info) {
    $field_name = 'term_' . $machine_name;

    if (field_info_instance('node', $field_name, 'recipe')) {
        continue;
    }
    $field = array(
        'field_name' => $field_name,
        'type' => 'taxonomy_term_reference',
        'settings' => array(
            'allowed_values' => array(
                array(
                    'vocabulary' => $machine_name,
                    'parent' => 0
                ),
            ),
        ),
    );
};
```

```

    field_create_field($field);

    $instance = array(
      'field_name' => $field_name,
      'entity_type' => 'node',
      'label' => $info['name'],
      'bundle' => 'recipe',
      'required' => true,
      'widget' => array(
        'type' => 'options_select'
      ),
      'display' => array(
        'default' => array('type' => 'hidden'),
        'teaser' => array('type' => 'hidden')
      )
    );

    field_create_instance($instance);
  }

```

The preceding code block splits the field creation and the instance creation (field attachment), meanwhile showing the structure of the field configuration (\$field) and field instance (\$instance) declarations. The `field_create_field()` and `field_create_instance()` function calls are highlighted again.

Add the following code block at the end of the `recipe.install` file in order to call the `recipe_install_fields()` function during a site upgrade:

```

/**
 * Install fields needed by recipe module.
 */
function recipe_update_7202(&$sandbox) {
  recipe_install_fields();
}

```

To complete the upgrade, we must copy data from the legacy fields to the Drupal 7 fields. Beneath the code you added, paste the following code:

```

/**
 * Migrates recipe data from old schema into fields.
 */
function recipe_update_7203(&$sandbox) {
  if (!isset($sandbox['progress'])) {
    $sandbox['progress'] = 0;
    $sandbox['current_nid'] = 0;
    $sandbox['max']
  }
}

```

```
        = db_query(
            "SELECT COUNT(DISTINCT nid) FROM {node} WHERE type = :type",
            array(':type' => 'recipe')
        )->fetchField();
    }

    $query = db_select('node', 'n');
    $query->join('recipe', 'r', 'n.nid = r.nid');
    $query->fields('r')
        ->condition('n.type', 'recipe')
        ->range($sandbox['progress'], 10);
    $result = $query->execute();

    foreach ($result as $record) {
        $sandbox['current_nid'] = $record->nid;
        $nwrapper = entity_metadata_wrapper('node', $record->nid);

        if (valid_url($record->source, TRUE)) {
            // Put the URL into the link field
            $nwrapper->field_source_link->url = $record->source;
        }
        $nwrapper->field_source = $record->source;
        $nwrapper->field_yield = $record->yield;
        $nwrapper->field_yield_unit = $record->yield_unit;
        $nwrapper->field_description->value = $record->description;
        $nwrapper->instructions->value = $record->instructions;
        $nwrapper->field_notes->value = $record->notes;
        $nwrapper->field_preptime = $record->preptime;
        $nwrapper->field_cooktime = $record->cooktime;

        $sandbox['progress']++;
    }

    $sandbox['#finished']
        = empty($sandbox['max'])
        ? TRUE : ($sandbox['progress'] / $sandbox['max']);
}
```

This function is the meat and potatoes of the update and it batch processes all of the existing nodes to migrate data from the recipe database table to the new fields. The three highlighted lines are all setting the value property of the field wrapper. These are all fields of type `text_formatted`. You may recall these are structure fields, so we need special handling here. What isn't handled in the previous code are text formats for these long text fields, because there isn't full support for that in recipe module. Remember to deal with that in any code that you write.

In the `recipe_install()` function, you need to also add the following line in order to install the fields when the recipe module is installed for the first time.

```
recipe_install_fields();
```

Now you can update your site database by using Drush, which will install and upgrade all the fields during a database update. Use the following command:

```
$ drush updatedb
```

On the **Manage fields** page at `admin/structure/types/manage/recipe/fields` you'll now see the following:

Label	Machine name	Field type	Widget	Operations	
+ Title	title	Node module element			
+ Yield unit	recipe_yield_unit	Text	Text field	edit	delete
+ Yield	recipe_yield	Decimal	Text field	edit	delete
+ Ingredients	recipe_ingredients	Recipe module element			
+ Instructions	recipe_instructions	Long text	Text area (multiple rows)	edit	delete
+ Source	recipe_source	Text	Text field	edit	delete
+ Source link	recipe_source_link	Link	Link	edit	delete
+ Additional notes	recipe_notes	Long text	Text area (multiple rows)	edit	delete
+ Cooking time	recipe_cooktime	Integer	Text field	edit	delete
+ Preparation time	recipe_preptime	Integer	Text field	edit	delete
+ Description	recipe_description	Long text	Text area (multiple rows)	edit	delete
+ Cuisine	term_cuisine	Term reference	Select list	edit	delete
+ Difficulty	term_difficulty	Term reference	Select list	edit	delete

Recipe node manage field page after the update

Summary

In this chapter we learned all about the common fields in Drupal 7: date, file, image, link, number, text, and term reference. We learned to differentiate between single-value and multi-value fields, and we then programmatically accessed all of the covered field types. Finally, we programmatically created fields, added them to a content type, and migrated data into them. Next, we will take a good look at programming field collections – another powerful Drupal 7 concept made possible thanks to the entity paradigm.

6

Developing with Field Collections

We've almost covered all there is to know about Drupal 7 entities. One last common entity type needs our attention: field collections. In this chapter we:

- Describe field collection entities, and how they fit into the Drupal landscape
- Programmatically access a field collection's content
- Programmatically add a field collection entity to a node
- Programmatically add a field collection to a content type

Before Drupal 7

In Drupal days of yore, there were two main ways of grouping fields together and attaching them to a node as a field. We could even have had that field be multi-value, allowing multiple groups of the same fields to be referenced.

This is illustrated in the following screenshot:

The screenshot shows a form titled "Ingredients" with a "Show row weights" link. Below the title is a table with five rows. Each row has a move icon (a crossed arrow) on the left. The table has four main columns: "Quantity", "Units", "Ingredient name", and "Processing/Notes". Each column has a sub-header and an input field. The "Quantity" sub-header is "Quantity" and the input is a text box. The "Units" sub-header is "Unit" and the input is a dropdown menu. The "Ingredient name" sub-header is "Name" and the input is a text box with a radio button. The "Processing/Notes" sub-header is "Note" and the input is a text box. At the bottom of the table is a button labeled "More ingredients".

	Quantity	Units	Ingredient name	Processing/Notes
+	Quantity	Unit	Name	Note
+	Quantity	Unit	Name	Note
+	Quantity	Unit	Name	Note
+	Quantity	Unit	Name	Note
+	Quantity	Unit	Name	Note

More ingredients

Example of field groups in a multi-value field

The previous screenshot shows an ingredients multi-value field with a table of five possible values. Each value or row is a group of the following fields:

- **Quantity**
- **Units**
- **Ingredient name**
- **Processing/Notes**

Each row can be moved up or down to change their order. To move them, the user drags the crossed arrow icon on the left of each row to a new position.

In Drupal 6, this type of data model was built by using multigroups that came along with CCK 3.x. Prior to that, such a model had to be built using a multi-value node reference field. Node references resulted in a poor user experience, because the node being referenced could not be edited directly on the edit form of the referencing node. In addition, referenced nodes soaked up storage space because properties, such as author, were often not needed.

Thanks to the entity paradigm, field collections were possible, and these gave us lightweight storage and a better user experience. Field collections are essentially a multi-value field storing entity references. When you create a new field collection, you are declaring a field collection entity bundle. That entity bundle's edit form is injected into the node edit form by the Field Collection module. This enables a user to edit the values directly on the referencing node's edit form, using an interface similar to that shown in the previous screenshot. When the edit form is submitted, any new values result in a new entity being created to store those values. That entity is then referenced in the field collection field of the node being saved.

Creating a field collection field

Our practical examples will require a field collection that will eventually replace the legacy ingredients hogwash in the old code. Let's create a new ingredients field collection in the recipe node.

After you have installed your Chapter 6 development site, surf over to `admin/structure/types/manage/recipe/fields`. In the **Add new field** section of the form you'll now see **Field collection** listed in the field type select list. In the **Label** text box, enter `Ingredients`. The machine name will automatically be generated as `field_ingredients`. This is fine for now. Select **Field collection** as the type, **Embedded** as the widget, and move the field up to be above `recipe_ingredients`, and then click on **Save**. Choose `unlimited` for the number of values, and keep the defaults for the remaining settings.

After you finish, your model will look as shown in the following screenshot:

Label	Machine name	Field type	Widget	Operations	
+ Title	title	Node module element			
+ Description	recipe_description	Long text	Text area (multiple rows)	edit	delete
+ Yield unit	recipe_yield_unit	Text	Text field	edit	delete
+ Yield	recipe_yield	Decimal	Text field	edit	delete
+ Ingredients	field_ingredients	Field collection	Embedded	edit	delete
+ Ingredients	recipe_ingredients	Recipe module element			
+ Additional notes	recipe_notes	Long text	Text area (multiple rows)	edit	delete
+ Instructions	recipe_instructions	Long text	Text area (multiple rows)	edit	delete
+ Source	recipe_source	Text	Text field	edit	delete
+ Preparation time	recipe_preptime	Integer	Text field	edit	delete
+ Cooking time	recipe_cooktime	Integer	Text field	edit	delete
+ Source link	recipe_source_link	Link	Link	edit	delete
+ Cuisine	term_cuisine	Term reference	Select list	edit	delete
+ Difficulty	term_difficulty	Term reference	Select list	edit	delete

Manage fields page of recipe content type after adding the ingredients field collection

Now we can add fields to the field collection by going to the field collection **Manage fields** page. Navigate to `admin/structure/field-collections/field-ingredients/fields`, and add fields matching the following new field specifications for the `recipe_ingredients` field collection:

New field specifications	Allowed values for field_unit_key
field_quantity (Decimal)	bunch bunch (bn)
Label: Quantity	can can (cn)
Minimum: 0	carton carton (ct)
	centigram centigram (cg)
field_unit_key (List (text))	centiliter centiliter (cl)
Label: Unit	clove clove (clv)
Allowed values: see right column	cup cup (c)
Widget: Select list	dash dash (ds)
	deciliter deciliter (dl)
field_ingredient (Entity Reference)	drop drop (dr)
Label: Ingredient	us fluid ounce fluid ounce (fl oz)
Widget: Autocomplete	us gallon gallon (gal)
Target type: File	gram gram (g)
Sort by: A property of the base table of the entity	kilogram kilogram (kg)
Sort property: filename	liter liter (l)
Size of text field: 25	loaf loaf (lf)
	milligram milligram (mg)
field_note (Text)	milliliter milliliter (ml)
Label: Processing/Notes	ounce ounce (oz)
Size of text field: 33	package package (pk)
	pinch pinch (pn)
	us liquid pint pint (pt)
	pound pound (lb)
	us liquid quart quart (q)
	slice slice (sli)
	tablespoon tablespoon (T)
	teaspoon teaspoon (t)
	unit unit
	unknown unknown

Note that we have a new field type: `entity reference`. We have selected File entities as the target type for `field_ingredient` as a placeholder; we will change this in the next chapter.

Field collection entities

Field collection entities are those referenced by the field collection field just discussed. Each individual field collection is an independent entity bundle, so all of them can have different fields attached. A field collection entity type has the following wrapper and entity properties:

Wrapper property	Type (Read/Write)	Description	Entity property
item_id	integer (R)	Field collection item ID	
revision_id	integer (R)	Revision ID	
field_name	string (R)	Whether or not the node is new	
archived	integer (R/W)	Node type (bundle name)	
url	string (R)	URL for viewing the field collection	N/A
host_entity	entity (R/W)	Entity containing the field collection	N/A
N/A	integer (R/W)	TRUE if this is the default revision	default_revision

Convert the first if block in `pde_field_value()` to the following code:

```
if (isset($info[$pkey]['field']) && $info[$pkey]['field']) {
    $value = pde_field_value($wrapper->$pkey);
    if (is_array($value)) {
        foreach ($value as $i => $v) {
            $key = $pkey."[$i] (" . $wrapper->$pkey->type() . ')';
            $rows[$key] = $v;
        }
    }
    else {
        $key = "$pkey (" . $wrapper->$pkey->type() . ')';
        $rows[$key] = $value;
    }
}
```

The changes improve the output formatting for multi-value fields in our Drush `print-entity` command. Now we will see multiple rows of `field_name[n]` for each value within the multi-value field. The `n` is the delta, or index, of the field value within the multi-value field.

We also need to add some special handling for field collections. As they are entities, let's change `pde_entity_value()` to print the full set of properties and fields of an entity. The `pde_drush_print_entity()` function already prints these, but we want the entity properties and fields to be printed on the right side of the table rather than as a nested table.

```
function pde_entity_value($entity_wrapper) {
    $info = $entity_wrapper->getPropertyInfo();

    $output = pde_output('identifier: '
        . $entity_wrapper->getIdentifier());
    $output .= pde_output('label: ' . $entity_wrapper->label());
    $output .= pde_output('type: ' . $entity_wrapper->type());

    foreach ($entity_wrapper as $pkey => $pwrapper) {
        if (isset($info[$pkey]['field']) && $info[$pkey]['field']) {
            $msg = $info[$pkey]['label'];
        }
        else {
            $msg = $pkey;
        }
        $msg .= ': ';

        if ($entity_wrapper->$pkey instanceof EntityDrupalWrapper) {
            $msg .= pde_output($entity_wrapper->$pkey->label());
        }
        else if ($entity_wrapper->$pkey instanceof EntityListWrapper) {
            $items = array();
            foreach ($entity_wrapper->$pkey as $key => $value) {
                $item_value = "$key ";
                if ($value instanceof EntityDrupalWrapper) {
                    $item_value .= $value->label();
                }
                else {
                    $item_value .= pde_field_value($value);
                }
                $items[] = $item_value;
            }

            $msg .= "\n\t";
            if (!count($items)) {
                $msg .= "<empty>";
            }
            $msg .= implode("\n\t", $items) . "\n";
        }
    }
}
```



```
    }
    else {
        $msg .= pde_field_value($entity_wrapper->$pkey);
    }
    $output .= $msg;
}
return $output;
}
```

This is quite a huge change from the previous implementation. The reason for this is that we get an infinite loop if we simply call the `pde_field_value()` function for each field. To prevent infinite loops we need to avoid sending entities to `pde_field_value()`, and instead we use the output from the `label()` method. The first highlighted line does so, when the field wrapper wraps an entity.

We also output multi-value fields in a pretty way by looping through all of the values, and use the `label()` method output for entities or, for everything else, the output from the `pde_field_value()` function. These are the second and third highlighted lines.

For all other fields we use the output from `pde_field_value()` as shown in the fourth highlighted line.

Adding a field collection to a node

There's nothing too challenging about adding a field collection to a node, because a field collection is an entity. You simply create the field collection entity and then attach it to the node.

Let's look at two examples of using the `field_ingredients` field collection that we just created. For these examples assume that node 2 exists, but has nothing added in `field_ingredients`. The first example, which uses no wrapper code, is as follows:

```
$node = node_load(2);
$data = array(
  'field_name' => 'field_ingredients',
);
// Create the field collection
$fc = entity_create('field_collection_item', $data);
// Set the host entity and field values then save
$fc->setHostEntity('node', $node);
$fc->field_quantity[LANGUAGE_NONE][0]['value'] = 1.0;
$fc->field_unit_key[LANGUAGE_NONE][0]['value'] = 'gram';
$fc->field_ingredient[LANGUAGE_NONE][0]['value'] = 'Salt';
$fc->field_note[LANGUAGE_NONE][0]['value'] = 'coarse';
$fc->save();
```

The same thing, but this time using wrappers, is shown in the following code:

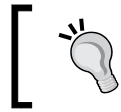
```
$target_wrapper = entity_metadata_wrapper('node', 2);
$data = array(
  'field_name' => 'field_ingredients',
);
// Create the field collection
$fc = entity_create('field_collection_item', $data);
$fc_wrapper =
  entity_metadata_wrapper('field_collection_item', $fc);
// Set the host entity and field values then save
$fc_wrapper->host_entity = $target_wrapper;
$fc_wrapper->field_quantity = 1.0;
$fc_wrapper->field_unit_key = 'gram';
$fc_wrapper->field_ingredient = 'Salt';
$fc_wrapper->field_note = 'coarse';
$fc_wrapper->save();
```

The comments in the code tell the story. The wrapper version has one, highlighted, extra line. The highlighted code in the non-wrapper version is slightly terser than the wrapper version.

Attaching a field collection to a content type

Programmatically creating a field collection can be done in much the same way as we did with fields. Because we have already created the field collection, we can now perform the following steps in order to have these added to recipe nodes at installation time or after an upgrade:

1. Export the field collection to code inside a feature.
2. Copy the exported declarations into the recipe module and rename `field_ingredients` to `recipe_ingredients` and change the prefixes of the fields within the field collection to `ri_`.
3. Tweak the recipe module code to remove the legacy fields from the field admin UI and node edit forms.
4. Add update code to attach the new field collection to recipe nodes. We will defer copying data from the `recipe_node_ingredient` table into the new field collections until the next chapter.



After these changes are completed, the examples in the previous section would need the field names changed to match the new names.

Exporting field collection and fields

Point your browser to `admin/structure/features/create` and enter values into the **General Information** section of the form so that it matches the example shown in the following screenshot:

General Information

Name

Machine name: recipe2 [\[Edit\]](#)
Example: Image gallery (Do not begin name with numbers.)

Description

Provide a short description of what users should expect when they enable your feature.

Package

Organize your features in groups.

Version

Examples: 7.x-1.0, 7.x-1.0-beta1

General Information fieldset of the create feature page

In the **Components** fieldset we expand the **Fields** section and select the checkbox for `node-recipe-field_ingredients`. Continue to select the checkboxes of all other fields beginning with `field_collection_item` and `node-recipe-recipe`. The **Dependencies (dependencies)**, **Field Bases (field_base)** and **Field Instances (field_instance)** sections will look as shown in the following screenshot:

The screenshot displays the 'Components' fieldset with three main sections: 'Dependencies (dependencies)', 'Field Bases (field_base)', and 'Field Instances (field_instance)'. Each section contains a list of checkboxes, many of which are checked.

- Dependencies (dependencies):**
 - ☒ Entity Reference
 - ☒ Features
 - ☒ Field collection
 - ☒ Link
 - ☒ List
 - ☒ Number
 - ☒ Options
- Field Bases (field_base):**
 - ☒ field_ingredient
 - ☒ field_ingredients
 - ☒ field_note
 - ☒ field_quantity
 - ☒ field_unit_key
 - ☒ recipe_cooktime
 - ☒ recipe_description
 - ☒ recipe_instructions
 - ☒ recipe_notes
 - ☒ recipe_preptime
 - ☒ recipe_source
 - ☒ recipe_source_link
 - ☒ recipe_yield
 - ☒ recipe_yield_unit
- Field Instances (field_instance):**
 - ☐ node-recipe-term_cuisine
 - ☐ node-recipe-term_difficulty
 - ☒ field_collection_item-field_ingredients-field_ingredient
 - ☒ field_collection_item-field_ingredients-field_note
 - ☒ field_collection_item-field_ingredients-field_quantity
 - ☒ field_collection_item-field_ingredients-field_unit_key
 - ☒ node-recipe-field_ingredients
 - ☒ node-recipe-recipe_cooktime
 - ☒ node-recipe-recipe_description
 - ☒ node-recipe-recipe_instructions
 - ☒ node-recipe-recipe_notes
 - ☒ node-recipe-recipe_preptime
 - ☒ node-recipe-recipe_source
 - ☒ node-recipe-recipe_source_link
 - ☒ node-recipe-recipe_yield
 - ☒ node-recipe-recipe_yield_unit

Part of the Components fieldset of the create feature page

Click on the **Download feature** button to download the feature to your computer.

Copying the code to the recipe module

In your terminal, navigate to the folder into which the feature was downloaded, and extract the feature file by using the following command:

```
$ tar xf recipe2.tar
```

Open `recipe2.features.field_base.inc` and `recipe2.features.field_instance.inc` found in the newly-created `recipe2` folder. Copy the `recipe2_field_default_field_bases()` and `recipe2_field_default_field_instances()` functions to `recipe.field.inc` after removing all functions in the latter file. You'll find `recipe.field.inc` in the recipe module folder `sites/all/modules/customized/recipe` of your development site.

In the pasted code, make the following replacements, in the given order:

1. `recipe2` to `recipe`
2. `field_ingredients` to `recipe_ingredients`
3. `field_ingredient` to `ri_ingredient`
4. `field_note` to `ri_note`
5. `field_quantity` to `ri_quantity`
6. `field_unit_key` to `ri_unit_key`
7. `'locked' => 0` to `'locked' => 1`

Tweaking recipe.module

Inside `recipe.module` we can now empty the array returned by the `recipe_field_extra_fields()` function since there are no longer any legacy fields. The top of the function now looks as shown in the following code snippet:

```
function recipe_field_extra_fields() {
  if (variable_get('recipe_fields_installed', FALSE)) {
    return array();
  }

  $extra = array();
  $extra['node']['recipe'] = array(
    ...
  );
}
```

The highlighted line shows the empty array. Replace the `recipe_form_recipe_node_form_alter()` function with the following code:

```
/**
 * Implementation of hook_form_FORM_ID_alter().
 */
```

```

function recipe_form_recipe_node_form_alter(&$form, &$form_state) {
  $language = $form['language']['#value'];
  $children =
    element_children($form['recipe_ingredients'][$language]);
  foreach ($children as $delta) {
    $fcoll = &$form['recipe_ingredients'][$language][$delta];
    $fcoll['ri_quantity'][$language][0]['value']['#size'] = 7;
  }

  $css = '.field-name-recipe-ingredients .form-wrapper { display:
inline-block; }';
  $form['recipe_ingredients']['#attached']['css'] = array(
    $css => array('type' => 'inline'),
  );
}

```

The first block of code sets the visible size of the quantity text field to 7. The second block makes all of the fields display inline rather than above one another on the node edit form. Surf to `node/add/recipe`, and scroll down to see the fruits of your work.

Field collection user interface for entering values

The final change is to `recipe.info`, where we add the following highlighted dependency lines:

```

dependencies[] = entity
dependencies[] = entityreference
dependencies[] = field_collection
dependencies[] = link
dependencies[] = list
dependencies[] = number
dependencies[] = options
dependencies[] = taxonomy
dependencies[] = text
dependencies[] = token

```

Updating code is unnecessary

For the changes we have made, there is no need for us to add any update code at this point. Calls to the `recipe_install_fields()` function that we added to the update and installation code in an earlier chapter will take care of installing and updating the field collection and its fields.

Summary

In this chapter we discovered field collections, their programmatic reading, creation, and addition to existing entities. Next, we will make custom database data available to Drupal 7 by exposing them as entities.

7

Expose Local Entities

So far in this book, we have played around with entities implemented by Drupal core and contributed modules. In our next installment, we will use a pressure cooker to create a typical, slow-cooked dish quickly: exposing custom data to Drupal. In this chapter, we do the following:

- Expose a database table as a non-fieldable entity
- Expose a database table as a fieldable entity
- Enable exporting, importing, and cloning of bundle configurations

Motivation for exposing entities

Sites that do anything useful tend to have a lot of legacy data that they use throughout the site. Because these data vary a lot and are tied into business processes that are unique to each business, it becomes necessary to build custom solutions per site. Until the entity paradigm existed, each of these custom solutions required reinvention of the wheel to make the data available to adequate numbers of solution modules. For example, one database table needed custom code for each of the `token` and `views` modules just so those modules would retrieve table data correctly. Thankfully, we are past this stone-age style of implementation, because we have entities!

We will add code to the recipe module piece-by-piece to gradually expose ingredients as fully-fledged entities that are able to please even the most discerning site builder's palate.

Fast track your data exposure

The fastest way to create custom entities is to use the **Entity Construction Kit (ECK)** module. ECK will do all of the dirty work for you after you enter an entity type name and an optional bundle name. After that, you add your fields by using the generated admin UI. The interface is similar to node field management. However, there's one thing ECK can't do at the time of writing – expose an existing database table to Drupal.

Exposing an existing database table as entities can be done easily by combining the entity module's helpers with one single hook implementation: `hook_entity_info`. Let's say we want to expose the `recipe_ingredient` table as a non-fieldable entity. The table schema would be as shown in the following screenshot:

recipe_ingredient	
id	serial
name	varchar
link	int

Schema of the `recipe_ingredient` table in `recipe` module

The `hook_entity_info` implementation, `recipe_entity_info`, inside the `recipe.module` file will look like the following:

```
/**
 * Implements hook_entity_info().
 */
function recipe_entity_info() {
  $info = array();
  $info['recipe_ingredient'] = array(
    'label' => t('Ingredient'),
    'plural label' => t('Ingredients'),
    'description' => t('Recipe ingredients.'),
    'entity class' => 'Entity',
    'controller class' => 'EntityAPIController',
    'base table' => 'recipe_ingredient',
    'fieldable' => FALSE,
    'entity keys' => array(
      'id' => 'id',
      'label' => 'name',
    ),
  );
  return $info;
}
```

What you see in the preceding code snippet is the minimal code needed to expose data as entities. The key directives within the preceding entity type declaration are as follows:

- `entity class`: This is the class used for entities returned from `entity_load`
- `controller class`: This is the class used to load entity objects
- `base table`: This is the database table containing our data
- `entity keys`: This contains the database field names for the entity ID and label properties

In our `recipe_ingredient` table (our base table) we have the `id` and `name` fields storing the entity ID and label respectively. We have specified these in the `entity keys` directive. We are also using `Entity` and `EntityApiController` as the entity and controller class respectively. These are provided by the `entity` module, and using them allows us to expose our data quickly. These classes can be overridden when special handling is needed.

Once you have the code in place, you can write some standard entity creation code such as the following:

```
$data = array(
  'name' => 'Salt',
  'link' => 0,
);
$entity = entity_create('recipe_ingredient', $data);
$wrapper = entity_metadata_wrapper('recipe_ingredient', $entity);
$wrapper->save();
```

Allow fields on your entity

Making your exposed entity type fieldable is as simple as changing the `fieldable` value in the entity type declaration to `TRUE`:

```
...
'fieldable' => TRUE,
...
```

Once this is in place and you have cleared the caches, you can programmatically add fields to the entity. Remember though, that the bundle name is the same as the entity type name. For our example, that would be `recipe_ingredient`. This is fine for scenarios where your entity type will have the same set of fields for all entities; in other words, a single bundle. For cases where multiple bundles are required, we need more tweaks.

Give it multiple bundles

To allow our entity type to have multiple bundles, we make two changes to the entity type declaration. We add the `bundle` directive to the set of `entity keys` and the `bundles` directive containing all bundles:

```
...
'entity keys' => array(
  'id' => 'id',
  'label' => 'name',
  'bundle' => 'type',
),
'bundles' => array(
  'standard' => array(
    'label' => t('Standard'),
  ),
),
...
```

The `bundle` directive in the `entity keys` array tells Drupal the database field name containing the bundle name of each record. The field is typically 32 characters long and of the `varchar` type. We will add the `type` field to the `recipe_ingredient` table in an update.

Each bundle is declared within the `bundles` array of the entity type declaration. In the previous code snippet, we have declared the `standard` bundle with only a label, which is all that is needed for it to work.

Let's create some code to add the `type` field to the `recipe_ingredient` table. Add the following code to the `recipe_ingredient` table declaration within the `recipe_schema()` function in `recipe.install`:

```
'type' => array(
  'description' => 'The type of this ingredient.',
  'type' => 'varchar',
  'length' => 32,
  'not null' => TRUE,
  'default' => 'standard',
),
```

Next, at the bottom of the file, add the following function:

```
/**
 * Make schema changes for ingredients as entities.
 */
function recipe_update_7205(&$sandbox) {
```

```
// Add type field
$type_schema = array(
  'description' => 'The type of this ingredient.',
  'type' => 'varchar',
  'length' => 32,
  'not null' => TRUE,
  'default' => 'standard',
);
db_add_field('recipe_ingredient', 'type', $type_schema);
}
```

After adding this code and updating the database, you can have multiple fieldable bundles for your entity type.

Administration interface and exportability

All of the previous code forces us to use programmatic ways to manage bundles and fields of our entity type. We want administrative users to be able to manage bundles and fields through the web interface. In addition to this, we would like to allow users to export and import bundle configurations. To do so we need the following things:

- A way to store information about the bundles created by the user
- Expose that information to Drupal as entities with some extra directives
- Provide access handling in order to prevent unauthorized users from changing our bundles

Storing bundle information

To store the bundle information, we will use a new database table called `recipe_ingredient_type`. Add the following table declaration to `recipe_schema` in the `recipe.install` file.

```
$schema['recipe_ingredient_type'] = array(
  'description'
    => 'Stores information about all defined ingredient types.',
  'fields' => array(
    'id' => array(
      'type' => 'serial',
      'not null' => TRUE,
      'description' => 'Primary Key: Unique ingredient type ID.',
    ),
    'type' => array(
      'description'
        => 'The machine-readable name of this ingredient type.',
```

```
        'type' => 'varchar',
        'length' => 32,
        'not null' => TRUE,
    ),
    'label' => array(
        'description'
            => 'The human-readable name of this ingredient type.',
        'type' => 'varchar',
        'length' => 255,
        'not null' => TRUE,
        'default' => '',
    ),
    'weight' => array(
        'type' => 'int',
        'not null' => TRUE,
        'default' => 0,
        'size' => 'tiny',
        'description' => 'The weight of this ingredient type in relation
to others.',
    ),
    'data' => array(
        'type' => 'text',
        'not null' => FALSE,
        'size' => 'big',
        'serialize' => TRUE,
        'description' => 'A serialized array of additional data related
to this ingredient type.',
    ),
    'status' => array(
        'type' => 'int',
        'not null' => TRUE,
        // Set the default to ENTITY_CUSTOM without using the
        // constant as it is not safe to use it at this point.
        'default' => 0x01,
        'size' => 'tiny',
        'description' => 'The exportable status of the entity.',
    ),
    'module' => array(
        'description' => 'The name of the providing module if the entity
has been defined in code.',
        'type' => 'varchar',
        'length' => 255,
        'not null' => FALSE,
    ),
),
```

```

    ),
    'primary key' => array('id'),
    'unique keys' => array(
        'type' => array('type'),
    ),
);

```

The `type` and `label` database fields store the machine name and human-readable names of the bundle. The `data` field is required by the Entity API classes, but we don't use it. The `status` field is a flag that tracks whether the bundle has been defined in code or has been custom built or edited through the web interface.

Next, add the following `update()` function to the bottom of `recipe.install`. The code adds the `recipe_ingredient_type` table to an existing installation of the `recipe` module. We have to declare the complete table schema rather than pulling it from `recipe_schema` in order to prevent future schema changes from breaking the upgrades.

```

/**
 * Make schema changes for ingredients as entities.
 */
function recipe_update_7206(&$sandbox) {
    // Add ingredient type table
    $table_name = 'recipe_ingredient_type';
    $schema[$table_name] = array(
        'description' => 'Stores information about all defined ingredient
types.',
        'fields' => array(
            'id' => array(
                'type' => 'serial',
                'not null' => TRUE,
                'description' => 'Primary Key: Unique ingredient type ID.',
            ),
            'type' => array(
                'description'
                => 'The machine-readable name of this ingredient type.',
                'type' => 'varchar',
                'length' => 32,
                'not null' => TRUE,
            ),
            'label' => array(
                'description'
                => 'The human-readable name of this ingredient type.',
                'type' => 'varchar',
                'length' => 255,
                'not null' => TRUE,
            ),
        ),
    );
}

```

```
        'default' => '',
    ),
    'weight' => array(
        'type' => 'int',
        'not null' => TRUE,
        'default' => 0,
        'size' => 'tiny',
        'description' => 'The weight of this ingredient type in
relation to others.',
    ),
    'data' => array(
        'type' => 'text',
        'not null' => FALSE,
        'size' => 'big',
        'serialize' => TRUE,
        'description' => 'A serialized array of additional data
related to this ingredient type.',
    ),
    'status' => array(
        'type' => 'int',
        'not null' => TRUE,
        // Set the default to ENTITY_CUSTOM without using the
        // constant as it is not safe to use it at this point.
        'default' => 0x01,
        'size' => 'tiny',
        'description' => 'The exportable status of the entity.',
    ),
    'module' => array(
        'description' => 'The name of the providing module if the
entity has been defined in code.',
        'type' => 'varchar',
        'length' => 255,
        'not null' => FALSE,
    ),
),
'primary key' => array('id'),
'unique keys' => array(
    'type' => array('type'),
),
);
db_create_table($table_name, $schema[$table_name]);
}
```

Run the database updates by using Drush, and the new table will be ready for the next step:

```
$ drush updatedb
```

Exposing bundle information and handling access rights

Bundle information exposure and access rights are both implemented by using extra directives in `hook_entity_info` and supporting functions. Let's begin with the top part of `recipe_entity_info`:

```
/**
 * Implements hook_entity_info().
 */
function recipe_entity_info() {
  $info = array();
  $info['recipe_ingredient'] = array(
    'label' => t('Ingredient'),
    'plural label' => t('Ingredients'),
    'description' => t('Recipe ingredients.'),
    'entity class' => 'Entity',
    'controller class' => 'EntityApiController',
    'base table' => 'recipe_ingredient',
    'fieldable' => TRUE,
    'view modes' => array(
      'full' => array(
        'label' => t('Full content'),
        'custom settings' => FALSE,
      ),
    ),
    'entity keys' => array(
      'id' => 'id',
      'bundle' => 'type',
      'label' => 'name',
    ),
    'bundles' => array(),
    'bundle keys' => array(
      'bundle' => 'type',
    ),
    'label callback' => 'entity_class_label',
    'uri callback' => 'entity_class_uri',
    'module' => 'recipe',
  );
}
```


We have changed the declaration of `recipe_ingredient` by assigning an empty array to the `bundles` directive and dynamically building the array elements from the `recipe_ingredient_type` table.

Highlighted lines in `recipe_ingredient` show the following extra directives that are needed:

- `bundle keys`: This has to be declared because we are using a load handler in the admin URI for each bundle. This directive tells Drupal which entity object property returns the bundle name. We'll discuss more about this in a moment.
- `label callback`: This function is called to retrieve the entity's label. The entity module implements the `entity_class_label` function, which calls the `label()` method of the loaded entity object.
- `uri callback`: This function is called to retrieve the entity's URI. The entity module implements `entity_class_uri`, which calls the `uri()` method of the loaded entity object.
- `module`: This tells the entity module which module exposes the entity type. This is used for calling hooks, generating file paths, and for declaring import dependencies in bundle exports. This directive also helps the entity module provide default integration with solution modules such as `views` and `token`.

The following is part two of `recipe_entity_info`:

```
// Add bundle info but bypass entity_load() as we cannot
// use it here.
$types = db_select('recipe_ingredient_type', 'rit')
  ->fields('rit')
  ->execute()
  ->fetchAllAssoc('type');

foreach ($types as $type => $tinfo) {
  $info['recipe_ingredient']['bundles'][$type] = array(
    'label' => $tinfo->label,
    'admin' => array(
      'path' => 'admin/structure/ingredients/manage/%recipe_ingredient_type',
      'real path' => 'admin/structure/ingredients/manage/' . $type,
      'bundle argument' => 4,
      'access arguments' => array('administer ingredients'),
    ),
  );
}
```

This code queries the `recipe_ingredient_type` table for user-defined bundles, and uses that information to dynamically build `bundle` directives. Notice the additional `admin` directive that we didn't use before.

The `entity` module uses the `admin` directive to build menu items for the bundle administration interface. The `path` directive specifies the URI with a load handler named `%recipe_ingredient_type`. Drupal's menu system takes this URI component (that is, index 4 slash delimited) and passes that component value to the `recipe_ingredient_type_load()` function. For example, when the user navigates to `admin/structure/ingredients/manage/standard`, Drupal calls `recipe_ingredient_type_load()` passing it the string value `standard`.

The `manage` part of the URI is also significant because the `entity` module appends `manage` to the `admin ui` path of the bundle definition entity. The **bundle definition entity** is an entity type that stores the defined bundles of another entity. The following is the final code block of `recipe_entity_info` in which we declare our bundle definition entity named `recipe_ingredient_type`:

```
$info['recipe_ingredient_type'] = array(
  'label' => t('Ingredient type'),
  'plural label' => t('Ingredient types'),
  'description' => t('Ingredient types for Recipe module.'),
  'entity class' => 'IngredientType',
  'controller class' => 'EntityAPIControllerExportable',
  'base table' => 'recipe_ingredient_type',
  'fieldable' => FALSE,
  'bundle of' => 'recipe_ingredient',
  'exportable' => TRUE,
  'entity keys' => array(
    'id' => 'id',
    'name' => 'type',
    'label' => 'label',
  ),
  'access callback' => 'recipe_ingredient_access',
  'module' => 'recipe',
  // Enable the entity API's admin UI.
  'admin ui' => array(
    'path' => 'admin/structure/ingredients',
    'file' => 'recipe.admin.inc',
    'controller class' => 'IngredientTypeUIController',
  ),
);

return $info;
}
```

The highlighted parts of the `recipe_ingredient_type` entity type declaration are as follows:

- `IngredientType`: This is used to override the `Entity` class in order to support entity export
- `EntityAPIControllerExportable`: This is a subclass of `EntityAPIController` that we are using so that our entity becomes exportable
- `bundle of`: This tells the entity module that this entity is the bundle definition entity for `recipe_ingredient`
- `exportable`: This is set to `TRUE` to make these entities exportable
- `name`: This tells the Entity API to use the specified field as an identifier for the entity instead of the numeric ID
- `access callback`: This function is called to check a user's right to manage entities
- `path`: This is the URI of the entity administration UI and a base path for the bundle administration UI of `recipe_ingredient` mentioned earlier
- `file`: This is the name of the file (relative to the module) containing the administration UI controller class declaration and supporting code
- `IngredientTypeUIController`: This enables overriding of the `EntityDefaultUIController` class in order to set the menu item description

Adding the support code

To complete the overhaul, we add some class declarations and functions to `recipe.module` and `recipe.admin.inc`. First, add the following code to `recipe.admin.inc`:

```
/**
 * Generates the recipe_ingredient_type editing form.
 */
function recipe_ingredient_type_form($form, &$form_state, $recipe_ingredient_type, $op = 'edit') {

  if ($op == 'clone') {
    $recipe_ingredient_type->label .= ' (cloned)';
    $recipe_ingredient_type->type = '';
  }

  $form['label'] = array(
    '#title' => t('Label'),
```

```

        '#type' => 'textfield',
        '#default_value' => $recipe_ingredient_type->label,
        '#description' => t('The human-readable name of this ingredient
type.'),
        '#required' => TRUE,
        '#size' => 30,
    );
    // Machine-readable type name.
    $form['type'] = array(
        '#type' => 'machine_name',
        '#default_value' => isset($recipe_ingredient_type->type) ?
$recipe_ingredient_type->type : '',
        '#maxlength' => 32,
        '#disabled' => $recipe_ingredient_type->isLocked() && $op !=
'clone',
        '#machine_name' => array(
            'exists' => 'recipe_ingredient_get_types',
            'source' => array('label'),
        ),
        '#description' => t('A unique machine-readable name for this
ingredient type. It must only contain lowercase letters, numbers,
and underscores.'),
    );

    $form['actions'] = array('#type' => 'actions');
    $form['actions']['submit'] = array(
        '#type' => 'submit',
        '#value' => t('Save ingredient type'),
        '#weight' => 40,
    );
    $form['weight'] = array(
        '#type' => 'weight',
        '#title' => t('Weight'),
        '#default_value' => $recipe_ingredient_type->weight,
        '#description' => t('When showing ingredients, those with lighter
(smaller) weights get listed before ingredients with heavier (larger)
weights.'),
        '#weight' => 10,
    );

    if (!$recipe_ingredient_type->isLocked() && $op != 'add' && $op !=
'clone') {
        $form['actions']['delete'] = array(
            '#type' => 'submit',
            '#value' => t('Delete ingredient type'),

```

```
        '#weight' => 45,  
        '#limit_validation_errors' => array(),  
        '#submit' => array('recipe_ingredient_type_form_submit_delete')  
    );  
}  
return $form;  
}
```

The preceding code is the Form API code for the entity edit form, and the following code defines the button submit handlers:

```
/**  
 * Form API submit callback for the save button.  
 */  
function recipe_ingredient_type_form_submit(&$form, &$form_state) {  
    $recipe_ingredient_type  
        = entity_ui_form_submit_build_entity($form, $form_state);  
    // Save and go back.  
    $recipe_ingredient_type->save();  
    $form_state['redirect'] = 'admin/structure/ingredients';  
}  
  
/**  
 * Form API submit callback for the delete button.  
 */  
function recipe_ingredient_type_form_submit_delete(&$form, &$form_  
state) {  
    $type = $form_state['recipe_ingredient_type']->type;  
    $form_state['redirect']  
        = "admin/structure/ingredients/manage/$type/delete";  
}
```

Add the following code to the top of the `recipe.module` file:

```
/**  
 * Use a separate class for ingredient types so we can specify some  
 * defaults modules may alter.  
 */  
class IngredientType extends Entity {  
    public $type;  
    public $label;  
    public $weight = 0;  
  
    public function __construct($values = array()) {  
        parent::__construct($values, 'recipe_ingredient_type');  
    }  
}  
  
/**
```

```
* Returns whether the ingredient type is locked, thus may not be
* deleted or renamed.
*
* Ingredient types provided in code are automatically treated as
* locked, as well as any fixed ingredient type.
*/
public function isLocked() {
    return isset($this->status) && empty($this->is_new)
    && (($this->status & ENTITY_IN_CODE)
        || ($this->status & ENTITY_FIXED));
}

/**
 * Overrides Entity::save().
 */
public function save() {
    parent::save();
    // Clear field info caches such that any changes to extra fields
    // get reflected.
    field_info_cache_clear();
}

/**
 * UI controller.
 */
class IngredientTypeUIController extends EntityDefaultUIController {
    /**
     * Overrides hook_menu() defaults.
     */
    public function hook_menu() {
        $items = parent::hook_menu();
        $items[$this->path]['description']
            = 'Manage ingredients, including fields.';
        return $items;
    }
}
```

The preceding `IngredientType` class declaration overrides the `Entity` class that we normally use to get off the ground quickly. We have implemented the following methods:

- `__construct()`: This allows the class to be called without specifying an entity type
- `isLocked()`: This helps the UI prevent deletion or editing
- `save()`: This flushes field information caches after the ingredient type is saved

The `IngredientTypeUIController` class overriding `EntityDefaultUIController` class is simply an example of how to put this into place. A trivial change to the menu item description is the only override. If you choose not to override the UI controller, then you should remember to change the `controller class` directive of `admin ui` to `EntityDefaultUIController`.

In more complicated use cases, it may be necessary to override `EntityDefaultUIController` in order to achieve the following:

- Implement extra operations that can be performed on your entities
- Show more details on the entity listing pages

To complete the entity exposure, add the following code under `recipe_entity_info` in `recipe.module`:

```
/**
 * Access callback for the entity API.
 */
function recipe_ingredient_access($op, $type = NULL, $account = NULL)
{
    return user_access('administer ingredients', $account);
}

/**
 * Gets an array of all ingredient types, keyed by the type name.
 *
 * @param $type_name
 *   If set, the type with the given name is returned.
 * @return IngredientType[]
 *   Depending whether $type is set, an array of ingredient types or
 *   a single one.
 */
function recipe_ingredient_get_types($type_name = NULL) {
    $types = entity_load_multiple_by_name('recipe_ingredient_type',
    isset($type_name) ? array($type_name) : FALSE);
```

```

    return isset($type_name) ? reset($types) : $types;
}

/**
 * Menu load handler.
 */
function recipe_ingredient_type_load($type_name) {
    return recipe_ingredient_get_types($type_name);
}

/**
 * Define default ingredient type configurations.
 *
 * @return
 *   An array of default ingredient types, keyed by ingredient type
 *   names.
 */
function recipe_default_recipe_ingredient_type() {
    $types['standard'] = new IngredientType(array(
        'type' => 'standard',
        'label' => t('Standard'),
        'weight' => 0,
        // 'status' => ENTITY_FIXED,
    ));
    return $types;
}

```

The following is some information about these functions:

- `recipe_ingredient_access()`: This is an access handler that returns `TRUE` when the user has the `administer ingredients` permission
- `recipe_ingredient_get_types()`: This is a helper function that loads information for all ingredient types or for a specified ingredient type
- `recipe_ingredient_type_load()`: This is a menu item load handler that is needed for the administration UI
- `recipe_default_recipe_ingredient_type()`: This is called by the `entity` module, and is an implementation of `hook_default_recipe_ingredient_type` that informs Drupal about the standard ingredient type

Other modules can also implement `hook_default_recipe_ingredient_type` in order to offer their own built-in ingredient types. The highlighted `status` directive can be uncommented to prevent any changes to the ingredient type.

Change the `recipe_permissions()` function by adding the `administer ingredients` permission declaration, so that the start of the function looks like the following code excerpt:

```
function recipe_permissions() {
  return array(
    'administer ingredients' => array(
      'title' => t('Administer ingredients'),
      'description' => t('Administer ingredients system-wide.'),
    ),
    'export recipes' => array(
      ...
    )
  );
}
```

To check whether the standard ingredient type is activated, you can clear the caches using Drush, and then run the `print-entity` Drush command in the `pde` module:

```
$ drush cc all
$ drush pe standard recipe_ingredient_type
Entity (recipe_ingredient_type) - ID# standard:

id      : 1
type     : standard
label    : Standard
weight   : 0
data:
status   : 2
module   : recipe
```

Notice here that we are using `standard` instead of the numeric ID of the entity. This is necessary because we specified `name` instead of `label` in the `entity keys` directive of the entity type declaration. For more information about when to use the `name` instead of the numeric ID, see the documentation for `entity_crud_hook_entity_info` in `entity.api.php` from the `entity` module.

You can now navigate in your browser to `admin/structure/ingredients` and play around with the administration UI of ingredient types!

Summary

In this code-heavy chapter, we covered the exposure of database tables as fieldable and non-fieldable entities. We also discussed enabling export, import, and cloning of the entity bundle configurations.

In the next and final chapter, we expose remote data as entities.

8

Expose Remote Entities

So far, we have cooked up some rather meaty portions of Drupal entity soup. We will finish the dish with a dollop of:

- Introducing the Remote Entity API
- Requirements for exposing a remote data source as Drupal entities
- Implementing a remote data source as entities in a recipe website

Introducing the Remote Entity API

Use cases may demand remote data be integrated with the Drupal site. Developers tend to use custom built import scripts or the `Feeds` module to pull data into entities. Another option would be to implement a custom entity controller class overriding the `load()` and `save()` methods. The latter is faster to implement, but has no built-in support for caching. Without caching, users will not see entities during Internet routing failures, and page loading times may be longer. Once the `Feeds` module is configured to import the data into a custom table, we would simply expose the data like we did in the previous chapter. As there are resources online and books available about using the `Feeds` module, it would be silly to cover it in this book.

A new option became possible with the introduction of the Remote Entity API module. Remote Entity API is intended to reduce the amount of coding necessary to expose remote entities to Drupal. It even caches remote data locally and, once exposed by a `hook_entity_info` implementation, will integrate well with solution modules that support `EntityFieldQuery`.

Remote Entity API provides the following benefits:

- **Fixed entity structure:** All remote entities have the same basic properties independent of the remote data structure
- **Remote property mapping:** Exposes properties of the remote entity as properties of the local entity
- **Administration UI:** Implements a menu item and associated handler for a page listing remote entities with edit and delete links
- **Numeric entity IDs:** Regardless of the remote entity's primary key, numeric entity IDs are exposed allowing full integration with most solution modules

At the time of writing, Remote Entity API is at alpha development stage, meaning the API is susceptible to change. The code we will extend in this chapter is not likely to change much and is written well. We will also keep our entities read-only to simplify things. However, do stay abreast of changes in the Remote Entity API module and check for addendums of this book.

Requirements for exposing remote entities

To expose remote data to Drupal using the Remote Entity API module, several requirements must be satisfied. The requirements are as follows:

- A web service exposing data for retrieve and index REST operations. Other services and protocols can also be used.
- Connection and resource classes for the `Clients` module to connect to the web service.
- Connection and resources configured for the web service.
- A database table for cached entities if caching is desired.
- Implementation of `hook_entity_info` to expose your entity to Drupal.
- An entity controller supporting remote entities.
- Custom code for caching and data import.

It's important to note that Remote Entity API does not cache remote data in a schema identical to the remote database. Instead, the API demands a particular base schema in which remote entity data is serialized into one field. Each entity type must have its own base table.

Implementing remote entity exposure

For the practical part of this chapter, we will expose a USDA food description list as entities in our recipe website. We will complete a new module named `usda_remote`, which depends on the `clients` and `remote_entity` modules for connection to a remote RESTful service exposing Drupal entities using the `services_entity` module. We will only concern ourselves with the client side of this setup.

You will find the `usda_remote` module in the `sites/all/modules/customized/usda_remote` folder inside the example code you downloaded earlier. The list of files within that folder is as follows:

- `usda_remote.admin.inc`: Contains administration UI code
- `usda_remote.batch.inc`: Batch entity import code
- `usda_remote.clients.inc`: Subclass for connection to the remote service
- `usda_remote.info`: Standard Drupal information file for a module
- `usda_remote.install`: Install file containing the schema for the database table needed by Remote Entity API
- `usda_remote.module`: Drupal module file
- `usda_remote.query.inc`: Subclasses of `RemoteEntityQuery`: `USDARemoteSelectQuery`, `USDARemoteInsertQuery`, and `USDARemoteUpdateQuery`

Most code we are using is copied from the `clients_ms_dynamics_soap` module available on www.drupal.org. There are some empty write-related class methods so that interfaces are implemented and code compiles. At the time of writing, the `clients_ms_dynamics_soap` module is the only publically available module using Remote Entity API. More will surface as the module matures.

Go ahead and install the code for this chapter. During the installation, you will be asked for a username and a password for accessing the remote service. In the first fieldset's description, click on the link to the **USDA Nutritional Database Service** registration page, create an account, and then enter those credentials into the form before proceeding with your installation.

Let's examine or add the necessary code file-by-file to bring the entities to life on our local site.

Database schema

In the `usda_remote.install` file, you will see the schema for the `usda_food_des` database table with the following fields:

- `eid`: Entity ID field
- `remote_id`: Remote entity ID that can be textual or numeric
- `type`: Entity bundle
- `entity_data`: Serialized remote object data
- `created`: UNIX timestamp for when the entity was created
- `changed`: UNIX timestamp for when the entity was changed
- `remote_saved`: UNIX timestamp for when the entity was remotely saved
- `needs_remote_save`: Flag indicating the entity needs to be saved remotely
- `expires`: UNIX timestamp for when the entity expires
- `deleted`: Flag indicating the entity is to be deleted

All the preceding fields are required by Remote Entity API. Additional fields can be added for your use case.

Connection code

In order to connect to a remote REST service, we must override the `Clients` module's `clients_connection_drupal_services_rest_7` class and implement the `ClientsRemoteEntityInterface` and `ClientsConnectionAdminUIInterface` interfaces. These interfaces are required for Remote Entity API to interface with the `Clients` module. It's likely this connection code will not be necessary for most connections once Remote Entity API matures.

The code in this file has been copied from `clients_ms_dynamics_soap.clients.inc` in the `clients_ms_dynamics_soap` module. The functions copied are: `remote_entity_load()`, `remote_entity_save()`, `entity_property_type_map()`, and `getRemoteEntityQuery()`. Add the following lines to the `getRemoteEntityQuery()` function:

```
switch ($query_type) {
  case 'select':
    return new USDARemoteSelectQuery($this);
  case 'insert':
    return new USDARemoteInsertQuery($this);
  case 'update':
    return new USDARemoteUpdateQuery($this);
}
```

The preceding code returns the appropriate instance of our `RemoteEntityQuery` subclass, which we will discuss next. We don't need to modify `entity_property_type_map()` because, at the time of writing, the remote query implementation in Remote Entity API is still being finalized.

Remote query code

Code for remote queries can be found in the `usda_remote.query.inc` file. The code is based on the `clients_ms_dynamics_soap` module's implementation and contains the following subclasses:

- `USDARemoteSelectQuery`
- `USDARemoteInsertQuery`
- `USDARemoteUpdateQuery`

Only the `USDARemoteSelectQuery` class contains code, as we are limiting the scope to read-only operations on the remote entities.

The `execute()` method has been amended to support the REST service including the `page` and `pagesize` HTTP query string parameters supported by the index endpoint. The latter will be used to import entities in chunks.

In the `usda_remote` module, you'll see `usda_remote_remote_entity_query_table_info`, which is only implemented to prevent a PHP warning. It should not be needed when Remote Entity API is released.

Entity exposure code

Inside the `usda_remote` module, you will see the empty `usda_remote_entity_info()` function. In this function, we expose the database table we defined earlier rather than the source entity schema. We also need additional directives for Remote Entity API's magic.

Add the following code into the `usda_remote_entity_info()` function:

```
$info = array();
$info['usda_food_des'] = array(
    'label' => t('USDA food description'),
    'entity class' => 'Entity',
    'controller class' => 'RemoteEntityAPIDefaultController',
    'metadata controller class'
        => 'RemoteEntityAPIDefaultMetadataController',
    'base table' => 'usda_food_des',
    'fieldable' => FALSE,
```

```
'entity keys' => array(
  'id' => 'eid',
  'label' => 'long_desc',
),
'view modes' => array(
  'full' => array(
    'label' => t('Full content'),
    'custom settings' => FALSE,
  ),
),
'label callback' => 'remote_entity_entity_label',
'uri callback' => 'entity_class_uri',
'module' => 'usda',
'access callback' => 'usda_remote_admin_access',
// Enable the entity API's admin UI.
'admin ui' => array(
  'path' => 'admin/content/usda',
  'file' => 'usda_remote.admin.inc',
  'controller class' => 'RemoteEntityEntityUIController',
),
// Remote Entity API directives
'remote base table' => 'usda_food_des',
'remote entity keys' => array(
  'remote id' => 'ndb_no',
  'label' => 'long_desc',
),
);

// Setup the property map
$remote_properties = _usda_remote_remote_properties();
foreach ($info as $key => $einfo) {
  $info[$key]['property map'] =
    drupal_map_assoc(array_keys($remote_properties[$key]));
}

return $info;
```

In the previous entity declaration, you'll see all the variations highlighted. We need to use different controller classes and a different label callback. After the Remote Entity API directives comment, we have added the following additional directives:

- remote base table: Can be used to build remote queries or endpoint URLs
- remote entity keys: Maps local properties to remote identifier properties
- property map: A map between all remote properties and local properties

Our `property map` implementation maps remote properties to local properties with the same name. Some other exposure code defines the user permission in `usda_remote_permission` and the access handler `usda_remote_admin_access`.

Entity metadata API integration

You'll also notice that our `property map` code mentioned previously refers to an array returned by `_usda_remote_remote_properties()`. We had to implement our own Entity Metadata API integration as the functionality in Remote Entity API is unfinished at the time of writing. We add the following code to `usda_remote_entity_property_info`:

```
$entity_types = array('usda_food_des');
$remote_properties = _usda_remote_remote_properties();

foreach ($entity_types as $entity_type) {
    $properties = &$info[$entity_type]['properties'];
    $entity_data = &$properties['entity_data'];
    $pp = &$remote_properties[$entity_type];
    $entity_data['type'] = 'remote_entity_'.$entity_type;

    foreach ($pp as $key => $pinfo) {
        $pp[$key]['label'] = $key;
        $pp[$key]['getter callback'] = 'entity_property_verbatim_get';
    }
    $entity_data['property info'] = $pp;
}
```

This code notifies the Entity Metadata API about the properties found in the `entity_data` field storing a serialized copy of remote data. We have set the type of `entity_data` in the `usda_food_des` entity type to `remote_entity_usda_food_des`. This is so that it's not treated as a text field causing errors. The inner `foreach` loop cycles through the properties of `entity_data` making each one accessible to wrapper code. The inner `foreach` loop and the assignment line after it are optional, as Remote Entity API automatically makes all remote properties declared in `property map` available at the top-level of the entity.

Import and administration code

The only thing remaining is an import UI to enable an administrator to import the remote entities. You will see a menu item declared in the `usda_remote_menu()` function along with supporting code in `usda_remote.admin.inc` and `usda_remote.batch.inc`. We set up the batch in the former file's `usda_remote_import_form_submit`, and then it invokes a batch function in the latter file. The batch processing function is `usda_remote_import_data()`, and it is invoked for each entity type returned by `usda_remote_entity_info()`.

Replace the comment in `usda_remote_import_data()` with the following code:

```
$controller = entity_get_controller($entity_type);
$query = $controller->getRemoteEntityQuery();
$query->pager['page'] = $context['sandbox']['progress'] / $query->pager['limit'];

try {
  $entities = $controller->executeRemoteEntityQuery($query);
  $context['sandbox']['current'] = count($entities);
  $context['sandbox']['progress'] += $context['sandbox']['current'];
}
catch (Exception $e) {
  ;
}
```

In the preceding code, we retrieve the entity controller, and then from it we retrieve the `RemoteEntityQuery` subclass instance we implemented earlier. In line three, we adjust the page number to match our progress then execute the query in the first line of the `try` block. We could also adjust the `pagesize` attribute of the query's `pager` property to change the number of records we retrieve.

Running

Once you have all your code in place, clear all Drupal caches. Point your browser to `admin/content/usda/import` and, on that page, click on the **Import** button. Once the import is complete, you'll be redirected back to the import page. Click on the **List** tab (`admin/content/usda`) and you'll see the imported remote entities.

USDA food descriptions		
<div>List</div> <div>Import data</div>		
Label	Operations	
Butter, salted	edit	delete
Butter, whipped, with salt	edit	delete
Butter oil, anhydrous	edit	delete
Cheese, blue	edit	delete
Cheese, brick	edit	delete
Cheese, brie	edit	delete
Cheese, camembert	edit	delete
Cheese, caraway	edit	delete
Cheese, cheddar	edit	delete
Cheese, cheshire	edit	delete
Cheese, colby	edit	delete
Cheese, cottage, creamed, large or small curd	edit	delete

Listing of imported remote USDA food description entities

Adding write support

While write support is out of the scope for this text, a quick mention of how to do it may help. The `usda_remote.query.inc` file has two write-related classes that need some implementation:

- `USDARemoteInsertQuery`
- `USDARemoteUpdateQuery`

To give administrators a user interface for editing entities, add code to the form handler named `usda_food_des_form` in the `usda_remote.admin.inc` file.

Customization for your use case

The code that we have prepared in this chapter can be easily used as a template for your own implementations. It must be emphasized that Remote Entity API is still in development, and you may need to reference chapter addendums online to get your code working with later releases of the module.

You will find in-depth information about all the supported hooks and hook extensions of the Remote Entity API in the `remote_entity.api.inc` file.

Summary

In this chapter, we introduced Remote Entity API and covered the requirements for exposing remote entities to a Drupal 7 site. We then implemented an example exposing remote USDA food descriptions to Drupal using Remote Entity API.

This brings our Drupal 7 entity cooking adventure to an end, and together we have consumed some light snacks along with some slow-cooked hearty stews. Your humble author hopes you will successfully prepare many Drupal entity dishes in the future using what you learned from this book.

That's right... "Good Codes!"

Index

Symbols

`$field_name` 49
`__construct()` method 98
`_usda_remote_remote_properties()` 107

A

access callback entity 94
administration code 108
archived property 74
author property 36, 39

B

base table key directive 85
body field
 about 37
 format 38
 safe_summary 38
 safe_value 38
 summary 38
 value 38
body property 37
box perspective 8
bundle
 about 10
 information, exposing 91-94
 information, storing 87-91
 multiple bundles 86, 87
bundle definition entity 93
bundle directive 86
bundle keys function 92
bundle of entity 94

C

cart module 13
changed field 104
changed property 36
checkout module 13
cid property 39
code
 adding 94-100
 copying, to recipe module 60, 61, 80
 updating 82
code snippet 19, 21
comment_body field 39
comment_body property 39
comment_count_new property 37
comment_count property 37
comment entity
 about 39, 40
 author property 40
 node property 40
comment entity, property
 author property 39
 cid property 39
 comment_body property 39
 created property 39
 edit_url property 39
 homepage property 39
 hostname property 39
 mail property 39
 name property 39
 node property 39
 parent property 39
 status property 39
 subject property 39
 url property 39

- comment property 37
- commerce products 13
- compound type 37
- connection code 104
- Content Construction Kit (CCK) 11
- content type
 - field collection, attaching 77
- controller class key directive 85
- created field 104
- created property 36, 39
- Create Retrieve Update Delete (CRUD) 10

D

- data
 - exposing 84, 85
- database schema 104
- data field 89
- date field 47
- date (ISO format) field 47
- datetime field 51, 53
- date (UNIX timestamp) field 47
- decimal field 47
- decode option 22
- deleted field 104
- delete() method 21
- description property 30, 40
- devel module 18
- dpm() function 18
- Drupal
 - history 71
 - URL 103
- Drupal core
 - bundles 12
 - entity type 11
 - fieldability 11
- Drush commands 18
- drush_pde_entity_delete() function 21
- drush_pde_entity_update() function 21
- drush_pde_print_entity() function 19
- dump-entity-properties (dep) Drush command 26

E

- ECK 84
- edit_url property 36, 39
- eid field 104

entity

- about 8
- box perspective 8
- bundles 10
- comment entity 39, 40
- exposing 83
- fieldable entity 35, 36
- fields 11
- fields, allowing 85
- file entity 26-29
- interface perspective 8
- introspection 18
- node entity 36-39
- non-fieldable 25, 26
- programming, limitation 22
- properties 17
- structure perspective 8
- term entity 40-45
- types 10
- use cases 12
- vocabulary entity 30, 31
- EntityAPIControllerExportable entity 94
- Entity Class API
 - URL 22
- entity class key directive 85
- Entity Construction Kit. *See* ECK
- entity-create (ec) Drush command 18
- entity, creating
 - code snippet 18, 19
 - Drush commands 18
- entity_data field 104
- EntityDefaultUIController class 98
- entity-delete (ed) Drush command 21
- entity, deleting
 - code snippet 21
 - Drush commands 21
- EntityDrupalWrapper class 16, 50
- entity exposure code 105-107
- entity_id() function 17
- entity keys 86
- EntityListWrapper class 15, 16
- EntityListWrapper instance 50
- entity metadata API
 - integrating 107
- entity metadata wrapper
 - about 15
 - object, creating 16

- URL 23
- using 18
- EntityMetadataWrapper class API**
 - URL 23
- EntityMetadataWrapperException 50**
- entity_metadata_wrapper() function**
 - about 17, 33
 - URL 22
- entity metadata wrapper, using**
 - create 18
 - delete 21
 - retrieve 19
 - update 21
- entity module 9, 93**
- entity, property**
 - identifying property 17
 - label property 17
- entity_property_type_map() function 104, 105**
- entity-read (er) Drush command 19**
- entity, retrieving**
 - code snippet 19
 - Drush commands 19
- EntityStructureWrapper class**
 - about 15, 50
 - EntityDrupalWrapper class 16
- entity_translation module 13**
- entity, types**
 - comment 12
 - file 12
 - node 12
 - term 12
 - user 12
 - vocabulary 12
- entity-update (eu) Drush command 21**
- entity, updating**
 - code snippet 21
 - Drush commands 21
- entity, use cases**
 - commerce products 13
 - internationalization 13
 - Stock-Keeping Unit (SKU) 13
 - user profiles 12
- EntityValueWrapper class 15**
- EntityValueWrapper object 50**
- execute() method 105**

- expires field 104**
- exportable entity 94**

F

- Feeds module 101**
- fid property 27**
- field**
 - about 11
 - allowing, on entity 85
 - creating 56
 - exporting 78, 79
 - exporting, to feature 58, 59
 - multi-value field 48-50
 - single-value field 48-50
 - structure fields 50
 - types 47, 48
- fieldability 11**
- fieldable entity 35, 36**
- field collection**
 - adding, to node 76
 - adding, to recipe node 77
 - archived property 74
 - attaching, to content type 77
 - code, copying to recipe module 80
 - code, updating 82
 - entities 74, 76
 - exporting 78, 79
 - field, creating 71, 72
 - field_name property 74
 - host_entity property 74
 - item_id property 74
 - recipe.module, tweaking 80, 81
 - revision_id property 74
 - url property 74
- Field Collection module 71**
- field_create_field() function 64, 65**
- field_create_instance() function 64, 65**
- field_info_extra_fields() 64**
- field_item_file wrapper 51**
- field_item_image wrapper 51**
- field_item_link wrapper 51**
- field_name property 74**
- field, types**
 - date field 47
 - date (ISO format) field 47
 - datetime field 51, 53

- date (UNIX timestamp) field 47
- decimal field 47
- file field 47, 51, 52
- float field 47
- image field 47, 51, 52
- Integer field 48
- link field 48, 51, 52
- Long text and summary field 48
- Long text field 48
- Text field 48
- text_with_summary field 51
- file entities, wrapper property**
 - fid property 27
 - mime property 27
 - name property 27
 - owner property 27
 - size property 27
 - timestamp property 27
 - url property 27
- file entity 26-29, 94**
- file field 47, 51, 52**
- file_save_data() function 28**
- float field 47**

G

- getIdentifier() method 20**
- getPropertyInfo() method 18, 52**
- getRemoteEntityQuery() function 104**

H

- help command 18**
- hierarchy property 30**
- homepage property 39**
- host_entity property 74**
- hostname property 39**

I

- identifying property 17**
- image field 47, 51, 52**
- import 108**
- Ingredient name field 70**
- IngredientType class 98**
- IngredientType entity 94**
- IngredientTypeUIController class 98**

- IngredientTypeUIController entity 94**
- instanceof operator 49**
- Integer field 48**
- interface perspective 8**
- internationalization 13**
- isLocked() method 98**
- is_new property 36**
- item_id property 74**
- IteratorAggregate interface 20**

L

- label callback function 92**
- label() method 20, 50, 76, 92**
- label property 17**
- language property 36**
- link field 48, 51**
- load() method 101**
- locked directive 62**
- log property 37**
- Long text and summary field 48**
- Long text field 48**

M

- machine_name property 30**
- mail property 39**
- Managed files 27**
- mime property 27**
- module code**
 - upgrading 55
- module function 92**
- modules, dealing with entities**
 - entity modules 9
 - solution modules 9
- multi-value field 48-50**
- myproperty 22**

N

- name entity 94**
- name property 27, 30, 39, 40**
- needs_remote_save field 104**
- nid property 36**
- node**
 - about 10
 - field collection, adding 76, 77
- node_count property 41**

- node entity** 36-38
- node entity, property**
 - author property 36
 - body property 37
 - changed property 36
 - comment_count_new property 37
 - comment_count property 37
 - comment property 37
 - created property 36
 - edit_url property 36
 - is_new property 36
 - language property 36
 - log property 37
 - nid property 36
 - promote property 36
 - revision property 37
 - source property 37
 - status property 36
 - sticky property 36
 - title property 36
 - type property 36
 - url property 36
 - vid property 36
- node property** 39
- non-fieldable entities** 25, 26

O

- owner property** 27

P

- pager property** 108
- parent property** 39, 41
- parents_all property** 41, 42
- parents property** 42
- path entity** 94
- pde_drush_print_entity()** function 75
- pde_entity_value()** function 52, 75
- pde_field_value()** function 74, 76
- pde module** 26
- pde_structure_value()** function 51, 54, 55
- print-entity command** 74
- Processing/Notes field** 70
- processing type** 38
- promote property** 36
- property map directive** 106
- property value** 20

Q

- Quantity field** 70

R

- raw() method** 20
- recipe content type**
 - code, copying to recipe module 60, 61
 - converting, to use fields 55
 - fields, creating 56, 57
 - fields, exporting to feature 58, 59
 - recipe.info, tweaking 61-63
 - recipe.module, tweaking 61-63
 - recipe.module, upgrading 63
- recipe_cooktime** 57
- recipe_default_recipe_ingredient_type()** function 99
- recipe_description** 56
- recipe_field_default_fields()** function 64
- recipe_field_extra_field()** function 62
- recipe_field_extra_fields()** function 80
- recipe_form()** function 63
- recipe.info**
 - tweaking 61, 62
- recipe_ingredient_access()** function 99
- recipe_ingredient_get_types()** function 99
- recipe_ingredient_type_load()** function 93, 99
- recipe_install_fields()** function 63, 65, 82
- recipe.install file** 31, 65
- recipe_install()** function 42, 67
- recipe_instructions** 57
- recipe_load()** function 63
- recipe.module**
 - code, copying 80
 - recipe_cooktime 57
 - recipe_description 56
 - recipe_instructions 57
 - recipe_notes 56
 - recipe_preptime 57
 - recipe_source 56
 - recipe_yield 56
 - recipe_yield_unit 56
 - tweaking 61-63, 80, 81
 - upgrading 63
- recipe.module file** 84

- recipe node**
 - field collection, adding 77
- recipe_node_info() function** 62
- recipe_notes** 56
- recipe_permissions() function** 100
- recipe_preptime** 57
- recipe_schema() function** 86
- recipe site vocabularies** 31, 32
- recipe_source** 56
- recipe_yield** 56
- recipe_yield_unit** 56
- remote base table directive** 106
- remote entity**
 - customizing, for use case 110
 - exposure, implementing 103
 - exposure, requisites 102
 - implementing 103
- Remote Entity API**
 - about 101
 - advantages 102
- remote_entity.api.inc file** 110
- remote entity exposure**
 - running 108
 - write support, adding 109
- remote entity exposure, implementing**
 - administration code 108
 - code 105-107
 - connection node 104
 - database schema 104
 - Entity metadata API, integrating 107
 - importing 108
 - remote query code 105
- remote entity keys directive** 106
- remote_entity_load() function** 104
- RemoteEntityQuery subclass** 108
- remote_entity_save() function** 104
- remote_id field** 104
- remote query code** 105
- remote_saved field** 104
- revision_id property** 74
- revision property** 37

S

- sanitize option** 22
- save() method** 18, 21, 26, 98, 101
- services_entity module** 103

- single-value field** 48-50
- size property** 27
- solution modules** 9
- source property** 37
- spaghetti code** 8
- status field** 89
- status property** 27, 36, 39
- sticky property** 36
- storage value** 20
- structure fields**
 - field type 51
- structure perspective** 8
- struct wrapper** 51
- subject property** 39
- system_retrieve_file() API function** 28

T

- taxonomy module** 30
- taxonomy_term entities** 41
- taxonomy_vocabulary entity** 41
- taxonomy_vocabulary_machine_name_load() function** 33
- term_count property** 30
- term entity** 41-45
- term entity, property**
 - description property 40
 - name property 40
 - node_count property 41
 - parent property 41
 - parents_all property 41
 - tid property 40
 - url property 41
 - vocabulary property 41
 - weight property 41
- Text field** 48
- text_formatted wrapper** 51
- text property values**
 - using 22
- text_with_summary field** 51
- tid property** 40
- timestamp field** 27
- timestamp property** 27
- title property** 36
- type field** 104
- type() method** 19, 51
- type property** 17, 36

U

- Units field 70
- Unmanaged files 27
- update() function 89
- uri callback function 92
- uri() method 92
- url property 27, 36, 39, 41, 74
- usda_remote.admin.inc file 103
- usda_remote.batch.inc file 103
- usda_remote.clients.inc file 103
- usda_remote_entity_info() function 105, 108
- usda_remote_import_data() function 108
- usda_remote.info file 103
- USDARemoteInsertQuery class 105, 109
- usda_remote.install file 103, 104
- usda_remote_menu() function 108, 110
- usda_remote module 103
- usda_remote.module file 103
- usda_remote.query.inc file 103, 105, 109
- USDARemoteSelectQuery class 105
- USDARemoteUpdateQuery class 105, 109
- use cases
 - about 12, 13
 - customizing, for remote entity 110
- user profiles 12

V

- value() function 37, 38
- value() method 19-22, 50, 53
- vid property 30, 36
- vocabularies
 - cuisine 31
 - difficulty 31
- vocabulary entity 30, 31
- vocabulary entity, wrapper property
 - description property 30
 - machine_name property 30
 - name property 30
 - term_count property 30
 - vid property 30
- vocabulary property 41

W

- weight property 41
- wrapper type
 - field_item_file wrapper 51
 - field_item_image wrapper 51
 - field_item_link wrapper 51
 - struct wrapper 51
 - text_formatted wrapper 51



Thank you for buying Programming Drupal 7 Entities

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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

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, 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.



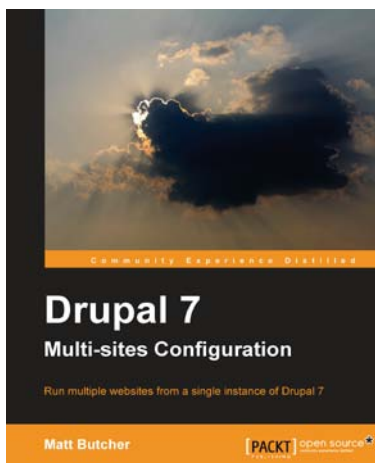
Drupal 7 Multilingual Sites

ISBN: 978-1-84951-818-5

Paperback: 140 pages

A hands-on, practical guide for configuring your Drupal 7 website to handle all languages for your site users

1. Prepare your Drupal site to handle content in different languages easily.
2. Apply the numerous multilingual modules to your Drupal site and configure it for any number of different languages.
3. Organize the multilingual pieces into logical areas for easier handling.



Drupal 7 Multi-sites Configuration

ISBN: 978-1-84951-800-0

Paperback: 100 pages

Run multiple websites from a single instance of Drupal 7

1. Prepare your server for hosting multiple sites.
2. Configure and install several sites on one instance of Drupal.
3. Manage and share themes and modules across the multi-site configuration.

Please check www.PacktPub.com for information on our titles



Migrating to Drupal 7

ISBN: 978-1-78216-054-0 Paperback: 158 pages

Learn how to quickly and efficiently migrate content into Drupal 7 from a variety of sources including Drupal 6 using automated migration and import processes

1. Learn how to import content and data into your Drupal 7 site from other websites, content management systems, and databases.
2. Upgrade your Drupal 6 site to Drupal 7 and migrate your CCK based content into the Drupal 7 fields based framework.
3. Use modules that will automate the import and migration process including the Feeds and Migrate modules.



Instant Drupal Rules How-to

ISBN: 978-1-84951-998-4 Paperback: 74 pages

Discover the power of the Rules framework to turn your Drupal 7 installation into an action-based, interactive application

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Leverage the power of Rules and Views Bulk Operations.
3. Re-use configurations using Components.
4. Create your own Events, Conditions and Actions.

Please check www.PacktPub.com for information on our titles