

PHP 5 E-commerce Development

Create a flexible framework in PHP for a powerful ecommerce solution

Michael Peacock

PACKT PUBLISHING

PHP 5 e-commerce Development

Create a flexible framework in PHP for a powerful e-commerce solution

Michael Peacock

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

PHP 5 e-commerce Development

Copyright © 2010 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: January 2010

Production Reference: 1140110

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847199-64-5

www.packtpub.com

Cover Image by Vinayak Chittar (vinayak.chittar@gmail.com)

Credits

Author
Michael Peacock

Reviewers
Chetankumar Akarte
Tahsin Hasan

Acquisition Editor
Douglas Paterson

Development Editor
Swapna V. Verlekar

Technical Editor
Ishita Dhabalia

Indexer
Rekha Nair

Proofreader
Sandra Hopper

Production Editorial Manager
Abhijeet Deobhakta

Editorial Team Leader
Gagandeep Singh

Project Team Leader
Lata Basantani

Project Coordinator
Poorvi Nair

Graphics
Geetanjali Sawant

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Author

Michael Peacock (`http://www.michaelpeacock.co.uk`) is a web developer from Newcastle on Tyne, UK with a degree in Software Engineering from the University of Durham. After meeting his business partner while studying at Durham, he co-founded Peacock Carter Limited (`http://www.peacockcarter.co.uk`), a Newcastle-based creative consultancy specializing in web design, web development, and corporate identity. Michael loves working on web-related projects and new business ideas, usually with interests in several companies.

He has been involved with a number of books, having written four books: *PHP 5 e-commerce Development*, *Drupal 6 Social Networking*, *Selling Online with Drupal e-Commerce*, and *Building Websites with TYPO3*, and acted as technical reviewer for two others—*Mobile Web Development* and *Drupal for Education and E-Learning*.

You can follow Michael on Twitter: `http://www.twitter.com/michaelpeacock`.

I'd like to thank everybody at Packt Publishing, in particular Douglas Paterson for working with me on building the idea of this book into a suitable structure; Poorvi Nair for helping to keep the book on track; Swapna Verlekar, the development editor; and of course, the technical reviewers, Chetan Akarte and Tahsin Hasan, who helped improve the quality of the book.

My thanks also go to my friends and family, in particular my fiancée Emma for her support while working on the book.

Finally, I'd like to thank you, the reader; I hope that you enjoy this book, and produce a fantastic e-commerce website of your own. I look forward to hearing your feedback and seeing what e-commerce sites you come up with!

About the Reviewers

Chetankumar Akarte is working on PHP since last 5 years. He has extensively worked on small- and large-scale PHP e-commerce, social networking, Wordpress, and Joomla-based web projects. Over the years, Chetan has been actively involved in "Xfunda Developers Community" and has regularly been blogging on Microsoft .NET technology at `http://www.tipsntracks.com`.

Chetan received a Bachelor of Engineering degree in Electronics from the Nagpur University, India in 2006. He likes to contribute on the newsgroups and forums, and has written articles for Electronics For You, DeveloperIQ, and Flash & Flex Developer's Magazine.

Chetan lives in Navi Mumbai, India. You can visit his websites `http://www.xfunda.com` and `http://www.tipsntracks.com` or get in touch with him at `chetan.akarte@gmail.com`.

I would like to thank my sister Poonam and Jijaji Vinay for their consistent support and encouragement. I'd also like to thank Packt Publishing and especially my Project Coordinator Poorvi for giving me the opportunity to do something useful.

Tahsin Hasan is a software engineer. He passed the Zend Certification Exam on August 09, 2009 and has become the seventeenth Zend Certified Engineer (ZCE) from Bangladesh. This is the topmost certification on PHP from Zend, the developer of this outstanding scripting language. He is a tech enthusiastic and always keeps himself well-equipped with latest technologies. He has completed his M.Sc. and B.Sc. in Computer Science and Engineering from Jahangirnagar University.

Tahsin Hasan has profound knowledge of LAMP environment. His advanced understanding of database environments and Apache web server is an asset. He has proficiency in scalability and optimizing of server performance. He has worked with Zend framework, CakePHP, Codeigniter, and Symfony.

I'd like to give thanks to my parents and my siblings for their encouragement and also a special thanks to Poorvi Nair and Swapna Verlekar from Packt Publishing.

Table of Contents

Preface 1

Chapter 1: PHP e-commerce 7

e-commerce: Who, what, where, why? 7

An overview of e-commerce 7

eBay 8

Amazon 8

Brick 'N Mortar stores 8

Service-based companies 8

Why use e-commerce? 9

Rolling out your own framework 9

Why PHP? 9

Why a framework? 9

When to use an existing package? 10

Existing products 10

A look at e-commerce sites 10

iStockphoto 11

WooThemes 11

eBay 12

Amazon 12

Play.com 12

e-commerce: What does it need to do/have? 13

Products 13

Checkout process 14

General 14

Our framework: What is it going to do? 14

Our framework: Why is it going to do it? 15

Juniper Theatricals 16

Summary 17

Chapter 2: Planning our Framework 19
Designing a killer framework 19
Patterns 20
Model-View-Controller (MVC) 20
Registry 21
Singleton 22
Structure 23
Building a killer framework 24
Pattern implementation 25
MVC 25
Registry 25
Singleton 27
Registry objects 29
Routing requests 54
An alternative: With a router 54
Processing the incoming URL within our registry object 55
index.php 56
.htaccess file 58
Configuration file 58
What about e-commerce? 59
An e-commerce registry? 59
Summary 60
Chapter 3: Products and Categories 61
What we need 61
Product information 62
Category information 62
Structuring content within our framework 63
Pages 63
Content 63
Versioning 64
Building products, categories, and content functionality into our framework 65
Database 65
Content 65
Content types 67
Content versions 68
Products 69
Categories 70
Pages within our framework 70
Model 70
View 73
Controller 74
Products 76
Model 76
View 80

Controller 81
Categories 83
Model 84
View 87
Controller 89
Some thoughts 92
Product and category images 92
Routing products and categories 92
Featured products 93
Embedding products 93
Summary 94
Chapter 4: Product Variations and User Uploads 95
Giving users choice 95
Simple variants 96
How could this work? 96
Combinations of variants 96
How will this work? 96
High-level overview 97
Database structure 98
Template switching 100
Templates 103
A look back at simple variants 104
Giving users control 104
How to customize a product? 105
Uploads 105
Custom text 105
Maintaining uploads 106
Security considerations 107
Database changes 107
Extending our products table 107
Template switching 108
Shopping basket preparation 110
Stock control 110
Product variations 111
Product customizations 111
Basket templates 111
Product subtotals 111
Summary 112
Chapter 5: Enhancing the User Experience 113
Juniper Theatricals 113
The importance of user experience 114
Search 114
Finding products 114

Search box 115
Controlling searches with the products controller 115
Search results 117
Improving searches 118
Filtering products 119
Product attributes 120
Filter options 122
Processing filter requests 125
Displaying filtered products 129
Improving product filtering 130
Providing wish lists 130
Creating the structure 131
Saving wishes 132
Wish-list controller 132
Add to wish list 135
Viewing a wish list 135
Controller changes 135
Wish-list view 137
Purchases 137
Gift purchases 138
Self purchases 138
Improving the wish list 138
Recommendations 139
Related products 139
Controlling the related products 141
Viewing the related products 142
E-mail recommendations 142
Help! It's out of stock! 143
Detecting stock levels 144
Changing our controller 144
Out of stock: A new template bit 144
Tell me when it is back in stock please! 145
Stock alerts database table 145
More controller changes 146
It is back! 148
Giving power to customers 148
Product ratings 148
Saving a rating 149
Viewing ratings 151
Product reviews 152
Processing reviews/comments 153
Displaying reviews/comments 154
Combining the two? 155
Any other experience improvements to consider? 155
Summary 156

Chapter 6: The Shopping Basket 157
Shopping baskets 157
Our basket 158
Per-page basket 158
Considerations for our shopping basket 159
Creating a basket 160
When to build a user's basket 160
Basket database 160
Basket contents 161
Viewing the basket 162
checkBasket method 162
The controller 164
Adding products 165
An addProduct method 165
The controller 168
A note on etiquette 170
Adding customizable products 170
Changing our basket database 171
Viewing the basket 171
Changing the model 171
The controller 172
Adding product variants 172
A new database table 173
Model changes 173
The controller 174
Editing quantities 174
From visitor to a user 177
The transferToUser function 177
Performing the transfer 177
Cleaning the basket 178
Expired contents 178
Displaying the basket on every page 178
Functionality 179
Summary 180
Chapter 7: The Checkout and Order Process 181
Some examples 181
Amazon 182
Limitations 183
Useful features 184
eBay 185
Interesting points of note 186
Play.com 187
Interesting points of note 188

The process **189**
The basket 189
Voucher codes 189
Shipping method 190
An overview 190
Authentication 190
Why should we authenticate the user at this stage? 191
Login 191
Register 191
Do nothing 191
Delivery address 191
Payment method 192
Offline payment method 192
Off-site payment method 192
On-site payment method 193
Confirmation 193
Payment details 193
Payment made 194
Order processed 194
Other points of note 194
Summary **195**
Chapter 8: Shipping and Tax **197**
Shipping **197**
Shipping methods 197
Shipping costs 199
Product-based shipping costs 200
Weight-based shipping costs 200
To think about: Location-based shipping costs 201
Shipping rules 202
Free shipping 204
Capped shipping 204
Tracking 204
Integrating shipping costs into the basket 205
Shipping methods and a default 205
Calculating shipping costs based on products 205
Calculating shipping costs based on product weights 206
Considering shipping rules, and adjusting prices accordingly 207
Tax **209**
Separately calculating tax values 210
To think about: Location-based tax costs 211
A look at our basket now **211**
Summary **212**

Chapter 9: Discounts, Vouchers, and Referrals 213
Discount codes 213
Discount codes data 214
Discount codes database 215
Discount codes functionality 215
Reducing the number of codes available 219
Purchasable voucher codes 219
Existing functionality 219
Discount codes 219
Product variations 220
Required additional functionality 220
Referrals 220
Database changes 221
New table: Referrers 221
Changes 221
Functionality 222
Checkout process consideration 222
Summary 222
Chapter 10: Checkout 223
Order process review 223
Authentication 225
Delivery address 227
Payment method 228
Confirmation 230
Storing orders in the database 230
Orders table 231
Order statuses 232
Order items 232
Order item attributes 233
Payment methods 233
Summary 233
Chapter 11: Taking Payment for Orders 235
Taking payment 235
Our payment system 235
Taking payment online 237
PayPal 237
The payment button 237
Processing payment to update the order 239
Direct with a credit/debit card 242
Storing card details 242
Not storing card details 243
Other payment gateways 244
Payment gateway tips 244

Taking payment offline **245**
Summary **245**
Chapter 12: User Account Features **247**
User account area **247**
Changing details **247**
Changing password 248
Changing default delivery address 249
Viewing orders **250**
Listing orders 250
Query 251
Viewing an order 251
Order model 251
Cancelling an order 253
Order model additions 254
Controller code 255
Expansion **256**
Summary **257**
Chapter 13: Administration **259**
Dashboard **260**
Products and categories **261**
Products 261
Creating a product 261
Editing a product 265
Categories 265
Creating a category 265
Editing a category 266
Deleting a category 266
Orders and customers **266**
Orders 267
Updating an order 267
Dispatch note 268
Refunds 268
Customers area 269
Listing customers 269
A customer's orders 269
Miscellaneous **269**
Shipping 269
Creating a shipping method 270
Voucher codes 270
Creating a voucher code 270
Summary **271**

Chapter 14: Deploying, Security, and Maintenance 273
Deploying 273
Hosting accounts and domain names 274
Hosting providers 275
Domain name registrars 276
Manual deployment 276
Setting up the database 276
Uploading our store 279
Settings 280
Automated deployment 280
Security 281
Server security 281
Software 281
Securing the site with a firewall 282
Passwords 282
SSL/TLS 283
CAPTCHA 283
Maintenance 283
Backing up and restoring 284
Using cPanel 284
Using the command line (SSH) 286
Summary 287
Chapter 15: Marketing, SEO, and Customer Retention 289
Marketing sites and stores powered by our framework (and other sites for that matter) 290
Online advertising 290
Buying advertising space 290
Pay-per-click advertisements 291
Advertisement networks provided by search engines 292
Newsletter advertising 293
A word of warning: Search engine penalization 294
Newsletters 295
Marketing materials 295
Affiliate marketing 296
Social marketing 296
Viral marketing 296
Twitter 296
RSS with FeedBurner 297
Search engine optimization 297
On-site SEO 297
Headings 297
Links 298
Up-to-date content 298
Meta tags 298

Sitemap and webmaster tools 299
Off-site SEO 300
Customer retention 300
Newsletters 301
Social features 301
Coupons and voucher codes 301
Summary 302
Appendix A: Interacting with Web Services 303
Google products 303
Adding the feed to the Google merchant center 304
Setting an update schedule 304
Creating the feed 304
Product feed controller 305
Other useful link 306
Alternative—Google Base Data API 306
Others 306
Google Analytics 306
Signing up 307
Tracking e-commerce 307
Add transaction 307
Add item 308
Track transaction 308
Further reading 309
Other services 309
Amazon 309
eBay 309
More to come 310
Summary 310
Appendix B: Downloadable Products 311
Extending products 311
Extending the payment and administration areas 312
Access database 312
Providing access 313
Rescinding access 314
Centralized download area 315
What else is needed? 315
Summary 316

Appendix C: Cookbook **317**
Authentication reminders **317**
Help! I forgot my password! 317
Generate the reset key, update the user record, and e-mail the customer 318
Reset the password 318
Help! I forgot my username! 319
E-mailing customers **319**
Integrating Campaign Monitor **320**
Integrating reCAPTCHA **320**
On the registration page 321
When processing the registration 321
Tweeting about happy customers **321**
Other uses 322
Summary **323**
Index **325**

Preface

The popularity of online shopping has increased dramatically over the past few years. There are plenty of options available if you not are planning to build your own e-commerce solution, but sometimes it's better to use your own solutions. It may be easy to find an e-commerce system but when it comes to extending it or using it, you might come across a lot of difficulties.

This book will show you how to create your own PHP framework that can be extended and used with ease, particularly for e-commerce sites. Using this framework you will be able to display and manage products, customize products, create wish lists, make recommendations to customers based on previous purchases, send e-mail notifications when certain products are in stock, rate the products online, and much more.

This book helps you build a Model-View-Controller style framework, which is then used to put together an e-commerce application. The framework contains template management, database management, and user authentication management. With core functionality in place, e-commerce-focused features are gradually added to the framework including products, categories, customizable products with different variations and customer input, wish lists, recommendations, the shopping basket, and a complete order process.

At the end of the book, you will have an e-commerce architecture that will take you from viewing or searching for products and adding them to your basket, through the checkout process and making payment for your order to your order being dispatched. Focus is placed on flexibility, so that the framework can be extended as the needs of a particular store change, as illustrated by one of the appendices, which goes through the process of modifying the store to sell downloadable products, as well as physical ones.

Supplementary information, such as how to market and promote an online store, in addition to taking regular backups and performing maintenance is also covered, ensuring you have every chance of success with your own e-commerce framework-backed store.

What this book covers

Chapter 1, *PHP e-commerce*, looks into the growing need and use of e-commerce, including various popular online retailers, and discusses what we are going to do throughout the book, and why.

Chapter 2, *Planning our Framework*, introduces you to several key architectural patterns, including MVC, Registry, and Singleton, as we develop the structure and core functionality for our framework including template management, database management, and user authentication.

Chapter 3, *Products and Categories*, takes a step further and demonstrates how to display and categorize products within our framework for our customers.

Chapter 4, *Product Variations and User Uploads,* moves on to enhancing the standard product listings with customizable products, product variations, and allowing customers to upload files with their orders.

Chapter 5, *Enhancing the User Experience*, discusses tips and tricks to enhance user experience by looking at search, product filtering, providing wish lists, sending e-mail notifications, and other useful enhancements for our customers.

Chapter 6, *The Shopping Basket*, demonstrates how to structure, build, and manage the shopping basket supporting both standard and customized products.

Chapter 7, *The Checkout and Order Process*, looks at the checkout and order process implemented by some of the popular e-stores and their pros and cons, to chalk out the process for our own framework.

Chapter 8, *Shipping and Taxes*, focuses on calculating shipping costs based on different methods, integrating third-party shipping APIs, sending shipping and tracking notifications on orders, and integrating tax costs into our system.

Chapter 9, *Discounts, Vouchers, and Referrals*, aims at extending our framework to encourage new customers and orders by promoting our store through discount codes, purchasable vouchers, and referral discounts.

Chapter 10, *Checkout*, ties everything together, as most of our checkout functionality is already in place, and extends our order process to leave our customers with a confirmed order, ready for their payment.

Chapter 11, *Taking Payment for Orders,* introduces payment processing to the framework, covering different modes of payment and various post-payment steps involved.

Chapter 12, *User Account Features*, walks through the development of a customers area where they can see as well as edit their orders and profile information.

Chapter 13, *Administration*, walks through the development of an administrators area where they can see orders, products, and settings, and add, edit, and remove these.

Chapter 14, *Deploying, Security, and Maintenance*, looks at deploying our site using the framework into a live environment and examines the security and maintenance concerns, introducing different ways to enhance security of our framework and to restore a live site from a backup.

Chapter 15, *Marketing, SEO, and Customer Retention*, discusses hints and tips for effectively marketing and promoting websites and e-commerce stores with online marketing techniques, search engine optimization, and customer retention strategies.

Appendix A, *Interacting with Web Services*, explains how we may interact with other e-commerce-related web services, such as Google products, Google Analytics, Amazon web services, and eBay developer center, in order to target new markets, or to make tasks easier for us.

Appendix B, *Downloadable products*, illustrates how to extend our store to allow downloadable products.

Appendix C, *Cookbook*, goes through a number of useful code snippets to enhance the framework and our store.

What you need for this book

In the course of this book, you will need the following software utilities to try out various code examples listed:

- Apache 1.3 or above (2 recommended)
- Apache `mod_rewrite` module
- MySQL 5.0
- PHP 5.0 (5.2+ recommended)

The above can be installed using a package such as WampServer 2.0 for Windows.

For development, a text editor is all that is required, though one with syntax highlighting would be beneficial.

Who this book is for

If you are a web developer, or anyone looking to increase your understanding of e-commerce site development, this book is for you. Primarily aimed at PHP developers, it is suitable for any web developer interested in enhancing their e-commerce knowledge, or developers looking to move towards PHP.

Intermediate knowledge of PHP development and object-oriented programming is assumed, and basic knowledge of e-commerce principles will be of benefit too.

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: "The suffix of `ecomframe` is used to allow us to store multiple database connection details within the same array."

A block of code is set as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>{title}</title>
  <meta http-equiv="content-type"
        content="text/html;
        charset=iso-8859-1" />
  <meta name="description" content="{metadescription}" />
  <meta name="keywords" content="{metakeywords}" />
</head>
<body>
```

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

```
SELECT v.name AS product_name, c.ID AS product_id,
    (SELECT GROUP_CONCAT( a.name,'--AV--', av.ID, '--AV--',
       av.name SEPARATOR '---ATTR---' )
     FROM product_attribute_values av,
        product_attribute_value_association ava,
        product_attributes a
     WHERE a.ID = av.attribute_id AND av.ID=ava.attribute_id
        AND ava.product_id=c.ID ORDER BY ava.order ) AS attributes,
```

```
    p.image AS product_image, p.stock AS product_stock,
    p.weight AS product_weight, p.price AS product_price,
    p.SKU AS product_sku, p.featured AS product_featured,
    v.heading AS product_heading,
    v.content AS product_description,
    v.metakeywords AS metakeywords,
    v.metarobots AS metarobots,
    v.metadescription AS metadescription
FROM content_versions v, content c, content_types t,
    content_types_products p
WHERE c.active=1 AND c.secure=0 AND c.type=t.ID
    AND t.reference='product' AND p.content_version=v.ID
    AND v.ID=c.current_revision AND c.path='{$productPath}'
```

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

```
Mysql -u username -p databasename < /home/junipert/backup.sql
```

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: "Once we have entered the username and password, we need to click on the **Next Step** button."



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 email to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, 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 for the book

Visit `http://www.packtpub.com/files/code/9645_Code.zip` to directly download the example code.

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 to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. 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 web site 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
PHP e-commerce

Welcome to building a PHP e-commerce framework! During the course of this book we are going to build a flexible e-commerce framework using PHP, which can be extended and modified for the purposes of any e-commerce site.

In this chapter, you will learn:

- The business logic behind e-commerce
- Why (and when) you should use your own system over an existing product
- The benefits of a "framework"
- About existing e-commerce sites and products

e-commerce: Who, what, where, why?

e-commerce, or electronic commerce, is the sale and purchase of goods or services through electronic means. In our case, this electronic means is the Internet. There are so many different applications of e-commerce on the Internet, including:

- Online shops selling products, such as Amazon, or the online counterparts to Brick 'N Mortar stores
- Online auctions, such as eBay
- Online services/web services such as BaseCamp, or subscription-based websites

An overview of e-commerce

e-commerce is an incredibly popular way of doing business, so let's look at who is using e-commerce and what they are using it for.

eBay

According to eBay's website, there are approximately 84 million active users of eBay, with users trading more than $1,900 worth of goods each second. That means 84 million of us are using eBay to buy and sell goods, either as a business sustaining a regular turnover, or to try and make a little extra cash by selling unwanted or unneeded things knocking about the house.

eBay is a social e-commerce site, operating as an online auction house, whereby they don't actually sell anything themselves, but instead allow their community of users to not only buy but also sell through their site. This not only illustrates the popularity of e-commerce, but also that there is money to be made in providing a stage for low (and high) volume online purchases.

Amazon

With revenue of over $19 billion in 2008, Amazon is one of the most popular e-commerce sites on the Internet. Research in early 2009 indicated that it was the favorite retailer for both video and music in the UK.

Brick 'N Mortar stores

Large, established Brick 'N Mortar stores such as Wal-Mart, Tesco, and Borders use online shops to sell the products they generally keep in store. With the likes of Wal-Mart and Tesco, customers often book a delivery timeslot for their groceries to be delivered. They also offer more than what is available in store, which they can easily bolt on for the convenience of their customers. With online retail, sellers are not confined to what they can stock on the shelves, but what they can store in their distribution warehouses.

Smaller, niche-based Brick 'N Mortar stores use online selling as a way to target their products to a wider audience, without the limitation of their existing physical presence.

Service-based companies

Companies such as 37signals are setting up online applications (such as Project Management tool, BaseCamp) with monthly subscription models. Other examples of such sites include large file distribution websites (allowing you to "e-mail" large files using a third-party website) and premium features on certain websites, for example Get Satisfaction.

Why use e-commerce?

The popularity of online shopping has increased dramatically over the past few years. Not only does it provide the convenience of allowing customers to shop in the comfort of their own home, it also allows businesses to trade on a global marketplace, targeting even more potential customers. Because everything is done electronically, e-commerce stores can also help generate recurring revenue, by recommending new products to customers based on previous purchases, and by keeping them up to date with the store's catalog.

Rolling out your own framework

Throughout the course of this book, we are going to build a framework of our own, using PHP, as opposed to making use of an existing product. Sometimes, it is more appropriate to use existing solutions; sometimes it is better to use your own solutions. As you are reading this book, hopefully you know why you want to create your own framework. However, let's look at why we are going to create ours.

Why PHP?

PHP is a very popular language, and because it isn't a framework in its own right, we can easily structure our framework out of it, however we wish. The main choice for a programming language is generally down to your own preference.

Most modern web hosts support PHP and MySQL, and while languages like Ruby on Rails are gaining popularity, at the moment hosting for them is not as common. This book assumes that you already have a reasonable understanding of PHP, so hopefully that will also be an important factor in why you want to use PHP; perhaps you need to develop something quickly, and don't want to use a language or platform that is out of your comfort zone.

Why a framework?

Instead of looking to create an e-commerce system, designed to perform all types of e-commerce tasks, we will create a framework. This will make it easy to extend the needs of any e-commerce project with minimal effort. Because we are creating our own framework, it is going to be something we will know and understand very well, meaning that if we do need to extend it or use it, we can do so easily.

While a typical e-commerce system may show products within a browse or search interface, a framework could allow us to integrate products into other areas of a website; for instance, pulling certain categories of product into relevant pages, particularly useful for a website that needs to do more than just sell online. For example, if we were selling books, we could have pages dedicated to certain authors with information, reviews, and other media about the author, and then integrate some of their popular products into the page.

When to use an existing package?

There are already a number of e-commerce systems available, written in various different languages, and sometimes it is more appropriate to use such a product, for example:

- When you have a tight deadline for a project, and you don't have a framework in place
- When there are lots of developers on the project; something with more documentation available would probably be more useful, at least initially (unless the framework was developed by most of the developers)
- When a client has indicated a preference
- When the features match—if another system has all the features you need and want, and it works in a way you are comfortable with, then it would be more appropriate to use the existing system

Existing products

Of the e-commerce applications that are already available, the following are amongst the most popular:

- **Magento**: This is a very modular and flexible e-commerce platform, which is becoming more widespread in its usage.
- **Drupal e-commerce**: Drupal is a popular content management system, which is easy to extend and modify. There are two packages of modules, Ubercart and Drupal e-commerce, which add a wealth of e-commerce functionality to the popular CMS.
- **CubeCart**: This is a simple-to-use e-commerce solution, available with both free and paid versions.

A look at e-commerce sites

We have already taken a brief look at who is using e-commerce; let's take a more detailed look at some popular e-commerce sites, and see how they work and what features they provide to their users.

iStockphoto

iStockphoto is a popular website for buying and selling stock photography. Photographers can register on the website and submit their photographs for approval. Approved photographs can then be purchased by customers for a number of credits, depending on the size of the photograph and the license they wish to purchase it under.

Features

- **Approval process for sellers**: Photographs are approved by iStockphoto before being available for purchase.
- **Flexibility for sellers**: Sellers can choose their prices, multiple image sizes, and the licenses they wish to sell the image under.
- **Credits**: Because most stock photograph purchases equate to only a few dollars, iStockphoto has a credits system whereby the customers purchase at least $10 of credits, which are assigned to their accounts. These credits are then deducted when they make a purchase.
- **Social**: Photographs can be rated and commented on, making the website very social and interactive.

WooThemes

WooThemes is quickly becoming a popular online shop for custom themes, but operates quite differently to most theme-selling websites. Purchases are either a theme package (the theme and accompanying color schemes), or a one-off fee along with a monthly subscription allowing the customer to purchase any themes they wish in a particular month, backed up with reassurances of a minimum number of themes each month.

WooThemes also invites established members of the web design community to create themes for the site, helping to raise the profile of the site and continue their ability to offer quality themes.

Features

- In-depth knowledge of industry and respected designers to help increase their product offering (not strictly a feature)
- Two types of purchases, each with two tiers, providing access to different downloads:
    - Recurring payments
    - Membership-based offers

eBay

We discussed earlier, that eBay is an online auction house, but what features does it have to support its business?

Features

- Powerful search feature to find products
- Purchase products directly—"Buy it now"
- Bid for products/express interest in purchasing
- Make payments and manage orders

Amazon

While Amazon doesn't operate as an online auction house, as eBay does, it still has a number of social aspects to it, including ratings, reviews, and recommendations. It also allows users of the site to sell their own copies of products listed within the store, through the Amazon Marketplace. This market place functionality is also integrated within their main product listing, informing customers that they can also purchase used and new copies of a product, from non-Amazon sellers, through its market place.

Features

On a basic level, Amazon.com provides the following features to its users:

- Browse and search for products
- Rate and review products
- Purchase products
- Make payments and manage orders
- Sell products through the Marketplace

Play.com

Play.com operates in a similar way to Amazon: it not only sells products, but also allows users to sell their own items (branded as PlayTrade). One notable difference with Play.com is the categorization of products, which also allows more dynamic categories such as products under a certain amount, or seasonal items (for example, Christmas present ideas).

Features

- Browse and search for products
- Rate and review products
- Purchase products
- Make payments and manage orders
- Sell products through PlayTrade

e-commerce: What does it need to do/have?

After looking at some popular e-commerce sites, our store obviously needs some key features. It needs the ability for users to search for and browse products, within different categories. Visitors to the site obviously need to be able to purchase these products, which leads to the need for a shopping basket to store products the visitor intends to purchase and a checkout process to manage delivery details, tax calculations, delivery charge calculations, payment processing, and of course order management for administrators. We will build upon these core features to build a basic feature list for our framework.

The exception with regards to those features is eBay, which forgoes the need for a shopping basket; however, it contains provisions for watching items, automatically bidding for items, and with "Buy it now" making an instant purchase.

Products

We need to have the following product-related features:

- **Finding products**: We need the functionality to both browse product listings and search for products to make it easy for customers to find products within our framework.
- **Viewing products**: Once customers have found a product that interests them, they obviously need to be able to view the product to find out more about it. This also means we need to take into consideration what type of information is related to products (for example name, price, weight, and so on). Community-orientated aspects link in nicely here too, such as ratings and reviews.
- **Expressing interest in a product**: This can be done either by adding a product to a basket, or to an interest list, for future purchase.

Checkout process

The checkout process essentially has the following three features or requirements:

- Group products into orders (unless it's an auction-style site)
- Accept and process payment for orders, or accept a note of how payment is going to be made
- Take delivery details from the customer

General

There are also some other supplementary features the framework will need to implement:

- Allow the store to be administered
- Allow customers to manage their orders, and change account information (username, password, default delivery address)

Our framework: What is it going to do?

We are going to create a framework that can do anything we need it to. Of course, the exact needs of a project vary from project to project, so we will ensure it has some fundamental features, which we can then extend to whatever we need. The following features will be the minimum that we will have our framework capable of doing:

- Displaying and managing products
- Displaying and managing categories of products
- Embedding products, listings, and categories into other aspects of a website or web application (after all, it is a framework we are creating!)
- Customizing products such as apparel
- Searching for products
- Filtering the product list based on the customers' preferences, such as brand, or other properties
- Providing wish lists, that is, lists of products that users wish to purchase at some point, or would like someone to purchase for them (including the provision to facilitate gift purchases)
- Generating recommendations based on previous purchases
- Sending e-mail notifications when certain products are in stock
- Publishing ratings and reviews of products

- Providing a shopping basket to store products and quantities of the products a customer wishes to purchase
- Calculating shipping cost
    - Based on products and/or their weights
    - Based on delivery address
    - Based on custom rules (for example free shipping to orders over a fixed amount)
- Tax cost calculations
- Managing discount codes
- Managing gift certificates
- Providing referral discounts
- Processing payments
- Allowing customers to manage their account
- Allowing us and other administrators to manage the store

Along with these features, we are also going to look at the following functionalities:

- Deploying the framework into a live store environment
- Backing up and restoring the store
- Enabling secure connections to the live store using SSL

To illustrate how our framework can be extended to meet the needs of any e-commerce situation, there are three appendices looking at different ways to extend the framework:

- Web service integration, for services such as Google Product Search
- Extending our store to support downloadable products
- Various code snippets in a cook book format, showing how to quickly extend this (and any other framework) to support some specific enhancements

Our framework: Why is it going to do it?

Most online stores are there for a particular purpose, either to sell a particular product, or to act as the online division of a Brick 'N Mortar store. Obviously, the point of creating a framework is to easily adapt to any purpose; however, for the purposes of this book, we are going to need a scenario site to create.

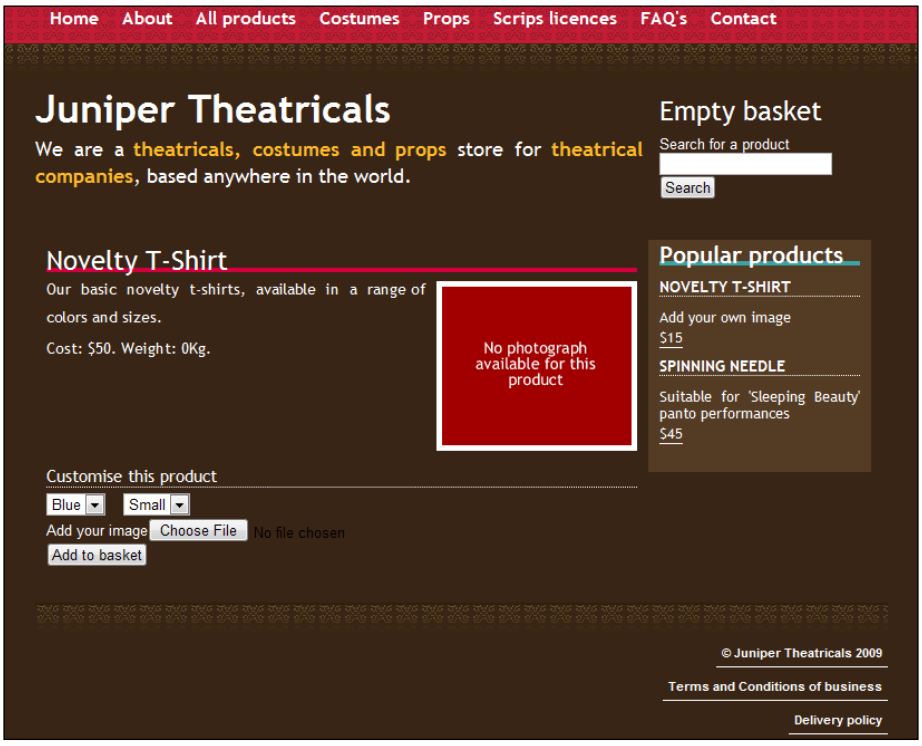
Juniper Theatricals

Juniper Theatricals is a fictional Brick 'N Mortar novelty and theatrical supplies store based in Newcastle upon Tyne, UK. They have some customers who place orders over the telephone, or over fax, and have a loyal local base of customers too. When building our framework, we are going to use it as a web presence for this fictional store. Some specific requirements of the store include, in addition to previously discussed features:

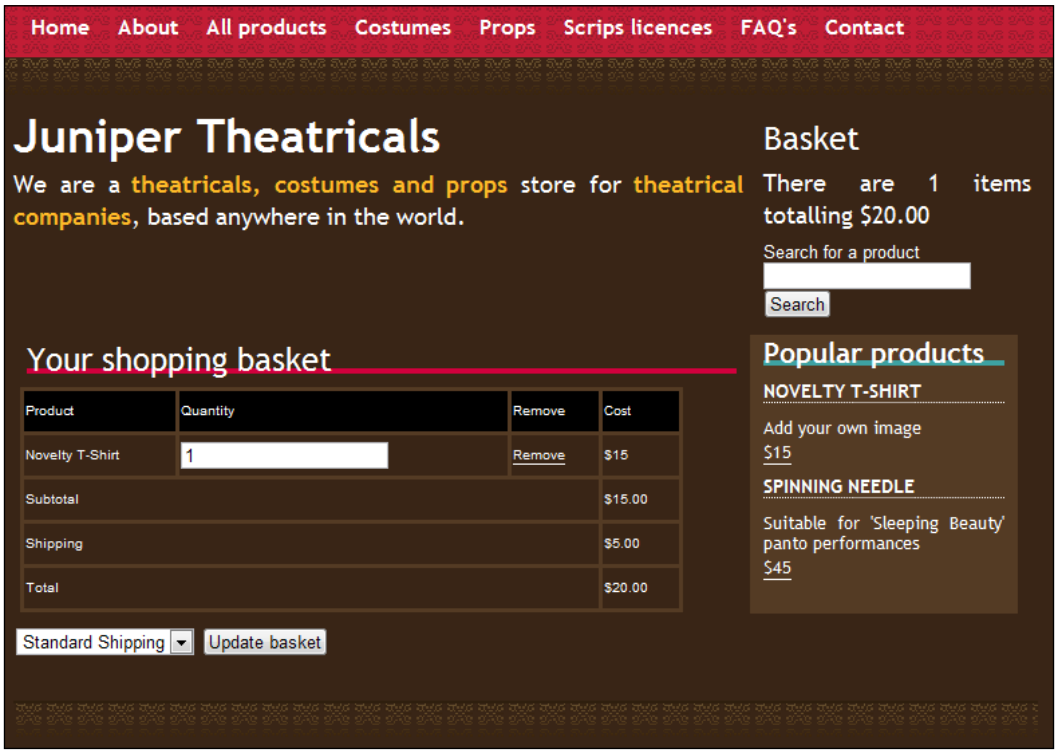
- **Customizable products**: Apparel.
- **Custom products**: User-defined images and text on apparel, perhaps the ability to list these for sale as well.
- **Virtual purchases**: Electronic tickets to events.

Because the store has no web presence, the framework needs to work for their entire website, integrating e-commerce functionality where appropriate. Although they are based in the UK, the website is designed to target new customers, primarily in the USA, so the site will use USD as its primary currency.

Let's look at what our framework will look like for our Juniper Theatricals store. Here's an insight to a product view powered by our framework:



And of course, the shopping basket itself:



Summary

In this chapter we looked at e-commerce and discussed the reasons for creating our own e-commerce framework in PHP, along with the features our framework is going to support. We also looked at some existing e-commerce setups and discussed the different types of e-commerce stores available on the Internet. Now that we know what we are going to create and why, we are ready to start building the structure and basic functionality of our framework, before adding a wealth of e-commerce functionality.

2
Planning our Framework

Now that we know more about what we are here to do, it is time to start planning our framework to ensure we get it off to the right start. In this chapter, you will learn:

- About design and architectural patterns in PHP, including:
    - Model-View-Controller
    - Registry
    - Singleton
- How to structure an extendable framework
- How the framework should work with settings for the site and e-commerce setups it powers

Let us start by designing our framework, and then building it based on our ideas for its design.

Designing a killer framework

There are many different ways to design and build frameworks. Some involve building very complicated frameworks, and others involve creating simple ones. In this book, we are going to quickly build an easy-to-use, easy-to-understand framework.

This book will serve as a guide to help you develop a framework of your own, different from the one created in this book, but better suited to your needs, ideas, and preferences. The emphasis in this book is on e-commerce, so if you already have a framework of your own, or would prefer to use an existing framework, this book will give you ideas to integrate e-commerce capabilities into any framework.

Patterns

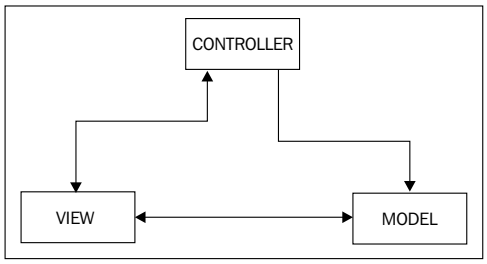
There are a number of design and architectural patterns that were designed to help provide some general, good practices and solutions to common problems within software design. There are a few patterns of particular interest to us, as we are looking to develop a framework:

- Model-View-Controller (MVC)
- Registry
- Singleton

Model-View-Controller (MVC)

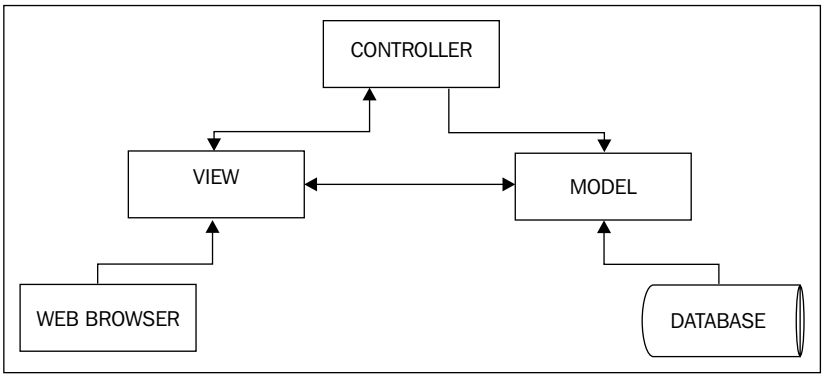
The Model-View-Controller architectural pattern provides a widely used solution to separate the user interface from the logic of an application. The user interface of the application (view) interacts with the data (model) using the controller, which contains the business rules needed to manipulate data sent to and from the model.

To put this into an e-commerce perspective, consider a customer adding a product to their shopping basket clicks on an **Add to basket** button within the view/user interface. The controller processes this request and interacts with the model (basket) to add the product to the basket. Similarly, the data from within the basket is relayed back to the user interface through the controller, to display how many products are in the basket, and the value of the contents.



Because we are creating a framework for use with websites and web applications, we can further extend the representation of the MVC pattern to reflect implementation in such a framework. As discussed earlier, the models represent data; this is primarily stored within the database. However, in our framework we will have a series of models, which take the data and store it within themselves in a more suitable format, or allowing the data to be manipulated more easily. So, we could in fact add our database to this diagram, to show the interaction with the models and the database. We are also viewing the end result of our website or web application in a web browser, which renders the views, and relays our interactions (for example mouse clicks or field submissions), back to the controller. So we could also add the

web browser to the diagram, to show its interaction with the views. This gives us a clearer understanding of how the MVC pattern will work within our framework, and where the three components sit within it.

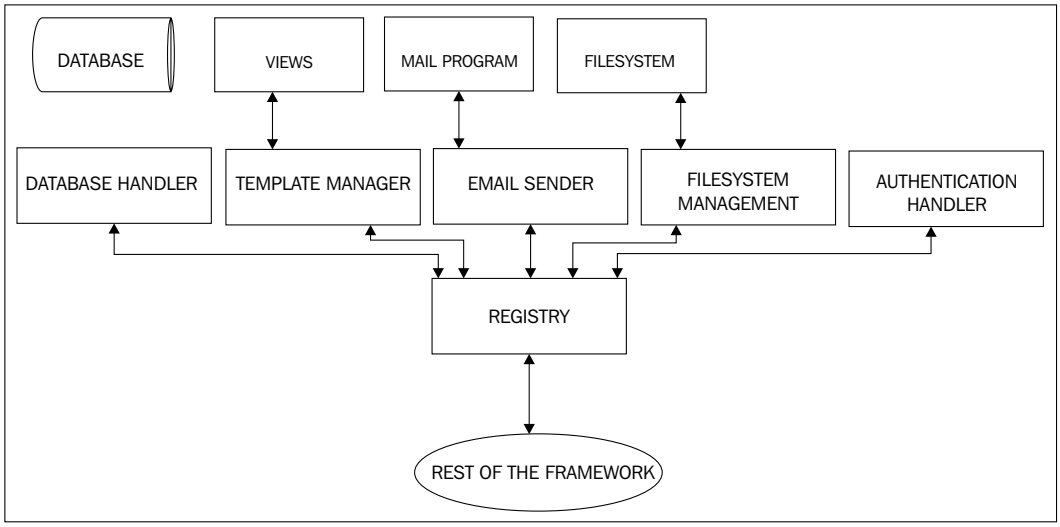


Registry

The registry pattern provides a means to store a collection of objects within our framework. The need for a registry arises from the abstraction provided with the MVC pattern. Each set of controllers and models we create (for example products, shopping basket, page viewing, or content management) need to perform some shared tasks, including:

- Querying the database
- Checking if the user is logged in, and if so, getting certain user data
- Sending data to the views to be generated (template management)
- Sending e-mails, for instance to confirm a purchase with the customer
- Interacting with the server's filesystem, for instance to upload photographs of products

Most systems and frameworks abstract these functions into objects of their own, and ours will be no exception. A registry allows us to keep these objects together. The registry can then be passed around the framework, providing a single point of contact to access these core functions. Let's have an overview of the registry pattern:



The framework interacts directly with the registry, which provides access to the other relevant objects. These objects can interact with one another using the registry itself, and have functionality to interact with aspects of the system they require; that is, the database handler can access the database, the template manager can access the templates stored on the filesystem, the e-mail sender can access the e-mail templates and also the systems mail program, the filesystem manager can access the filesystem, and the authentication handler reads and writes to session variables and cookies to maintain an authenticated user's session throughout their visit to the site.

Singleton

There are certain situations where we may require an object to only ever have one instance of it available. For instance, we will make use of a database handler and multiple instances of this could lead to results from different queries being supplied, depending on how it is used. The singleton pattern is designed to prevent this from occurring, by restricting an object to one instance only.

However, we won't use the singleton pattern in this way. We will instead use it to ensure we have only one instance of our registry available in the framework at any point of time.

Structure

The next important stage in the design of our framework is its structure. We need to design a suitable file structure for our framework. We need the structure to provide for:

- Models
- Views (We may wish to integrate the ability to switch styles for the websites we power with our framework, so one folder for each unique set of templates, styles, and views would be a good idea.)
- Controllers (We may wish to have each controller within its own folder, as we may have accompanying functions in additional files, so we can keep them organized.)
- Administration controllers (If we are to add administration tools to our framework, we should provide some administration controllers. These would be controllers for the various models we have; however, they would be designed for administrative tasks, and would be accessible only to administrators.)
- Registry
- Registry objects
- User/administrator uploaded files
- Third-party libraries
- Any other code

Taking this framework structure into account, a suitable directory structure would be as follows:

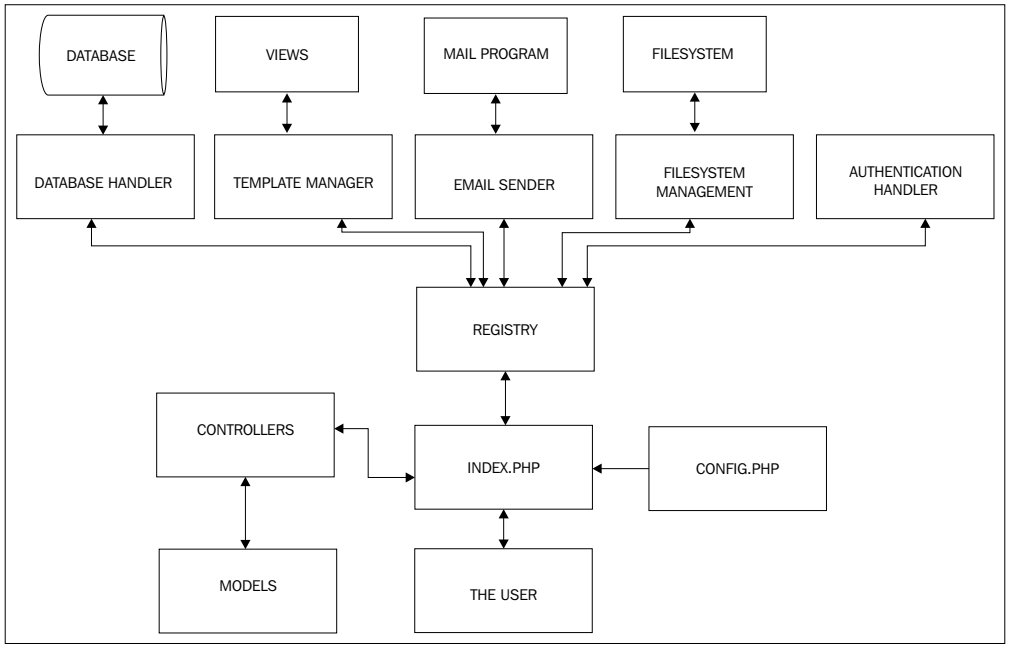
- Models
- Views
    - View A (that is, a folder per set of views)
    - Templates
    - Images
    - JavaScript
- Controllers
    - Controller A (that is, a folder per controller)
        - ControllerA
        - ControllerAAdmin

- Registry
    - Objects
    - Database objects
- Assets
- Uploads
    - To be expanded when we add products and images to our framework!
- Libraries
- Miscellaneous

Building a killer framework

Now that we have designed our framework, it is time to start building it! Let's start by implementing the patterns we discussed earlier in the chapter.

If we now look at an overview of our framework, the user visits the site through the `index.php` file, which in turn instantiates the registry, creates or instantiates the relevant controllers, and passes the registry to those. The controllers in turn create models where appropriate, and both the models and controllers can interact with the registry (as it was passed to the objects), generating and manipulating views as appropriate. The following diagram illustrates this:



Pattern implementation

There are a number of different ways to implement the patterns we discussed. Many different frameworks utilize these patterns, and implement them in a wide range of different ways. The way we are going to implement them is just one of these methods. You may find it best to alter this implementation to better suit your needs, or perhaps this will provide you with a great structure to build your own framework from. Let us begin.

MVC

The actual implementation of the MVC pattern at this stage is quite difficult, as it essentially requires us to be at a stage where we wish to implement the main features that the sites powered by our framework will use. We are, of course, not yet at this stage. We only have our folder structure in place; however, we can create some example or basic models, views, and controllers to illustrate its workings, or alternatively we can create some interfaces for the models and views to ensure they all have a certain, common structure for all of our models and controllers.

Registry

The registry implementation is quite straightforward; the main difficulty is with all of the objects it holds! The registry on its own is very simple. It needs to have a method to create certain objects and store them with a key; it needs another method, which when passed with a key as a parameter, returns the object in question.

We can also store some useful, central functions within the registry object itself; although if we wished, we could abstract them into objects of their own. (If you feel your framework would be better suited with these functions being contained within a separate object(s), please do so!) Such functions include:

- Processing the incoming URL, so our `index.php` file can route the request correctly
- Building URLs based on a series of parameters, a query string, and the URL display/generation method we use (that is whether we have `mod_rewrite` enabled on our server, or not)
- Pagination

Settings

The registry is our primary store for both settings and commonly used objects, so we need to make provision for settings management.

Data

Database stored settings in web applications often range from large text areas to Boolean tick boxes. This sort of flexibility isn't something we can easily add into the database easily. The simplest method is to store settings in a single table, consisting of a key and setting value pair, with the setting value stored as longtext. Some settings could be stored within the code too. The registry needs a simple method to store a copy of these settings for the framework to use.

Code

The code that follows makes up the basics of our registry, with two arrays: one for objects, one for settings, and a store and get method for each of them. The `storeObject` method also has provisions to detect if the object is a database object, and if so, it opens the object from a different folder.

```
/**
  * The array of objects being stored within the registry
  * @access private
  */
  private static $objects = array();
/**
  * The array of settings being stored within the registry
  * @access private
  */
  private static $settings = array();
/**
  * Stores an object in the registry
  * @param String $object the name of the object
  * @param String $key the key for the array
  * @return void
  */
  public function storeObject( $object, $key )
  {
    if( strpos( $object, 'database' ) !== false )
    {
      $object = str_replace( '.database', 'database', $object);
      require_once('databaseobjects/' . $object
          . '.database.class.php');
    }
    else
    {
      require_once('objects/' . $object . '.class.php');
    }
    self::$objects[ $key ] = new $object( self::$instance );
```

```
    }
  /**
    * Gets an object from within the registry
    * @param String $key the array key used to store the object
    * @return object - the object
    */
    public function getObject( $key )
    {
      if( is_object ( self::$objects[ $key ] ) )
      {
        return self::$objects[ $key ];
      }
    }
  /**
    * Stores a setting in the registry
    * @param String $data the setting we wish to store
    * @param String $key the key for the array to access the setting
    * @return void
    */
    public function storeSetting( $data, $key )
    {
      self::$settings[ $key ] = $data;
    }
  /**
    * Gets a setting from the registry
    * @param String $key the key used to store the setting
    * @return String the setting
    */
    public function getSetting( $key )
    {
      return self::$settings[ $key ];
    }
```

Singleton

The singleton pattern is very easy to implement, as it requires only a few minor changes to a standard PHP class, to ensure that it is only ever instantiated once.

```
  /**
    * The instance of the registry
    * @access private
    */
    private static $instance;
```

```
    /**
      * Private constructor to prevent it being created directly
      * @access private
      */
      private function __construct()
      {
      }
    /**
      * singleton method used to access the object
      * @access public
      * @return
      */
      public static function singleton()
      {
        if( !isset( self::$instance ) )
        {
          $obj = __CLASS__;
          self::$instance = new $obj;
        }

        return self::$instance;
      }
    /**
      * prevent cloning of the object:
      * issues an E_USER_ERROR if this is attempted
      */
      public function __clone()
      {
        trigger_error( 'Cloning the registry is not permitted',
            E_USER_ERROR );
      }
```

The constructor method (which is used to create an instance of this class as an object) is set to `private`, which means we cannot call it from outside of the class. This is important as it restricts how the object is instantiated, and allows us to control how many copies of it are created.

We next have a singleton method; this is used to access the registry. If there is no instance of the object already, then it creates a new instance and stores it within the `$instance` variable. If the instance variable is already set, then the object (itself) is returned.

Finally, to prevent the object being cloned, we have a `trigger_error` call, so should we ever add a functionality that clones our registry, we will find that an `E_USER_ERROR` error is produced.

Registry objects

As discussed earlier, the registry on its own is very straightforward, the more complicated aspects are the objects that the registry will actually store and manage access to. These objects include:

- Database handler
- User authentication
- Template management
- E-mail sending

Let's look at creating those objects now.

Database

Our database object needs to have functionality for:

- Connecting to the database
- Managing multiple database connections
- Performing queries
- Returning common query information such as number of rows affected, the last insert ID, and so on
- Caching queries, particularly so we can integrate a result set with the views (through the template handler), and pass it a cache reference, so it can generate the results into the view
- Making common queries easier (for example, inserts, updates, and deletes) by having the queries pre-formatted within certain methods of the object

There is obviously a lot more a database object could do; we will discuss that in a moment.

Our database object

This database class abstracts the MySQL functions from the rest of the framework into a single file, which manages the database connection:

```
<?php
/**
  * Database management / access class: basic abstraction
```

```
     *
     * @author Michael Peacock
     * @version 1.0
     */
    class mysqldatabase {
    /**
     * Allows multiple database connections
     * each connection is stored as an element in the array,
     * and the active connection is maintained in a variable (see below)
     */
     private $connections = array();
     /**
     * Tells the DB object which connection to use
     * setActiveConnection($id) allows us to change this
     */
     private $activeConnection = 0;
     /**
     * Queries which have been executed and the results cached for
     *  later, primarily for use within the template engine
     */
     private $queryCache = array();
     /**
     * Data which has been prepared and then cached for later usage,
     *  primarily within the template engine
     */
     private $dataCache = array();
     /**
     * Number of queries made during execution process
     */
     private $queryCounter = 0;
     /**
     * Record of the last query
     */
     private $last;

     /**
     * Construct our database object
     */
     public function __construct() { }
```

Let's look through what it does, method by method:

- `newConnection`: This method creates a new connection to a database. This is separate from the constructor for two reasons: firstly, if it were in the constructor, we would need to pass the connection details to it, which would mean it would need to be created separately from the other objects within the registry. Secondly, this method allows us to maintain several connections to databases within the same object.

```
/**
  * Create a new database connection
  * @param String database hostname
  * @param String database username
  * @param String database password
  * @param String database we are using
  * @return int the id of the new connection
*/
public function newConnection( $host, $user, $password,
    $database )
{
    $this->connections[] = new mysqli( $host, $user,
        $password, $database );
    $connection_id = count( $this->connections )-1;
    if( mysqli_connect_errno() )
    {
      trigger_error('Error connecting to host. '
          .$this->connections[$connection_id]->error,
          E_USER_ERROR);
    }
    return $connection_id;
}
```

- `closeConnection`: This method closes the active connection (while there are multiple connections within the object, the current object being used is stored as a variable to keep track of it).

```
/**
  * Close the active connection
  * @return void
*/
  public function closeConnection()
  {
    $this->connections[$this->activeConnection]->close();
  }
```

- `setActiveConnection`: This method allows us to toggle which database connection is the active one to be used.

```
/**
  * Change which database connection is actively used for the
  * next operation
  * @param int the new connection id
  * @return void
*/
public function setActiveConnection( int $new )
{
  $this->activeConnection = $new;
}
```

- `cacheQuery`: This method allows us to store a query to be processed later, primarily to be replaced in a loop into the view.

```
/**
  * Store a query in the query cache for processing later
  * @param String the query string
  * @return the pointed to the query in the cache
  */
  public function cacheQuery( $queryStr )
  {
    if( !$result = $this->connections
        [$this->activeConnection]->query( $queryStr ) )
    {
      trigger_error('Error executing and caching query: '
        .$this->connections[$this->activeConnection]
        ->error, E_USER_ERROR);
      return -1;
    }
    else
    {
      $this->queryCache[] = $result;
      return count($this->queryCache)-1;
    }
  }
```

- `numRowsFromCache`: This method tells us the number of rows a cached query contains.

```
/**
  * Get the number of rows from the cache
  * @param int the query cache pointer
```

```
  * @return int the number of rows
  */
  public function numRowsFromCache( $cache_id )
  {
    return $this->queryCache[$cache_id]->num_rows;
  }
```

- `resultsFromCache`: This method gets the results from a cached query.

```
  /**
  * Get the rows from a cached query
  * @param int the query cache pointer
  * @return array the row
  */
  public function resultsFromCache( $cache_id )
  {
    return $this->queryCache[$cache_id]->
                                        fetch_array(MYSQLI_ASSOC);
  }
```

- `cacheData`: This method caches an array of data, as if it were results in a query, stored within this object, as it is a `database` class, so we could use it to store data too.

```
/**
  * Store some data in a cache for later
  * @param array the data
  * @return int the pointed to the array in the data cache
  */
  public function cacheData( $data )
  {
    $this->dataCache[] = $data;
    return count( $this->dataCache )-1;
}
```

- `dataFromCache`: This method returns data from the cache of stored data.

```
/**
  * Get data from the data cache
  * @param int data cache pointed
  * @return array the data
  */
public function dataFromCache( $cache_id )
  {
    return $this->dataCache[$cache_id];
```

```
  }

/**
  * Delete records from the database
  * @param String the table to remove rows from
  * @param String the condition for which rows are to be removed
  * @param int the number of rows to be removed
  * @return void
  */
```

- `deleteRecords`: This method takes the table, conditions, and a limit as parameters and builds them into a delete statement, which is then executed.

```
public function deleteRecords( $table, $condition, $limit )
{
  $limit = ( $limit == '' ) ? '' : ' LIMIT ' . $limit;
  $delete = "DELETE FROM {$table} WHERE {$condition} {$limit}";
  $this->executeQuery( $delete );
}
```

- `updateRecords`: This method takes the table, an array of changes (fields as keys, values as values), and a condition and builds them into an update statement, which is then executed.

```
/**
  * Update records in the database
  * @param String the table
  * @param array of changes field => value
  * @param String the condition
  * @return bool
  */
  public function updateRecords( $table, $changes, $condition )
  {
    $update = "UPDATE " . $table . " SET ";
    foreach( $changes as $field => $value )
    {
      $update .= "`" . $field . "`='{$value}',";
    }
    // remove our trailing ,
    $update = substr($update, 0, -1);
    if( $condition != '' )
    {
      $update .= "WHERE " . $condition;
```

```
    }
    $this->executeQuery( $update );
    return true;
  }
```

- `insertRecords`: This method takes the table and an array of data (fields as keys, value as value) and builds them into an `insert` statement, which is then executed.

```
/**
  * Insert records into the database
  * @param String the database table
  * @param array data to insert field => value
  * @return bool
  */
  public function insertRecords( $table, $data )
  {
    // setup some variables for fields and values
    $fields  = "";
    $values = "";
    // populate them
    foreach ($data as $f => $v)
    {
      $fields  .= "`$f`,";
      $values .= ( is_numeric( $v ) && ( intval( $v ) == $v ) ) ?
           $v."," : "'$v',";
    }
    // remove our trailing ,
    $fields = substr($fields, 0, -1);
    // remove our trailing ,
    $values = substr($values, 0, -1);
    $insert = "INSERT INTO $table ({$fields}) VALUES({$values})";
    $this->executeQuery( $insert );
    return true;
  }
```

- `executeQuery`: This method runs a query.

```
/**
  * Execute a query string
  * @param String the query
  * @return void
  */
```

```
public function executeQuery( $queryStr )
{
  if( !$result = $this->connections[$this->activeConnection]->
                                                query( $queryStr ) )
  {
    trigger_error('Error executing query: '.$this->
        connections[$this->activeConnection]->error,
        E_USER_ERROR);
  }
  else
  {
    $this->last = $result;
  }
}
```

- `getRows`: This method returns the number of rows from a query.

```
/**
  * Get the rows from the most recently executed query,
  * excluding cached queries
  * @return array
  */
  public function getRows()
  {
    return $this->last->fetch_array(MYSQLI_ASSOC);
  }
```

- `affectedRows`: This method returns the number of rows affected by the last executed query.

```
    /**
     * Gets the number of affected rows from the previous query
     * @return int the number of affected rows
     */
    public function affectedRows()
    {
       return $this->$this->connections[$this->activeConnection]-
>affected_rows;
    }
```

- `sanitizeData`: This method cleans data to ensure it is safe to be put into an SQL statement.

```
/**
  * Sanitize data
```

```
  * @param String the data to be sanitized
  * @return String the sanitized data
  */
  public function sanitizeData( $data )
  {
    return $this->connections[$this->activeConnection]->
                                    real_escape_string( $data );
  }
```

- `__deconstruct`: The deconstructor, closes all the connections that were opened with the database.

```
/**
  * Deconstruct the object
  * close all of the database connections
  */
  public function __deconstruct()
  {
    foreach( $this->connections as $connection )
    {
      $connection->close();
    }
   }
}
?>
```

Extending the database object

So, how could we extend the database object, and the way we have designed it?

- **Inheritance**: We could have a database interface, which defines some base methods for any database object we create, making it easier to swap the type of database we use (for example, from MySQL to pgSQL, or MSSQL).
- **Abstracting the logic to the queries**: To further simplify the use of various database engines, we could abstract the logic from our queries into the database object itself. This way, instead of passing queries to the object, we pass the make up of the queries—for example, table, fields, fields to order by (to define how the results should be ordered), types of joins, and so on—and the object inserts every bit of SQL that is required. This is the only true way to have complete database abstraction within a framework.
- **Debug information**: We could add provisions for logging the performance of our queries, recording slow queries to allow us to debug, and optimize the queries we use.

User authentication

Our user authentication class needs to:

- Process login requests
- Check to see if the user is logged in
- Log out the user
- Maintain information about the currently logged-in user (we could extend this to use a User object if we wish)

Firstly, we need our class and some methods:

```
<?php
/**
 * Authentication manager
 *
 *
 * @version 1.0
 * @author Michael Peacock
 */
class authentication {
  private $userID;
  private $loggedIn = false;
  private $admin = false;
  private $groups = array();
  private $banned = false;
  private $username;
  private $justProcessed = false;
  public function __construct() {}
```

These are just the core properties we need to maintain, and will need to access. The next stage is to check for any authentication requests or current login—this will be called by our framework once the database has been connected to. This should first check to see if a user may be logged in; if this is the case, it should verify this. If not, it should then check to see if a user is trying to log in. The following function does this, and passes control to an appropriate method depending on the situation.

```
public function checkForAuthentication()
{
  if( isset( $_SESSION['phpecomf_auth_session_uid'] ) &&
      intval( $_SESSION['phpecomf_auth_session_uid'] ) > 0 )
  {
    $this->sessionAuthenticate( intval(
        $_SESSION['phpecomf_auth_session_uid'] ) );
```

```
    }
    elseif( isset(  $_POST['ecomf_auth_user'] ) &&
        $_POST['ecomf_auth_user'] != '' &&
        isset( $_POST['ecomf_auth_pass'] ) &&
        $_POST['ecomf_auth_pass'] != '')
    {
      $this->postAuthenticate(
         PeacockCarterFrameworkRegistry::getObject('db')->
         sanitizeData( $_POST['ecomf_auth_user'] ),
         md5( $_POST['ecomf_auth_pass'] ) );
    }
    //echo $this->userID;
  }
```

We can authenticate a user who is logged in from session data: if we store the user's ID in a session, we can check this is valid and the user is active.

```
  private function sessionAuthenticate( $uid )
  {
    $sql = "SELECT u.ID, u.username, u.active, u.email, u.admin,
      u.banned, u.name, (SELECT GROUP_CONCAT( g.name SEPARATOR
      '-groupsep-' ) FROM groups g, group_memberships gm
      WHERE g.ID = gm.group AND gm.user = u.ID ) AS groupmemberships
      FROM users u WHERE u.ID={$uid}";
    PeacockCarterFrameworkRegistry::getObject('db')->
                                                executeQuery( $sql );
    if( PeacockCarterFrameworkRegistry::getObject('db')->
                                                     numRows() == 1 )
      {
```

Even if the user exists, we can't just log them in. But, what if their user account is not active, or has been marked as "banned"?

```
        $userData = PeacockCarterFrameworkRegistry::getObject('db')->
                                                           getRows();
        if( $userData['active'] == 0 )
        {
          $this->loggedIn = false;
          $this->loginFailureReason = 'inactive';
          $this->active = false;
        }
        elseif( $userData['banned'] != 0)
        {
          $this->loggedIn = false;
          $this->loginFailureReason = 'banned';
          $this->banned = false;
        }
```

```
    else
    {
      $this->loggedIn = true;
      $this->userID = $uid;
      $this->admin = ( $userData['admin'] == 1 ) ? true : false;
      $this->username = $userData['username'];
      $this->name = $userData['name'];
```

All of a user's group memberships are returned as a single field from the user lookup query. We can then split this into the individual groups and store them in the object.

```
      $groups = explode( '-groupsep-',
          $userData['groupmemberships'] );
      $this->groups = $groups;
    }
  }
  else
  {
    $this->loggedIn = false;
    $this->loginFailureReason = 'nouser';
    if( $this->loggedIn == false )
    {
      $this->logout();
    }
  }
```

If the user is trying to log in, we must look up his or her username and password to verify them. This is very similar to the above function, except it uses the username and password provided by the user, rather than a session-stored user ID.

```
  private function postAuthenticate( $u, $p )
  {
    $this->justProcessed = true;
    $sql = "SELECT u.ID, u.username, u.email, u.admin, u.banned,
        u.active, u.name, (SELECT GROUP_CONCAT( g.name SEPARATOR
        '-groupsep-' ) FROM groups g, group_memberships gm WHERE
        g.ID = gm.group AND gm.user = u.ID ) AS groupmemberships
        FROM users u WHERE u.username='{$u}'
        AND u.password_hash='{$p}'";
    //echo $sql;
    PeacockCarterFrameworkRegistry::getObject('db')->
                                              executeQuery( $sql );
    if( PeacockCarterFrameworkRegistry::getObject('db')->
                                                  numRows() == 1 )
    {
      $userData = PeacockCarterFrameworkRegistry::getObject('db')->
                                                      getRows();
```

As with before, once we find a user, we must check to see that they are active, and not banned from the site.

```
    if( $userData['active'] == 0 )
    {
      $this->loggedIn = false;
      $this->loginFailureReason = 'inactive';
      $this->active = false;
    }
    elseif( $userData['banned'] != 0)
    {
      $this->loggedIn = false;
      $this->loginFailureReason = 'banned';
      $this->banned = false;
    }
    else
    {
      $this->loggedIn = true;
      $this->userID = $userData['ID'];
      $this->admin = ( $userData['admin'] == 1 ) ? true : false;
      $_SESSION['phpecomf_auth_session_uid'] = $userData['ID'];
      $groups = explode( '-groupsep-',
          $userData['groupmemberships'] );
      $this->groups = $groups;
    }
  }
  else
  {
    $this->loggedIn = false;
    $this->loginFailureReason = 'invalidcredentials';
  }
}
```

Logging out can be done simply by cleaning the session data for the user.

```
function logout()
{
  $_SESSION['phpecomf_auth_session_uid'] = '';
}
```

Finally, we need some getter methods to return various properties of the current user.

```
public function getUserID()
{
  return $this->userID;
```

```
    }
    public function isLoggedIn()
    {
      return $this->loggedIn;
    }
    public function isAdmin()
    {
      return $this->admin;
    }
    public function getUsername()
    {
      return $this->username;
    }
    public  function isMemberOfGroup( $group )
    {
      if( in_array( $group, $this->groups )
      {
        return true;
      }
      else
      {
        return false;
      }
    }
}
?>
```

Template management

The template management functionality is easily broken down into two aspects: an object to manage the actual content (a `page` object), and a template object to manage the interaction with the content along with the parsing of the content within it.

Let's take a look at the code for `template.class.php`:

```
<?php
/**
 * Views: Template manager
 * Page content and structure is managed with a seperate page object.
 *
 * @version 1.0
 * @author Michael Peacock
 */
class template {
```


This material is copyright and is licensed for the sole use by jackie tracey on 23rd February 2010
953 Quincy Drive, , Brick, , 08724

Our constructor includes the `page` class, and creates a new `page` object that is stored within the `page` variable within the `template` object. This allows the `template` object to manipulate the page as required by the framework.

```
private $page;

/**
 * Include our page class, and build a page object to manage
 *  the content and structure of the page
 */
 public function __construct()
 {
   include( FRAMEWORK_PATH . '/registry/objects/page.class.php');
   $this->page = new Page();
 }
```

In some cases, we may wish to insert one template into another. For instance, we may wish to display a shopping basket summary on every page. However, if there is nothing in the basket, we may wish to insert a different template, or just some text. This method takes a template and places it into another template when it finds the appropriate template tag.

```
/**
 * Add a template bit from a view to our page
 * @param String $tag the tag where we insert the template
 * e.g. {hello}
 * @param String $bit the template bit (path to file,
 * or just the filename)
 * @return void
 */
public function addTemplateBit( $tag, $bit )
{
  if( strpos( $bit, 'views/' ) === false )
  {
    $bit = 'views/'
         . PHPEcommerceFrameworkRegistry::getSetting('view')
         . '/templates/' . $bit;
  }
  $this->page->addTemplateBit( $tag, $bit );
}
```

To make it easier to add multiple templates into another template, we can add the templates and the template tags to an array, which is then iterated through when the templates are parsed.

```
/**
 * Take the template bits from the view and insert them into
 * our page content
 * Updates the pages content
 * @return void
 */
private function replaceBits()
{
  $bits = $this->page->getBits();
  // loop through template bits e.g.
  foreach( $bits as $tag => $template )
  {
    $templateContent = file_get_contents( $template );
    $newContent = str_replace( '{' . $tag . '}',
        $templateContent, $this->page->getContent() );
    $this->page->setContent( $newContent );
  }
}
```

As we will also want to insert dynamically-generated data into our templates, we need a function to do that—this replaces all of the template tags with the values we wish to be associated with them. It also checks to see if a template variable is data, or if it is a cached query (or cached data) and if so, replaces the tag with results from a query.

```
/**
 * Replace tags in our page with content
 * @return void
 */
private function replaceTags()
{
  // get the tags in the page
  $tags = $this->page->getTags();
  // go through them all
  foreach( $tags as $tag => $data )
  {
  // if the tag is an array, then we need to do more than a
  // simple find and replace!
  if( is_array( $data ) )
  {
    if( $data[0] == 'SQL' )
```

```
            {
             // it is a cached query...replace tags from the database
             $this->replaceDBTags( $tag, $data[1] );
            }
            elseif( $data[0] == 'DATA' )
            {
             // it is some cached data...replace tags from cached data
             $this->replaceDataTags( $tag, $data[1] );
            }
          }
          else
          {
          // replace the content
          $newContent = str_replace( '{' . $tag . '}', $data,
              $this->page->getContent() );
          // update the pages content
          $this->page->setContent( $newContent );
        }
      }
    }
```

If we have cached the results of a database query and assigned them to a template variable, we need to replace the relevant template variables with the results of the query. The following function does this, repeating the contents of `<!--START template_tag -->` and `<!--END template_tag -->` for however many records there are for that query, and then replacing tags within that block of code, with the results of the query.

```
  /**
    * Replace content on the page with data from the database
    * @param String $tag the tag defining the area of content
    * @param int $cacheId the queries ID in the query cache
    * @return void
    */
  private function replaceDBTags( $tag, $cacheId )
  {
    $block = '';
    $blockOld = $this->page->getBlock( $tag );
    // foreach record relating to the query...
    while ($tags = PHPEcommerceFrameworkRegistry::getObject('db')->
        resultsFromCache( $cacheId ) )
    {
      $blockNew = $blockOld;
      // create a new block of content with the results replaced
      //  into it
```

```
      foreach ($tags as $ntag => $data)
        {
          $blockNew = str_replace("{" . $ntag . "}", $data, $blockNew);
        }
        $block .= $blockNew;
      }
      $pageContent = $this->page->getContent();
      // remove the seperator in the template, cleaner HTML
      $newContent = str_replace( '<!-- START ' . $tag . ' -->'
          . $blockOld . '<!-- END ' . $tag . ' -->', $block,
          $pageContent );
      // update the page content
      $this->page->setContent( $newContent );
    }
```

Here we do the same as `replaceDBTags`, only with data we have cached (as opposed to a query).

```
    /**
      * Replace content on the page with data from the cache
      * @param String $tag the tag defining the area of content
      * @param int $cacheId the data's ID in the data cache
      * @return void
      */
    private function replaceDataTags( $tag, $cacheId )
    {
      $block = $this->page->getBlock( $tag );
      $blockOld = $block;
      while ($tags = PHPEcommerceFrameworkRegistry::getObject('db')-
>dataFromCache( $cacheId ) )
      {
        foreach ($tags as $tag => $data)
        {
          $blockNew = $blockOld;
          $blockNew = str_replace("{" . $tag . "}", $data, $blockNew);
        }
        $block .= $blockNew;
      }
      $pageContent = $this->page->getContent();
      $newContent = str_replace( $blockOld, $block, $pageContent );
      $this->page->setContent( $newContent );
    }
```

This returns the `page` object, so we can directly call public methods from within it, if we should need to from outside the scope of the template object.

```
/**
 * Get the page object
 * @return Object
 */
public function getPage()
{
  return $this->page;
}
```

Our page is comprised of a number of templates. So generally, this is the first thing we need to do for a new view: we build the view from a number of templates. The templates must be passed, in the correct order, to the method (that is, header, main content, and footer) for them to be appropriately displayed.

```
/**
 * Set the content of the page based on a number of templates
 * pass template file locations as individual arguments
 * @return void
 */
public function buildFromTemplates()
{
  $bits = func_get_args();
  $content = "";
  foreach( $bits as $bit )
  {
    if( strpos( $bit, 'views/' ) === false )
    {
      $bit = 'views/'
        . PHPEcommerceFrameworkRegistry::getSetting('view')
        . '/templates/' . $bit;
    }
    if( file_exists( $bit ) == true )
    {
      $content .= file_get_contents( $bit );
    }

  }
  $this->page->setContent( $content );
}
```

This is particularly useful when working in the model or controller with a single row of results from a query. We may wish to convert them to template tags; this function facilitates that, and allows us to prefix the tags, which helps eliminate naming conflicts.

```
/**
 * Convert an array of data into some tags
 * @param array the data
 * @param string a prefix which is added to field name to create
 * the tag name
 * @return void
 */
public function dataToTags( $data, $prefix )
{
  foreach( $data as $key => $content )
  {
    $this->page->addTag( $key.$prefix, $content);
  }
}
```

We set the title of a page directly to the `page` object. However, this needs to be inserted directly into the template, as this function takes the title value and inserts it between the title tags within the template itself.

```
/**
 * Take the title we set in the page object, and insert them
 * into the view
 */
public function parseTitle()
{
  $newContent = str_replace('<title>', '<title>'
      . $this->page->getTitle(), $this->page->getContent() );
  $this->page->setContent( $newContent );
}
```

Finally, when we have finished assigning template variables, new templates need to be inserted into a template and so on. We need to call the various replace and parse functions. This method consolidates these calls, so we simply call it from our framework. Our completed view is now ready to be sent to the user's browser.

```
/**
 * Parse the page object into some output
 * @return void
 */
public function parseOutput()
{
```

```
      $this->replaceBits();
      $this->replaceTags();
      $this->parseTitle();
    }
  }
  ?>
```

Let's take a look at the code for `page.class.php`.

Our `page` object makes it easier to encapsulate the data and templates that we have compiled during the framework's execution to create the appropriate finalized view for the customer. It stores information such as the title of the page, the template variables and their corresponding data values, the contents of various templates, and any template bits we wish to insert into the others.

```
<?php
/**
 * Page object for our template manager
 *
 * @author Michael Peacock
 * @version 1.0
 */
class page {
  // page elements

  // page title
  private $title = '';
  // template tags
  private $tags = array();
  // tags which should be processed after the page has been parsed
  // reason: what if there are template tags within the database
  //   content, we must parse the page, then parse it again for post
  //   parse tags
  private $postParseTags = array();
  // template bits
  private $bits = array();
  // the page content
  private $content = "";

  /**
   * Create our page object
  */
    function __construct() { }

    /**
```

```
     * Get the page title from the page
     * @return String
     */
    public function getTitle()
    {
          return $this->title;
    }

    /**
     * Set the page title
     * @param String $title the page title
     * @return void
     */
    public function setTitle( $title )
    {
      $this->title = $title;
    }

    /**
     * Set the page content
     * @param String $content the page content
     * @return void
     */
    public function setContent( $content )
    {
      $this->content = $content;
    }

    /**
     * Add a template tag, and its replacement value/data to the page
     * @param String $key the key to store within the tags array
     * @param String $data the replacement data (may also be an
     *  array)
     * @return void
     */
    public function addTag( $key, $data )
    {
      $this->tags[$key] = $data;
    }

    /**
     * Get tags associated with the page
     * @return void
     */
```

```
public function getTags()
{
  return $this->tags;
}

/**
 * Add post parse tags: as per adding tags
 * @param String $key the key to store within the array
 * @param String $data the replacement data
 * @return void
 */
public function addPPTag( $key, $data )
{
  $this->postParseTags[$key] = $data;
}

/**
 * Get tags to be parsed after the first batch have been parsed
 * @return array
 */
public function getPPTags()
{
  return $this->postParseTags;
}

/**
 * Add a template bit to the page, doesnt actually add
 * the content just yet
 * @param String the tag where the template is added
 * @param String the template file name
 * @return void
 */
public function addTemplateBit( $tag, $bit )
{
  $this->bits[ $tag ] = $bit;
}

/**
 * Get the template bits to be entered into the page
 * @return array the array of template tags and template
 * file names
 */
public function getBits()
{
```

```
        return $this->bits;
      }

      /**
       * Gets a chunk of page content
       * @param String the tag wrapping the block
       * ( <!-- START tag --> block <!-- END tag --> )
       * @return String the block of content
       */
      public function getBlock( $tag )
      {
        preg_match ('#<!-- START '. $tag . ' -->(.+?)
            <!-- END '. $tag . ' -->#si', $this->content, $tor);
        $tor = str_replace ('<!-- START '. $tag . ' -->', "", $tor[0]);
        $tor = str_replace ('<!-- END '  . $tag . ' -->', "", $tor);
        return $tor;
      }

      public function getContent()
      {
        return $this->content;
      }

}
?>
```

Extending the template management object

We could extend our template management object and `page` object to make things easier or more powerful should we wish. Some examples include:

- Making it possible to password protect pages
- Restricting access to pages based on permissions
- Making it easy to add CSS and JavaScript files into the page
- Easily adding some `onLoad` JavaScript to the page's `<body>` tag

E-mail sending

Most sites require the need to send e-mails. This is even more so the case with e-commerce sites, as they will need to send e-mails to confirm purchases, confirm dispatches, and to inform customers when products they were interested in are back in stock.

Other potential modules

We could of course have many more objects managed with our registry; if we wished to further extend our framework in the future, this could include:

- E-mail parsing
- Security management
- Filesystem management

Let's have a brief look at what would be involved in such objects, and potential uses for them.

Email parsing

With an e-mail parsing object, we could easily process incoming e-mails. This could be useful for:

- Running an online help desk: We could process incoming e-mails, and log them as support tickets, even assign them to different categories or departments depending on the e-mail address the e-mails were sent to.
- Running a basic online e-mail service for our customers.

Security management

By providing a security management object within our registry, we could make it easier to:

- Manage a ban list of:
    - E-mail addresses
    - Usernames
    - IP addresses (to prevent troublesome users registering, and make it easy to block automated spam bots)
- Validate certain forms of data, including:
    - E-mail addresses
    - Phone numbers
    - Prices
    - Postal codes and zip codes

Filesystem management

Filesystem access could also be abstracted, making it easier for us to:

- Create files and folders on the server
- Manage permissions of files
- Process uploaded files
- Delete files
- Display file and directory listings

This would be useful if we wish to add functions to make it easier to manage file uploads; for instance when we integrate the provisions for allowing customers to upload files with certain products (for example a photograph to be printed onto a t-shirt they order), we could use this to easily manage the uploads, place the uploads into monthly or weekly folders, and keep things easily organized.

Routing requests

We now have our core tasks managed within a single instance of a registry, which is available to the rest of the framework. These core tasks are generally required for each area of the site. We also have a structure in place allowing different code to be run, such as for a product to be displayed, an order to be placed, or a page to be viewed. While this additional code is easy to slot in, we still require a way for the framework to know when that code should be run, based on the URL our users are visiting.

The simplest way to route requests is to first search for any pages in the database that have a search engine-friendly reference that matches the incoming URL. If no match is found, we call the controller, which matches the first part of the URL, and have it determine the correct action to take based on the rest of the URL. For example, `products/view/some-product` would tell the products controller to "view" the product with the reference "some product".

This isn't the most flexible method, as we may want requests of certain formats to be sent to different models, or be handled in a different way; however, it is the quickest and easiest method. One way to extend this is with a router.

An alternative: With a router

An alternative approach is to use a router. The router matches the pattern of the URL of the visitor to the website and routes the request depending on the format, or against a series of rules which are defined as routes.



Try it yourself
Why not try and integrate a router into the framework yourself?

Processing the incoming URL within our registry object

A single function allows us to easily process the current URL and breaks it down to make it easy for us to route the request to the correct part of the framework.

This function works by taking the URL path from the `page` GET variable (for example, `?page=products/view/some-product`). The framework is set to process everything between forward slashes as a different part of the URL. The first section, for example `products`, indicates which controller to use. Generally the controller will take the second part, say `view`, to determine what it needs to do, and how it should use data from the model, and often the third part as an indication as to the data the model should represent.

```
/**
  * Gets data from the current URL
  * @return void
  */
  public function getURLData()
  {
  $urldata = $_GET['page'];
  self::$urlPath = $urldata;
  if( $urldata == '' )
  {
    $this->urlBits[] = 'home';
  }
  else
  {
```

We split the URL data at each instance of a forward slash with the explode function.

```
    $data = explode( '/', $urldata );
      while ( !empty( $data ) && strlen( reset( $data ) ) === 0 )
      {
        array_shift( $data );
      }
      while ( !empty( $data ) && strlen( end( $data ) ) === 0)
      {
        array_pop($data);
```

```
        }
        self::$urlBits = $this->array_trim( $data );
      }
    }
```

Two variables are required to store the URL and the bits of the URL as an array. Two additional functions are also required, to return the URL and the bits of the URL.

index.php

Our `index.php` file needs to load up our registry, connect to the database, call the authentication checks, process the URL, and pass control to the appropriate controller—which is quite a lot of work. Most of the actual work is done by the registry and the registry objects we have already created.

The first stage is to load the registry and its objects. Once this is done, we can connect to the database, and then we can perform our authentication checks. Before any code is written, we should start our sessions; this can't be done after anything has been outputted to the user, so it's best to do it before any other code to make sure this is the case.

```
// first and foremost, start our sessions
session_start();
// setup some definitions
// The applications root path, so we can easily get this path from
files located in other folders
define( "FRAMEWORK_PATH", dirname( __FILE__ ) ."/" );
// require our registry
require_once('registry/registry.class.php');
$registry = PHPEcommerceFrameworkRegistry::singleton();
$registry->getURLData();
// store core objects in the registry.
$registry->storeObject('mysql.database', 'db');
$registry->storeObject('template', 'template');
$registry->storeObject('authentication', 'authenticate');
// database settings
include(FRAMEWORK_PATH . 'config.php');
// create a database connection
$registry->getObject('db')->newConnection($configs['db_host_
ecomframe'], $configs['db_user_ecomframe'], $configs['db_pass_
ecomframe'], $configs['db_name_ecomframe']);
// process any authentication
$registry->getObject('authenticate')->checkForAuthentication();
```

Next we need to load a list of controllers our framework has (we should maintain a database table of these). Once the registry object has parsed the URL the user is accessing the site with, we can check the first part of this against our controller list, and pass control to the relevant one.

```
// populate our page object from a template file
$registry->getObject('template')->
    buildFromTemplates('header.tpl.php', 'main.tpl.php',
    'footer.tpl.php');

$activeControllers = array();
$registry->getObject('db')->executeQuery('SELECT controller FROM
    controllers WHERE active=1');
while( $activeController = $registry->getObject('db')->getRows() )
{
  $activeControllers[] = $activeController['controller'];
}
$currentController = $registry->getURLBit( 0 );
if( in_array( $currentController, $activeControllers ) )
{
  require_once( FRAMEWORK_PATH . 'controllers/'
      . $currentController . '/controller.php');
  $controllerInc = $currentController.'controller';
  $controller = new $controllerInc( $registry, true );
}
else
{
  require_once( FRAMEWORK_PATH . 'controllers/page/controller.php');
  $controller = new Pagecontroller( $registry, true );
}
// parse it all, and spit it out
$registry->getObject('template')->parseOutput();
print $registry->getObject('template')->getPage()->getContent();

exit();
```

.htaccess file

We have our `index.php` file set up to process the incoming request and send it to the relevant controller. However, URLs which have the format of `index.php?page=some/page/on/our/site` or `index.php?page=products/view/some-product` are not as attractive or memorable as those with just `some/page/on/our/site` or `products/view/some-product`. With the Apache module `mod_rewrite` we can get our site to rewrite the more friendly URLs into the less friendly ones for our framework to understand.

```
ErrorDocument 404 /index.php
DirectoryIndex index.php
<IfModule mod_rewrite.c>
  RewriteEngine on
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule ^(.*)$ index.php?page=$1 [L,QSA]
</IfModule>
```

This `.htaccess` file instructs the web server (Apache) to use `index.php` as the index file within a directory. It also instructs Apache that if the `mod_rewrite` module is enabled, the requests that are not for valid files or directories should be rewritten to the main `index.php` file. However, anything that occurs after the directory containing the `.htaccess` file, should be appended to the page `$_GET` variable.

For example, `oursite.com/pagea` would be rewritten to `oursite.com/index.php?page=pagea`.

Configuration file

We also need a configuration file, to store our database connection settings. Most other settings will be stored in the database. However, the actual connection details cannot be stored within it, as we need to know the details *before* we connect to the database; otherwise, we cannot connect.

```
<?php
$configs = array();
$configs['db_host_ecomframe'] = 'localhost';
$configs['db_user_ecomframe'] = 'root';
$configs['db_pass_ecomframe'] = '';
$configs['db_name_ecomframe'] = 'phpecommerce';
?>
```

The simplest way to store the settings is within an array in a configuration file, with the keys of the array relating to what the value is used for. The suffix of `ecomframe` is used to allow us to store multiple database connection details within the same array.

What about e-commerce?

Looking at frameworks is important; however, we haven't really covered anything specific to e-commerce yet. So where does e-commerce fit into this? Most of our e-commerce functionality fits into this by adding models and controllers that perform the relevant e-commerce tasks we require, such as managing products, handling the order processing and checkout process, and so on.

An e-commerce registry?

One thing that we could consider is an e-commerce registry. We could perhaps have our shopping basket as a registry, containing a collection of products. This would make it simple for each page to access the shopping basket and return the number of items contained within. For the framework we are going to create in this book, we are not going to use this method. However, it may be something you wish to think about with your own framework. After saying that, you may be wondering how we are going to provide access to the basket to all areas of the framework. The answer is with some call backs. At certain key points of execution within the framework, certain functions will be called. Inside this, we provide code to interact with the shopping basket. Exactly where these callback functions are, and when they are called, is a discussion for another chapter.

Summary

In this chapter, we created a sound start to our framework which will be extended with e-commerce functionality during the rest of this book. We specifically covered:

- Various software architectural and design patterns, looking into best practices for implementing certain aspects of our framework
- Developing a directory structure for our framework
- Creating a registry to store core objects
- How we will use the MVC pattern to structure and operate our framework
- Routing page requests around our framework and an overview of routers
- Creating our single point of contact for accessing the framework, as well as an `htaccess` and configuration file

Next we move onto storing, displaying, and managing products and their categories, some true e-commerce functionality!

3
Products and Categories

With a basic structure to our framework in place, we can now start to think about the e-commerce aspects to it. In this chapter you will learn:

- How to structure content within the framework, including:
    - Page structure
    - Product structure
    - Categories structure
- How to access and display products and categories with models and controllers
- How to design views to interact with our framework

As we discussed in Chapter 1, *PHP e-commerce*, Juniper Theatricals requires a framework to power their website as well as the e-commerce features, so page management is a must.

What we need

Before we start building products and categories into our framework, let's think about what information we need, both to display to our customers and for the use of the store administrator.

To provide our customers with sufficient product information, we need to inform them of the name of the product, a detailed description of the product, and the price of the product. We may also wish to show them a photograph of the product and a number of additional images related to the product. Additionally, we may wish to make them aware of the weight and the cost of shipping, number of items we have in stock, as well as any categories the product is contained within. From an administration perspective, we need a reference number, or ID number, and we may need a Stock Keeping Unit reference. We may also require a search engine friendly name, which is used within the URL to view the product.

Product information

Taking into account what we have just discussed, at a minimum we need to store the following information:

Data	Description
ID	A reference number for the framework to reference the product
Name	The name of the product
Search Engine Friendly Name	A search engine friendly name for the product to be displayed in URLs.
Description	A detailed description of the product
SKU	A stock keeping reference (usually supplier's reference, or for integration with stock keeping systems)
Price	The cost of the product
Stock	The number of these products which are currently in stock
Primary image	An image of the product
Additional images	A number of additional images which are displayed as thumbnails and then toggled into the place of the main page

Shipping costs and information will be discussed in Chapter 8, *Shipping and Taxes*, so we don't need to take those aspects into consideration just yet.

Category information

We need to be able to contain our products within categories, so what information would we need to collect for our categories?

Data	Description
ID	A reference number for the framework to reference the category
Name	The name of the category
Description	A detailed description of the category
Search Engine Friendly Name	A search engine friendly name for the category, to be used for display within URLs.

Structuring content within our framework

We could go ahead now and implement a data structure and functionality to display products and categories within our framework; however, if we did so at this stage, we would lose out on a lot of potential flexibility. Most content displayed on any website or contained within any web application has some common data. If we find this common data, and create an abstract content type, then we will have a more flexible framework. This is because we could easily integrate additional functionality to each of these content types without the need for duplicating the functionality or the code. Such additional functionality could include:

- Various versions of content
- Access permissions
- Commenting on content, pages, or products
- Rating pages or products, or other content

Pages

Pages are an essential type of content. Even if we were creating a website, which was just to be an online store, we would still need some standard pages, for contact details, delivery information, terms and conditions, and privacy policies among other things. So, what data might we wish to store for pages?

Data	Description
ID	A reference for the framework to refer to the pages
Name	The name of the page
S.E.F. Name	A search engine friendly name for the page to be used in URLs
Heading	A page heading, generally something we would store within an `<h1>` tag
Title	A title of the page (displayed within the `<title>` tags of the page)
Content	The content for the page
Keywords	Metadata for the page—keywords
Description	Metadata for the page—description

Content

Pages are the most fundamental content type we would need, and most of the fields required are shared throughout most of the data we would store. Product categories could operate using the same data as a page; however, products and other advanced content types would need more data, and we would extend the data stored for these content types.

Data	Description
ID	A reference for the framework to refer to the content
Name	The name of the content entity
S.E.F Name	A search engine friendly name for the content, for use within URLs
Content	The content itself, for example page, product details, and category description
Type	The type of content this content entry is (for example page, product, category, and so on)
Order	The order of the content within a group, for example pages in a menu
Parent	The parent element for this entity, useful to indicate subproducts and subpages
Meta keywords	Metadata (keywords) for this content entity
Meta description	Metadata (description) for this content entity
Date created	The date the content entity was created
Creator	The user who created the content
Active	If the content is active (publicly visible) or not
Secure	If the content requires the user to log in to see it (doesn't take into account fine-grained permissions)

Versioning

To effectively manage versions of content, we need to keep the ID and some other aspects consistent, while still maintaining different versions of our content. We can do this by keeping a reference to content and the version of the content separate, and maintaining a record of previous versions, to allow us to roll back to a previous version should we need to.

So, the content we discussed earlier would actually be a version of content, to reference the active version of content we would need to also store:

Data	Description
ID	A reference number for the framework to refer to active content
Current revision	The active version of the content

Building products, categories, and content functionality into our framework

Now that we know the data we need to store, we now need to think about exactly how we will store this data, and how we will manage and access it from within our framework.

Database

The first stage is to design the relevant database tables. Then we can develop our framework to query the database and render the relevant data. We know roughly the data we need to store for each of our content types which we have discussed; let us now translate that into a database structure. First, let's review what tables we will need in our database:

Table	Description
Content	To store references to the active version of content as well as some information on content that doesn't change with each version (for example, initial author) or things that can change without affecting the version (active/secure toggles)
Versions	To store the actual content data, one record for each version of content
Content types	Record the types of content in the framework, relating to content entities to refer to the type of content they are (for example, page, product, category)
Products	An extension of the versions table for product-specific data; when combined with the appropriate version's record, this makes up the product data
Revision history	Maintains a history of revisions and their content entities, allowing us to roll back to a previous version, should we need to

Content

All of the content within our framework will stem from a standard format; however, as we may also wish to take advantage of versions of content, this is where we must start: a table where the records relate to the active version of content for particular content elements.

So, if for instance we have three pages in our database, and two products, these would have a record in the content table. There would also be a record in our revisions table for each revision of that content. The ID of the content entry in the database will never change, and this entry maintains a relation to the active record in the revisions table.

This table requires the following fields:

Field	Type	Description
ID	Integer (Primary Key, Auto Increment)	A reference for the framework to refer to the content entity by
Current_revision	Integer	A reference to the current version of this content entity
Active	Boolean	Indicates if the content is active or inactive, and thus if it should be shown to users
Secure	Boolean	Indicates if the content is secure or insecure, and thus if the user must be logged in before they can see the content
Parent	Int	A reference to the parent content item, if appropriate
Order	Int	A reference to the order of the content (where appropriate, primarily used for pages, to automatically build menus and site maps based on structure)
Author	Int	A reference to the user who created the first version of this content entity
Type	Int	A reference to the type of content this entity is, for example page, product, and so on
Path	Varchar	Search engine friendly reference for the page, product, or other type of content, for example pages may be accessed by `ourwebsite.com/the/content/path`, whereas products may be accessed by `ourwebsite.com/products/view/the/product/path`
Created	Timestamp	When the content entity was created

The SQL for this table is as follows:

```
CREATE TABLE `content` (
    `ID` int(11) NOT NULL auto_increment,
    `current_revision` int(11) NOT NULL,
    `active` tinyint(1) NOT NULL,
    `secure` tinyint(1) NOT NULL,
    `parent` int(11) NOT NULL,
    `order` int(11) NOT NULL,
    `author` int(11) NOT NULL,
    `type` int(11) NOT NULL,
    `path` varchar(255) NOT NULL,
    PRIMARY KEY  (`ID`),
```

```
    KEY `current_revision` (`current_revision`,`active`,`type`),
    KEY `type` (`type`),
    KEY `author` (`author`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1
  COMMENT='Content Elements Table' AUTO_INCREMENT=1 ;
```

The table should have the following references to foreign keys:

```
ALTER TABLE `content`
  ADD CONSTRAINT `content_ibfk_10` FOREIGN KEY (`type`)
    REFERENCES `content_types` (`ID`) ON UPDATE CASCADE,
  ADD CONSTRAINT `content_ibfk_8` FOREIGN KEY (`current_revision`)
    REFERENCES `content_versions` (`ID`) ON UPDATE CASCADE,
  ADD CONSTRAINT `content_ibfk_9` FOREIGN KEY (`author`)
    REFERENCES `users` (`ID`) ON UPDATE CASCADE;
```

Content types

Because we are going to store all content centrally within a few set tables, we also need a way to reference the types of the content, for example if the content is a page, a product, a category, or something else.

Field	Type	Description
ID	Integer (Primary Key, Auto Increment)	An ID to be referred to by other tables
Reference	Varchar	A machine-friendly reference string for the type, since we can't assume all products have a type of ID X, but we can assume they all have a type with a reference which is "product"
Name	Varchar	A user-friendly string to represent the name of the content type

The SQL for this table is as follows:

```
CREATE TABLE `content_types` (
  `ID` int(11) NOT NULL auto_increment,
  `reference` varchar(15) NOT NULL,
  `name` varchar(25) NOT NULL,
  PRIMARY KEY  (`ID`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

Content versions

The actual content of pages and other content types is stored in the `content_versions` table, which stores versions of the content, and the active version for each content element is referenced directly by the content table. This table requires the following structure:

Field	Type	Description
ID	Integer (Auto increment, Primary Key)	A reference for the framework to refer to different versions
Name	Varchar	The name of the content
Title	Varchar	The page title for the content (shown in the `<title>` tags of the page)
Heading	Varchar	The heading for the page, commonly displayed within `<h1>` tags.
Content	Longtext	The actual HTML content for the content (the page, product, or other suitable content type)
Metakeywords	Varchar	The keywords metadata
Metadescription	Varchar	The description metadata field
Metarobots	Varchar	The robots metadata field; this could be from a predefined list, so we may wish to have a metarobots table, or make this an ENUM field
Author	Integer	A reference to the user who created this version of the content
Timestamp	Timestamp	The time and date that the version was created

The SQL for this table is as follows:

```
CREATE TABLE `content_versions` (
    `ID` int(11) NOT NULL auto_increment,
    `name` varchar(255) NOT NULL,
    `heading` varchar(255) NOT NULL,
    `content` longtext NOT NULL,
    `metakeywords` varchar(255) NOT NULL,
    `metadescription` varchar(255) NOT NULL,
    `metarobots` varchar(255) NOT NULL,
    `author` int(11) NOT NULL,
    `created` timestamp NOT NULL default CURRENT_TIMESTAMP,
    PRIMARY KEY  (`ID`),
    KEY `author` (`author`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

This table should have the following references to foreign keys:

```
ALTER TABLE `content_versions`
  ADD CONSTRAINT `content_versions_ibfk_1` FOREIGN KEY (`author`)
    REFERENCES `users` (`ID`) ON UPDATE CASCADE;
```

Products

As discussed earlier, products will be represented by extending the revisions table and combining it with a products table.

Field	Type	Description
ID	Integer (Primary Key, Auto increment)	A reference to the entry in the table.
Content_version	Integer	Ties the product version data to a content version, associating relevant fields from that and from its associated record in the content table too.
SKU	Varchar	The stock keeping unit reference for the product.
Image	Varchar	A reference to an image path where the primary image for this product is stored.
Weight	Int	The weight of the product.
Featured	Boolean	If the product is featured. We may wish to use this to display products on the home page, or perhaps in a "featured products" box.

The following SQL represents this table:

```
CREATE TABLE `content_types_products` (
    `ID` int(11) NOT NULL auto_increment,
    `content_version` int(11) NOT NULL,
    `price` float NOT NULL,
    `weight` int(11) NOT NULL,
    `SKU` varchar(255) NOT NULL,
    `stock` int(11) NOT NULL,
    `image` varchar(255) NOT NULL,
    `featured` tinyint(1) NOT NULL,
    PRIMARY KEY  (`ID`),
    KEY `content_version` (`content_version`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

The table needs the following references to foreign keys:

```
ALTER TABLE `content_types_products`
  ADD CONSTRAINT `content_types_products_ibfk_1`
    FOREIGN KEY (`content_version`)
    REFERENCES `content_versions` (`ID`)
    ON DELETE CASCADE ON UPDATE CASCADE;
```

Categories

Categories themselves don't need to extend the data held within the content versions table, so we can use that table as is for categories.

Pages within our framework

To enable pages within our framework, we need to add a model, controller, and some templates (a view) to it. The model will connect to the database and represent the data contained within a page. The controller will work with the model to lookup the current page, and display it within the view. There is a lot of additional functionality, which we could look at implementing here, such as menu and submenu generation, breadcrumb generation, and so on. However, that is more of a content management system focus, and beyond the scope of this book.

Model

Our model requires the ability to connect to the database, search for a page given the path of a page, and represent the data of that page. It also needs to inform the framework if the path provided does not represent a page, so that we can generate a "Page not found" error.

The constructor

The simplest way for us to do this, is to have the constructor perform the page lookup, by passing a page path to it as a parameter. The constructor should then set the values of various properties within the model object to the values from within the database. If a page was not found, it needs to set a particular property to reflect this.

Other methods

Other methods we need within our model are:

- `isValid()`: This method is used by the controller to lookup if the page is valid or not.
- `isActive()`: This method is used by the controller to lookup if the page is active or not.
- `isSecure()`: This method is used by the controller to determine if the page is secure, and if the visitor is permitted to see the page.
- `getProperties()`: This method returns the properties of the page, so they can be integrated with the view.

This gives us the following code for our model:

```
<?php
// models/pages/model.php
class Pagemodel {

  private $registry;
  private $valid = false;
  private $active = true;
  private $secure = false;
  private $pageheading;
  private $title;
  private $pagecontent;
  private $metakeywords;
  private $metadescription;
  private $metarobots;

  public function __construct( PHPEcommerceFrameworkRegistry
      $registry, $urlPath )
  {
    $this->registry = $registry;
    $urlPath = $this->registry->getObject('db')->sanitizeData
                                                    ( $urlPath );

    $sql = "SELECT c.ID, c.active, c.secure, v.title, v.name,
              v.heading, v.content, v.metakeywords,
              v.metadescription, v.metarobots FROM content c,
              content_types t, content_versions v
            WHERE c.type=t.ID AND t.reference='page'
              AND c.path='{$urlPath}'
              AND v.ID=c.current_revision LIMIT 1";
    $this->registry->getObject('db')->executeQuery( $sql );
    if( $this->registry->getObject('db')->numRows() != 0 )
    {
```

```
        $this->valid = true;
        $pageData = $this->registry->getObject('db')->getRows();
        $this->active = $pageData['active'];
        $this->secure = $pageData['secure'];
        $this->pageheading = $pageData['heading'];
        $this->title = $pageData['title'];
        $this->pagecontent = $pageData['content'];
        $this->metakeywords = $pageData['metakeywords'];
        $this->metadescription = $pageData['metadescription'];
        $this->metarobots = $pageData['metarobots'];
      }
    }
    public function isValid()
    {
      return $this->valid;
    }

    public function isActive()
    {
      return $this->active;
    }
    public function isSecure()
    {
      return $this->secure;
    }
    public function getProperties()
    {
      $tor = array();
      foreach( $this as $field => $value )
      {
        if( !is_object( $value ) )
        {
          $tor[ $field ] = $value;
        }
      }
      return $tor;
    }
  }
  ?>
```

View

The view for a page should consist of the following template files:

- **Header**: To have a common header used on most pages and areas of the website.
- **Footer**: To have a common footer used on most pages and areas of the website.
- **Page**: To display the page to the user.

We will also need some additional templates, depending on the page itself, including:

- **404 error template**: To indicate that the page was not found.
- **Login template**: For secure pages where the current user is not logged in.
- **"Page disabled" page**: For inactive pages, although we may of course prefer to generate a 404 error page, so that the visitor does not realize that the page they tried to view exists, but is just disabled.

Header template

Our header template will contain the first bit of HTML for a particular page, so obviously this needs to contain our doctype, page title (empty as this is populated later), style references, and so on.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>{title}</title>
  <meta http-equiv="content-type"
        content="text/html;
        charset=iso-8859-1" />
  <meta name="description" content="{metadescription}" />
  <meta name="keywords" content="{metakeywords}" />
</head>
<body>
```

Footer template

The footer template is for content and HTML, which is displayed after a page, product, or other primary area of the site. For now, this just needs to close our `<body>` and `<html>` tags.

```
</body>
</html>
```

Page template

The page template itself needs very little, just a heading and some content. The content is stored in the database as full HTML, so content can be marked up and then displayed in the page.

```
<h1>{pageheading}</h1>
{pagecontent}
```

404 error template

Content for our error template would just be the following:

```
<h1>Page not found</h1>
<p>Sorry, we could not find the page or file you were looking for,
   please return to the home page and try again.</p>
<!-- 404 error -->
```

Other templates

We may also wish to have a login template and a "Page disabled" template. These are the aspects we will focus on a little more in our framework's development.

Controller

The controller needs to be able to:

- Clean the path requested (to prevent any security issues, for example MySQL injection)
- Pass the path to the page model
- If the page is valid, convert the page properties into tags for use in the template system, and output the page view
- If the page is not valid, display the 404 error view
- If the page is valid and secure, display the page if the visitor is a logged-in user, or else display a login page.

This gives us the following controller:

```
<?php
// controllers/pages/controller.php
class Pagecontroller
{
  private $registry;
```

Our constructor receives the registry as a parameter so that the object can use the registry when it needs to. It also receives a Boolean value for `$directCall` to indicate if the controller should assume the user is accessing the controller, or another controller is piggy-backing on this one, to call some of its functions.

```
  public function __construct( PHPEcommerceFrameworkRegistry
$registry, $directCall )
  {
    $this->registry = $registry;
    if( $directCall == true )
    {
      $this->viewPage();
    }
  }
```

Assuming the user is trying to view a page (at present, that is all this controller supports!), we must lookup the page using the page model, generate the view, and insert the relevant data.

```
  private function viewPage()
  {
    // We require the page model, so we create a new page model
    // passing the URL path as a reference to allow it to look up the
    // page.
    require_once( FRAMEWORK_PATH . 'models/page/model.php');
    // Page model needs different class name, as page is used for
    // the template handler!
    $this->model = new Pagemodel( $this->registry,
        $this->registry->getURLPath() );
    // If the page is valid, or not valid, the relevant templates are
    // displayed, and if appropriate, the pages data fields are
    // assigned to template variables, to display in the view.
    if( $this->model->isValid() )
    {
      $pageData = $this->model->getProperties();
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php', 'main.tpl.php',
          'footer.tpl.php');
      $this->registry->getObject('template')->
          dataToTags( $pageData, '' );
      $this->registry->getObject('template')->getPage()->
          setTitle( $pageData['title'] );
    }
    else
    {
      $this->pageNotFound();
    }
  }
```

If the page isn't found, we need to display a suitable error message.

```
    private function pageNotFound()
    {
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php', '404.tpl.php',
          'footer.tpl.php');
    }
```

Our framework should also have scope to expand for pages that are disabled, or require the user to be logged in (and the user isn't logged in), thus displaying a login page.

```
    private function pageRequiresLogin()
    {
      // TODO
    }
    private function pageDisabled()
    {
      // TODO
    }
}
?>
```

Products

Building functionality for our products can be broken down into three main areas. Firstly we need to create a model, which represents a product in our code. In time we will extend this model to be able to easily update and save individual products as well as clone them, but for now it will just represent a product's data. Secondly we need a controller, to interpret the user's page request, interface with the model, and work with the view to display the data to the end user. Finally we need a view or a series of templates, to display the information to the end user.

Model

At this stage in our framework's development the model has a very simple function: to query the database for a specified product, and represent the data for the product in object form. It also needs to determine if a product is valid or not, so that our constructor can either display the product or an error message (for example invalid product) to the end user.

To fully understand the main lookup query this model must perform, let us reflect on our database structure, which has been developed previously within the chapter. First we have our `content` table; from here we lookup the product's unique search engine friendly name, as the path to the content element within this table. We also need to ensure that the content element is set to be active. The next stage is to check that the type of content is that of a product, by referring to the `content_types` table. Then we need to look up the `content_versions` table, to get the main content such as the name of the product, and a description of the product. Finally, we need to link to the products table itself, to get the product-specific data such as price, weight, and so on. The following query nicely takes care of that for us:

```
SELECT v.name AS product_name, c.ID AS product_id,
  p.image AS product_image, p.weight AS product_weight,
  p.price AS product_price, p.SKU AS product_sku,
  p.featured AS product_featured, v.heading AS product_heading,
  v.content AS product_description, v.metakeywords AS metakeywords,
  v.metarobots AS metarobots, v.metadescription AS metadescription
FROM content_versions v, content c, content_types t,
  content_types_products p
WHERE c.active=1 AND c.secure=0 AND c.type=t.ID
  AND t.reference='product' AND p.content_version=v.ID
  AND v.ID=c.current_revision AND c.path='{$productPath}'
```

At this stage, the model needs the following distinct functionality:

- Lookup a product based on a product path
- Determine if a product exists or not, based on said path
- Store product data
- Return product data as an array
- Return a Boolean value indicating if a product is valid or not

Further into our framework development, we will want and need to extend this quite a lot, but for simply representing and displaying products, this will suffice for now.

```
<?php
class Product{
  private $registry;
  private $ID;
  private $name;
  private $SKU;
  private $description;
  private $price;
  private $weight;
  private $image;
  private $stock;
  private $heading;
```

```
private $metakeywords;
private $metadescription;
private $metarobots;
private $active;
private $secure;
private $activeProduct = false;
// Our constructor takes two arguments: the registry, so it can use
// it, and the product path, so it can try and lookup the product
// in question. The first thing the constructor does, is check if
// the path is set to something, and try and perform a lookup.
public function __construct( PHPEcommerceFrameworkRegistry
    $registry, $productPath )
{
  $this->registry = $registry;
  if( $productPath != '' )
  {
    $productPath = $this->registry->getObject('db')->
        sanitizeData( $productPath );
    // The lookup query is quite long because there are quite a few
    // tables involved relating the content, versions, products and
    // content type tables.
    $productQuery = "SELECT v.name AS product_name,
                      c.ID AS product_id,
                      p.image AS product_image,
                      p.stock AS product_stock,
                      p.weight AS product_weight,
                      p.price AS product_price,
                      p.SKU AS product_sku,
                      p.featured AS product_featured,
                      v.heading AS product_heading,
                      v.content AS product_description,
                      v.metakeywords AS metakeywords,
                      v.metarobots AS metarobots,
                      v.metadescription AS metadescription
                    FROM content_versions v, content c,
                      content_types t, content_types_products p
                    WHERE c.active=1 AND c.secure=0 c.type=t.ID
                      AND t.reference='product'
                      AND p.content_version=v.ID
                      AND v.ID=c.current_revision
                      AND c.path='{$productPath}'";
    //echo $productQuery;
    $this->registry->getObject('db')->
        executeQuery( $productQuery );
    // The query is then executed, and if there is a record,
    // appropriate object variables are set.
    if( $this->registry->getObject('db')->numRows() == 1 )
    {
      // tells the controller we have a product!
      $this->activeProduct = true;
```

```
        // grab the product data, and associate it with the relevant
        // fields for this object
        $data = $this->registry->getObject('db')->getRows();
        $this->ID = $data['product_id'];
        $this->name = $data['product_name'];
        $this->price = $data['product_price'];
        $this->weight = $data['product_weight'];
        $this->image = $data['product_image'];
        $this->heading = $data['product_heading'];
        $this->description = $data['product_description'];
        $this->SKU = $data['product_sku'];
        $this->stock = $data['product_stock'];
        // secure and active were set in the query, we will probably
        // want to change this later
        $this->secure = 0;
        $this->active = 1;
        $this->metakeywords = $data['metakeywords'];
        $this->metadescription = $data['metadescription'];
        $this->metarobots = $data['metarobots'];

      }
    }
    else
    {
      // here we may want to do something else...
    }

  }
```

So far, we only need two simple methods: one for the controller to check the product is valid, and one to return the data from the model. This works by returning any object variable that isn't an object itself, in a single array.

```
  public function isValid()
  {
    return $this->activeProduct;
  }
  public function getData()
  {
    $data = array();
    foreach( $this as $field => $fdata )
    {
      if( ! is_object( $fdata ) )
      {
        $data[ $field ] = $fdata;
      }
    }
    return $data;
  }
```

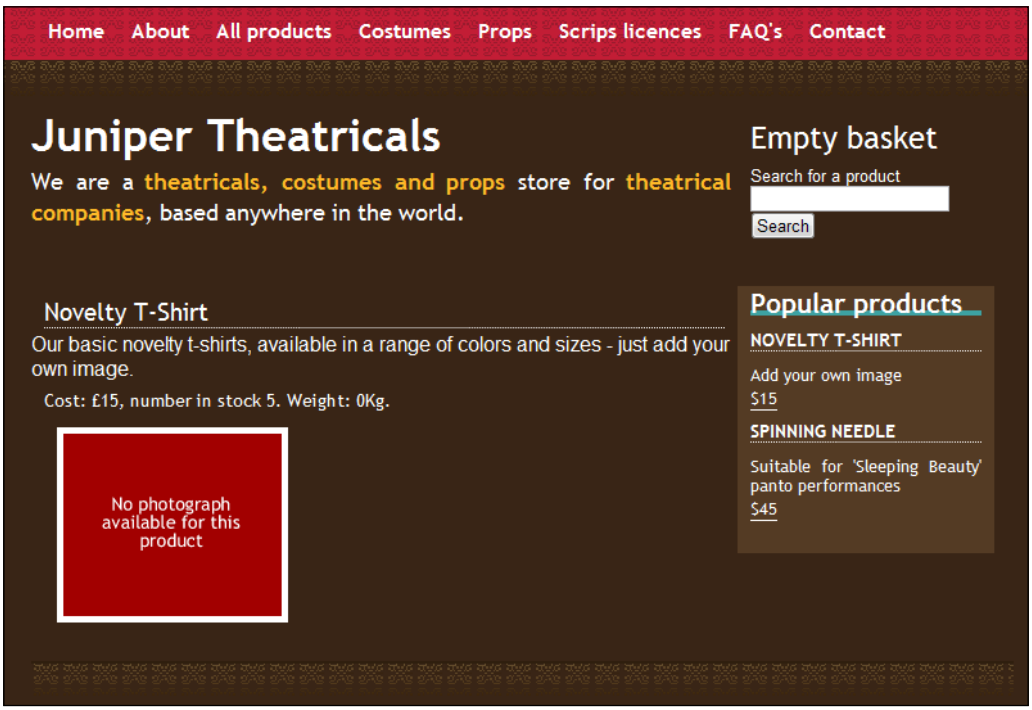
```
    /*
      Also useful: getters and setters for various fields, as well as
      a save method, to update a database entry
    */
  }
  ?>
```

View

We can build the view using the following snippet:

```
<h2>{product_name}</h2>
{product_description}
<p>Cost: &pound;{product_price}, number in stock {product_stock}
    . Weight: {product_weight}Kg.</p>
<p>
<img src="product_images/{product_image}"
     alt="{product_name} image" />
</p>
```

If we were to create records in the database for a product, to view the product we would need to visit `http://localhost/path/to/our/framework/products/view/product-path`.



Controller

The controller needs to include our product model, pass the URL path (with the exception of `products/view/`, which is used by the controller to determine what operation we are performing) to it, and if a valid product is represented by the model, display it; if not, then it needs to display a "Product not found" page.

The following code handles this nicely for us, and as with our product model, will be extended quite a lot as we progress with our framework. Both this and the model take the framework's registry as a parameter in the constructor; this ensures the objects know of the registry, and allows them to store a reference to it themselves. The controller's constructor also has a `directCall` parameter, to indicate if the controller is being accessed directly by the framework, or from another controller, to embed or extend functionality.

```
<?php
/**
 * Products Controller
 *
 * @author Michael Peacock
 * @version 1.0
 */
class Productscontroller{

  /**
  * Registry object reference
  */
  private $registry;

  /**
  * Product model object reference
  */
  private $model;

  /**
  * Controller constructor - direct call to false when being embedded
  * via another controller
  * @param PHPEcommerceFrameworkRegistry $registry our registry
  * @param bool $directCall - are we calling it directly via the
  * framework (true), or via another controller (false)
  */
  public function __construct( PHPEcommerceFrameworkRegistry
      $registry, $directCall )
  {
    $this->registry = $registry;
```

```
    if( $directCall == true )
    {
      $this->viewProduct();
    }
  }

  /**
  * View a product
  * @return void
  */
  private function viewProduct()
  {
```

When viewing a product, the URL path is taken, and the controller-specific part of that (`products/view/`, which instructs the controller that the user wishes to view a product) is removed, leaving us with the product's path.

This path is then passed in the constructor to the product model, which looks up the appropriate data.

```
    $pathToRemove = 'products/view/';
    $productPath = str_replace( $pathToRemove, '', $this->
        registry->getURLPath() );

    require_once( FRAMEWORK_PATH . 'models/products/model.php');
    $this->model = new Product( $this->registry, $productPath );
    if( $this->model->isValid() )
    {
      // Assuming the model is valid, the controller builds the view,
      // and populates the view with data.
      $productData = $this->model->getData();
      $this->registry->getObject('template')->
          dataToTags( $productData, 'product_' );
      $this->registry->getObject('template')->getPage()->
          addTag( 'metakeywords', $productData['metakeywords'] );
      $this->registry->getObject('template')->getPage()->
          addTag( 'metadescription',
          $productData['metadescription'] );
      $this->registry->getObject('template')->getPage()->
          addTag( 'metarobots', $productData['metarobots'] );
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php',
          'product.tpl.php', 'footer.tpl.php');
      $this->registry->getObject('template')->getPage()->
          setTitle('Viewing product' . $productData['name'] );
    }
```

```
    else
    {
      // If the product path wasn't valid, a product not found method
      // is called.
      $this->productNotFound();
    }
  }
  /**
  * Display invalid product page
  * @return void
  */
  private function productNotFound()
  {
    $this->registry->getObject('template')->
        buildFromTemplates('header.tpl.php',
        'invalid-product.tpl.php', 'footer.tpl.php');
  }
}
?>
```

The product data, which is represented within the model is taken and converted into tags for use within the template system. Metadata is extracted separately and converted into tags individually; as the product tags were prefixed, we don't want these tags to be prefixed, as they are common template tags throughout the site.

Categories

Our category model and controller will be similar in complexity to the products model and controller, with some minor differences:

- Not as many tables need to be referenced to get the details of a category (as the `content_versions` table fulfills the data requirements of a category)
- Categories also need to list products contained or referenced within them

One aspect that is important with categories, which we are not yet going to focus on, is a structure for categories. Categories may have parent or child categories, so we may wish to implement a hierarchical structure for our categories. However, we will put some basic provisions for this into place for now.

The controller also needs to omit or extend the view depending on if the category has any products within it, or if it has any child categories.

Model

The model for categories is a little more complicated than the products, as it needs to lookup subcategories and products associated with it. The first stage to doing this is a simple subquery, which counts the number of products and number of subcategories. If these values are greater than zero, then additional queries are performed. These queries are then cached, and a reference to the query is passed to the template engine. This allows the template engine to easily replace the results of the query within the view for the categories. The second stage is adding some additional functions to tell the controller about these cached queries, and if there are any subcategories or products.

An example model for this is as follows:

```
<?php
class Category{

  private $registry;
  private $subcatsCache = 0;
  private $productsCache = 0;
  private $numSubcats=0;
  private $isValid = false;
  private $numProducts = 0;
  private $name;
  private $title;
  private $content;
  private $metakeywords;
  private $metadescription;
  private $metarobots;
  private $active;
  private $secure;
```

The constructor simply takes the registry and the path to the category as parameters, and then calls the `getCategory` method.

```
  public function __construct( PHPEcommerceFrameworkRegistry
      $registry, $catPath )
  {
    $this->urlPath = $catPath;
    $this->getCategory();
  }
```

The "get category" method looks up the category in the database, and sets appropriate variables depending on if a category was found.

```
private function getCategory()
{
  $sql = "SELECT c.ID, c.active, c.secure, v.title, v.name,
            v.heading, v.content, v.metakeywords,
            v.metadescription, v.metarobots, ( SELECT COUNT(*)
            FROM content cn, content_types ct
            WHERE ct.ID=cn.type AND ct.reference='category'
            AND cn.parent=c.ID )
          AS num_subcats, ( SELECT COUNT(*) FROM content cn,
              content_types_products_in_categories pic
            WHERE cn.active=1 AND cn.ID=pic.product_id
              AND pic.category_id=c.ID ) AS num_products
            FROM content c, content_types t, content_versions v
            WHERE c.type=t.ID AND t.reference='category'
              AND c.path='{$this->urlPath}'
              AND v.ID=c.current_revision LIMIT 1";
  $this->registry->getObject('db')->executeQuery( $sql );
  if( $this->registry->getObject('db')->numRows() == 1 )
  {
    $this->isValid = true;
    $data = $this->registry->getObject('db')->getRows();
    $this->numSubcats = $data['num_subcats'];
    $this->numProducts = $data['num_products'];

    // If the category has subcategories, these should be looked up
    // and cached, as we may wish to generate a list or submenu
    // based off these.
    if( $this->numSubcats != 0 )
    {
      $catid = $data['ID'];
      $sql = "SELECT v.name AS category_name,
                c.path AS category_path
              FROM content c, content_versions v,
              WHERE c.parent={$catid} AND v.ID=c.current_revision
                AND c.active=1 ";
      $cache = $this->registry->getObject('db')->
          cacheQuery( $sql );
      $this->subCats = $cache;
    }

    // If the category has products within it, we should cache these
    // too, as we will want to display these products on the
    // category view.
    if( $this->numProducts != 0 )
    {
```

```
            $catid = $data['ID'];
            $sql = "SELECT p.price AS product_price,
                      v.name AS product_name, c.path AS product_path,
                    FROM content c, content_versions v,
                      content_types_products p,
                      content_types_products_in_categories pic
                    WHERE pic.product_id=c.ID
                      AND pic.category_id={$catid}
                      AND p.current_id=v.ID AND v.ID=c.current_revision
                      AND c.active=1 ";
            $cache = $this->registry->getObject('db')->
                cacheQuery( $sql );
            $this->productsCache = $cache;
          }

          $this->name = $data['name'];
          $this->title = $data['title'];
          $this->content = $data['content'];
          $this->title = $data['title'];
          $this->metakeywords = $data['metakeywords'];
          $this->metadescription = $data['metadescription'];
          $this->metarobots = $data['metarobots'];
          $this->active = $data['active'];
          $this->secure = $data['secure'];
          $this->heading = $data['heading'];
        }
      }
```

Finally, we have some getter methods, which inform the controller about the various values set.

```
      public function isValid()
      {
        return $this->isValid;
      }
      public function isEmpty()
      {
        return ($this->numProducts == 0) ? true : false;
      }
      public function numSubcats()
      {
        return $this->numSubcats;
      }
      public function getProperties()
      {
        $tor = array();
```

```
        $tor['title'] = $this->title;
        $tor['name'] = $this->name;
        $tor['content'] = $this->content;
        $tor['heading'] = $this->heading;
        $tor['metakeywords'] = $this->metakeywords;
        $tor['metadescription'] = $this->metadescription;
        $tor['metarobots'] = $this->metarobots;
        return $tor;
      }
      public function getSubCatsCache()
      {
        return $this->subcatsCache;
      }
      public function getProductsCache()
      {
        return $this->productsCache;
      }
    }
    ?>
```

View

A number of templates are needed to build the view for our product categories, these include:

- Category template
- Subcategories template
- Products template, containing products within the category (a separate template, because if there are none, the template tag is simply nulled out)

Category template

The category template can be laid out as follows:

```
<h1>{category_heading}</h1>
{category_content}
{catproducts}
{subcats}
```

Subcategories template

And here is the subcategories template:

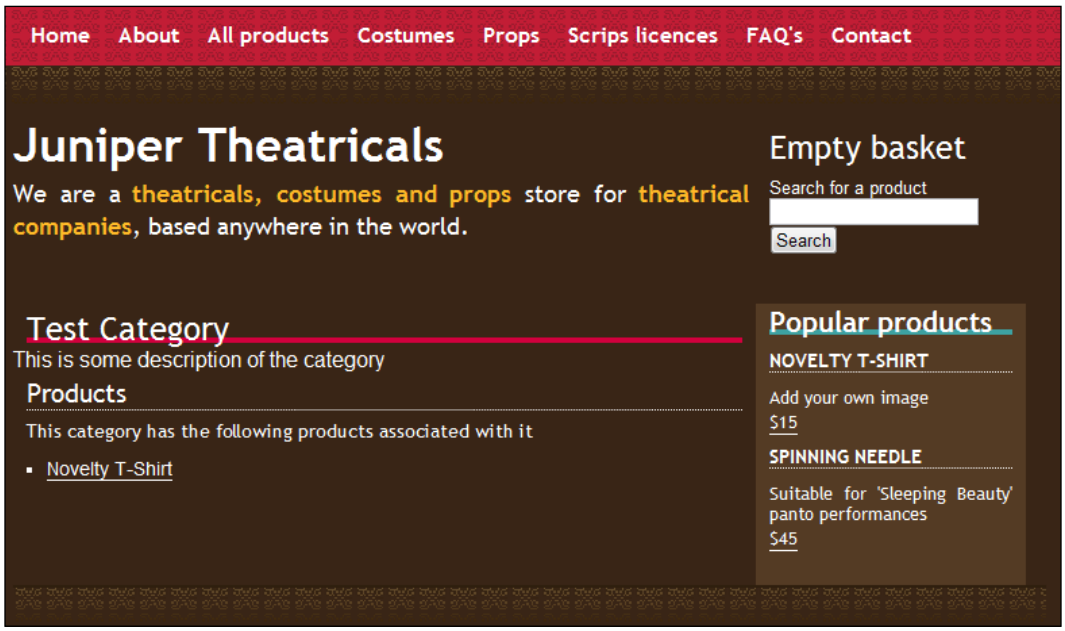
```
<h2>Subcategories</h2>
<p>This category has the following subcategories</p>
<ul>
<!-- START subcatslist -->
<li>
  <a href="categories/view/{category_path}">{category_name}</a>
</li>
<!-- END subcatslist -->
</ul>
```

Products template

The products template is as follows:

```
<h2>Products</h2>
<p>This category has the following products associated with it</p>
<ul>
<!-- START productlist -->
<li><a href="products/view/{product_path}">{product_name}</a></li>
<!-- END productlist -->
</ul>
```

The end result is a product category view.



Controller

Based off our previous controllers and the difference in the categories model, the following is our controller for viewing categories:

```
<?php
/**
 * Categories Controller
 *
 * @author Michael Peacock
 * @version 1.0
 */
class Categories{
 /**
  * Registry object reference
  */
  private $registry;
  /**
   * Categories model object reference
   */
  private $model;
  // The constructor simply checks that the object is being created
  // by a direct call to the category section by the user (and isn't
  // being used by another controller, which won't want the default
  // behavior), and calls the view category method.
  /**
   * Controller constructor - direct call to false when being
   * embedded via another controller
   * @param PHPEcommerceFrameworkRegistry $registry our registry
   * @param bool $directCall - are we calling it directly via the
   * framework (true), or via another controller (false)
   */
  public function __construct( PHPEcommerceFrameworkRegistry
      $registry, $directCall )
  {
    $this->registry = $registry;
    if( $directCall == true )
    {
        $this->viewCategory();
    }
  }
  // The viewCategory method creates a category model, passes the
  // category path, and then builds the view out of the appropriate
  // templates, depending on if the category has products within it
  // and a number of subcategories.
  /**
```

```
     * View a category
     * @return void
     */
    private function viewCategory()
    {
      $pathToRemove = 'categories/view/';
      $categoryPath = str_replace( $pathToRemove, '',
          $this->registry->getURLPath() );
      require_once( FRAMEWORK_PATH . 'models/categories/model.php');
      $this->model = new Category( $this->registry, $categoryPath );
      if( $this->model->isValid() )
      {
        if( $this->model->isEmpty() && $this->model->
            numSubcats() == 0 )
        {
          $this->emptyCategory();
        }
        else
        {
          $categoryData = $this->model->getProperties();
          $this->registry->getObject('template')->
              dataToTags( $categoryData, 'category_' );
          $this->registry->getObject('template')->getPage()->
              addTag( 'metakeywords', $categoryData['metakeywords'] );
          $this->registry->getObject('template')->getPage()->
              addTag( 'metadescription',
              $categoryData['metadescription'] );
          $this->registry->getObject('template')->getPage()->
              addTag( 'metarobots', $categoryData['metarobots'] );
          $this->registry->getObject('template')->
              buildFromTemplates('header.tpl.php', 'category.tpl.php',
              'footer.tpl.php');
          $this->registry->getObject('template')->getPage()->
              setTitle('Viewing category ' . $categoryData['name'] );

          // Here the category has no subcategories.
          if(  $this->model->numSubcats() == 0 )
          {
            $this->registry->getObject('template')->getPage()->
                addTag( 'subcats', '' );
          }

          // Here the category has sub categories.
          else
          {
            $this->registry->getObject('template')->
```

```
                addTemplateBit( 'subcats', 'subcategories.tpl.php' );
              $this->registry->getObject('template')->getPage()->
                  addTag( 'subcatslist', array('SQL',
                  $this->model->getSubCatsCache() ) );
            }

            // Similarly, if the category is empty, or has products
            // within it, different templates are used.
            if( $this->model->isEmpty() )
            {
              $this->registry->getObject('template')->getPage()->
                  addTag( 'catproducts', '' );
            }
            else
            {
              $this->registry->getObject('template')->
                  addTemplateBit( 'catproducts',
                  'categoryproducts.tpl.php' );
              $this->registry->getObject('template')->getPage()->
                  addTag( 'productslist', array('SQL', $this->model->
                  getProductsCache() ) );
            }
          }
        }
        else
        {
          $this->categoryNotFound();
        }
      }
      /**
       * Display invalid category page
       * @return void
       */
      private function categoryNotFound()
      {
        $this->registry->getObject('template')->
            buildFromTemplates('header.tpl.php',
            'invalid-category.tpl.php', 'footer.tpl.php');
      }
      private function emptyCategory()
      {
        $this->registry->getObject('template')->
            buildFromTemplates('header.tpl.php',
            'empty-category.tpl.php', 'footer.tpl.php');
      }
    }
    ?>
```

Some thoughts

At present, all of our models and controllers follow an incredibly similar format; one potential area for improvement would be for us to make use of inheritance within PHP, for our controllers and models to inherit their core methods from an interface, and, where appropriate, deviate from the standard page model and controller. I leave this improvement for you to consider and implement if you so wish.

Product and category images

One very important aspect which we have so far neglected is photographs of our products, and photographs we may wish to have to represent our categories.

Having an image associated with our products is great, but most customers want more than just one image, so let's implement additional image functionality for our products. Ideally, we will also want our customers to be able to toggle between different images from the "product view" page too.

These are aspects we will come to in the next chapter.

Routing products and categories

The first part of our URL path should indicate the area of the site the user is trying to access, for example products. We need to maintain a list of active controllers in use by the framework, and if this part of the URL matches an active controller, we should include the controller file, create a controller object for that controller, and pass control of the framework to that controller. If the first part of the URL isn't an active controller, then we should include the page controller and pass control to that, which should in turn either display a page, or detect that the page isn't valid and therefore display a 404 error ("Page not found") page.

```
$activeControllers = array();
$registry->getObject('db')->executeQuery('SELECT controller
                                          FROM controllers
                                          WHERE active=1');
while( $activeController = $registry->getObject('db')->getRows() )
{
  $activeControllers[] = $activeController['controller'];
}
$currentController = $registry->getURLBit( 0 );
if( in_array( $currentController, $activeControllers ) )
{
```

```
    require_once( FRAMEWORK_PATH . 'controllers/'
        . $currentController . '/controller.php');
    $controllerInc = $currentController.'controller';
    $controller = new $controllerInc( $registry, true );
  }
  else
  {
    require_once( FRAMEWORK_PATH . 'controllers/page/controller.php');
    $controller = new Pagecontroller( $registry, true );
  }
```

Featured products

Within our page controller, we can add some logic to detect if we are viewing the default home page, or any other page we wish to view featured products within. This would be done simply by referencing the path of the page, checking if it was blank (the home page), or matching a list of pages we have defined as pages to contain featured products.

If this is the case, we simply instantiate the products constructor, passing `false` as the `directCall` parameter to ensure it doesn't react as if this was a user directly trying to access the products controller to view a page.

This then allows us to call a featured products method within the controller. This method within the controller would lookup the featured products, cache the results, assign them to a template variable, and where appropriate, insert a featured products template into the page too.

Embedding products

As with featured products, we can make use of the `directCall` parameter in our products controller to include product details in other pages (and even in other controllers). If we have a specific area of our site where we wish to embed specific products, we can simply include the products controller, pass a `directCall` parameter of `false`, which prevents the controller from performing its default actions, and then do our relevant logic.

One example would be to include a specific product on a specific page.

1. We would need to modify our page controller to detect when we were viewing that particular page, if we were, then the products controller would be included.
2. We would need to create a new method in the products controller to lookup a specific product, take some of the data (name, description, image, price and URL path), and convert these to (post parse) template tags. This would be similar to the `viewProduct` method, however, without so much data, and without the dependency upon product templates.
3. Within the content for this page, we would insert relevant template tags, which would be populated with product data by our products controller.

Summary

In this chapter, we discussed the data that needs to be stored for us to effectively maintain an online product catalog. We took this outline of data, and used it to construct a set of database tables for our framework. From here we were then able to construct a series of models, views, and controllers for our framework to interact with the user, the products, and categories stored within the database. We also looked at how to manage multiple images associated with products, so that our customers can toggle between them, to see different views and perspectives of the products within the store, a feature that is becoming the norm with most e-commerce stores.

Now that we have products and categories working in our store, the next stage is to look at extending this to manage and display customizable products, and multiple variations of products, for example, a blue t-shirt or a red t-shirt.

4
Product Variations and User Uploads

Having a store with products and categories is great, but we need to be able to offer greater flexibility with our products. We looked at how to extend the information stored about our products, and extending products that way; but certain types of product, like apparel, need to allow the user to customize the product, often by selecting a variation of the product, or uploading images or text as part of the order. In this chapter, you will learn:

- How to create customizable products
- How to assign uploaded files to individual product orders
- How we will maintain these uploads
- How to assign custom user-submitted data with individual product orders

One important point to note is that this chapter links in greatly with Chapter 6, *The Shopping Basket*; so some aspects of this chapter may be preparation for that, and some aspects of that chapter may require some looking back at this one. This chapter will primarily focus on integrating support for these customizable products to our framework as it is at the moment.

Giving users choice

Many products in e-commerce stores require some sort of choice from the customer, be it the size, color, or even the material. At the moment, we only have very basic products, which can simply be viewed by our customers. We need to extend this to allow customers to see variations of products, and to be able to choose their own variation of the product, before purchasing it.

Simple variants

The simplest form of variant would be a single drop-down box of variations of a product. If we took t-shirts as an example, we would just have a variation of size. This would be quite simple to implement, as we would only need to make a reference to the variant of the product the user decides to purchase in the end, essentially meaning that when we create our database table for items in the shopping basket; it needs only an extra field to record the variant the user is purchasing.

How could this work?

If we were to implement this method of product variations, how might we do it? Well, firstly we would need a list of possible variations, and secondly we would need to associate these with various products along with any additional cost implications they would have on the product.

Combinations of variants

Obviously, simple variants are quite limiting. If we want to have variations of both size and color, administrators would need to create a variant for each combination of these, which wouldn't really be practical. When it comes to developing our shopping basket later, however, there will obviously be some additional complications, which will need to be considered.

How will this work?

The easiest way for this to work is to have our framework work with a list of administrator-definable variation types or attributes, such as:

- Size
- Color
- Finish

Each of these attributes would be associated with a number of variations, for example:

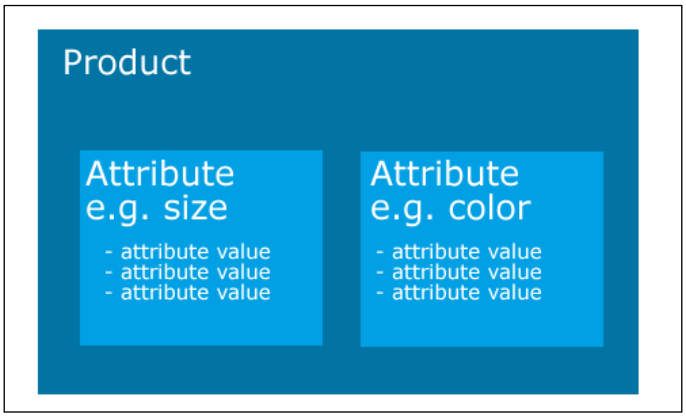
- Size:
    - Small
    - Medium
    - Large
    - XL
    - XXL

- Colors:
    - Red
    - Green
    - Blue
    - Yellow
    - Black
    - White
- Finish:
    - Matt
    - Gloss
    - Chrome

Then, each product will have a number of these variations associated with it, grouped by their variation type. The framework would also need to store and manage potential cost differences with different versions of a product; for example, if we were to sell printed photographs or canvases, larger sizes would almost definitely cost more, and require the product cost to be altered.

High-level overview

If we take a look at a high-level overview of what we have discussed, we can see more clearly how this all may relate.



The attribute values selected by the customer make up the end product that the customer chooses to purchase. This is opposed to each variation, or combination of variations, being a separate product.

Database structure

We need to create two tables to record the variation data itself, and some additional tables to maintain the relationship between products and their variants. We won't be associating a product with variation types (attributes), because not all products associated with those types of variation would need all of the variations associated with them. For example, if we were to sell both t-shirts and ballroom gowns, we may stock and sell t-shirts in sizes from Small to XXL, whereas with ballroom gowns, we may use a different way to describe the size, or we may not stock all of the different sizes available, so we wouldn't want the customer to be able to choose an option, that was not available.

Attributes table

The attributes table requires only two fields:

Field	Type	Description
ID	Integer (Primary Key, Auto Increment)	To reference the attribute from the attribute values table
Name	Varchar	The name of the attribute, for example size, color, and so on

The following SQL code represents this database table:

```
CREATE TABLE `product_attributes` (
    `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `name` VARCHAR( 50 ) NOT NULL)
    ENGINE = INNODB COMMENT = 'Product Attributes
    e.g. Color, Size, etc.';
```

Attribute values table

The attribute values table requires three fields:

Field	Type	Description
ID	Integer (Primary Key, Auto Increment)	To reference the attribute from the association with the products table
Name	Varchar	The name for the attribute value, for example, Blue, Large, and so on
Attribute_id	Integer	The attribute this value is associated with

The following code represents this database table:

```
CREATE TABLE `product_attribute_values` (
    `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `name` VARCHAR( 50 ) NOT NULL,
    `attribute_id` INT NOT NULL,
    INDEX ( `attribute_id` ))
    ENGINE = INNODB COMMENT = 'Product Attribute Values
    for example Blue, Large, etc.';
```

Product-attribute-value-association table

This table requires four fields:

Field	Type	Description
Product_id	Integer	The ID of the product we are associating with the attribute value
Attribute_value	Integer	The ID of the attribute value the product is being associated with
Order	Integer	Determines the order in which the value should be displayed in the attribute list
Cost_difference	Integer	Indicates if the variant product has a cost implication, for example ordering an extra large prop, or curtain may increase the cost

The following SQL code represents this table:

```
CREATE TABLE `product_attribute_value_association` (
  `product_id` int(11) NOT NULL,
  `attribute_id` int(11) NOT NULL,
  `order` int(11) NOT NULL,
  `cost_difference` double NOT NULL,
  KEY `product_id` (`product_id`,`attribute_id`),
  KEY `attribute_id` (`attribute_id`)
  ) ENGINE=InnoDB DEFAULT CHARSET=latin1 COMMENT='Association of
      products and attribute values';
--
-- Constraints for table `product_attribute_value_association`
--
ALTER TABLE `product_attribute_value_association`
  ADD CONSTRAINT `product_attribute_value_association_ibfk_2`
      FOREIGN KEY (`attribute_id`)
      REFERENCES `product_attribute_values` (`ID`)
      ON DELETE CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `product_attribute_value_association_ibfk_1`
      FOREIGN KEY (`product_id`) REFERENCES `content` (`ID`)
      ON DELETE CASCADE ON UPDATE CASCADE;
```

Template switching

Depending on if the product has variations, we need to change the template used to generate the product view to the customer. If the product has no variations, then they need to see the template we created in the previous chapter; if the product does have variations, then we need to display a template that can support the ability for the customer to choose their variants. For each variant type that a product can have, we would obviously need a new section within this template; this could be drop-down boxes, or some other method for selecting a variation. (For instance, we could have a series of boxes of each variation color, and clicking a box could set a hidden field to the value of the option.)

One such example of an extended template, as shown follows, lists options for colors and sizes of t-shirts:



Changing our product query

In order for the framework to detect if a product has any variations or changeable attributes, we need to modify the query we use to get the product data, and change this to perform a subquery, which groups together all of the attribute values and their attributes that can be associated with the product.

If there are no attribute values, then the model should have a property set to define that it has no customizable attributes; otherwise, the property should indicate that there are customizable attributes. This can be done using the following:

```
SELECT v.name AS product_name, c.ID AS product_id,
    (SELECT GROUP_CONCAT( a.name,'--AV--', av.ID, '--AV--',
       av.name SEPARATOR '---ATTR---' )
     FROM product_attribute_values av,
        product_attribute_value_association ava,
        product_attributes a
     WHERE a.ID = av.attribute_id AND av.ID=ava.attribute_id
        AND ava.product_id=c.ID ORDER BY ava.order ) AS attributes,
    p.image AS product_image, p.stock AS product_stock,
    p.weight AS product_weight, p.price AS product_price,
    p.SKU AS product_sku, p.featured AS product_featured,
    v.heading AS product_heading,
    v.content AS product_description,
    v.metakeywords AS metakeywords,
    v.metarobots AS metarobots,
    v.metadescription AS metadescription
FROM content_versions v, content c, content_types t,
    content_types_products p
WHERE c.active=1 AND c.secure=0 AND c.type=t.ID
    AND t.reference='product' AND p.content_version=v.ID
    AND v.ID=c.current_revision AND c.path='{$productPath}'
```

The highlighted code here shows the addition of this subquery.

Switching the template

Our controller can then check this property value with the model, and generate the relevant templates to form the view accordingly.

```
if( $this->model->hasAttributes() )
{
  $attrdata = $this->model->getAttributes();
  $attrs = array_keys( $attrdata );
  $temp = array();
  $aftertags =  array();
```

If the product has attributes, we iterate through them, and for each attribute we generate a list of values associated with it. The way we send these values to the template means that attributes are processed first, which repeat an empty drop-down list for each attribute type we have. For each of these lists duplicated, template tags are inserted, which allow us to dynamically insert the appropriate values for each of them.

```
    foreach( $attrs as $attribute )
    {

      $temp[] = array( 'attribute_name' => $attribute );
      $vtemp = array();
      foreach( $attrdata[ $attribute ] as $key => $value )
      {
        $vtemp[] = array('value_id'=> $value['attrid'],
            'value_name'=>$value['attrvalue']);
      }
      $cache = $this->registry->getObject('db')->cacheData( $vtemp );
      $aftertags[] = array( 'cache'=>$cache, 'tag' => 'values_'
          . $attribute);
    }
    $cache = $this->registry->getObject('db')->cacheData( $temp );
    $this->registry->getObject('template')->getPage()->
        addTag( 'attributes', array('DATA', $cache ) );
    foreach( $aftertags as $key => $data )
    {
      $this->registry->getObject('template')->getPage()->
           addTag( $data['tag'], array('DATA', $data['cache'] ) );
    }
    $this->registry->getObject('template')->
        buildFromTemplates('header.tpl.php',
        'product-attributes.tpl.php', 'footer.tpl.php');
}
```

If the product doesn't have attributes, we just display the standard product template for the view.

```
else
{

  $this->registry->getObject('template')->
      buildFromTemplates('header.tpl.php', 'product.tpl.php',
      'footer.tpl.php');
}
```

One important aspect of this code is that, each set of attribute values is cached, and then added as a tag for the template engine after the attribute sets themselves have been converted into tags. This is because the template system processes the tags in order, so if it doesn't do the attributes first, it won't find the tags it generates, which are replaced with the attribute values, as explained in the next section, *Templates*.

Templates

We have looked into how to alter the framework to use different templates depending on if the product has customizable attributes. The next stage is to actually make these templates.

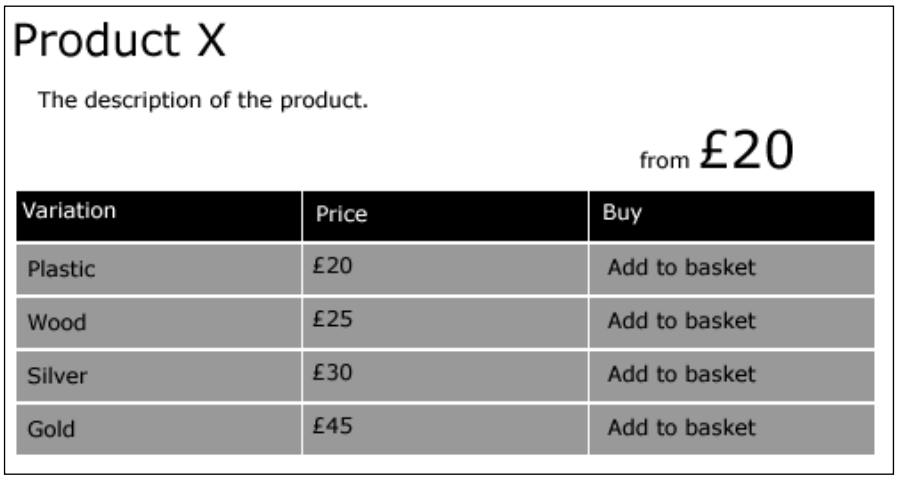
We need a template for the product with attributes. This template requires a nested loop of template tags: the outer loop of template tags will be to generate a drop-down list for each attribute, the inner loop to generate the attributes for each of the attribute drop-down lists.

```
<h2>{product_name}</h2>
{product_description}
<p>Cost: &pound;{product_price}, number in stock {product_stock}
    . Weight: {product_weight}Kg.
</p>
<p>
<img src="product_images/{product_image}" alt="{product_name}
     image" />
</p>
<!-- START attributes -->
<select name="attribute_{attribute_name}">
<!-- START values_{attribute_name} -->
<option value="">{value_name}</option>
<!-- END values_{attribute_name} -->
</select>
<!-- END attributes -->
```

This is the same code as our product template, with a `select` field. A drop-down list is generated for each set of attributes associated with the product, and then list items are generated where appropriate.

A look back at simple variants

We discussed simple product variants earlier. One advantage that they have over multiple variants becomes obvious when page design becomes a concern. If we were to have a system that utilized simple variants, we could display a simple table of product variants on the "main products" page, listing the names of products, cost or cost difference, and a purchase button.



Of course, this is something we could look at implementing as subproducts, should we wish to.

Giving users control

Along with giving users a choice for products in our store, we may also wish to give them some control over the products; for example, this could include:

- Uploading a photograph or image, for our Juniper Theatricals store; this will be for customers to order and purchase customized novelty t-shirts
- Supplying some custom text, again for our Juniper Theatricals store; this will be for customers to customize their novelty t-shirts with a punch line of their own, to fit into the products template

How to customize a product?

We need to make it possible for our customers to customize the product through both file and image uploads, and then entering of free text.

Uploads

If the product is to allow the customer to upload an image, then the template requires a file upload field within it, to facilitate that. We are only going to look at allowing a single file upload for each product; if a product required multiple images, for instance print media, where back and front designs may be required, the customer can compress them into a single file. It may be worth considering implementing multiple file uploads per product at a later stage, should the requirements of our store demand it.

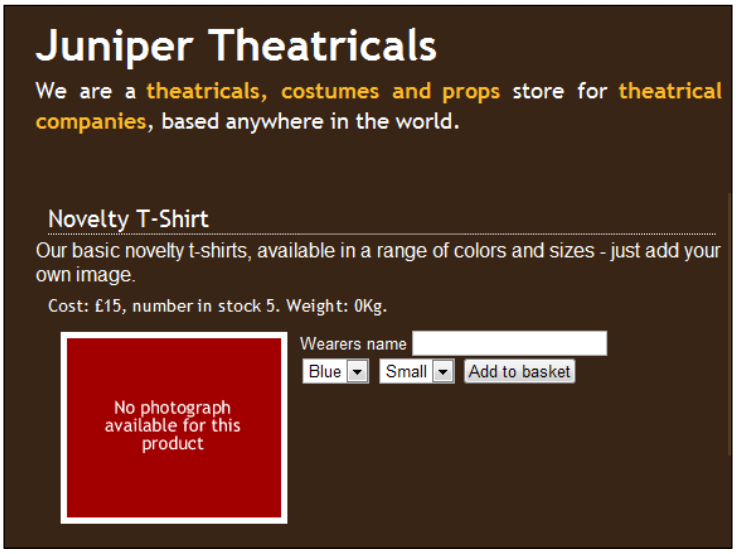
Custom text

The simplest way to handle custom text is to have at most one free text permitted per product. This would simply involve having an extra field in the table to contain items held within a customer's basket, relating to the value from this field, and also a field in the products table indicating that the product can accept free text as an input.

For many situations, this should be sufficient; however, let's look at a potential way we could support multiple text fields. In our Juniper Theatricals store, we will want to advertise customizable t-shirts for sale, with novelty text or images within them. If the customer is to supply some text, we may wish to provide them with options for entering text for the front, back, and perhaps even the sleeves, or below a logo, which may appear on the breast of the shirt.

If we have an additional field in our products table containing a list of free text attributes, we may wish to collect it from the user, and then generate text boxes for each of these attributes, and the submitted values could be serialized into an array and stored within a single field in the customer's entry for the product in the basket items table, when we develop that in Chapter 6.

When viewing a product with custom text inputs, the appropriate text boxes appear within the view, as the following screenshot illustrates.



Limitations to this method

This method obviously has limitations. Primarily, because we store all of the custom text inputs available and their values in a single database field, instead of one per value, it isn't going to be easy to search for product purchases based on the values submitted into these. Another limitation is that all of the text fields would need to be text input boxes, and we could extend this to support both input boxes and text areas, among other relevant and useful input boxes.

Maintaining uploads

In Chapter 6, we will look at processing these uploads; however, we will also need to consider maintaining them. If a file is uploaded and a product is added to a basket, and that basket is never converted into an actual order, we will want to remove that file. Similarly, once an order has been paid for, and processed fully, we would also want to remove it.

Security considerations

There are also a number of security considerations which we must bear in mind:

- By allowing customers to upload files, we could be open to abuse from someone repeatedly uploading images to our server. We could implement time delays to prevent this.
- Which types of files will we allow customers to upload? We should check both the type of the file uploaded and the file extension against a list of suitable values.
- What would the maximum file size be for files customers upload? If we set this value to be too large, our server will get filled up quickly with custom files, or could be abused by someone purposely uploading very large files.
- What safeguards are in place to prevent customers finding uploads of other customers?

Database changes

To allow customers to customize products, we obviously need to make some changes to our database structure to indicate that a particular product is customizable, and can be customized either by the customer uploading a file or entering some text.

Extending our products table

The changes required to our products table are actually quite simple; we only need to add two fields to the table:

- `allow_upload (Boolean)`: This field is used to indicate if the customer is permitted or able to upload a file when adding the product to their basket.
- `custom_text_inputs (longtext)`: This field is used to hold a serialized array of free text fields, which we may wish to collect from our customers.

The following SQL query will modify the table for us:

```
ALTER TABLE `content_types_products` ADD `allow_upload`
    BOOL NOT NULL, ADD `custom_text_inputs` LONGTEXT NOT NULL ;
```

Template switching

As with product variations, the templates can be used to generate the view to show the customer depending on if and how a product can be customized. If a product has no customization options, then we would just show them the standard template; otherwise, we will show them a template, which also supports the uploading of files and or supplying some text with the product. This needs to be done in conjunction with the template switching for the product variations, so let's edit the code we discussed earlier to account for additional text fields and upload form templates.

The first section of the code is the same as when we previously looked into switching templates.

```
if( $this->model->hasAttributes() )
{
  $attrdata = $this->model->getAttributes();
  $attrs = array_keys( $attrdata );
  $temp = array();
  $aftertags =  array();
  foreach( $attrs as $attribute )
  {
    $temp[] = array( 'attribute_name' => $attribute );
    $vtemp = array();
    foreach( $attrdata[ $attribute ] as $key => $value )
    {
      $vtemp[] = array('value_id'=> $value['attrid'],
          'value_name'=>$value['attrvalue']);
    }
    $cache = $this->registry->getObject('db')->cacheData( $vtemp );
    $aftertags[] = array( 'cache'=>$cache, 'tag' => 'values_'
        . $attribute);
  }
  $cache = $this->registry->getObject('db')->cacheData( $temp );
  $this->registry->getObject('template')->getPage()->
    addTag( 'attributes', array('DATA', $cache ) );
  foreach( $aftertags as $key => $data )
  {
    $this->registry->getObject('template')->getPage()->
        addTag( $data['tag'], array('DATA', $data['cache'] ) );
  }
  // The change to the code appears here, we detect if the product
  //  has custom text inputs.
  if( $this->model->hasCustomTextInputs() )
  {
```

```
        //If it does, we also check to see if the product allows uploads
        //  to be submitted by the customer.
        if( $this->model->allowUploads() )
        {
          // Assuming this is the case, we then build the list of fields
          //  and use a template, which accommodates both custom text
          //  inputs and file uploads.
          $fieldsdata = $this->model->getCustomTextInputs();
          $tags = array();
          foreach( $fieldsdata as $fieldkey => $name )
          {
           $tags[] = array('fieldkey' => $fieldkey, 'fieldname' => $name );
          }
          $cache = $this->registry->getObject('db')->cacheData( $tags );
          $this->registry->getObject('template')->getPage()->
              addTag('fields', array( 'DATA', $cache ) );
          $this->registry->getObject('template')->
              buildFromTemplates('header.tpl.php',
              'product-attributes-custom-upload.tpl.php',
              'footer.tpl.php');
        }
```

Building the list of fields is simply a case of caching an array of data, listing the name and key of the field—the key is used as the ID and name of the input fields, and the name value is for the label of the field, informing the customer of what information they need to supply.

```
        else
        {
          $fieldsdata = $this->model->getCustomTextInputs();
          $tags = array();
          foreach( $fieldsdata as $fieldkey => $name )
          {
            $tags[] = array('fieldkey' => $fieldkey,
                'fieldname' => $name );
          }
          $cache = $this->registry->getObject('db')->cacheData( $tags );
          $this->registry->getObject('template')->getPage()->
              addTag('fields', array( 'DATA', $cache ) );
          $this->registry->getObject('template')->
              buildFromTemplates('header.tpl.php',
              'product-attributes-custom.tpl.php', 'footer.tpl.php');
        }
      }
```

If the product doesn't have custom text inputs, we still check to see if uploads are permitted; if they are, we use a template which displays the upload field, but not the custom text inputs.

```
    elseif( $this->model->allowUploads() )
    {
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php',
          'product-attributes-uploads.tpl.php',
          'footer.tpl.php');
    }
    else
    {
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php',
          'product-attributes.tpl.php', 'footer.tpl.php');
    }
  }
  else
  {
    $this->registry->getObject('template')->
        buildFromTemplates('header.tpl.php', 'product.tpl.php',
        'footer.tpl.php');
  }
```

Shopping basket preparation

Although we won't be developing our shopping basket until Chapter 6, let us have a brief think about the consequences of customizable products.

Stock control

With product variations, stock control becomes an interesting issue. If we only had a single set of variations for each products (as we discussed under the *Simple variants* section, earlier in the chapter), we could simply disable an option if it was out of stock. However, with multiple variations we may have small blue t-shirts, but not large blue t-shirts. The logic for detecting if this is in stock obviously lies with the shopping basket itself: when we click on **Add to basket**, it will need to detect to see if there are any in stock.

This obviously isn't an ideal situation; however, an alternative would be to utilize AJAX to enable the view to alert the customer that a particular combination is out of stock, by performing a lookup as and when the customer selects their variation of the product.

Product variations

The shopping basket needs to detect which variations have been selected, and store a record of this as the customer's intended purchase. Because each product order could be a product with more than one variation, for instance both size and color, this would need to be maintained in a database table of its own.

Product customizations

For products where the customer can upload images or files, and also supply custom text, we need to record the file(s) and the text. And depending on the file type we may wish to do something with that; for instance, if it was an image we may wish for that to be displayed in the basket as the product's image.

Both this and the product variations have another interesting implication when it comes to the shopping basket. If we were to add a product to our shopping basket on an e-commerce store, and then go back to the products page and add it again, we would expect to see the product listed once, with a quantity of two. However, if we add one product, with a variation or uploaded file, and then add the product again, we would need for these to display as two separate listings in the shopping basket.

Basket templates

Because of the different templates we have created to represent the product view, there is a minor implication for our shopping basket. A standard product would just require a customer to click on a link to add a product to their basket. With the customizable products, the user would need to click on a form submit button to pass the uploaded files and custom text, so our basket will need to add products based on both methods.

Product subtotals

With the costs or cost differences associated with various product variations, we will need to take this into consideration when we have the basket calculate totals and subtotals of orders.

Summary

In this chapter, we have taken our e-commerce framework and extended the products' provisions to allow for variations of products and products to be customized by the customer using upload and free-text fields. We have also discussed the implications of these features upon the shopping basket, something which we will come to again later in Chapter 6, *The Shopping Basket*.

5
Enhancing the User Experience

We now have an e-commerce framework, which allows us to manage products and show them to our customers; we have also customized this functionality to allow our customers to be able to customize their purchases. Before moving on to the shopping basket, let us take a step back to focus on some important user experience aspects to our framework. In this chapter, you will learn how to enhance the user experience by:

- Allowing customers to search our product catalog effectively
- Enhancing this search by allowing our customers to filter products
- Providing wish lists for our customers
- Generating recommendations for customers based on previous purchases
- Informing customers when their desired products are back in stock
- Enabling social aspects such as product ratings and reviews from customers

Juniper Theatricals

Juniper Theatricals want to have a lot of products on their online store, and as a result they fear that some products may get lost within the website, or not be as obvious to their customers. To help prevent this problem, we will integrate product searching to make products easy to find, and we will add filters to product lists allowing customers to see products that match what they are looking for (for example, ones within their price range).

As some products could still be lost, they want to be able to recommend related products to customers when they view particular products. If a customer wants a product, and it happens to be out of stock, then they want to prevent the customer from purchasing it elsewhere; so we will look at stock notifications too.

The importance of user experience

Our customers' experience on the stores powered by our framework is very important. A good user experience will leave them feeling wanted and valued, whereas a poor user experience will leave them feeling unwanted, unvalued, and may leave a bad taste in their mouths.

Search

The ability for customers to be able to search, find, and filter products is vital, as if they cannot find what they are looking for they will be frustrated by our site and go somewhere where they can find what they are looking for much more easily.

There are two methods that can make it much easier for customers to find what they are looking for:

- **Keyword search**: This method allows customers to search the product catalog based on a series of keywords.
- **Filtering**: This method allows customers to filter down lists of products based on attributes, refining larger lists of products into ones that better match their requirements.

Finding products

The simplest way for us to implement a search feature is to search the product name and product description fields. To make the results more relevant, we can place different priorities on where matches were found; for instance, if a word or phrase is found in both the name and description then that would be of the highest importance; next would be products with the word or phrase in the name; and finally, we would have products that just have the word or phrase contained within the product description itself.

So, what is involved in adding search features to our framework? We need the following:

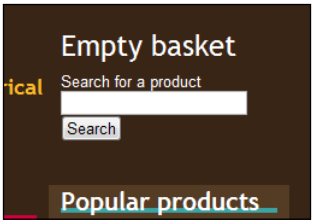
- **Search box**: We need a search box for our customers to type in words or phrases.
- **Search feature in the controller**: We need to add some code to search the products database for matching products.
- **Search results**: Finally, we need to display the matching products to the customer.

Search box

We need a search box where our customers can type in words or phrases to search our stores. This should be a simple `POST` form pointing to the path `products/search` with a search field of `product_search`. The best place for this would be in our website's header, so customers can perform their search from anywhere on the site or store.

```
<div id="search">
  <form action="products/search" method="post">
    <label for="product_search">Search for a product</label>
    <input type="text" id="product_search" name="product_search" />
    <input type="submit" id="search" name="search" value="Search" />
  </form>
</div>
```

We now have a search box on the store:



Controlling searches with the products controller

A simple modification to our products controller will allow customers to search products. We need to make a small change to the constructor, to ensure that it knows when to deal with search requests. Then we need to create a search function to search products, store the results, and display them in a view.

Constructor changes

A simple switch statement can be used to detect if we are viewing a product, performing a search, or viewing all of the products in the database as a list.

```
$urlBits = $this->registry->getURLBits();
if( !isset( $urlBits[1] ) )
{
  $this->listProducts();
}
else
{
  switch( $urlBits[1] )
```

```
    {
      case 'view':
      $this->viewProduct();
      break;
      case 'search':
      $this->searchProducts();
      break;
      default:
      $this->listProducts();
      break;
    }
  }
```

This works by breaking down the URL and, depending on certain aspects of the URL, different methods are called from within the controller.

Search function

We now need a function to actually search our products database, such as the following:

```
  private function searchProducts()
    {
    // check to see if the user has actually submitted the search form
    if( isset( $_POST['product_search'] ) &&
        $_POST['product_search'] != '' )
    {
```

Assuming the customer has actually entered something to search, we need to clean the search phrase, so it is suitable to run in our database query, and then we perform the query. The phrase is checked against the name and description of the product, with the name taking priority within the results. The highlighted code illustrates the query with prioritization.

```
      // clean up the search phrase
      $searchPhrase = $this->registry->getObject('db')->
          sanitizeData( $_POST['product_search'] );
      $this->registry->getObject('template')->getPage()->
          addTag( 'query', $_POST['product_search'] );
      // perform the search, and cache the results, ready for the
      // results template
      $sql = "SELECT v.name, c.path,
                IF(v.name LIKE '%{$searchPhrase}%', 0, 1) AS priority,
                IF(v.content LIKE '%{$searchPhrase}%', 0, 1)
                AS priorityb
              FROM content c, content_versions v, content_types t
              WHERE v.ID=c.current_revision AND c.type=t.ID
```

```
            AND t.reference='product' AND c.active=1
            AND ( v.name LIKE '%{$searchPhrase}%' OR v.content
            LIKE '%{$searchPhrase}%' )
          ORDER BY priority, priorityb ";
    $cache = $this->registry->getObject('db')->cacheQuery( $sql );
    if( $this->registry->getObject('db')->
        numRowsFromCache( $cache ) == 0 )
    {
      // no results from the cached query, display the no results
      // template
    }
```

If there are some products matching the search, then we display the results to the customer.

```
    else
    {
    // some results were found, display them on the results page
    // IMPROVEMENT: paginated results
    $this->registry->getObject('template')->getPage()->
        addTag( 'results', array( 'SQL', $cache ) );
    $this->registry->getObject('template')->
        buildFromTemplates('header.tpl.php',
        'products-searchresults.tpl.php', 'footer.tpl.php');
    }
  }
  else
  {
    // search form not submitted, so just display the search box page
    $this->registry->getObject('template')->
        buildFromTemplates('header.tpl.php',
        'products-searchform.tpl.php', 'footer.tpl.php');
  }
}
```

As the results from the query are stored in a cache, we can simply assign this cache to a template tag variable, and the results will be displayed. Of course, as we need to account for the fact that there may be no results, we must check to ensure there are some results, and if there are none, we must display the relevant template.

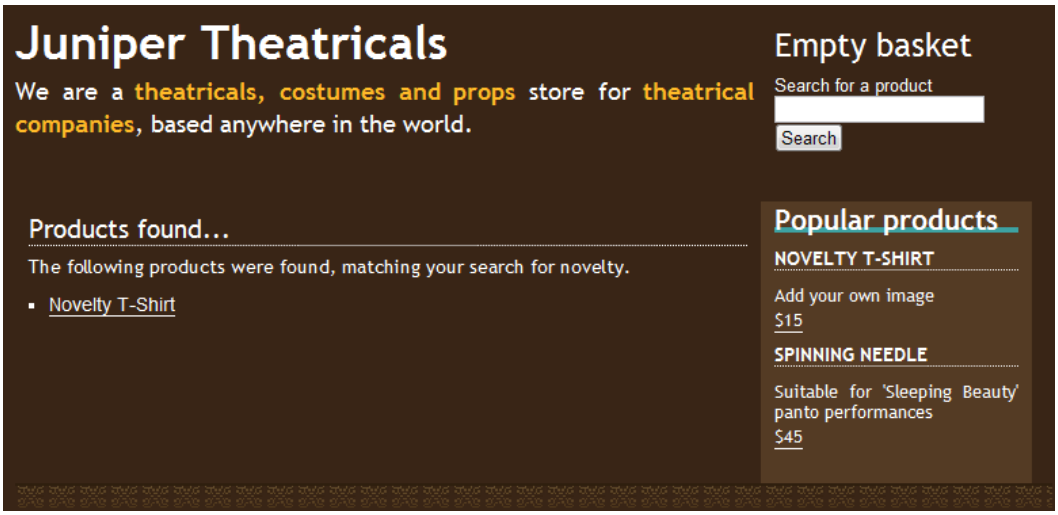
Search results

Finally, we need a results page to display these results on.

```
<h2>Products found...</h2>
<p>The following products were found, matching your search for
   {query}.</p>
```

```
<ul>
<!-- START results -->
<li><a href="products/view/{path}">{ name}</a></li>
<!-- END results -->
</ul>
```

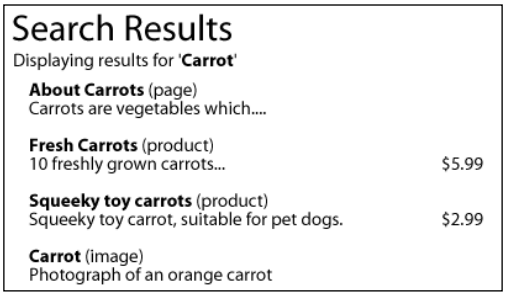
Our search results page looks like this:



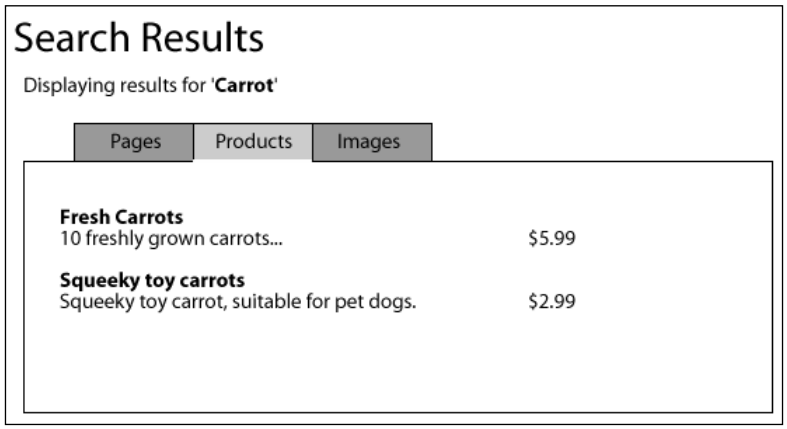
Improving searches

We could improve this search function by making it applicable for all types of content managed by the framework. Obviously if we were going to do this, it would need to be taken out of the products controller, perhaps either as a controller itself, or as a core registry function, or as part of the main content/pages controller.

The results could either be entirely in a main list, with a note of their type of content, or tabbed, with each type of content being displayed in a different tab. The following diagrams represent these potential **Search Results** pages.



And, of course, the tab-separated search results.



Filtering products

Another useful way to allow customers to better find the products they are looking for is with filtering. Customers can filter down lists of products based on attributes, such as price ranges, manufacturer, weight, brands, and so on.

Price range filtering should be simple enough. However, with attributes such as manufacturer or brands, we would need to extend the database and models representation of a product to maintain this additional information, and allow us to filter down based on these attributes.

There are a few different ways in which we can store filtered results:

- **In the user's session**: This will be lost when the user closes their browser.
- **In a cookie**: This information will stay when the user closes their browser.
- **In the URL**: This would allow the customer to filter results and send the link of those results to a friend.
- **In POST data**: The information will only be stored for the one instance the filter is used.

Let's try using the URL to store filter data. If we format filter data within the URL as `filter/attribute-type/attribute-value-ID`, then we can simply iterate through the bits of the URL, find bits containing `filter`, and then take the next two parts of the URL to help build the filter. This way we can filter down products based on a number of attributes, for example `filter/price/5/filter/weight/6`. Of course, there is a limit to this, and that is the maximum length of a URL.

Product attributes

Some attributes are already stored within the product, such as the weight and the price. However, we still need to store some ranges of these for our customers to filter by. As we discussed earlier, that we will store the attribute type as well as the attribute value within the URL, we can take the attribute type and either filter based on attribute values associated in the database (for example, products associated with brands for filtering by brand) or if the type is price or weight, we can detect that these should be filtered based on values stored in the products table.

Database changes

We are going to need to create three new database tables to effectively support product filtering as we have discussed. We will need:

- An attribute types table to manage types of attributes; for example, price, weight, brand, manufacturer, color, and so on
- An attribute values table to manage values and ranges of attributes; for example, < $5, $5 - $10, < 5 KG, Nike, Adidas, gold, red, and so on
- An associations table to associate products with attribute values

Filter attribute types

The attribute types table needs to be able to act as a grouping reference for attribute values, and also detect if an attribute value should be referenced against the products table, or the attribute associations table. Prices and weights would be referenced against the products table, where as brands, colors, and manufacturers would be referenced against the associations table.

Field	Type	Description
ID	Integer (Primary Key, Auto Increment)	A database reference for the attribute type.
Reference	Varchar	
Name	Varchar	The name of the type of attribute, for example price.
ProductContainedAttribute	Boolean	Specifies if the attribute is part of a field defined in the products table, such as price or weight, or not.

The following SQL represents this table:

```
CREATE TABLE `product_filter_attribute_types` (
  `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
  `reference` VARCHAR( 25 ) NOT NULL,
  `name` VARCHAR( 50 ) NOT NULL ,
  `ProductContainedAttribute` BOOL NOT NULL
) ENGINE = INNODB COMMENT = 'Product Attributes for Filtering
    Product Lists';
```

Filter attribute values

The attribute values table needs to store the name of the attribute and its relevant attribute type. Required fields are:

Field	Type	Description
ID	Integer (Primary Key, Auto Increment)	
Name	Varchar	The name of the attribute, for example < $10
Attribute type	Integer	A reference to the type of attribute (for example size, price)
Lower value	Integer	Used for attributes that are referenced within the products table (see below)
Upper value	Integer	Used for attributes that are referenced within the products table (see below)
Order	Integer	The order of the attribute in a list

The upper and lower values are used when referencing against the products table, so we can get our framework to quickly construct queries using the lower and upper values as ranges for the `WHERE` clause of the query.

The following SQL represents this table:

```
CREATE TABLE `product_filter_attribute_values` (
  `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `name` VARCHAR( 100 ) NOT NULL,
  `attributeType` INT NOT NULL,
  `order` INT NOT NULL,
  `lowerValue` INT NOT NULL,
  `upperValue` INT NOT NULL,
  INDEX ( `attributeType` )
) ENGINE = INNODB COMMENT = 'Attribute values for filtering products'
ALTER TABLE `product_filter_attribute_values`
```

```
ADD FOREIGN KEY ( `attributeType` )
REFERENCES `book4`.`product_filter_attribute_types` (`ID`)
ON DELETE CASCADE ON UPDATE CASCADE;
```

Attribute associations

The final table we need is the one to associate various attributes with various products, the data we need to store is:

- The product ID
- The attribute ID

The following SQL represents the previous table:

```
CREATE TABLE `product_filter_attribute_associations` (
  `attribute` INT NOT NULL,
  `product` INT NOT NULL,
  PRIMARY KEY ( `attribute` , `product` )
) ENGINE = INNODB COMMENT = 'Product attribute associations
    for filtering product lists';
ALTER TABLE `product_filter_attribute_associations`
ADD FOREIGN KEY ( `attribute` )
REFERENCES `book4`
    .`product_attribute_values` (`ID`)
ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE `product_filter_attribute_associations`
ADD FOREIGN KEY ( `product` )
REFERENCES `book4`.`content` (`ID`)
ON DELETE CASCADE ON UPDATE CASCADE;
```

Filter options

To display these attributes to our customers, and allow them to click on them to perform a filter, we need to build a list of attributes, build suitable URLs based on the attributes, and display them within the product list view.

Displaying these attributes will involve some nested-looped template tags. The first loop will be to generate headings and empty lists (with suitable template tags within) for the attribute types. Then we need to insert the loops of values into these.

The simplest way to do this would be to do a query of all of the attribute types, cache it, and assign it to a template variable, and then do this for each set of values. Of course, that isn't a very good way, as we end up doing one query per set of attribute types, which isn't very efficient. We need to query the attribute types, query all attribute values and then process them into groups, and associate these groups with relevant template tags.

Let's look at this as a step-by-step process:

1. We query the database for attribute types.
2. We cache the results of this query.
3. The cache is associated with a template tag. (This allows the template engine to generate a list of attribute types, and for each attribute type, it can build an empty list, surrounded by template tags, which will eventually contain the attribute values.)
4. We query the database for all attribute types, ordering by their own order. (Although the order is their order within their group, this does not matter, as we filter them out.)
5. We iterate through the results, putting each value into an array for its corresponding attribute type.
6. For each attribute type, we cache the array, and assign it to a template tag, allowing each group of values to populate the appropriate list for the attribute type.

Our modified controller now looks like this, with our aforementioned six steps commented in for reference:

```
private function generateFilterOptions()
{
  // 1. Query the database for attribute types
  $attrTypesSQL = "SELECT reference, name
                   FROM product_filter_attribute_types";
  $this->registry->getObject('db')->executeQuery( $attrTypesSQL );
  if( $this->registry->getObject('db')->numRows() != 0 )
  {
    $attributeValues = array();
    $attributeTypes = array();
    while( $attributeTypeData = $this->registry->
        getObject('db')->getRows() )
    {
      $attributeValues[ $attributeTypeData['reference'] ] = array();
      $attributeTypes[] = array(
         'filter_attr_reference' => $attributeTypeData['reference'],
         'filter_attr_name' => $attributeTypeData['name'] );
    }
    // 2. cache the results of this query
    $attributeTypesCache = $this->registry->getObject('db')->
        cacheData( $attributeTypes );
    // 3. The cache is associated with a template tag
    $this->registry->getObject('template')->getPage()->
```

```
            addTag( 'filter_attribute_types',
            array( 'DATA', $attributeTypesCache ) );
        // 4. We query the database for all attribute types,
        //    ordering by their own order
        $attrValuesSQL = "SELECT v.name AS attrName,
                                 t.reference AS attrType,
                                 v.ID AS attrID
                          FROM product_filter_attribute_values v,
                               product_filter_attribute_types t
                          WHERE t.ID=v.attributeType
                          ORDER BY v.order ASC";
        $this->registry->getObject('db')->executeQuery( $attrValuesSQL );
        if( $this->registry->getObject('db')->numRows() != 0 )
        {
          // 5. We iterate through the results, putting each value into
          //    an array for its corresponding attribute type.
          while( $attributeValueData = $this->registry->getObject('db')->
                                                          getRows() )
          {
            $data = array();
            $data['attribute_value'] = $attributeValueData['attrName'];
            $data['attribute_URL_extra'] = 'filter/'
                . $attributeValueData['attrType'] . '/'
                . $attributeValueData['attrID'];
            $attributeValues[ $attributeValueData['attrType'] ][] =
                                                                  $data;
          }
        }
        // 6. For each attribute type, we cache the array, and assign it
        //    to a template tag, allowing each group of values to
        //    populate the appropriate list for the attribute type.
        foreach( $attributeValues as $type => $data )
        {
          //echo '<pre>' . print_r( $attributeValues, true ) . '</pre>';
          $cache = $this->registry->getObject('db')->cacheData( $data );
          $this->registry->getObject('template')->getPage()->
              addPPTag( 'attribute_values_'
              . $type, array( 'DATA', $cache ) );
        }
      }
    }
```

Processing filter requests

With the relevant database structure in place, and functionality available for our customers to select attributes for which they wish to filter their product viewings, we need a method to process the request and actually filter the products listing.

This involves iterating through the bits within the URL, and for every instance of `filter` found, storing the following two values. Once all bits of the URL have been processed, the saved bits should be processed to build a suitable query to filter the products.

We will need some variables within our controller to store some of the data we will be processing. These would include:

- An array containing the filter attribute types, so we can pass the components of the URL to it in order to determine if the attribute value is from part of the products table itself, or if it is from an attribute association
- An array containing the filter attribute values, so when we find an attribute type that refers to the products table, we can get the upper- and lower-bound values for this
- An array of pieces of SQL to search for attribute associations
- An array of pieces of SQL to search for attribute values within the products table
- A counter for the number of filters by association, as we will group this part of the search into a subquery, returning the results of a count, and we will know if we have a match if the count matches the number of conditions to the subquery

These variables are displayed as follows:

```
// Filter count: to count how many attributes by association
// must match
private $filterCount=0;
// SQL statement parts where products are associated with
// attributes
private $filterAssociations = array();
// SQL statement parts where products are filtered by their own
// direct properties i.e. price, weight.
private $filterDirect = array();
// Array of filter attribute types
private $filterTypes = array();
// Array of filter attribute values
private $filterValues = array();
// our SQL statement for filtered products
private $filterSQL = '';
```

We now need a function to search through the URL and another function to add query pieces to our various arrays when it is passed the filter type and filter value once an occurrence of the word `filter` is found in the URL.

So, firstly we'll see a function to go through the URL.

```
/**
 * Generate an SQL statement for filtering products, based on URL
 * paramaters
 * @param array $bits the bits contained within the URL
 * @return void
 */
```

We first get all of the attribute types available, and then we get all of the attribute values.

```
private function filterProducts( $bits )
{
  // get our attribute types
  $attributeTypesSQL = "SELECT ID, reference, name,
                               ProductContainedAttribute
                        FROM  product_filter_attribute_types ";
  $this->registry->getObject('db')->executeQuery( $attributeTypesSQL
);
  while( $type = $this->registry->getObject('db')->getRows() )
  {
    $this->filterTypes[ $type['reference'] ] =
               array( 'ID' => $type['ID'],
               'reference'=>$type['reference'],
               'ProductContainedAttribute'=>
                    $type['ProductContainedAttribute'] );
  }
  // get our attribute values
  $attributeValuesSQL = "SELECT ID, name, lowerValue, upperValue
                         FROM product_filter_attribute_values";
  $this->registry->getObject('db')->
      executeQuery( $attributeValuesSQL );
  while( $value = $this->registry->getObject('db')->getRows() )
  {
    $this->filterValues[ $value['ID'] ] =
        array( 'ID' => $value['ID'],
        'name' => $value['name'],
        'lowerValue' => $value['lowerValue'],
        'upperValue' => $value['upperValue'] );
  }
```

For each part of the URL, we go through and find anything that relates to the filter functionality, which is of the format `filter/attribute-type/attribute-value`.

```
// process the URL
foreach( $bits as $position => $bit )
{
  // if we find filter in the URL
  if( $bit == 'filter' )
  {
    // send the next two bits to the addToFilter method
    $this->addToFilter( $bits[ $position+1], $bits[ $position+2] );
  }
}
```

We assume there are no filter requests being made, and set the basic filter query. Then we check if we have any filters that are not based on the product table values; if there are, we set the `somethingToFilter` variable, then we do the same for the filters based on the product table values. Each filter found adds additional restrictions to the basic filter SQL query.

```
// assume no filter requests
$somethingToFilter = false;
// basic filter query
$sql = "SELECT  p.price AS product_price, v.name AS product_name,
                c.path AS product_path
        FROM content c, content_types t, content_versions v,
             content_types_products p
        WHERE v.ID=c.current_revision AND c.active=1
              AND p.content_version=v.ID AND t.reference='product'
              AND c.type=t.ID ";
if( !empty( $this->filterAssociations ) )
{
  // we have some filter requests
  $somethingToFilter = true;
  // build the query
  $sqla = " AND ( SELECT COUNT( * )
            FROM product_filter_attribute_associations pfaa
            WHERE ( ";
  $assocs = implode( " AND ", $this->filterAssociations );
  $sqla .= $assocs;
  $sqla .= " )AND pfaa.product = c.ID )={$this->filterCount}";
  $sql .= $sqla;
}
if( !empty( $this->filterDirect ) )
{
  // we have some filter requests
```

```
      $somethingToFilter = true;
      // build the query
      $sql .= " AND ";
      $assocs = implode( " AND ", $this->filterDirect );
      $sql .= $assocs;
    }
    if( $somethingToFilter )
    {
      // since we have some filter requests, store the query.
      $this->filterSQL = $sql;
    }
  }
```

And secondly, we look at a function to build our SQL statement.

```
  /**
    * Add SQL chunks to our filter arrays, to help build our query,
    * based on actual filter requests in the URL
    * @param String $filterType the reference of the attribute type we
    * are filtering by
    * @param int $filterValue the ID of the attribute value
    * @return void
    */
  private function addToFilter( $filterType, $filterValue )
  {
    if( $this->filterTypes[ $filterType ]
                                  ['ProductContainedAttribute'] == 1 )
    {
      $lower = $this->filterValues[ $filterValue ]['lowerValue'];
      $upper = $this->filterValues[ $filterValue ]['upperValue'];
      $sql = " p.{$filterType} >= {$lower}
               AND p.{$filterType} < {$upper}";
      $this->filterDirect[] = $sql;
    }
    else
    {
      $this->filterCount++;
      $sql = " pfaa.attribute={$filterValue} ";
      $this->filterAssociations[] = $sql;
    }
  }
```

Displaying filtered products

Assuming we call our `filterProducts()` method within the products controller at some point, we can filter our products list quite easily. In our "products list" page, for instance, we can simply detect if the filter SQL field is empty; if it is not, we can replace the list query with the filter query. Of course, we should also swap our template, to indicate that the results are a filtered subset.

```
private function listProducts()
{
  if( $this->filterSQL == '' )
  {
    $sql = "SELECT p.price as product_price,
                   v.name as product_name,
                   c.path as product_path
            FROM content c, content_versions v,
                 content_types_products p
            WHERE p.content_version=v.ID AND v.ID=c.current_revision
                  AND c.active=1 ";
  }
  else
  {
    $sql = $this->filterSQL;
  }
  $cache = $this->registry->getObject('db')->cacheQuery( $sql );
  $this->registry->getObject('template')->getPage()->
    addTag( 'products', array( 'SQL', $cache ) );
  $this->registry->getObject('template')->
      buildFromTemplates('header.tpl.php',
      'list-products.tpl.php',
      'footer.tpl.php');
  $this->generateFilterOptions();
}
```

Remember, we must first call our `filterProducts` method, so I've added this to the switch statement within the controller's constructor.

```
$urlBits = $this->registry->getURLBits();
$this->filterProducts( $urlBits );
```

If we have filters in place with respect to price and weight (which are based off the products table), they would look like this:



If we click on one of the options, the products list would update to show products matching that criterion.

Improving product filtering

As with everything, there is always room for improvement. For the filter feature, potential improvements include:

- Displaying the number of products matching a filter next to it.
- Pagination—limiting the number of products displayed to the initial Y results, allowing the customer to move to the next set of results, so they are not overwhelmed with products.
- Updating this number to account for filters already in place (that is, if there are 100 brand X products and we filter the price to < $5, there may only be 20 matching brand X products, and the number should update to reflect this).
- Filter options with no matching products could be hidden, to prevent the customer from clicking them, and finding that it made no change.

Providing wish lists

Wish lists allow customers to maintain a list of products that they would like to purchase at some point, or that they would like others to purchase for them as a gift.

Creating the structure

To effectively maintain wish lists for customers, we need to keep a record of:

- The product the customer desires
- The quantity of the product
- If they are a logged-in customer, their user ID
- If they are not a logged-in customer, some way to identify their wish-list products for the duration of their visit to the site
- The date they added the products to their wish list
- The priority of the product in their wish lists; that is, if they really want the product, or if it is something they wouldn't mind having

Let's translate that into a suitable database table that our framework can interact with:

Field	Type	Description
ID	Integer (Primary Key, Auto Increment)	A reference for the database
Product	Integer	The product the user wishes to purchase
Quantity	Integer	The number of them the user would like
Date added	Datetime	The date they added the product to their wish list
Priority	Integer	Relative to other products in their wish list, and how important is this one
Session ID	Varcharr	The user's session ID (so they don't need to be logged in)
IP Address	Varchar	The user's IP address (so they don't need to be logged in)

By combining the session ID and IP address of the customer, along with the timestamp of when they added the product to their wish list, we can maintain a record of their wish list for the duration of their visit. Of course, they would need to register, or log in, before leaving the site, for their wish list to be permanently saved. This also introduces an element of maintenance to this feature, as once a customer who has not logged in closes their session, their wish-list data cannot be retrieved, so we would need to implement some garbage collection functions to prune this table.

The following SQL represents this table:

```
CREATE TABLE `wish_list_products` (
             `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
             `product` INT NOT NULL,
             `quantity` INT NOT NULL,
             `user` INT NOT NULL,
             `dateadded` TIMESTAMP NOT NULL
             DEFAULT CURRENT_TIMESTAMP,
             `priority` INT NOT NULL,
             `sessionID` VARCHAR( 50 ) NOT NULL,
             `IPAddress` VARCHAR( 50 ) NOT NULL,
             INDEX ( `product` )
) ENGINE = INNODB COMMENT = 'Wish list products'
ALTER TABLE `wish_list_products` ADD FOREIGN KEY ( `product` )
REFERENCES `book4`.`content` (`ID`)
ON DELETE CASCADE ON UPDATE CASCADE;
```

Saving wishes

Now that we have a structure in place for storing wish-list products, we need to have a process available to save them into the database. This involves a link or button placed on the product view, and either some modifications to our product controller, or a wish-list controller, to save the wish. As wish lists will have their own controller and model for viewing and managing the lists, we may as well add the functionality into the wish-list controller.

So we will need:

- a controller
- a link in our product view

Wish-list controller

The controller needs to detect if the user is logged in or not; if they are, then it should add products to the user's wish list; otherwise, it should be added to a session-based wish list, which lasts for the duration of the user's session.

The controller also needs to detect if the product is valid; we can do this by linking it up with the products model, and if it isn't a valid product, the customer should be informed of this. Let's look through a potential `addProduct()` method for our wish-list controller.

```
/**
  * Add a product to a user's wish list
  * @param String $productPath the product path
  * @return void
  */
```

We first check if the product is valid, by creating a new product model object, which informs us if the product is valid.

```
private function addProduct( $productPath )
{

  // check product path is a valid and active product
  $pathToRemove = 'wishlist/add/';
  $productPath = str_replace( $pathToRemove, '',
      $this->registry->getURLPath() );
  require_once( FRAMEWORK_PATH . 'models/products/model.php');
  $this->product = new Product( $this->registry, $productPath );
  if( $this->product->isValid()
  {
    // check if user is logged in or not
    if( $this->registry->getObject('authenticate')->
        loggedIn() == true )
    {
      //Assuming the user is logged in, we can also store their ID,
      // so the insert data is slightly different. Here we insert the
      // wish into the database.
      $wish = array();
      $pdata = $this->product->getData();
      $wish['product'] = $pdata['ID'];
      $wish['quantity'] = 1;
      $wish['user'] = $this->registry->getObject('authenticate')->
          getUserID();
      $this->registry->getObject('db')->
          insertRecords('wish_list_products', $wish );

      // inform the user
      $this->registry->getObject('template')->getPage()->
        addTag('message_heading', 'Product added to your wish list');
      $this->registry->getObject('template')->getPage()->
          addTag('message_heading', 'A ' . $pdata['name']
```

```
            .' has been added to your wish list');
        $this->registry->getObject('template')->
            buildFromTemplates('header.tpl.php', 'message.tpl.php',
                               'footer.tpl.php');
    }
```

The customer isn't logged into the website, so we add the wish to the database, using session and IP address data to tie the wish to the customer.

```
    else
    {

      // insert the wish
      $wish = array();
      $wish['sessionID'] = session_id();
      $wish['user'] = 0;
      $wish['IPAddress'] = $_SERVER['REMOTE_ADDR'];
      $pdata = $this->product->getData();
      $wish['product'] = $pdata['ID'];
      $wish['quantity'] = 1;
      $this->registry->getObject('db')->
          insertRecords('wish_list_products', $wish );
      // inform the user
      $this->registry->getObject('template')->getPage()->
          addTag('message_heading',
          'Product added to your wish list');
      $this->registry->getObject('template')->getPage()->
          addTag('message_heading', 'A ' . $pdata['name']
          .' has been added to your wish list');
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php', 'message.tpl.php',
          'footer.tpl.php');
    }
  }
```

The product wasn't valid, so we can't insert the wish, so we need to inform the customer of this.

```
  else
  {

    // we can't insert the wish, so inform the user
    $this->registry->getObject('template')->getPage()->
        addTag('message_heading', 'Invalid product');
    $this->registry->getObject('template')->getPage()->
        addTag('message_heading', 'Unfortunately, the product you
        tried to add to your wish list was invalid, and was not
        added, please try again');
```

```
        $this->registry->getObject('template')->
            buildFromTemplates('header.tpl.php', 'message.tpl.php',
                               'footer.tpl.php');
    }
}
```

Add to wish list

To actually add a product to our wish list, we need a simple link within our products view. This should be `/wishlist/add/product-path`.

```
<p>
  <a href="wishlist/add/{product_path}"
     title="Add {product_name} to your wishlist">
        Add to wishlist.
  </a>
</p>
```

We could encase this link around a nice image if we wanted, making it more user friendly. When the user clicks on this link, the product will be added to their wish list and they will be informed of that.

Viewing a wish list

As our customers are now able to create their own wish lists, we need to allow them to view and manage their wish lists.

Controller changes

Our controller needs to be modified to list items in a user's wish list; this involves detecting if the user is logged in or not, as this will determine the query it must use to lookup products. In addition to a function to display the list to the customer, we need to detect if the customer is trying to add a product to the list, or if they are trying to view the list, though a switch statement in the constructor.

```
private function viewList()
{
  $s = session_id();
  $ip = $_SERVER['REMOTE_ADDR'];
  $uid = $this->regisry->getObject('authenticate')->getUserID();
  if( $this->registry->getObject('authenticate')->loggedIn() )
  {
    $when = strtotime("-1 week");
    $when = date("Y-m-d h:i:s", $when);
```

```
  $sql = "SELECT p.price AS product_price,
                 v.name AS product_name,
                 c.path AS product_path
          FROM content c,
               content_versions v,
               content_types_products p,
               wish_list_items w
          WHERE c.ID=w.product
                AND p.content_version=v.ID
                AND v.ID=c.current_revision
                AND c.active=1
                AND ( w.user='{$uid}'
                OR ( w.sessionID='{$s}' AND w.IPAddress='{$ip}'
                     AND w.dateadded > '{$when}' ) )";
}
else
{
  $sql = "SELECT p.price AS product_price,
                 v.name AS product_name,
                 c.path AS product_path
          FROM content c,
               content_versions v,
               content_types_products p,
               wish_list_items w
          WHERE c.ID=w.product
                AND p.content_version=v.ID
                AND v.ID=c.current_revision
                AND c.active=1
                AND w.user=0
                AND ( w.sessionID='{$s}' AND w.IPAddress='{$ip}'
                      AND w.dateadded > '{$when}'   )";
}
$cache = $this->registry->getObject('database')->
    cacheQuery( $sql );
if( $this->registry->getObject('database')->
    numRowsFromCache( $cache ) == 0 )
{
  $this->registry->getObject('template')->getPage()->
      addTag('message_heading', 'No products');
  $this->registry->getObject('template')->getPage()->
      addTag('message_heading', 'Unfortunately, there are no
              products in your wish-list at this time.');
  $this->registry->getObject('template')->
      buildFromTemplates('header.tpl.php',
                         'message.tpl.php',
                         'footer.tpl.php');
}
else
```

```
    {
      $this->registry->getObject('template')->getPage()->
          addTag( 'wishes', array( 'SQL', $cache ) );
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php', 'wishlist.tpl.php',
                             'footer.tpl.php');
    }
  }
```

The highlighted section within that code segment illustrates the difference between a user being logged out or logged in. For a logged-in user, the wish list displays products both associated to their user account and also ones associated with their session and IP address details. For users who are logged out, it ensures the user ID is set to 0; otherwise, they may end up viewing other customers' wish-list products. It limits the timeframe for which session-based products are viewed. This is because session IDs and IP addresses are not a secure method to "authenticate" against. We should investigate some form of garbage collection to ensure out-of-date session-based wish-list products are removed, and also something to detect if the logged-in user has some wish-list products that are not associated with their user ID, and transfer them to their user account to prevent them from losing them.

Wish-list view

Although we would need to extend this later, to include a purchase button, a priority, and quantity, this code would suffice for a basic view to show our customers their desired products for the time being:

```
<h2>Your wishlist</h2>
<ul>
  <!-- START wishes -->
  <li><a href="products/view/{product_path}">{product_name}</a></li>
  <!-- END wishes -->
</ul>
```

Purchases

The purchases aspect of this feature is the most complicated, as it needs to facilitate both customers making purchases for themselves, and also others making a gift purchase for someone. We can't actually implement this aspect yet, as we don't have a shopping basket or a complete order process. However, we can discuss what will be involved.

Gift purchases

The complication of gift purchases is that the purchases need to be stored with the delivery address from the customer who created the wish list. The difficulty here is that, at any stage, the delivery address should not be presented to the user making the purchase, as this is private information.

Self purchases

Self purchases should be very straightforward to handle. Essentially, all the customer would be doing is adding their own wish-list product to their own shopping basket, the only difference being that we must maintain a record that this is from their wish list up until the point of their order being finalized; then we must remove the product from their wish-list completely, to prevent them from making a duplicate purchase.

Improving the wish list

There are a number of ways in which we could improve the wish-list feature we have added to our framework, including:

- Multiple lists per customer, allowing customers to maintain separate lists
- Garbage collection for session-based wish-list products, ensuring we don't have useless data in our database
- Transferring of session-based wish-list products to user account-based wish-list products when a user is logged in
- Model, as we didn't implement a model with this wish list, and we should do so to make it easier to extend
- Priority isn't considered or displayed to the customer, or anyone who would like to buy the product as a gift for someone
- Quantities, at present, they are not considered when adding a product to a list; perhaps we should look for existing products in the wish list and increment their quantity
- Public and private lists, allowing customers to have a private list, and also a public list of items they may wish for others to purchase for them



These improvements are ones you should investigate by adding yourself.

Recommendations

Sometimes, we may find that certain products go hand in hand, or that customers interested in certain products also find another group of products interesting or relevant. If we can suggest some relevant products to our customers, we increase the chances of them making a new purchase or adding something else to their shopping basket. There are two methods of recommendation that we should look into:

- Displaying related products on a products page
- E-mailing customers who have made certain purchases to inform them of some other products they may be interested in

Related products

The simplest way to inform customers of related products from within the product view is to maintain a relationship of related products within the database and within the products model, so we could cache the result of a subset of these related products. This way, the controller needs to only detect that there are more than zero related products, insert the relevant template bit into the view, and then associate the cached query as the template tag variable to ensure that they are displayed.

There are a few ways in which we can maintain this relationship of related products:

- Within the products table we maintain a serialized array of related product IDs
- We group related products together by themes
- We relate pairs of related products together

A serialized array isn't the most effective way to store related product data. Relating them by themes would prove problematic with multiple themes, and also when it comes to the administrator relating products to each other, as they would have to select or create a new theme. Relating pairs of products together would require a little trick with the query to get the product name, as the ID of the product being viewed could be one of two fields, as illustrated by the following table structure:

- ID (Integer, Primary Key, Auto Increment)
- ProductA (Integer)
- ProductB (Integer)

The difference between using productA or productB to store a particular product reference would be determined by the administration panel we develop. So if we were to view and edit a product in the administration panel, and we chose to set a related product, the product we were currently viewing would be productA and the related one, productB.

The SQL for this table structure is as follows:

```
CREATE TABLE `product_relevant_products` (
  `ID` int(11) NOT NULL auto_increment,
  `productA` int(11) NOT NULL,
  `productB` int(11) NOT NULL,
  PRIMARY KEY  (`ID`),
  KEY `productB` (`productB`),
  KEY `productA` (`productA`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=3 ;
ALTER TABLE `product_relevant_products`
  ADD CONSTRAINT `product_relevant_products_ibfk_2`
  FOREIGN KEY (`productB`) REFERENCES `content` (`ID`)
  ON DELETE CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `product_relevant_products_ibfk_1`
  FOREIGN KEY (`productA`) REFERENCES `content` (`ID`)
  ON DELETE CASCADE ON UPDATE CASCADE;
```

We can get round the issue of the fact that the current product ID could be found in both the productA and productB columns in the database with an IF statement within our query. The IF statement would work by checking to see if productA is the product the customer is viewing; if it is, then the name of productB is returned; otherwise, the name of productB is returned. This gives us a query such as the following, where `CURRENT_PRODUCT_ID` is the ID of the product the customer is currently viewing.

```
SELECT IF(rp.productA<>CURRENT_PRODUCT_ID,v.name,vn.name)
           AS product_name,
       IF(rp.productA<>CURRENT_PRODUCT_ID,c.path,cn.path)
           AS product_path, rp.productA, rp.productB,
          c.path AS cpath, cn.path AS cnpath, c.ID AS cid,
          cn.ID AS cnid
FROM content c, content cn, product_relevant_products rp,
     content_versions v, content_versions vn
WHERE (rp.productA= CURRENT_PRODUCT_ID
       OR rp.productB= CURRENT_PRODUCT_ID)
      AND c.ID=rp.productA AND cn.ID=rp.productB
      AND v.ID=c.current_revision AND vn.ID=cn.current_revision
```

As we may have a lot of related products, we may wish to put a limit on the number of related products displayed, and randomize the results.

```
SELECT IF(rp.productA<>CURRENT_PRODUCT_ID,v.name,vn.name)
           AS product_name,
       IF(rp.productA<>CURRENT_PRODUCT_ID,c.path,cn.path)
           AS product_path, rp.productA, rp.productB,
          c.path AS cpath, cn.path AS cnpath, c.ID AS cid,
          cn.ID AS cnid
FROM content c, content cn, product_relevant_products rp,
     content_versions v, content_versions vn
WHERE (rp.productA= CURRENT_PRODUCT_ID
      OR rp.productB= CURRENT_PRODUCT_ID) AND c.ID=rp.productA
      AND cn.ID=rp.productB AND v.ID=c.current_revision
      AND vn.ID=cn.current_revision
ORDER BY RAND() LIMIT 5
```

Controlling the related products

If we create a new function within our controller to run our random related products query, cache the result, and associate the cached results with a template variable, all we would need to do is call this function from within the product view function, passing the product ID as the parameter.

```
private function relatedProducts( $currentProduct )
{
  $relatedProductsSQL = "SELECT ".
    " IF(rp.productA<>{$currentProduct},v.name,vn.name)
          AS product_name,
      IF(rp.productA<>{$currentProduct},c.path,cn.path)
          AS product_path, rp.productA, rp.productB, c.path as cpath,
         cn.path AS cnpath, c.ID AS cid, cn.ID AS cnid ".
                       " FROM ".
    " content c, content cn, product_relevant_products rp,
      content_versions v, content_versions vn".
                       " WHERE ".
    " (rp.productA={$currentProduct}
         OR rp.productB={$currentProduct}) ".
    " AND c.ID=rp.productA ".
    " AND cn.ID=rp.productB ".
    " AND v.ID=c.current_revision ".
    " AND vn.ID=cn.current_revision ".
                       " ORDER BY RAND() ".
    " LIMIT 5";
  $relatedProductsCache = $this->registry->getObject('db')->
    cacheQuery( $relatedProductsSQL );
  $this->registry->getObject('template')->getPage()->
    addTag('relatedproducts', array( 'SQL', $relatedProductsCache ));
}
```

The following line calls our new function:

```
$this->relatedProducts( $productData['ID'] );
```

Viewing the related products

Related products are now cached and associated with a template variable. We now need to add relevant mark up into our view product template, to display related products.

```
<h2>Related products</h2>
<!-- START relatedproducts -->
<div class="floatingbox">
  <a href="products/view/{product_path}">{product_name}</a>
</div>
<!-- END relatedproducts -->
```

Of course, we can style this to however we wish: we could display them as small boxes alongside each other, or we could also use JavaScript to toggle between them. However, for now, a simple list or floating `<div>` should suffice.

Here, beneath the product details, we have a list of related products:



E-mail recommendations

There is only so much we can do for this feature at the moment, as it requires our customers to have made some purchases, and at the moment, we don't have the functionality available for customers to make a purchase. However, we can discuss what would be involved in creating this feature:

1. Search customers with previous purchases that match a subset of the product catalog (for example customers who purchased a red t-shirt and a red baseball cap).
2. Select products that are related to the subset we defined earlier and we think those customers would be interested in.
3. Generate an e-mail based on those products, a set template, and other content we may wish to supply.
4. Send the e-mail to all of the customers found in step 1.

Help! It's out of stock!

If we have a product that is out of stock, we need to make it possible for our customers to sign up to be alerted when they are back in stock. If we don't do this, then they will be left with the option of either going elsewhere, or regularly returning to our store to check on the stock levels for that particular product. To try and discourage these customers from going elsewhere a "tell me when it is back in stock" option saves them the need to regularly check back, which would be off-putting. Of course, it is still likely that the customer may go elsewhere; however, if our store is niche, and the products are not available elsewhere, then if we give the customer this option they will feel more valued.

There are a few stages involved in extending our framework to support this:

1. Firstly, we need to take into account stock levels.
2. If a product has no stock, we need to insert a new template bit with an "alert me when it is back in stock" form.
3. We need a template to be inserted when this is the case.
4. We then need functionality to capture and store the customer's e-mail address, and possibly their name, so that they can be informed when it is back in stock.
5. Next, we need to be able to inform all of the customers who expressed an interest in a particular product when it is back in stock.
6. Once our customers have been informed of the new stock level of the product, we need to remove their details from the database to prevent them from being informed at a later stage that there are more products in stock.
7. Finally, we will also require an e-mail template, which will be used when sending the e-mail alerts to our customers.

Detecting stock levels

With customizable products, stock levels won't be completely accurate. Some products may not require stock levels, such as gift vouchers and other non-tangible products. To account for this, we could either add a new field to our database to indicate to the framework that a products stock level isn't required for that particular product, or we could use an extreme or impossible value for the stock level, for example -1 to indicate this.

Changing our controller

We already have our model set to pull the product stock level from the database; we just need our controller to take this value and use different template bits where appropriate. We could also alter our model to detect stock levels, and if stock is required for a product.

```
if( $productData['stock'] == 0 )
{
  $this->registry->getObject('template')->
      addTemplateBit( 'stock', 'outofstock.tpl.php' );
}
elseif( $productData['stock'] > 0 )
{
  $this->registry->getObject('template')->
      addTemplateBit( 'stock', 'instock.tpl.php' );
}
else
{
  $this->registry->getObject('template')->getPage()->
      addTag( 'stock', '' );
}
```

This simple code addition imports a template file into our view, depending on the stock level.

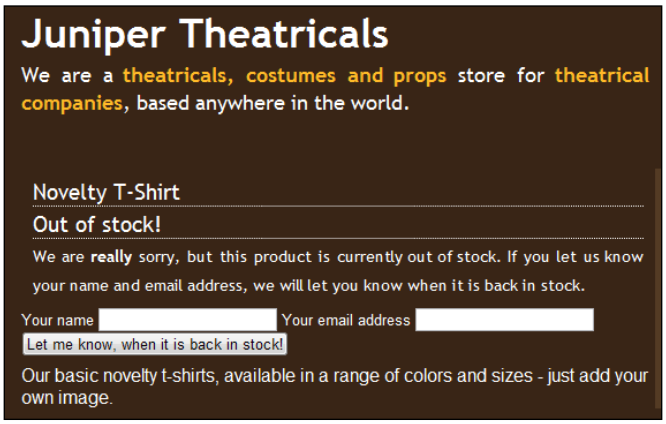
Out of stock: A new template bit

When the product is out of stock, we need a template to contain a form for the user to complete, so that they can register their interest in that product.

```
<h2>Out of stock!</h2>
<p>
  We are <strong>really</strong> sorry, but this product is currently
  out of stock. If you let us know your name and email address, we
  will let you know when it is back in stock.
```

```
</p>
<form action="products/stockalert/{product_path}" method="post">
<label for="stock_name">Your name</label>
<input type="text" id="stock_name" name="stock_name" />
<label for="stock_email">Your email address</label>
<input type="text" id="stock_email" name="stock_email" />
<input type="submit" id="stock_submit" name="stock_submit"
       value="Let me know, when it is back in stock!" />
</form>
```

Here we have the form showing our product view, allowing the customer to enter their name and e-mail address:



Tell me when it is back in stock please!

Once a customer has entered their name, e-mail address, and clicked on the submit button, we need to store these details and associate them with the product. This is going to involve a new database table to maintain the relationship between products and customers who wish to be notified when they are back in stock.

Stock alerts database table

We need to store the following information in the database to manage a list of customers interested in being alerted when products are back in stock:

- Customer name
- Customer e-mail address
- Product

In terms of a database, the following table structure would represent this:

Field	Type	Description
ID	Integer (Primary Key, Auto Increment)	The ID for the stock alert request
Customer	Varchar	The customer's name
Email	Varchar	The customer's e-mail address
ProductID	Integer	The ID of the product the customer wishes to be informed about when it is back in stock

The following SQL represents this table:

```
CREATE TABLE `product_stock_notification_requests` (
  `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
  `customer` VARCHAR( 100 ) NOT NULL ,
  `email` VARCHAR( 255 ) NOT NULL ,
  `product` INT NOT NULL ,
  `processed` BOOL NOT NULL ,
  INDEX ( `product` , `processed` )
) ENGINE = INNODB COMMENT = 'Customer notification requests for
                             new stock levels'
ALTER TABLE `product_stock_notification_requests`
ADD FOREIGN KEY ( `product` ) REFERENCES `book4`.`content` (`ID`)
ON DELETE CASCADE ON UPDATE CASCADE ;
```

More controller changes

Some modifications are needed to our product's controller to process the customer's form submission and save it in the stock alerts database table.

In addition to the following code, we must also change our switch statement to detect that the customer is visiting the `stockalert` section, and that the relevant function should be called.

```
private function informCustomerWhenBackInStock()
{
  $pathToRemove = 'products/stockalert/';
  $productPath = str_replace( $pathToRemove, '',
                              $this->registry->getURLPath() );
  require_once( FRAMEWORK_PATH . 'models/products/model.php');
  $this->model = new Product( $this->registry, $productPath );
```

Once we have included the model and checked that the product is valid, all we need to do is build our insert array, containing the customer's details and the product ID, and insert it into the notifications table.

```
    if( $this->model->isValid() )
    {

      $pdata = $this->product->getData();
      $alert = array();
      $alert['product'] = $pdata['ID'];
      $alert['customer'] = $this->registry->getObject('db')->
          sanitizeData( $_POST['stock_name'] );
      $alert['email'] = $this->registry->getObject('db')->
          sanitizeData( $_POST['stock_email'] );
      $alert['processed'] = 0;
      $this->registry->getObject('db')->
         insertRecords('product_stock_notification_requests', $alert );
      // We then inform the customer that we have saved their request.
      $this->registry->getObject('template')->getPage()->
          addTag('message_heading', 'Stock alert saved');
      $this->registry->getObject('template')->getPage()->
          addTag('message_heading', 'Thank you for your interest in
          this product, we will email you when it is back in stock.');
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php', 'message.tpl.php',
                             'footer.tpl.php');
    }
```

If the product wasn't valid, we tell them that, so they know the notification request was not saved.

```
    else
    {

      $this->registry->getObject('template')->getPage()->
          addTag('message_heading', 'Invalid product');
      $this->registry->getObject('template')->getPage()->
          addTag('message_heading', 'Unfortunately, we could not find
                  the product you requested.');
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php', 'message.tpl.php',
                             'footer.tpl.php');
    }
  }
```

This code is very basic, and does not validate e-mail address formats, something which must be done before we try to send any e-mails out.

It is back!

Once the product is back in stock, we need to then alert those customers that the product which they were interested in is back in stock, and that they can proceed to make their purchase. This isn't something we can implement now, as we don't have an administrative interface in place yet. However, we can discuss what is involved in doing this:

1. The administrator alters the stock level.
2. Customers interested in that product are looked up.
3. E-mails for each of those customers are generated with relevant details, such as their name and the name of the product being automatically inserted.
4. E-mails are sent to the customers.

The database contains a processed field, so once an e-mail is sent, we can set the processed value to 1, and then once we have alerted all of our customers, we can delete those records. This covers us in the unlikely event that all the new stock sells out while we are e-mailing customers, and a new customer completes the notification form.

Giving power to customers

There are two very powerful social-oriented features, which we can implement into our framework.

Product ratings

Product ratings are quite simple to add to our framework: we simply need to record a series of ratings between one and five, and display the average of these on the product view. We can enhance the view by making the rating system a clickable image, where the customer can click on the number of stars they wish to give the product and their rating is saved.

There are a few minor considerations that need to be taken into account. However, if the logged-in customer has already rated a product, we should then update their rating. If the customer is not logged in, we must record some information about them such as their IP address and the date and time of the rating. This way we prevent duplicate ratings from the same customer.

From a database perspective, we would need to capture the following information:

- ID (Integer, Primary Key, Auto Increment)
- ContentID (Integer)
- Rating (Integer)
- User ID (Integer)
- Timestamp (datetime)
- Session ID (Varchar)
- IP Address (Varchar)

The following SQL represents that table in our database:

```
CREATE TABLE `content_ratings` (
  `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
  `contentID` INT NOT NULL ,
  `rating` INT NOT NULL ,
  `userID` INT NOT NULL ,
  `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
  `sessionID` VARCHAR( 255 ) NOT NULL ,
  `IPAddress` VARCHAR( 50 ) NOT NULL
) ENGINE = INNODB;
```

Saving a rating

When a rating is made, we need to check to see if the current user has already rated that content element; if they have, then we must update that rating. For users who are not logged in, we should use their session name and IP address to lookup a potential rating from the past 30 days; if a rating is found, that should be updated.

Saving the rating should be made simple by processing values from the URL. This way, we can have a graphic of five stars or hearts, which when clicked, contain a link to the corresponding number of stars, to save the suitable rating.

As ratings and reviews (comments) will not be content specific, we will need a separate controller for these. To return the customer to the page they were on previously, we could investigate looking up the referring page, and then redirecting the user to that page once their rating has been saved.

The constructor of the controller needs to parse the URL bits to extract the content ID and the rating value, ensure that the rating is within allowed limits, and then call the `saveRating` function, which either inserts or updates a rating as appropriate.

To check if the user has rated the product already, we query the database; depending on if the user is logged in, this query is different. For users who are not logged in, we assume users with the same IP address and session data within the past 30 days were the current users.

```
private function saveRating( $contentID, $rating )
  {

  if( $this->regisry->getObject('authenticate')->isLoggedIn() )
  {
    $u = $this->registry->getObject('authenticate')->getUserID();
    $sql = "SELECT ID FROM content_ratings
            WHERE contentID={$contentID} AND userID={$u}";
  }
  else
  {
    $when = strtotime("-30 days");
    $when = date( 'Y-m-d h:i:s', $when );
    $s = session_id();
    $ip = $_SERVER['REMOTE_ADDR'];
    $sql = "SELECT ID FROM content_ratings
            WHERE content_id={$contentID} AND userID=0
                  AND sessionID='{$s}' AND IPAddress='{$ip}'
                  AND timestamp > '{$when}'";
  }
  $this->registry->getObject('db')->executeQuery( $sql );
```

If the product has already been rated, we update the rating.

```
  if( $this->regisry->getObject('db')->numRows() == 1 )
  {

    // update
    $data = $this->registry->getObject('db')->getRows();
    $update = array();
    $update['rating'] = $rating;
    $update['timestamp'] = date('Y-m-d h:i:s');
    $this->registry->getObject('db')->
     updateRecords( 'content_ratings', $update, 'ID=' . $data['ID']);
    $this->registry->getObject('template')->getPage()->
        addTag('message_heading', 'Rating changed');
    $this->registry->getObject('template')->getPage()->
        addTag('message_heading', 'Your rating has been changed');
    $this->registry->getObject('template')->
        buildFromTemplates('header.tpl.php', 'message.tpl.php',
                           'footer.tpl.php');
  }
```

Otherwise, we insert a new record in the ratings table.

```
    else
    {

      // insert
      $rating = array();
      $rating['rating'] = $rating;
      $rating['contentID'] = $contentID;
      $rating['sessionID'] = session_id();
      $rating['userID'] = ( $this->registry->
          getObject('authenticate')->isLoggedIn() == true ) ?
          $this->registry->getObject('authenticate')->getUserID() : 0;
      $rating['IPAddress'] = $_SERVER['REMOTE_ADDR'];
      $this->registry->getObject('db')->
          insertRecords( 'content_ratings', $rating );
      $this->registry->getObject('template')->getPage()->
          addTag('message_heading', 'Rating saved');
      $this->registry->getObject('template')->getPage()->
          addTag('message_heading', 'Your rating has been saved');
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php', 'message.tpl.php',
                             'footer.tpl.php');
    }
  }
```

Viewing ratings

To display the ratings, we need to alter the content or products query to also perform a subquery, which averages out the rating.

```
( SELECT sum(rating)/count(*)
  FROM content_ratings
  WHERE contentID=c.ID ) AS rating
```

Improving the user interface for ratings

Displaying ratings as nice graphics, which display both the current rating and allow the user to select their own rating from them, are chapters in themselves. There are a number of Internet tutorials that document this process; you may find them useful:

- http://www.komodomedia.com/blog/2005/08/creating-a-star-rater-using-css/
- http://www.search-this.com/2007/05/23/css-the-star-matrix-pre-loaded/

Product reviews

Product reviews can work as a simple comment form for the products, taking the name and e-mail address of the customer, as well as their review. Product reviews can be represented in the same way that we would represent comments on pages or blog entries, and because we have set up our database to store pages, products, and other types of content with a reference to a single database table, we can reference our reviews or comments to any content type. From a database perspective, a table with the following fields would suffice:

Field	Type	Description
ID	Integer (Auto Increment, Primary Key)	Review ID
Content	Integer	The content entity the user is reviewing
Customer name	Varchar	The customer
Customer email	Varchar	The customer's e-mail address
Review	Longtext	The customer's review
IPAddress	Varchar	The user's IP address
Date Added	Timestamp	The date they added the review
Approved	Boolean	If the review is approved and shown on the site

The following SQL code represents that table in our database:

```
CREATE TABLE `content_comments` (
  `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `content` INT NOT NULL,
  `authorName` VARCHAR( 50 ) NOT NULL,
  `authorEmail` VARCHAR( 50 ) NOT NULL,
  `comment` LONGTEXT NOT NULL,
  `IPAddress` VARCHAR( 40 ) NOT NULL,
  `dateadded` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `approved` BOOL NOT NULL,
  INDEX ( `content` )
) ENGINE = INNODB COMMENT = 'Content comments - also for product
                             reviews';
ALTER TABLE `content_comments`
ADD FOREIGN KEY ( `content` ) REFERENCES `book4`.`content` (`ID`)
ON DELETE CASCADE ON UPDATE CASCADE ;
```

Processing reviews/comments

With a database in place for our product reviews (or page comments), we need to provide a form for our customers to enter their review, and then process this submission and save it in the database.

Submission form

The submission form can be quite simple; we will only be collecting the customer's name, e-mail address, and their review:

```
<h2>Review this product</h2>
<form action="contentcomment/{ID}" method="post">
<label for="comment_name">Your name</label>
<input type="text" id="comment_name" name="comment_name" />
<label for="comment_email">Your email address</label>
<input type="text" id="comment_email" name="comment_email" />
<label for="comment">Your review</label>
<textarea name="comment" id="comment"></textarea>
<input type="submit" id="savecomment" name="savecomment"
       value="Add review" />
</form>
```

Adding the review

When it comes to saving the review, it is a simple case of sanitizing our data, and inserting it into the database.

I've not added checks to ensure the page or product exists, and the error checking for name and e-mail addresses is basic. Ideally, we would want to return the customer to the contact form, with the invalid fields highlighted.

```
private function saveComment( $contentID )
{
  //We build our insert array of data for the review record.
  $insert = array();
  $insert['content'] = $contentID;
  $insert['authorName'] = strip_tags($this->registry->
      getObject('db')->sanitizeData( $_POST['comment_name'] ) );
  $insert['authorEmail'] = $this->registry->getObject('db')->
      sanitizeData( $_POST['comment_email'] );
  $insert['comment'] = strip_tags( $this->registry->getObject('db')->
      sanitizeData( $_POST['comment'] ) );
  $insert['IPAddress'] = $_SERVER['REMOTE_ADDR'];
  $valid = true;
  if( $_POST['comment_name'] == '' ||
      $_POST['comment_email'] == '' || $_POST['comment'] == '' )
  {
    $valid = false;
  }
```

If enough information was provided, we insert the review into the database.

```
if( $valid == true )
{
  $this->registry->getObject('db')->
      insertRecords( 'content_comments', $insert );
  $this->registry->getObject('template')->getPage()->
      addTag('message_heading', 'Review added');
  $this->registry->getObject('template')->getPage()->
      addTag('message_heading', 'Your review has been added');
  $this->registry->getObject('template')->
      buildFromTemplates('header.tpl.php',
                         'message.tpl.php',
                         'footer.tpl.php');
}
```

Otherwise, we display an error to the customer.

```
else
{
  $this->registry->getObject('template')->getPage()->
      addTag('message_heading', 'Error');
  $this->registry->getObject('template')->getPage()->
      addTag('message_heading', 'Either your name, email address or
              review was empty, please try again');
  $this->registry->getObject('template')->
      buildFromTemplates('header.tpl.php',
                         'message.tpl.php',
                         'footer.tpl.php');
}
}
```

Displaying reviews/comments

What do we need to do to display reviews and comments?

1. Check to see if there are any comments.
2. Display either the comments or a "no comments" template.
3. Check to see if new comments are allowed.
4. Display either the comments form or a "no comments allowed" template.

Displaying and allowing comments is a very generic feature, which we could either add into our product's controller or in a controller that all of our controllers inherit from, ensuring all content types have support for it. I'll leave the choice down to you; however, it would be useful if you chose to use inheritance in order to override it within the products, so the templates replace the word comments with reviews, and perhaps have a larger comments box.

Combining the two?

e-commerce sites such as Amazon combine these two features: allowing customers to select a rating when they leave a review and have this rating display alongside their review. What would be involved in combining the two features?

1. Associate ratings with a review or account for rating reviews when working out the average rating. We could either extend the ratings table to include a potential reference to a review, which the review can pull in, or we can store the ratings along with the reviews and have the product's average rating calculated by combining both standard ratings, and ratings that were left with a review.
2. Save a rating at the same time as processing a review.
3. If a review has a rating associated with it, then it should be displayed.



Try it yourself

Once you have decided which of the two methods you think is best for your particular framework, why not try combining the two features yourself, should this be a useful feature for your framework?

Any other experience improvements to consider?

A fantastic user experience for our customers by no means ends here; there are many other ways in which we could improve our customers' experience. Let's discuss a few of them:

- **Suggest a product**: We could allow customers to suggest new products for our store.
- **Suggest a related product**: We could allow customers to suggest a product that is related to another product.
- **Report inaccurate information**: We could allow customers to report inaccurate product information.
- **Report an inappropriate comment**: We could allow customers to bring to our attention inappropriate comments left on product pages.

- **Pre-orders**: If we looked into registering customers' interest when a product was out of stock, we could extend and improve this to indicate that a product is not in stock because it is currently only available for pre-order.
- "**Feedback about this page**": A simple link taking customers to a contact form to leave feedback about a specific page could be useful, as it could allow them to inform us that a page is too thin on details, the information is out of date, or that a link or image is broken and needs to be fixed.

Summary

In this chapter we have discussed the advantages of improving the user experience for our customers, and we have worked to improve their experience by:

- Introducing product search and filtering options
- Recommending relevant products to our customers
- Giving our customers wish lists
- Informing our customers when products they are interested in are back in stock
- Allowing customers to rate and review products

We have also discussed how we could extend and enhance these user experience improvements, which we have implemented, including:

- How to improve our search feature
- Maintaining multiple wish lists
- Maintaining public and private wish lists
- Combining the ratings and reviews feature

Now that the user experience is improved, and we have some ideas on how to further improve it, we can move onto the shopping basket, bringing us one step closer to being able to trade online using our e-commerce framework.

6
The Shopping Basket

The first major step in effectively selling products online is the shopping basket, as this directly leads into the checkout process. In this chapter, you will learn:

- How to structure and create a shopping basket
- How to manage the contents of the shopping basket
- How to deal with a visitor signing up, and transferring their basket to their user account

The shopping basket should be a relatively easy process, as its function is to store a collection of products, which customers are intending to purchase, and relate them to the relevant customer.

Shopping baskets

Shopping baskets are a very important aspect in e-commerce websites, and in most websites, they are the first stage in enabling an online purchase. However, there are a number of other methods that can facilitate e-commerce, including:

- **One-click payments**: An example of these would be a PayPal payment button on a product view, or Amazon's one-click ordering. One-click payments, such as PayPal's payment button, take all of their data from the payment processor, and are generally used on websites with a small selection of products. The customer clicks on the button on the product page, their payment is processed, and the payment processor notifies the administrator of the product, the customer, delivery details, and the amount paid—the product and cost data is defined within the payment button. One-click ordering makes things easier for the customer, reducing the need to go through an entire payment process; however, it also has disadvantages—customers can easily order things by mistake, customers need to be sure their default delivery details are correct in advance, it isn't easy to add voucher codes without adding more clicks, and discounts for multiple purchases or bundled shipping can't be taken into account.

- **Service subscription payments**: These are generally similar to one-click payments; you click on the subscription level, and then you pay. Most subscription services also make it easy to upgrade or downgrade accounts at any point, resulting in the customer being charged pro-rata based on how long their account was at each subscription level. Subscription sites give access to products or services for the duration of a subscription, from a business perspective. This often allows measurable recurring income, and can reduce transaction fees. For example, on a music download website, the customer may wish to make 25 purchases a month, and each purchase would incur a transaction fee. With a subscription payment method, the transaction fees apply to less regular, larger payments, which tend to work out less overall.
- **Auctions**: Auctions lead to bidding for products. This involves the customer committing to purchase the product at a certain price, provided no other customer commits to a higher price within the auction time window. Often, auction sites are automated, so the customer enters a maximum price, and over the duration of the auction, the website will increase the customer's bid, with respect to their maximum bid, as and when other bids come in.

For our framework, a shopping basket is going to be the most appropriate option; however, this is a framework. After all, there is nothing stopping us from implementing other methods of facilitating purchases.

Our basket

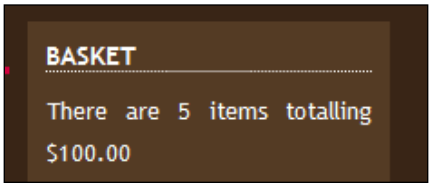
At the end of this chapter, we will have a "shopping basket" page and a smaller shopping basket displayed on each page.

Per-page basket

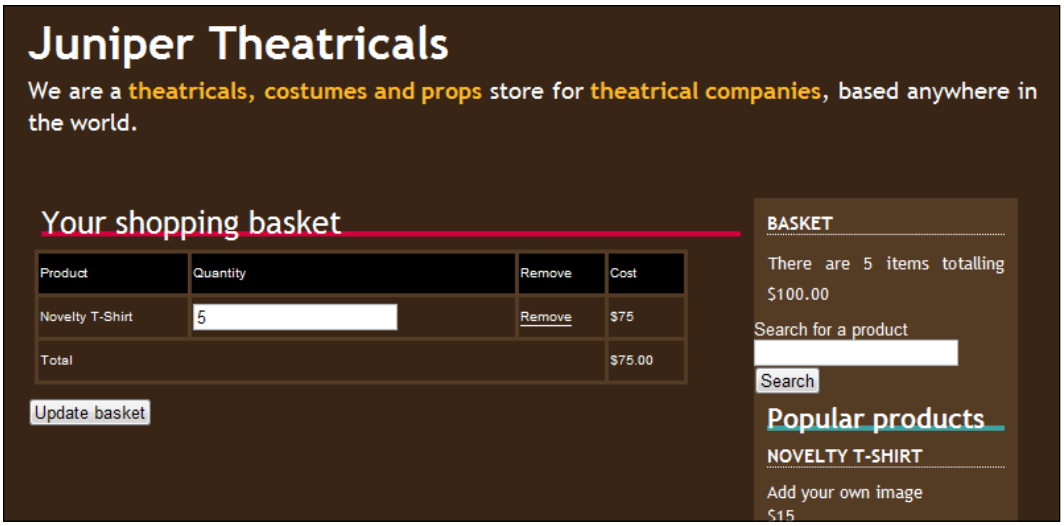
If there is nothing in the customer's basket, they will be shown an empty basket message, like the following, on each page:



If the customer has products in their basket, the small basket message on each page reflects something similar to this:



And, our main basket page should look like this:



Considerations for our shopping basket

We discussed a number of considerations we need to take into account within our shopping basket in Chapter 4, *Product Variations and User Uploads*. Let's recap on those, and discuss how we shall implement those suggestions.

- **Stock levels**: In some cases, we need to determine if there is sufficient stock when a customer adds a product to their basket. When we add a product to the basket, we can simply query the products database, and ensure the level in stock is either more than the quantity the customer wishes to purchase, or is not relevant for that product (for example digital delivery products, services, and so on).

- **Product variations**: When a customer adds a product variation to their basket, we need to record the variation that it is. The way products are maintained in an array for the basket must differentiate products and product variants, to ensure that when a customer adds a second instance of the product, the variant choices are preserved, allowing the customer to purchase any number of any different variations of a product.
- **Product customizations**: If the customer customizes a product, we need to record any customization data they have left, as with the product variations.
- **Templates**: We need a number of different templates to show the basket: an empty basket, a summary of the basket on every page, and so on.
- **Subtotals**: We need to calculate subtotals for each product in the basket.

Creating a basket

Let us create our shopping basket in stages, starting with basic functionality, and then extending it as we go along to support all of the considerations we have discussed.

When to build a user's basket

Our shopping basket will probably be stored on most pages, so we need to ensure that we can build up the contents of a shopping basket regardless of where a user is within the site being powered by our framework. Obviously, we may not always want to have this available, but more often than not, we would. Another consideration is with regards to user authentication: if the visitor is a logged-in user, then the basket will be built in a different way, so we need to ensure we build the basket after any authentication processing is done.

Basket database

We looked at creating a wish list in Chapter 5, *Enhancing the User Experience*; although the wish list was only suitable for standard products, which couldn't be customized and didn't have variants. The shopping basket needs to work in a similar way to this. A single database table is required that relates these products to the customers. Let's take the data from our wish-list table, and extend it to make it more suitable for a shopping basket.

Field	Type	Description
ID	Integer (Auto increment, Primary Key)	
Session_id	Varchar	To relate products in the basket to customers who are not logged into the site.
User_id	Integer	To relate products in the basket to customers who are logged into the site.
Quantity	Integer	The quantity of the particular product that the customer wishes to purchase.
IPAddress	Varchar	This is also needed to help relate the products in the basket to users who are not logged in.
Timestamp	Timestamp	To maintain active and expired contents. This is primarily for customers who are not logged in; after a certain period of time, we need to assume that the session ID and IP address now belong to another customer.

This database is represented with the following SQL code:

```
CREATE TABLE `book4database`.`basket_contents` (
`ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
`session_id` VARCHAR( 50 ) NOT NULL,
`user_id` INT NOT NULL ,
`product_id` INT NOT NULL,
`quantity` INT NOT NULL,
`ip_address` VARCHAR( 50 ) NOT NULL,
`timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
) ENGINE = MYISAM;
```

Basket contents

We have a basket structure in our database, but we now need to allow our customers to control the contents of the basket. This involves allowing them to:

- View the basket
- Add products to their basket
- Add customizable products to their basket
- Add variations of a product to their basket
- Edit quantities of products in their basket

Let's go through these requirements, and implement them into our framework.

Viewing the basket

This may seem strange, discussing viewing the basket before our customers are even able to add products to the basket. However, the reason for this is that we need functionality in place for checking the basket contents before we can add a product to the basket. If a customer adds a product to their basket twice, the second instance should increment the quantity of the first instance of the product, which means we need to be able to determine if a product exists in the basket.

The following stages are involved in checking our shopping basket:

1. Select relevant data from the basket table in the database.
2. Build an array of data representing the contents of the basket.
3. Set some variables, including basket cost, if the basket is empty, and that we have checked the baskets contents.

checkBasket method

We need a function that performs all of the aspects of checking the basket, so that we can display the basket to our customers.

```
/**
  * Checks for the users basket contents
  * @return void
  */
```

The first stage is to generate a number of variables to use later. We indicate that the basket has been checked, which prevents the framework from unnecessarily calling this method again once it has already been called, and the data processed.

We also need the user's session ID and IP address, primarily for customers who are not logged in, and if the user is logged in, we need to get their username.

```
public function checkBasket()
{
  // set out basket checked variable - this is to prevent this
  // function being called unnecessarily
  // if we run this on page load to generate a mini-basket, we
  // don't need to reload it to display the main basket!
  $this->basketChecked = true;
  // get user identifiable data
  $session_id = session_id();
  $ip_address = $_SERVER ['REMOTE_ADDR'];
  // if the customer is logged in, our query is different
  if( $registry->getObject('authentication')->
                                          isLoggedIn() == true )
```

```
{
  // they are logged in, get their ID
  $u = $this->registry->getObject('authentication')->getUserID();
```

The `checkBasket()` function runs one of two queries to lookup products in the basket: one if the customer is logged in, which checks for products based off the customer's ID, and another if they are not logged in, which uses the IP address and session ID to determine which products are from the current customer's basket.

```
  $sql = "SELECT b.ID AS basket_id,
                 b.quantity AS product_quantity,
                 c.ID AS product_id,
                 v.name AS product_name,
                 p.stock AS product_stock,
                 p.weight AS product_weight,
                 p.price AS product_price,
                 p.SKU AS product_sku
          FROM content_versions v, content c, content_types t,
               content_types_products p, basket_contents b
          WHERE c.active=1 AND c.secure=0 AND c.type=t.ID
                AND t.reference='product'
                AND p.content_version=v.ID
                AND v.ID=c.current_revision
                AND c.ID=b.product_id AND b.user_id={$u}";
}
else
{
  $sql = "SELECT b.ID AS basket_id,
                 b.quantity AS product_quantity,
                 c.ID AS product_id,
                 v.name AS product_name,
                 p.stock AS product_stock,
                 p.weight AS product_weight,
                 p.price AS product_price,
                 p.SKU AS product_sku,
          FROM content_versions v, content c, content_types t,
               content_types_products p, basket_contents b
          WHERE c.active=1 AND c.secure=0 AND c.type=t.ID
                AND t.reference='product'
                AND p.content_version=v.ID
                AND v.ID=c.current_revision AND c.ID=b.product_id
                AND b.user_id=0 AND b.session_id='{$session_id}'
                AND b.ip_address='{$ip_address}'";
}
// do the query
$this->registry->getObject('db')->executeQuery( $sql );
```

If the query yielded results, we indicate that the basket isn't empty, and then iterate through the results, building our basket. We will need to modify this later to handle customizable products.

```
    if( $this->registry->getObject('db')->numRows() > 0 )
    {
      // we have some products in our basket
      // set the relevant variable
      $this->basketEmpty = false;
      while( $contents = $this->registry->getObject('db')->getRows() )
      {
        // for each product, add them to the basket object
        $this->contents[ 'standard-' . $contents['product_id'] ] =
            array( 'unitcost' => $contents['product_price'],
                   'subtotal' => ($contents['product_price']
                       * $contents['product_quantity']),
                   'weight' => $contents['product_weight'],
                   'quantity' => $contents['product_quantity'],
                   'product' => $contents['product_id'],
                   'basket' => $contents['basket_id'],
                   'name' => $contents['product_name'] );
        $this->numProducts = $this->numItems + $contents['quantity'];
        $this->cost = $this->cost
            + ( $contents['price'] * $contents['quantity'] );
      }
    }
  }
```

The controller

At this stage in the chapter, our controller needs to:

- Detect if the customer is trying to view the basket
- Get the basket contents from the model
- Cache basket contents and associate them with a template tag

The following code does this, as described within:

```
public function viewBasket()
{
  // Build the view from the appropriate template
  $this->registry->getObject('template')->
    buildFromTemplates('header.tpl.php', 'viewbasket.tpl.php',
    'footer.tpl.php');
```

Get the contents of the basket from the basket model, and build it into an array suitable for caching and sending to the template engine.

```
    $contents = $this->basket->getContents();
    $products = array();
    foreach( $contents as $reference => $data )
    {
      $data['basket_id'] = $data['basket'];
      $data['basket'] = '';
      $products[] = $data;
    }
    // Send the basket data to the template engine, and assign other
    // information such as total cost and number of products to
    // template variables.
    $basketCache = $this->registry->getObject('db')->c
        acheData( $products );
    $this->registry->getObject('template')->getPage()->
        addTag( 'products', array( 'DATA', $basketCache ) );
    $this->registry->getObject('template')->getPage()->
        addTag( 'basket_total', $this->basket->getTotal() );
}
```

Adding products

Adding a standard product to the basket should be straightforward; we only need to record the product's ID number and information of the user (such as their user ID or if they are not a logged-in user, some session data).

As our basket will be maintained as an array, with product IDs acting as the keys, and other data such as price, name, quantity, and so on, acting as the elements, we should prefix the ID to ensure we don't have problems with adding product variants or customizable products to our basket later on. For example, if a product can have an image uploaded, and we add two of these products with different images, we don't want the basket treating this as one product, but with a quantity of two. However, with a product that can't be customized, this is exactly what we do want!

Let's start with the functionality for adding a standard product to our shopping basket.

An addProduct method

When a customer clicks on the **Add to basket** button on our store, what do we need to do?

1. First, we need to check that the product is valid and active, after all we can't add a product that does not exist to a user's shopping basket.

2. Secondly, we need to check that it is not a customizable product, or product that has variations (as this function will just deal with standard products for the time being).
3. Finally, we check to see if the product is already in the basket—if it is, then we increment the product's quantity in the basket; if it isn't, then we add it to the basket.

The following code represents a suitable `addProduct` method. Although quantity is a parameter for the method, in most instances, we would want this to default to one, as when a customer clicks on **Add to basket** they would only want one added to the basket.

```
/**
  * Add product to the basket
  * @param String productPath the product reference
  * @param int $quantity the quantity of the product
  * @return String a message for the controller
  */
public function addProduct( $productPath, $quantity=1 )
{
 // have we run the checkBasket method yet?
 if( ! $this->basketChecked == true ) { $this->checkBasket(); }
 // check the product exists
 $productQuery = "SELECT v.name AS product_name,
                         c.ID AS product_id,
                         p.allow_upload AS allow_upload,
                         p.stock AS product_stock,
                         p.weight AS product_weight,
                         p.price AS product_price,
                         p.SKU AS product_sku,
                         p.featured AS product_featured,
                         v.heading AS product_heading,
                         v.content AS product_description,
                         v.metakeywords AS metakeywords,
                         v.metarobots AS metarobots,
                         v.metadescription AS metadescription
                  FROM content_versions v, content c,
                       content_types t, content_types_products p
                  WHERE c.active=1 AND c.secure=0 AND c.type=t.ID
                        AND t.reference='product'
                        AND p.content_version=v.ID
                        AND v.ID=c.current_revision
                        AND c.path='{$productPath}'";
 $this->registry->getObject('db')->executeQuery( $productQuery );
 if( $this->registry->getObject('db')->numRows() == 1 )
 {
  // get the ID, etc
  $data = $this->registry->getObject('db')->getRows();
```

```
// check if it already in the basket
if( array_key_exists( 'standard-' . $data['product_id'],
    $this->contents ) == true )
{
 // check stock
 if( $data['product_stock'] == -1 ||
     $data['product_stock'] >= ( $this->contents['standard-'
                 . $data['product_id']]['quantity'] + $quantity ) )
 {
  // increment the quantity
  $this->contents['standard-'
   . $data['product_id']]['quantity'] = $this->contents['standard-'
   . $data['product_id']]['quantity']+$quantity;
  // update the database
  $this->registry->getObject('db')->
   updateRecords('basket_contents',
                 array('quantity'=> $this->contents[ 'standard-'
                 . $product]['quantity'] ),
   'ID = ' . $this->contents['standard-'.$product]['basket'] );
   return 'success';
 }
 else
 {
  // error message
  return 'stock';
 }
}
else
{
 if( $data['product_stock'] == -1 ||
                             $data['product_stock'] >= $quantity )
 {
  // add product
  // insert the new listing into the basket
  $s = session_id();
  $u = ( $this->registry->getObject('authentication')->
         isLoggedIn() ) ? $this->registry->
       getObject('authentication')->getUserID() : 0;
  $ip = $_SERVER['REMOTE_ADDR'];
  $item = array( 'session_id' => $s,
    'user_id' => $u,
    'product_id' => $data['product_id'],
    'quantity' => $quantity,
    'ip_address' => $ip );
  $this->registry->getObject('db')->
                          insertRecords( 'basket_contents', $item );
  $bid = $this->registry->getObject('db')->lastInsertID();
  // add the product to the contents array
  $this->contents['standard-' . $data['product_id'] ]
                     = array( 'unitcost' => $data['product_price'],
```

```
            'subtotal' => ($data['product_price'] * $quantity ),
            'weight' => $data['product_weight'],
            'quantity' => $quantity,
            'product' => $data['product_id'],
            'basket' => $bid,
            'name' => $data['product_name'] );
          // return that all was successful
          return 'success';
        }
        else
        {
          // error message
          return 'stock';
        }
      }
    }
    else
    {
      // product does not exist: Error message
      return 'noproduct';
    }
  }
```

The controller

In addition to the features discussed earlier, at this stage in the chapter, our controller needs to also:

- Detect if the customer is trying to add a product to the basket, that is, by clicking on an **Add to basket** button, and then processing this accordingly. Without this, products would never be added to a customer's basket.
- Pass data to the model to add a product to the basket. We need to tell the basket model which product the customer wants to add to their basket, so that it knows which product to associate with the current customer in the basket table.
- Display an error message if the product is not found, so the customer realizes the product has not been added to their basket and are not confused when they see their basket is missing the product.
- Display an error message if the product is out of stock, so the customer realizes the product has not been added to their basket, and are not confused when they see their basket is missing the product.

- If the product was found, and was in stock, it needs to display a confirmation message to the customer, so they know the product has been added to their basket. It should perhaps also redirect the customer to another page once the product was added successfully, such as the basket page, or the products page.

Although this is quite a lot to do, the code for it is surprisingly simple: first we have to extend the switch statement in the controller's constructor, so it can detect if the customer is visiting the basket page.

```
case 'add-product':
  echo $this->addProduct( $urlBits[2], 1);
  break;
```

Then we have our `addProduct` function, which is called when the customer tries to add a product to the basket. This function tries to add a product to the basket model, and depending on the response displays appropriate messages to the customer.

```
/**
  * Add product to the basket
  * @param String productPath the product reference
  * @param int $quantity the quantity of the product
  * @return String a message for the controller
  */
```

We check to see if the basket has already been checked, to save us from doing it again. We need to check the basket to determine if we are incrementing the quantity of a product in the basket, or adding a new product to the basket—so we already need to have our basket populated.

```
public function addProduct( $productPath, $quantity=1 )
{
  // have we run the checkBasket method yet?
  if( ! $this->basket->isChecked == true )
  { $this->basket->checkBasket(); }
  // We then call the addProduct method in the basket model and
  // make a note of the response it returns.
  $response = $this->basket->addProduct( $productPath, $quantity );
```

Depending on the response, the appropriate message is displayed to the customer.

```
  if( $response == 'success' )
  {
    $this->registry->redirectUser('products/view/'
        . $productPath, 'Product added',
        'The product has been added to your basket', false );
  }
  elseif( $response == 'stock' )
  {
```

```
            $this->registry->getObject('template')->
                buildFromTemplates('header.tpl.php', 'message.tpl.php',
                'footer.tpl.php');
            $this->registry->getObject('template')->getPage()->
                addTag('header', 'Out of stock' );
            $this->registry->getObject('template')->getPage()->
                addTag('message', 'Sorry, that product is out of stock,
                and could not be added to your basket.' );
        }
        elseif( $response == 'noproduct' )
        {
            $this->registry->getObject('template')->
                buildFromTemplates('header.tpl.php',
                'message.tpl.php','footer.tpl.php');
            $this->registry->getObject('template')->getPage()->
                addTag('header', 'Product not found' );
            $this->registry->getObject('template')->getPage()->
                addTag('message', 'Sorry, that product was not found.' );
        }
    }
```

A note on etiquette

When a customer adds products to their basket, it is important that only what they add to their basket is added. From personal experience, I know of a number of sites that try to auto-select other products, which then get added to your basket. It can be quite easy for a customer to miss this, and inadvertently purchase products they don't want. This will leave the customer unhappy, generate negative reviews about the site, and damage the store's reputation.

Adding customizable products

Earlier in this book, we discussed customizable products, which allowed customers to both upload files to associate with their product order, and to enter text related to a number of fields associated with their product order.

To facilitate the purchase of such products, there are a number of things we need to do:

- Restructure the basket database
- Change how we view the shopping basket
- Change how we add a product to the basket
- Process the customized information, if the product was customizable

Changing our basket database

In our products table, we have two fields of interest, `allow_upload` and `custom_text_inputs`. In our basket table, we could benefit from adding a field to store the uploaded file and a field to store the values from the custom text inputs.

```
ALTER TABLE `basket_contents`
ADD `uploaded_file` VARCHAR( 255 ) NOT NULL,
ADD `custom_text_values` LONGTEXT NOT NULL
```

Viewing the basket

When viewing the basket, we *may* wish to display different information relating to the fact that a product has been customized. This involves changing our `checkBasket` method in the model, and also making some changes to the controller.

The changes to the model need to detect if the basket entry has a file uploaded or a number of custom text submissions; if it does, then instead of adding it to the basket array with a prefix of `standard`, we use a prefix of `customized`. The reason for this prefix is to ensure when the customer clicks on **Add to basket** a second time, we don't duplicate an existing product in the basket, but instead take into account the customizations that the customer made.

Changing the model

Changes required to our model are:

- Check to see if the product is customizable; if the product is customizable, then we need to process the request differently.
- Check to see if a file is being uploaded; if the product is customizable, and customers are permitted to upload files with their purchase (such as an image or photograph), the model needs to check if the customer has chosen to upload a file, so that it can process it.
- If the customer has chosen to upload a file, it needs to be uploaded to the server, and moved to a suitable location, such as the `basket_uploads` directory. The name and location of this file then needs to be stored in the `basket_contents` table.

- Depending on the nature of the store, and the type of files we allow the customer to upload, we may also wish to store whether or not the file is an image. If the file is an image, we could display a thumbnail of this image on the customer's "basket view" page, so they can see the image they uploaded along with their product. If the product allows the customer to complete any custom text fields, we need to process these, and serialize the data, and store them in the database. The data needs to be serialized, so that the data from any number of fields (each product can store any variety of custom inputs) can be stored with their corresponding field names, within one database field.
- If the product is customizable, the product should be stored in the basket contents array with a prefix of `customizable-`. This way the standard and customizable products can be handled differently as required.
- In addition to the prefix and the product ID, we also need to store a unique reference, to indicate that the product is customized by means of the variations selected. This combined with the prefix allows us to list each customization separately within the basket view. After all, the customer could purchase three t-shirts, but each of them of different sizes or with custom images uploaded with them. If this is the case, they should appear as three separate products listed in the basket. If they are all the same version, then they would only appear once, with a quantity of three.

The controller

In addition to the features discussed earlier, at this stage in the chapter, our controller now also needs to:

- Display custom text inputs when viewing basket contents, so that the customer can see which product in their basket has been customized in which way, enabling them to change quantities and remove specific customizations from their basket should they wish to.
- Display an uploaded image, if we wish, for the same reason as the previous one, but this would also make the basket more appealing to the customer.
- Display a link to download the file that has been uploaded by the customer allowing them to verify that they had in fact uploaded the correct file for their order, before proceeding.

Adding product variants

When it comes to adding products that have variations to our shopping basket, there are again a number of changes that need to be made to our current database and codebase.

A new database table

A table like the one we created in Chapter 4, *Product Variations and User Uploads*, to relate products to potential variations, is needed to relate chosen product variants with products in a customer's shopping basket. The exception is that we don't need to retain the order field or the cost difference.

Field	Type	Description
Basket_id	Integer	A reference to the product in a customers basket
Attribute_id	Integer	A reference to a particular attribute for that product, indicating how that instance of the product in the basket should be customized or which variant of the product it is.

The following code represents the new table:

```
CREATE TABLE IF NOT EXISTS `basket_attribute_value_association` (
  `basket_id` int(11) NOT NULL,
  `attribute_id` int(11) NOT NULL,
  KEY `basket_id` (`basket_id`,`attribute_id`),
  KEY `attribute_id` (`attribute_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
COMMENT='Association of basket contents and attribute values';
```

Model changes

As far as our model is concerned, we need to:

- Store products that have variations, which have been added to the customer basket, with an array key prefix of `variation-`. This allows the basket to process these products differently when displaying them in the basket, and also when it comes to the checkout process.
- In addition to the prefix and the product ID, we also need to store a unique reference, to indicate that the product is customized by means of the variations selected. This means if a customer has purchased two variations of one product, they are listed as two products in their basket. Their reference numbers allow the customer to update quantities and remove them. If the reference was not unique, removing one product variant from the basket would remove the other, which wouldn't be what the customer wanted.
- Check if products in the basket have variations associated with them so that the details of the variant can be displayed.
- Store references to the variations within the array for basket contents.

- When adding a product, check relevant POST data for variations that are being selected. The variants of a product (sizes, colors, and so on) are all displayed in drop-down lists on the product view. When the customer adds one of these products to the basket, it is done by a form submit button, which passes the values of these drop-downs as POST data, so we need to check for this data, to determine which variations and attributes need to be stored.

The controller

In addition to the features discussed earlier, at this stage in the chapter, our controller needs to also display attributes from variations when viewing the basket, so the customer is aware of exactly which products (or which variations of the products) they are purchasing, and can change quantities of/remove them from their basket.

Editing quantities

Once a product has been added to the basket, we want to make it easy for the customer to change the quantity. The hope here is obviously that they will increase the quantities and purchase more, but equally, we need to allow them to reduce quantities or remove products, until they are happy with the contents of their basket and want to proceed.

When viewing the basket, each entry in our basket contents array will be a new row in a table within the view, and within this row will be a field allowing us to change the quantity. This field should have a reference the same as the key it has in the basket contents array. This is to ensure that we update the correct customized product; after all, if we have two customized instances of a product, and we wish to purchase two copies of one of them, we only want to alter the quantity of that record in our model and our database.

The following code in our controller updates the basket based on new **quantities** supplied by the customer:

```
/**
  * Update the shopping basket
  */
  private function updateBasket()
  {
  // First, this populates the basket model, unless this has already
  // been done.
    if( ! $this->basket->isChecked == true )
    { $this->basket->checkBasket(); }
```

We then go through each entry in the basket table (that is, each unique product in the customer's basket) and where there is a quantity that the customer has submitted, this is updated; if the customer removes the quantity for a product, it is then removed from the basket.

```
    foreach( $this->basket->getContents() as $pid => $data )
    {
      // get the product rows basket ID
      $bid = $data['basket'];
      if( intval( $_POST['qty_' . $bid ] ) == 0 )
      {
        $this->basket->removeProduct( $bid );
      }
      else
      {
        $this->basket->updateProductQuantity( $bid,
            intval( $_POST['qty_' . $bid] ) );
      }
    }
    // save the extra processing by marking embedded as false
    $this->embedded = false;
    // We then redirect the user to the basket page, informing them
    // that their changes have been saved.
    $this->registry->redirectUser('basket', 'Basket updated',
        'Your shopping basket has been updated', false );
  }
```

If the quantities are zero, or if the customer clicks on a **remove product** link, we need a `removeProduct` function to remove the product from the basket.

```
  public function removeProduct( $bid )
  {
    $this->basket->removeProduct( $bid );
    $this->registry->redirectUser('basket' , 'Product removed',
        'The product has been removed from your basket', false );
  }
```

Finally, the constructor needs some additional cases in its switch statement to call the appropriate functions depending on the page the customer is visiting, be it an "update basket" page, or the "remove product from basket" page.

```
  switch( $urlBits[1] )
  {
    case 'view':
      $this->viewBasket();
      break;
    case 'add-product':
      echo $this->addProduct( $urlBits[2], 1);
```

```
    break;
  case 'update':
    $this->updateBasket();
    break;
  case 'remove-product':
    $this->removeProduct( intval(  $urlBits[2] ) );
    break;
  default:
    $this->viewBasket();
    break;
}
```

Both the update and the remove product functions rely on methods in the basket model; these methods are very simple. To change the quantity of a product, we simply update the basket table with the new quantity.

```
public function updateProductQuantity( $basketItemId, $quantity )
{
  $s = session_id();
  $u = ( $this->registry->getObject('authenticate')->
         isLoggedIn() ) ? $this->registry->
                          getObject('authenticate')->getUserID() : 0;
  $ip = $_SERVER['REMOTE_ADDR'];
  $changes = array( 'quantity' => $quantity );
  $this->registry->getObject('db')->
      updateRecords( 'basket_contents',
                     $changes, " session_id='{$s}'
                     AND user_id={$u} AND ID={$basketItemId}
                     AND ip_address='{$ip}' ");
}
```

To remove a product, we simply remove an entry from the basket table, which corresponds to that particular instance of a product in the customer's basket.

```
public function removeProduct( $basketItemId )
{
  $s = session_id();
  $u = ( $this->registry->getObject('authenticate')->isLoggedIn() )
     ? $this->registry->getObject('authenticate')->getUserID() : 0;
  $ip = $_SERVER['REMOTE_ADDR'];
  $this->registry->getObject('db')->
     deleteRecords( 'basket_contents', " session_id='{$s}'
     AND user_id={$u} AND ID={$basketItemId}
     AND ip_address='{$ip}' ", 1 );
}
```

From visitor to a user

When a visitor comes to our site, adds products to their basket, and then signs up, we need to transfer the basket to them, as a user. This would allow them to access this basket on other computers too.

The transferToUser function

A function within our basket model would make this easy; all it needs to do is find any products in the basket table, which are associated with the user's session data and update these records to have the user's ID contained.

```
/**
  * Transfer the basket to another user
  * @param int user id
  * @return bool
  */
  public function transferToUser( $user )
  {
    $changes = array( 'user_id' => $user );
    $s = session_id();
    $ip = $_SERVER ['REMOTE_ADDR'];
    $this->registry->getObject('db')->
        updateRecords( 'basket_contents', $changes,
                       " SESSION_ID='{$s}' AND ip='{$ip}' ");
    return true;
  }
```

One thing you may notice is that we are not filtering this based on recent entries. The reason for that is, out-of-date entries in the basket table should be pruned automatically, which we will cover in the next section.

Performing the transfer

The `transferToUser( $user )` method should be called when processing a user login. However, we would need to use the `directCall` parameter in the controller to ensure that the controller does not try to display a basket-related page when the customer is not trying to view such a page.

Cleaning the basket

Shopping baskets need to be emptied, but this should be done only when:

- The customer wants to empty their basket
- The customer confirms an order
- The basket contents are old and are not tied to a customer account

Expired contents

We could create a function to empty a user's basket, upon their request. However, this can't be used for the instance of expired contents, as it isn't tied to a specific user; we need to purge any data in the basket table that has expired. We need to do this, because after a certain period of time, the customer won't be able to see the products in their basket anyway, as they may have a new IP address from their Internet Service Provider, or they may have initiated a new session, and so their session ID will not be the same. Essentially, these are orphaned products in the basket—and need to be removed to keep the database up to date, and free of redundant data.

Displaying the basket on every page

Most e-commerce websites display a small shopping basket on each page, often at the top of the page or the side of the page, reminding the customer how many products are in their basket, the cost of the contents of their basket, and providing a link for them to view their basket in detail and checkout their order.

To display the basket on each page we will need:

- An empty basket template file, so that if the customer has no products in the basket, the page accurately reflects that with a suitable message.
- A basket template file, which displays the number of products in the customer's basket, the cost of the order, and a link allowing the customer to proceed to the checkout process.
- A template tag in our main templates, where the basket can be inserted; when the basket is checked, the appropriate basket or empty basket will be inserted where this template tag is.
- Something in our framework to link into the basket on each page, not only when the basket controller is called directly. Without this, the customer would only see the basket if they visited the basket page.

Functionality

Within our framework, we could link the basket to each page, by calling the controller from our `index.php` file, but without using the `directCall` parameter; this would prevent the framework from trying to process the URL and running pre-defined parts of the controller. Instead, we simply call the `checkBasket` method, import the relevant template, and use the number of products and total cost variables from the model as template tags, and we have a small basket on the page.

The following is the code within our `index.php` file; it simply creates a `Basketcontroller` object, and then calls the `smallBasket` method of that object.

```
// basket
require_once('controllers/basket/controller.php');
$basket = new Basketcontroller( $registry, false );
$basket->smallBasket();
```

Within our `Basketcontroller`, we need to add the `smallBasket` method to insert the appropriate data and templates into the view.

```
/**
  * Small basket - prepare small embedded basket
  * @return void
  */
```

The first thing this needs to do is to check the basket in order to get the number of products and the cost of the baskets contents. As discussed earlier, when we do check the basket, we mark a variable to indicate that, to prevent us from unnecessarily doing this twice. We check to see if we haven't already checked the basket, and if we haven't, we then call the `checkBasket` function, to prepare the information we need.

```
public function smallBasket()
{
  if( $this->basket->isChecked() == false )
  { $this->basket->checkBasket(); }
  // set our embedded property
  $this->embedded = true;
```

If the basket isn't empty then we insert the basket template into the view, and set the appropriate values.

```
        // check that the basket is not empty
        if( $this->basket->isEmpty() == false )
        {
          // basket isn't empty so use the basket template,
          // and set the numBasketItems and basketCost template variables
          $this->registry->getObject('template')->
            addTemplateBit('basket',  'basket.tpl.php');
          $this->registry->getObject('template')->getPage()->
            addPPTag('numBasketItems', $this->basket->getNumProducts() );
          $this->registry->getObject('template')->getPage()->
            addPPTag('basketCost', $this->basket->getTotal());
          $this->registry->getObject('template')->getPage()->
            addPPTag('shippingCost', $this->basket->getShippingCost());
        }
```

If the basket is empty, then we insert the empty basket template.

```
        else
        {
          // basket is empty - so use the empty basket template
          $this->registry->getObject('template')->
            addTemplateBit('basket', 'basket-empty.tpl.php');
        }
      }
```

Summary

In this chapter, we have created our shopping basket, facilitating the first phase of allowing customers to make a purchase with our framework. We started with support for standard products, and extended it to work with products that customers can customize and products that have variations.

Now we can move on to the checkout and order process, as the shopping basket we have developed in this chapter is essentially the first stage in the order process, and with that in place, we can extend this to allowing customers to place an order and for it to be fulfilled.

7
The Checkout and Order Process

Our e-commerce framework is really starting to take shape now, and as we have made steps into facilitating online purchasing, things are starting to get very interesting. The next stage for us is the checkout and order process, which generally has quite a lot of features or requirements that we must take into account. We need to look through what is required of this process, look at how often it is structured, and decide how we should structure our order process.

In this chapter, you will learn:

- About the different processes involved in the checkout and order process
- How a number of other e-commerce websites, large and small, deal with their checkout and order process
- How we should structure our checkout and order process

Some examples

Let us start by reviewing some existing e-commerce websites from the perspective of their checkout and order process, to see how they go about it. We will look at the following stores:

- Amazon
- eBay
- Play.com

Amazon

Amazon is one of the most popular online stores available.



Let us take a look at the stages involved in its order process:

1. Select products to purchase.
2. Detailed basket displayed on side of each page.
3. Authentication:
    - Login
    - Register
    - Do nothing—already logged in
4. Select delivery address.
5. Select delivery method.
6. Select gift wrapping.
7. Enter payment details.
8. Enter voucher code.
9. Confirm details.
10. Process payment.
11. Order processed.
12. Order dispatched.
13. Happy customer.

Once the customer has selected the products they wish to purchase, they are displayed in great detail on the left-hand side of each page. When the customer clicks on the **checkout** button, they are either taken to the delivery-address page (if they are logged in) or they are taken to the login/sign-up page, where they can either log in to the site, or begin the first stage of the authentication process.

Once authenticated, the customer is taken to the delivery-method page. This page lists previously used delivery addresses, allowing the customer to either select, edit one of them, or enter a new delivery address for this order.

Next the customer selects their delivery method; this is generally a free method, a priority/first class method, or an express method. The free method is not automatically selected, so by default an additional charge is applied to the customer's order. Also, on this page is the option for the customer to select if they wish to have their order gift wrapped, useful if the order is being sent direct to the recipient of a present.

The next page allows the customer to enter their payment details, and the option to enter a voucher code.

Finally, the customer can review their order, confirm they wish to purchase, and have their credit or debit card charged for the relevant amount. Once the payment is processed, Amazon processes the order and dispatches it, resulting in a happy customer!

Limitations

Despite Amazon being one of the most popular e-commerce websites in the world, it has a number of limitations, and features that are not customer friendly, within this order process. These include:

- Moving backwards between stages in the order process is not possible without completely leaving the order process and returning to the shopping basket. This means if a customer wishes to make a change to something they have already submitted, they must go through the order process again, which can be off putting to the customer. Simply adding links so that the customer can quickly go back to a previous stage would solve this problem.
- The default shipping method adds a charge to the order, a customer who is not paying complete attention may inadvertently be charged for this when they actually may have wanted the free option. If this happens, the customer would likely be annoyed that they were charged for the shipping, and be left feeling disgruntled. If this were more obvious, this would be reduced, and more customers should be pleased with the service.

- The page where the customer enters their payment details is missing the price of the order, and although the page does mention that the customer will be provided the opportunity to review the order before being charged, this is not very clear. Despite payment not being taken at this stage, customers may be wary about waiting until the next stage to confirm the order, and may return to the start of the order process, potentially leading to more abandoned baskets.

Useful features

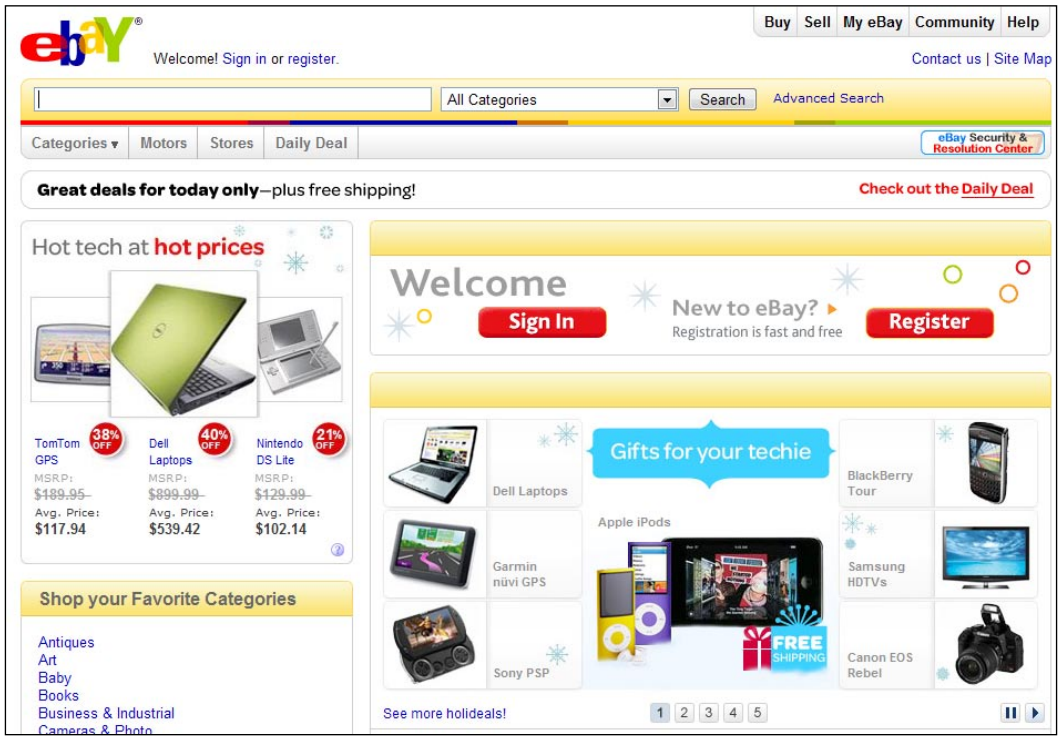
Of course, Amazon also has some excellent features; some particular points of note from the order process include:

- **Streamlined authentication**: The customer can both log in and start the registration process from the same page, without the need for going to a separate page for one or the other, making the process easier for the customer. If the authentication was just a login page, with a link to the registration page, this would introduce another step or two (at least one extra click), adding further barriers between the customer and a completed order. The fewer barriers in place, the higher conversion rate.
- **Detailed basket**: The shopping basket shown on each page is very detailed, this makes it very clear to the customer, what it is they are purchasing, for instance, so they can see in the basket, that they have not ordered the paperback copy instead of hardback copy of a book. The details it shows include:
    - Product names
    - Links to the products
    - Price
    - Quantity
    - Author
    - Type (for example Paperback)
    - Condition (for example new/used)
    - Subtotal
- **Gift wrapping**: The customer can choose to have their order gift wrapped, useful for sending it direct to a recipient when it is intended as a gift for them. This can make last-minute gift purchases online more appealing. Without this, if the customer needed a gift for someone quickly, they would need to order it, wait for delivery, wrap it themselves, and then take or deliver it to the recipient. With this, the customer can purchase gifts nearer to the occasion, where typically visiting a brick and mortar store would be the only option.

- **Flexibility to choose delivery address**: Multiple delivery addresses are saved, and the customer can easily switch between which they wish to use, or add a new one. This makes it really easy for customers to have orders sent to different addresses when they need to; for instance, a weekend order may get delivered to their home address, a weekday order to their office address—ensuring someone is there to collect the package.
- **Tracking of payment history**: On the payment page, the previously used card is recorded (with the last few digits being displayed for the customer's reference); however, there is also a form allowing the customer to enter a new card's details. This saves the customer needing to look for their card, and to prevent someone who may have hacked in from ordering things; changes to the delivery address require confirmation of the card (unless the address is saved).

eBay

As discussed earlier in this book, eBay is a very different type of e-commerce website; it is an auction website, whereby customers can either bid on items, or in some cases buy them for a set asking price.



The sellers are other eBay users, and not eBay itself. Because of the way eBay works, we can safely assume that the order process will be very different—and it is! It goes roughly like this:

1. The customer views an item of interest.
2. The customer indicates intention to purchase or bid on the item.
3. The customer either signs in or registers (*authentication*).
4. The customer confirms their intention to purchase (assuming *Buy It Now*, or successful winning of auction).
5. The customer can either make their purchase, or continue shopping.
6. When the customer is ready to pay, they first enter their delivery address.
7. Payment details are then entered, along with the ability for the customer to enter a voucher code.
8. The customer then reviews and confirms their order; this is generally for them to confirm the details are correct, as they are technically not permitted to back out of the purchase at this stage.
9. eBay processes payment.
10. The seller processes the order.
11. Happy customer.

The main difference to other sites is the confirmation of intent to purchase: the customer must confirm their intentions before supplying payment details; if they change their mind later, they can be penalized. At this stage the customer is shown the cost of the item. Once the customer has indicated their intent to purchase, they can go and make other purchases, if they wish to group their payment into a single transaction, or proceed with the single purchase.

Another difference is that eBay generally processes the payment (unless the customer indicated that they wished to pay through a check in the post to the seller), and then informs the seller of the outcome, so they can process the order. Generally, this is done through their own payment provider, PayPal.

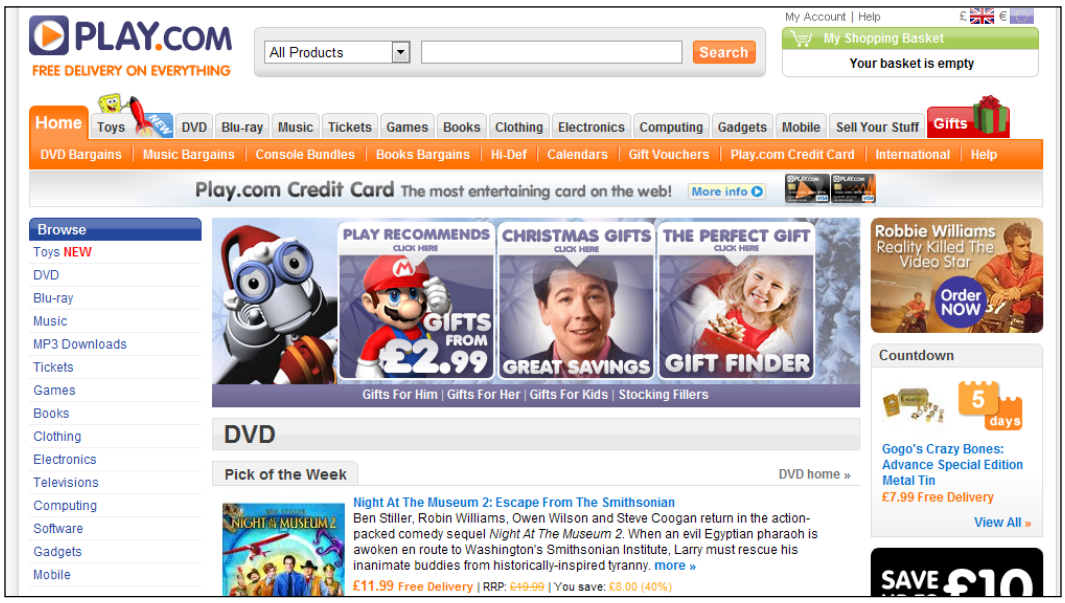
Interesting points of note

Aside from the points mentioned here, eBay also has some other interesting points worth noting.

PayPal's payment process is integrated within eBay; the customer can log in to PayPal, review their purchase, and then make the payment, from within the eBay site itself. While still using a third-party service (although technically not, as eBay owns PayPal), the process is streamlined, so the customer isn't confused by being bounced around various sites. Some customers can be confused by that, wondering why they are entering payment details into a site which isn't the one they placed an order with.

Play.com

Play is a very popular UK online retailer.



Let us discuss their order process, to see how they operate.

1. First the customer selects the products they wish to purchase.
2. A basic basket is displayed at the top of each page.
3. Authentication:
    - Login
    - Go to register page
    - Do nothing—already logged in
4. The customer selects a delivery address.
5. The customer then reviews and confirms the order.

6. The payment is processed.
7. The order is processed.
8. Happy customer.

This is a similar process to Amazon, although it is much shorter, and the customer is kept well informed throughout the process. Once the customer has added products to their basket, the checkout process then requires them to either log in or register. The registration process is done on a separate page (less streamlined than with Amazon), and then the customer can select their delivery address. Similar to Amazon, all delivery addresses are displayed, although one address is highlighted as the address that is associated with the relevant card the customer has on file. Once the delivery address is selected, the customer can then review and confirm their order (based on the card held on file). Once confirmed, the order is processed and dispatched, resulting in a happy customer.

Interesting points of note

There are a few interesting points worth noting from Play.com's order process, these are:

- Unlike many other e-commerce sites, including Amazon and eBay, the ability for the customer to enter a voucher code is not directly integrated with the order process—the customer must click on a button at the side of the page, which takes them to a separate page entirely, where they can enter their voucher code. From a customer's perspective, this takes additional steps, and can be easy for a customer to miss. This may reduce customer's confidence if they try and purchase something with a discount code, and end up paying full price for the order, because they didn't spot the discount code section.
- As mentioned earlier, one of the delivery addresses is the cardholder's billing address, and is highlighted as such.
- The basket contents are displayed at each stage of the process (except for authentication), even on the un-integrated voucher page; this ensures the customer is well informed at every stage of the process. While making the checkout process easy to move backwards and forwards allows the customer to confirm details, having the information on each page negates even this requirement, reducing more potential barriers between the customer and their order being completed.
- The order process is very quick, provided the customer has a card and delivery address on file. There are less stages involved; this makes the previous point about the customer being well informed even more important, to ensure accidental purchases are avoided.

The process

After reviewing the order process of these three popular sites, we can establish a suitable order process of our own for our framework. In general, the process will look like this:

1. View the basket.
    - Enter voucher code.
    - Select shipping method.
    - Review cost based on shipping and voucher code.
2. Authentication:
    - Log in.
    - Register.
    - Do nothing—already logged in.
3. Select delivery address.
4. Select payment method.
5. Order confirmation.
6. Display payment details.
7. Payment is made.
8. Order is processed.

Let's discuss this in more detail now.

The basket

We have the basket, which we implemented in Chapter 6, *The Shopping Basket*. In here are all of the products the customer chooses to purchase. At the basket stage, we can add two simple features, which in some stores are separate stages in their own right.

- We can allow the customer to enter a voucher code
- We can allow the customer to select their preferred shipping method

Voucher codes

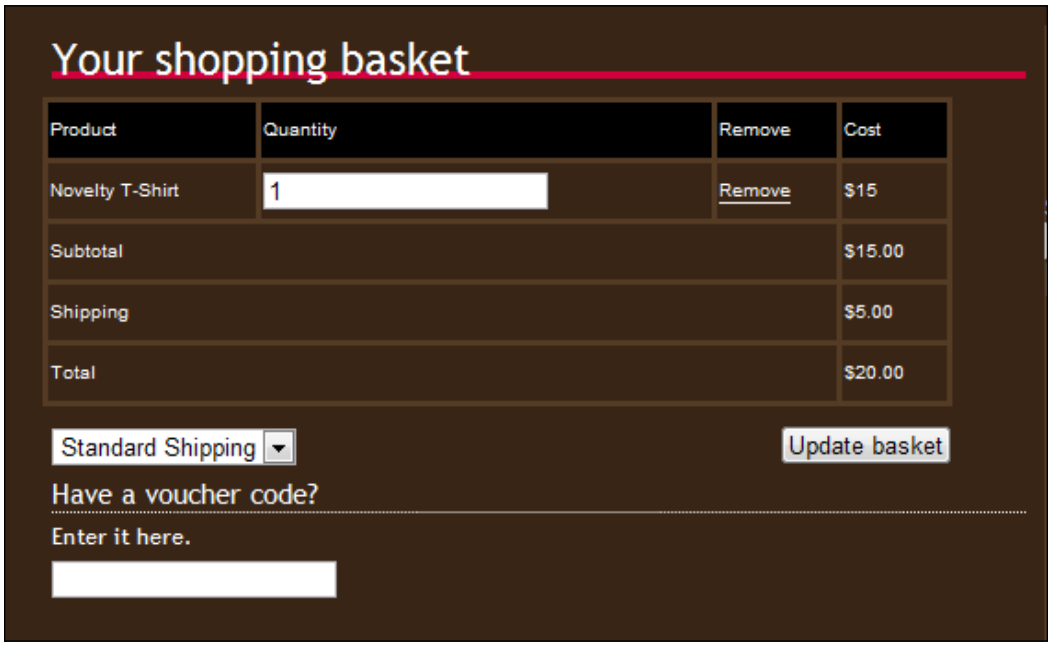
By adding the voucher code feature at this stage, the customer can immediately see the cost they should be paying for their order; this makes everything more open and transparent to the customer, saving them from waiting until near the end of the process before all of this is confirmed.

Shipping method

Different shipping methods often change the end cost of an order; again, by having this at the basket stage, the customer knows the price they will be paying, at the first stage of the process.

An overview

The basket should now display all products the customer wishes to purchase, list their quantities (in text boxes, so the customer can change it), have a text box for the customer to enter a voucher code, a drop-down list for the customer to select their preferred shipping method, a delete button for each set of products in their basket, a button allowing the customer to save their changes to the basket, and finally a button allowing them to proceed to the next stage of the process.



Authentication

While a customer having a user account isn't essential, most stores work on this basis; it is especially useful for customers making repeat orders, as we can save some of their details, making the purchase processes quicker for them.

Why should we authenticate the user at this stage?

Although the next stage is for the customer to enter their delivery address, we may be able to save them the need to re-enter their delivery address, as they may have a delivery address saved within their user account. If we authenticate the user at this stage, we can then look to see if they have a delivery address on file, and populate the delivery address form with these details. If the customer wishes to use a different delivery address, they could just change the values of the fields (which wouldn't alter their default delivery address incidentally).

Login

If the customer has a user account already, the customer can simply enter their username and password. Once the framework verifies their credentials and logs them in, this default delivery address can then be populated into the delivery form.

Register

If the customer is not logged in, and doesn't have a user account yet, they will need to register. This would consist of a username, e-mail address and password, and their default delivery address. The default delivery address would be saved for future orders, and pre-populated into the delivery address fields in the checkout process.

Do nothing

If the customer has already logged into the site prior to placing the order, then we won't need them to log in or register, as they are already logged in. In which case, this stage is skipped and the customer goes straight to the next one.

Delivery address

The customer's default delivery address will then be shown, within a series of text boxes, allowing the customer to select a new delivery address. This address should eventually be saved with the order itself, not within the user's account, as their default delivery address should still remain the same. (We would give the customer provisions to change their default delivery address elsewhere, for instance in a user account settings area.)

Payment method

The customer should then be able to select their payment method, from the methods we have installed and configured within our framework. Generally, such methods could include:

- An offline payment method, such as a check in the post, payment on delivery, or payment through a telephone hotline for processing orders
- An off-site online payment method, such as standard PayPal payments, where the customer is redirected to another website to make the payment, before being returned to the store
- An on-site online payment method, such as PayPal payments pro, where card details are taken securely on the site, and then passed to a payment processing API that processes the card details

Offline payment method

The offline payment method allows customers without credit or debit cards, or those who have them but don't wish to use them, to order online.

Advantages	Disadvantages
Enables more people to make a purchase	Payment isn't instant
No transaction costs (aside from any costs incurred from our business bank)	If using a check, we need to wait for it to clear before dispatching
	If using payment on delivery, we need to dispatch before receiving payment, and it may have been a fraudulent order

Off-site payment method

The off-site payment method is the simplest way to allow online payments.

Advantages	Disadvantages
Low barrier for entry	Takes the customer off-site
Sometimes, depending on the exact payment gateway, we can facilitate recurring billing	Some off-site payment gateways have stigma associated with them

On-site payment method

The on-site payment method is the most professional way to allow online payments.

Advantages	Disadvantages
Keeps the customer on-site	If we store the payment details, there are a number of security considerations, and PCI-DSS audit may be required
Generally, recurring billing can be done with this method	Requires an SSL certificate to be installed on the server, which in turn requires some additional development, and a static IP address for the website

Confirmation

At some point, we must go from a collection of products in a customer's basket, to an order stored in our database. This should be done at this stage, after the customer has confirmed the contents of their basket, and the other options (which can be changed later if required), the basket contents should be transferred into an order.

Once the customer has reviewed their order, a simple link or button should be presented, which once clicked, converts the basket contents into a new order in the system.

Payment details

As the customer has now confirmed their order and the order details, they should be presented with payment details:

- For offline payments this may be a postal address, a reference number, a name to make checks payable to, and instructions.
- For off-site online payment methods, this will be a link or a button to the off-site payment gateway, such as PayPal.
- For on-site online payment methods, the page must be securely encrypted, and contain relevant text boxes for the customer to enter their card details. This may also require a separate address field, for their billing address, unless we implement that elsewhere.

Payment made

The payment should then be made for the order; this should either be the customer posting a check, entering their card details, or paying through a service such as PayPal.

Order processed

Once payment has been made, the order should be marked as "pending dispatch"; this would either be done automatically when card details are verified or when an off-site gateway returns a successful notification, or manually when the administrator receives a check and marks it as pending dispatch. Generally, we would want to automatically e-mail the customer to confirm the payment was successful and the order is being fulfilled; the administrator should also be informed allowing them to fulfill the order. Once posted, the order should be updated to reflect this and the customer informed, potentially with a custom message (perhaps with a tracking number for the shipping provider).

Other points of note

In addition to the order process we have established in this chapter so far, there are some other considerations we must keep in mind while implementing the order process; these include:

- We should make the ability to move back and forth between stages in the process seamless for customers. This means that if the customer wishes to go back and make a change to their order information, or confirm some information, they can do so easily, without having to start at the beginning of the order process.
- Authentication should be seamless—either the login form should also be the first section of a registration process for new customers, or the page shown to the customer should contain both a form for logging in and a form for registering to use the site. This reduces potential barriers between the customer and them placing an order successfully.

Summary

In this chapter, we have looked in detail at the order process on a number of popular, successful e-commerce websites, and discussed their methods. We have used these observations to detail a process of our own, which we will use for our framework. This now gives us a clear plan of what we have left to do:

- Implement shipping and tax handling
- Implement payment processing
- Implement delivery address handling
- Implement order processing, fulfillment, and administration
- Implement voucher code discount provisions

We can now continue developing our store, with a better understanding of what we are going to do, and why.

8
Shipping and Tax

After discussing the checkout and order process in detail in the previous chapter, we now need to start building the functionality for this. In this chapter you will learn:

- How to calculate shipping costs based on:
    - Product
    - Weight
    - Location
    - "Shipping rules"
- About third-party shipping APIs
- How to integrate shipping and tracking notifications on orders
- How to integrate tax costs into our system

Shipping

Shipping is a very important aspect of an e-commerce system; without it customers will not accurately know the cost of their order. The only situation where we wouldn't want to include shipping costs is where we always offer free shipping. However, in that situation, we could either add provisions to ignore shipping costs, or we could set all values to zero, and remove references to shipping costs from the user interface.

Shipping methods

The first requirement to calculate shipping costs is a shipping method. We may wish to offer a number of different shipping methods to our customers such as standard shipping, next-day shipping, International shipping, and so on.

The system will require a default shipping method, so when the customer visits their basket, they see shipping costs calculated based off the default method. There should be a suitable drop-down list on the basket page containing the list of shipping methods; when this is changed, the costs in the basket should be updated to reflect the selected method.

We should store the following details for each shipping method:

- An ID number
- A name for the shipping method
- If the shipping method is active or not, indicating if it should be selectable by customers
- If the shipping method is the default method for the store
- A default shipping cost, this would:
    - Be pre-populated in a suitable field when creating new products; however, when the product is created through the administration interface, we would store the shipping cost for the product with the product.
    - Automatically be assigned to existing products in a store when a new shipping method is created to a store that already contains products.

This could be suitably stored in our database as the following:

Field	Type	Description
ID	Integer, Primary Key, Auto Increment	ID number for the shipping method
Name	Varchar	The name of the shipping method
Active	Boolean	Indicates if the shipping method is active
Default_cost	Float	The default cost for products for this shipping method.

This can be represented in the database using the following SQL:

```
CREATE TABLE `shipping_methods` (
    `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
    `name` VARCHAR( 50 ) NOT NULL ,
    `active` BOOL NOT NULL ,
    `is_default` BOOL NOT NULL ,
    `default_cost` DOUBLE NOT NULL ,
    INDEX ( `active` , `is_default` )
) ENGINE = INNODB COMMENT = 'Shipping methods';
```

Shipping costs

There are several different ways to calculate the costs of shipping products to customers:

- We could associate a cost to each product for each shipping method we have in our store
- We could associate costs for each shipping method to ranges of weights, and either charge the customer based on the weight-based shipping cost for each product combined, or based on the combined weight of the order
- We could base the cost on the customer's delivery address

The exact methods used, and the way they are used, depends on the exact nature of the store, as there are implications to these methods. If we were to use location-based shipping cost calculations, then the customer would not be aware of the total cost of their order until they entered their delivery address. There are a few ways this can be avoided: the system could assume a default delivery location and associated costs, and then update the customer's delivery cost at a later stage. Alternatively, if we enabled delivery methods for different locations or countries, we could associate the appropriate costs to these methods, although this does of course rely on the customer selecting the correct shipping method for their order to be approved; appropriate notifications to the customer would be required to ensure they do select the correct ones.

For this chapter we will implement:

- **Weight-based shipping costs**: Here the cost of shipping is based on the weight of the products.
- **Product-based shipping costs**: Here the cost of shipping is set on a per product basis for each product in the customer's basket.

We will also discuss location-based shipping costs, and look at how we may implement it. To account for international or long-distance shipping, we will use varying shipping methods; perhaps we could use:

- Shipping within state X.
- US shipping outside of state X.
- International shipping. (This could be broken down per continent if we wanted, without imposing on the customer *too* much.)

Product-based shipping costs

Product-based shipping costs would simply require each product to have a shipping cost associated to it for each shipping method in the store. As discussed earlier, when a new method is added to an existing store, a default value will initially be used, so in theory the administrator only needs to alter products whose shipping costs *shouldn't* be the default cost, and when creating new products, the relevant text box for the shipping cost for that method will have the default cost pre-populated.

To facilitate these costs, we need a new table in our database storing:

- Product IDs
- Shipping method IDs
- Shipping costs

The following SQL represents this table in our database:

```
CREATE TABLE `shipping_costs_product` (
    `shipping_id` int(11) NOT NULL, `product_id` int(11) NOT NULL,
    `cost` float NOT NULL, PRIMARY KEY (`shipping_id`,`product_id`) )
    ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Weight-based shipping costs

Depending on the store being operated from our framework, we may need to base shipping costs on the weights of products. If a particular courier for a particular shipping method charges based on weights, then there isn't any point in creating costs for each product for that shipping method. Our framework can calculate the shipping costs based on the weight ranges and costs for the method, and the weight of the product.

Within our database we would need to store:

- The shipping method in question
- A lower bound for the product weight, so we know which cost to apply to a product
- A cost associated for anything between this and the next weight bound

The table below illustrates these fields in our database:

Field	Type	Description
ID	Integer, primary key, Auto increment	A unique reference for the weight range
Shipping_id	Integer	The shipping method the range applies to
Lower_weight	Float	For working out which products this weight range cost applies to
Cost	Float	The shipping cost for a product of this weight.

The following SQL represents this table:

```
CREATE TABLE `shipping_costs_weight` (
    `ID` int(11) NOT NULL auto_increment,
    `shipping_id` int(11) NOT NULL,
    `lower_weight` float NOT NULL,
    `cost` float NOT NULL,
    PRIMARY KEY  (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

To think about: Location-based shipping costs

One thing we should still think about is location-based shipping costs, and how we may implement this. There are two primary ways in which we can do this:

- Assign shipping costs or cost surpluses/reductions to delivery addresses (either countries or states) and shipping methods
- Calculate costs using third-party service APIs

These two methods have one issue, which is why we are not going to implement them—that is the costs are calculated later in the checkout process. We want our customers to be well informed and aware of all of their costs as early as possible.

As mentioned earlier, however, we could get round this by assuming a default delivery location and providing customers with a guideline shipping cost, which would be subject to change based on their delivery address. Alternatively, we could allow customers to select their delivery location region from a drop-down list on the main "shopping basket" page. This way they would know the costs right away.

Regional shipping costs

We could look at storing:

- Shipping method IDs
- Region types (states or countries)
- Region values (an ID corresponding to a list of states or countries)
- A priority (in some cases, we may need to only consider the state delivery costs, and not country costs; in others cases, it may be the other way around)
- The associated costs changes (this could be a positive or negative value to be added to a product's delivery cost, as calculated by the other shipping systems already)

By doing this, we can then combine the delivery address with the products and lookup a price alteration, which is applied to the product's delivery cost, which has already been calculated. Ideally, we would use all the shipping cost calculation systems discussed, to make something as flexible as possible, based on the needs of a particular product, particular shipping method or courier, or of a particular store or business.

Third-party APIs

The most accurate method of charging delivery costs, encompassing weights and delivery addresses is via APIs provided by couriers themselves, such as UPS. The following web pages may be of reference:

- `http://www.ups.com/onlinetools`
- `http://answers.google.com/answers/threadview/id/429083.html`

Using such an API, means our shipping cost would be accurate, assuming our weight values were correct for our products, and we would not over or under charge customers for shipping costs. One additional consideration that third-party APIs may require would be dimensions of products, if their costs are also based on product sizes.

Shipping rules

Hopefully by using product and/or weight-based shipping methods, we can provide accurate shipping costs; however, some couriers cap their shipping costs for dispatches, or we may wish to offer incentives such as free shipping on certain orders. We may also find that we need to charge more for shipping, depending on the customer's location.

To store these rules, we need to record:

- A name for the rule
- The shipping method the rule is associated with
- The order of the rule, so if more than one rule were applicable, they would be applied in order
- The type of match to perform, either against total product cost, or the shipping cost (product cost would allow us to offer free shipping for orders over $X, and against shipping costs allow us to cap the costs at $Y)
- The amount to match against
- The operator to compare the match amount against the product or basket cost (this would be an operator such as greater than, less than, less than or equal to, greater than or equal to, not equal to, or equal to)
- The rule amount; this would be a value that would be applied to the shipping cost by a rule operator
- The rule operator, to determine how the rule amount would be applied to the shipping cost (this would be an operator such as plus, minus, divide by, multiply by, or set value to)

The following SQL represents this in our database:

```
CREATE TABLE ` shipping_rules` (
  `ID` int(11) NOT NULL auto_increment,
  `shipping_id` int(11) NOT NULL,
  `match_amount` float NOT NULL,
  `match_type` enum('shipping','products') NOT NULL,
  `match_operator` enum('<','>','<=','>=','<>','==') NOT NULL,
  `rule` varchar(255) NOT NULL,
  `rule_amount` float NOT NULL,
  `rule_operator` enum('+','-','=','*','/') NOT NULL,
  `order` int(11) NOT NULL,
  PRIMARY KEY  (`ID`),
  KEY `shipping_id` (`shipping_id`,`order`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;
```

Let's look at some example shipping rules, and potential values for these.

Free shipping

If we wished to offer free shipping to all customers whose orders were greater than or equal to $50, we would use the following values:

- **Name**: Free Shipping
- **Order**: 2 (assuming we use both this and the following rule)
- **Type of match**: Product
- **Match amount**: 50
- **Match comparison operator**: Greater than or equal to
- **Rule amount**: 0
- **Rule operator**: Set equal to

Capped shipping

If we wished to cap shipping costs to $20, to ensure no customer paid more than that, we would use the following values:

- **Name**: Max shipping cost
- **Order**: 1
- **Type of match**: Shipping
- **Match amount**: 20
- **Match comparison operator**: Greater than
- **Rule amount**: 20
- **Rule operator**: Set equal to

Of course we can also use these rules to do all sorts of calculations, such as discounted shipping for bulk orders, and so on. We could also extend these rules to take into account delivery locations.

Tracking

When products are shipped to customers, they may wish to be informed about tracking information. It may be possible for us to integrate with shipping provider APIs to do this. However, the simplest method (which could also eventually be integrated with such an API) is to allow store administrators to supply a message to the customer when they update an order's status to "dispatched"; this is something we will discuss in the Chapter 13, *Administration*.

Integrating shipping costs into the basket

We should integrate these shipping cost systems into our framework in the following stages:

1. List of shipping methods and a default method.
2. Calculate product-based shipping costs.
3. Calculate weight-based shipping costs.
4. Consider shipping rules and adjust shipping costs accordingly.

Shipping methods and a default

We can store a default shipping method in the framework's settings. When a customer selects an alternative shipping method, we should store that in an appropriate session variable. At this stage, all we need to do is check if the session variable is set. If the session variable is set, then that is the shipping method we must use; if it is not, then we must use the default shipping method.

```
// get the shipping method
  if( isset( $_SESSION['shipping_method'] ) )
  {
    // user-selected
    $this->shippingMethodID = intval( $_SESSION['shipping_method'] );
  }
  else
  {
    // system default
    $this->shippingMethodID = $this->registry->
        getSetting('default_shipping_method');
  }
```

Calculating shipping costs based on products

To calculate shipping costs based on products, we need to lookup the shipping cost for each product in the basket associated with the current shipping method.

```
// shipping costs: product based
$shippingCosts = $this->getShippingProductCosts( $this->productIDs );
```

Once we have these shipping costs, it is a case of looking up the product ID in the `$shippingCosts` array to get the shipping cost, and multiplying this by the quantity of the product in the basket.

```
$this->shippingCost = $this->shippingCost + ( $shippingCosts[
$contents['product_id'] ] * $contents['product_quantity'] );
```

Calculating shipping costs based on product weights

To calculate shipping costs based on product weight, we must build an array of shipping costs based on weight ranges.

```
// shipping costs: weight based
$weightCosts = $this->getShippingWeightCosts();
```

Once we have our array of shipping weight costs, while iterating through products in the basket, we then iterate through the *ordered* weights until we find an upper limit to the product in question. Once found, we get our shipping cost. This cost is then multiplied by the quantity of the product in the basket, and added to the rolling shipping cost.

```
// shipping costs: weight based
$currentWeight = 0;
while( $weightFound == false )
{
  if( $contents['product_weight'] >=
                              $weightCosts[$currentWeight]['weight'] )
  {
    $weightFound = true;
    $this->shippingCost = $this->shippingCost +
        ( $weightCosts[$currentWeight]['cost'] *
          $contents['product_quantity'] );
  }
  else
  {
    if( count( $weightCosts ) == $currentWeight )
    {
      // we don't want to do this forever!
      $weightFound = true;
    }
    else
    {
      $currentWeight++;
    }
  }
}
```

Considering shipping rules, and adjusting prices accordingly

The final shipping feature is shipping rules; this requires us looking up the shipping rules from the database, and iterating through them. For each rule, we need to check the type of rule, then check if the shipping cost or the basket cost is at least that of the rule amount; if it is, then we perform our rule calculation.

```
/**
  * Takes any shipping rules into account with regards to the shipping
costs
  * @return void
  */
  private function considerShippingRules()
  {
    // get the rules
    $rules_sql = "SELECT * FROM shipping_rules
                  WHERE shipping_id={$this->shippingMethodUD}
                  ORDER BY `order`";
    $this->registry->getObject('db')->executeQuery( $rules_sql );
    // go through them
    while( $rule = $this->registry->getObject('db')->getRows() )
    {
      // rule depends on the shipping cost
```

Here we have established that the current rule is based on shipping cost, which means we then check to see if the shipping cost meets the rule.

```
      if( $rule['match_type'] == 'shipping' )
      {
        $match = false;
        $match_operator = $rule['match_operator'];
        // check to see our shipping cost meets the rule
        if( $match_operator == '==' )
          { if( $this->shippingCost == $rule['match_amount'] )
            { $match = true; }  }
        elseif( $match_operator == '<>' )
        { if( $this->shippingCost <> $rule['match_amount'] )
          { $match = true; }  }
        elseif( $match_operator == '>=' )
        { if( $this->shippingCost >= $rule['match_amount'] )
          { $match = true; }  }
        elseif( $match_operator == '<=' )
        { if( $this->shippingCost <= $rule['match_amount'] )
          { $match = true; }  }
        elseif( $match_operator == '>' )
```

```
    { if( $this->shippingCost > $rule['match_amount'] )
      { $match = true; }  }
    elseif( $match_operator == '<' )
    { if( $this->shippingCost < $rule['match_amount'] )
      { $match = true; }  }
```

If a rule match was found, we then take the rule into account.

```
    if( $match == true )
    {
      // set the shipping cost based on the rule operator and
      // the rule amount
      $rule_operator = $rule['rule_operator'];
      if( $rule_operator == '=' )
      { $this->shippingCost = $rule['rule_amount']; }
      elseif( $rule_operator == '+' )
      { $this->shippingCost = $this->shippingCost
          + $rule['rule_amount']; }
      elseif( $rule_operator == '-' )
      { $this->shippingCost = $this->shippingCost
          - $rule['rule_amount']; }
      elseif( $rule_operator == '*' )
      { $this->shippingCost = $this->shippingCost
          * $rule['rule_amount']; }
      elseif( $rule_operator == '/' )
      { $this->shippingCost = $this->shippingCost
          / $rule['rule_amount']; }
    }
  }
```

If the product is based on the basket cost, we then do the same as before, except that the rule matching depends on the cost of the shopping basket.

```
  elseif( $rule['match_type'] == 'products' )
  {
    // rule depends on the basket cost
    $match = false;
    $match_operator = $rule['match_operator'];
    // check to see our basket cost meets the rule
    if( $match_operator == '==' )
    { if( $this->shippingCost == $rule['match_amount'] )
      { $match = true; }  }
    elseif( $match_operator == '<>' )
    { if( $this->cost <> $rule['match_amount'] )
      { $match = true; }  }
    elseif( $match_operator == '>=' )
    { if( $this->cost >= $rule['match_amount'] )
      { $match = true; }  }
```

```
        elseif( $match_operator == '<=' )
        { if( $this->cost <= $rule['match_amount'] )
          { $match = true; }  }
        elseif( $match_operator == '>' )
        { if( $this->cost > $rule['match_amount'] )
          { $match = true; }  }
        elseif( $match_operator == '<' )
        { if( $this->cost < $rule['match_amount'] )
          { $match = true; }  }
        if( $match == true )
        {
          // set the shipping cost based on the rule operator
          // and the rule amount
          $rule_operator = $rule['rule_operator'];
          if( $rule_operator == '=' )
          { $this->shippingCost = $rule['rule_amount']; }
          elseif( $rule_operator == '+' )
          { $this->shippingCost = $this->shippingCost
              + $rule['rule_amount']; }
          elseif( $rule_operator == '-' )
          { $this->shippingCost = $this->shippingCost
              - $rule['rule_amount']; }
          elseif( $rule_operator == '*' )
          { $this->shippingCost = $this->shippingCost
              * $rule['rule_amount']; }
          elseif( $rule_operator == '/' )
          { $this->shippingCost = $this->shippingCost
              / $rule['rule_amount']; }
        }
      }
    }
  }
```

Tax

There are three main ways to tackle tax costs in an e-commerce environment:

- We include tax in our product prices
- We assign tax codes to products to separately calculate and display tax costs
- We calculate tax based on the location of the buyer

The exact requirement for a particular store depends on the store itself and the laws applicable in that country or state. In some situations we can include the tax for a product in its price; it doesn't need to be displayed to the customer. In others, we may wish for tax to be shown and calculated for the customer, if they are able to reclaim this tax (for example UK/EU VAT), and in some states in the US different states have different taxes depending on the buyer or seller, where some customers may be taxed, others not, or the tax may be based on the state the seller resides in themselves.

Most situations can be handled by associating products with tax calculations, so let's focus on that. However, we will also discuss how we may implement a location-based tax system, to charge tax depending on the customer's delivery or billing address.

Separately calculating tax values

We could either have:

- Tax included in a product price, and a tax rule calculating how much of the product's price should be tax
- Product costs stored without tax, and associated with their relevant tax calculations

The main difference to the way tax calculations would need to work and the shipping costs is that tax costs actually need to be integrated before the basket; that is, the products themselves should incorporate tax costs.

We will look at the second of these two options.

This would require:

- Products to have a tax code associated with them
- A table of tax codes to be stored in our database, along with calculation details

The tax codes (just a reference for the type of tax; for instance, at the time of writing in the UK we would have: zero-rated VAT—0%, standard rate VAT—15%, reduced rate VAT—5%, and different products may have different tax codes associated with them) would have a calculation value and operation associated with them, similar to our shipping rules, this allows the framework to easily add/subtract/divide/multiply the product cost with the calculation value.

Field	Type	Description
ID	Integer, Primary Key, Auto Increment	The ID for the tax code
Tax_code	Varchar	Name of the tax code
Calculation_value	Double	The value applied to the order cost
Calculation_operation	Enum	The arithmetic operation applied to the order cost and the calculation value, to compute the tax.

The following SQL represents this table:

```
CREATE TABLE `tax_codes` (
`ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
`tax_code` varchar(255) NOT NULL,
`calculation_value` DOUBLE NOT NULL,
`calculation_operation` ENUM( '+', '-', '*', '/', '=' ) NOT NULL,
INDEX ( `tax_code` )
) ENGINE = INNODB;
```

Again, for example, UK standard rate VAT would have a value of 1.15, and an operation of multiply by.

To create a truly flexible tax system for an e-commerce system would involve a book of its own. The simplest methods that we have discussed are relatively straightforward to implement, especially because we have done some very similar work with our shipping methods.

To think about: Location-based tax costs

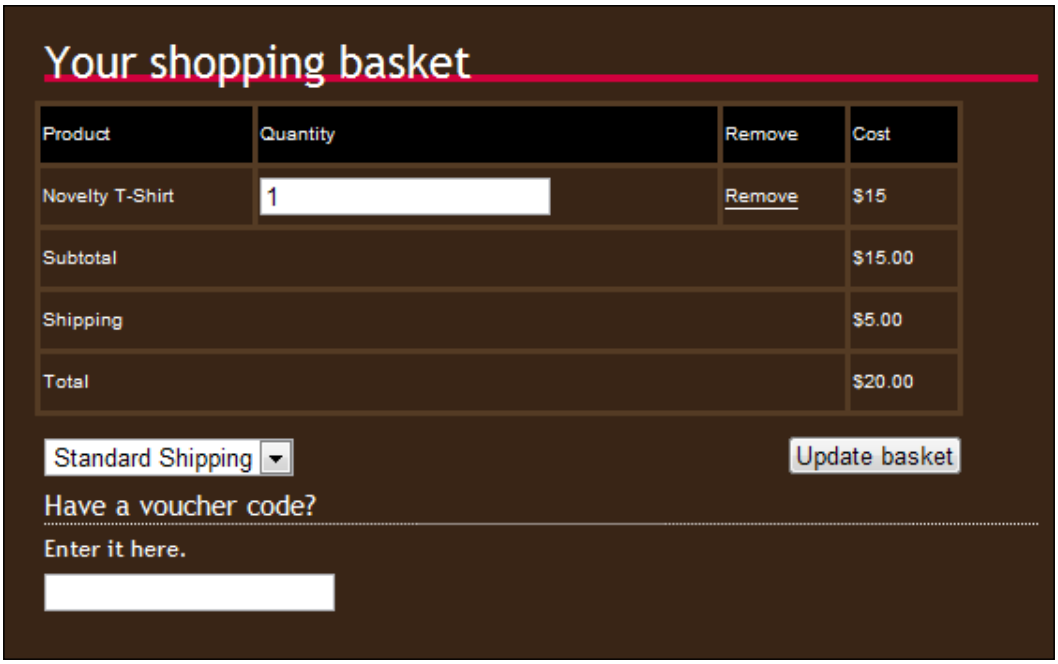
In some situations, we may have different taxes applicable depending on the locations of the buyers and sellers respective to one another. This may be something we would wish to implement. Advice from a tax professional is recommended to determine if this is required for a particular use or implementation of your framework in a particular store.

A look at our basket now

Now that we have implemented shipping costs to our store, our basket has some changes since we last looked at it:

- We have a row in the products table for the shipping costs
- The order total includes the shipping costs
- There is a drop-down list of shipping methods allowing customers to change the shipping method, which updates costs accordingly.

Here's a view of our new basket:



Summary

In this chapter, we discussed different ways to approach shipping costs and tax values for products within our e-commerce store. This included:

- Creating shipping methods
- Creating shipping rules to cap, reduce, wipe, or alter shipping costs based on the cost of a basket, or have the shipping cost otherwise calculated
- Setting shipping costs for each product based on the product and the shipping method
- Setting shipping costs for products based on weights and the shipping method
- How we would introduce tax costs to products

Now that we have looked into shipping and tax in detail, we can look at discount codes, purchasable voucher codes, and referrals in the next chapter.

9
Discounts, Vouchers, and Referrals

With shipping and tax issues taken into account, the next logical step is discount codes, as these need to be entered at the shopping basket stage. Going hand in hand with discount codes are voucher codes and referral discounts. In this chapter, you will learn:

- How to create a discount code system
    - How to offer different types of discounts
    - How to take the discount into account at the shopping basket stage
- How to sell voucher codes on your store
- How to offer discounts to customers who bring us referral business

Discount codes

Discount codes are a great way to both entice new customers into a store, and also to help retain customers with special discounts. The discount code should work by allowing the customer to enter a code, which will then be verified by the store, and then a discount will be applied to the order.

The following are discount options we may wish to have available in our store:

- A fixed amount deducted from the cost of the order
- A fixed percentage deducted from the cost of the order
- The shipping cost altered, either to free or to a lower amount
- Product-based discounts (although we won't cover this one in the chapter)

It may also be useful to take into account the cost of the customer's basket; after all if we have a $5 discount code, we probably wouldn't want that to apply for orders of $5 or lower, and may wish to apply a minimum order amount.

Discount codes data

When storing discount codes in the framework, we need to store and account for:

- The voucher code itself, so that we can check that the customer is entering a valid code
- Whether the voucher code is active, as we may wish to prepare some voucher codes, but not have them usable until a certain time, or we may wish to discontinue a code
- A minimum value for the customer's basket, either as an incentive for the customer to purchase more or to prevent loss-making situations (for example a $10 discount on a $5 purchase!)
- The type of discount:
    - **Percentage**: To indicate that the discount amount is a percentage to be removed from the cost
    - **Fixed amount deducted**: To indicate that the discount amount is a fixed amount to be removed from the order total
    - **Fixed amount set to shipping**: To indicate that the discount amount is to be the new value for the shipping cost
- Discount amount; that is, the amount of discount to be applied
- The number of vouchers issued, if we wish to limit the number of uses of a particular voucher code
- An expiry date, so that if we wish to have the voucher code expire, codes with a date after the stored expiry date would no longer work

Discount codes database

The following table illustrates this information as database fields within a table:

Field	Type	Description
ID	Integer (Primary Key, Auto Increment)	For the framework to reference the code
Vouchercode	Varchar	The code the customer enters into the order
Active	Boolean	If the code can be used
Min_basket_cost	Float	The minimum cost of the customer's basket for the code to work for them
Discount_operation	ENUM('-','%','s')	The type of discount
Num_vouchers	Integer	The number of times the voucher can be used
Expiry	timestamp	The date the voucher code expires, and is no longer usable

The default value for `num_vouchers` is `-1`, which we will use for vouchers that are not limited to a set number of issues.

The following code represents this data in our database:

```
CREATE TABLE `discount_codes` (
  `ID` INT( 11 ) NOT NULL AUTO_INCREMENT ,
  `vouchercode` VARCHAR( 25 ) NOT NULL ,
  `active` TINYINT( 1 ) NOT NULL ,
  `min_basket_cost` FLOAT NOT NULL ,
  `discount_operation` ENUM( '-', '%', 's' ) NOT NULL ,
  `discount_amount` FLOAT NOT NULL ,
  `num_vouchers` INT( 11 ) NOT NULL DEFAULT '-1',
  `expiry` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
PRIMARY KEY ( `ID` )
) ENGINE = INNODB DEFAULT CHARSET = latin1 AUTO_INCREMENT =1;
```

Discount codes functionality

The functionality for discount codes can be encapsulated into a single function; this function should be called when the basket is loaded and updated. (That is, if the customer changes the quantity of products in the basket, we must run this; also, if the customer adds a voucher code to their order, we must obviously call this function.)

The function needs to:

1. Check to see if the customer has entered a voucher code with their order.
2. If they have, it must look up the voucher code to see if it exists, or doesn't exist.
3. If the voucher code exists, it must check to see if the code has expired and is still able to be used (that is, that `num_vouchers` is greater than `0` or equal to `-1`).
4. Assuming this is the case, it must then check to see that the customer's basket cost is at least that of the minimum order amount in the discount codes record in the database.
5. If the voucher is able to be used, it must then determine the type of voucher, and make the relevant discount to either the basket cost, or the shipping costs.

The following code does all of this; some of the relevant sections described are highlighted within the code:

```
/**
  * Consider and apply voucher codes
  * Types of voucher code
  * - = FIXED AMOUNT OFF
  * % = PERCENTAGE OFF
  * s = SET NEW SHIPPING COST
  * @param voucherCode String
  * @return void
  */
  private function considerVouchers( $voucherCode )
  {
    //The voucher code value is checked to ensure it is not empty
    // (as per point 1)
    if( $voucherCode != '' )
    {
      // we need the date, to see if the voucher has expired
      $cts = date('Y-m-d H:i:s');
      $voucher_sql = "SELECT *, if('{$cts}' > expiry, 1, 0)
                        AS expired FROM discount_codes
                      WHERE vouchercode='{$voucherCode}' LIMIT 1";
      $this->registry->getObject('db')->executeQuery( $voucher_sql );
      if( $this->registry->getObject('db')->numRows() == 0 )
      {
        $this->voucher_notice = 'Sorry, the voucher code you entered
            is invalid';
```

```
}
else
{
  $voucher = $this->registry->getObject('db')->getRows();
  if( $voucher['active'] == 1 )
  {
    if( $voucher['expired'] == 1 )
    {
      $this->voucher_notice = 'Sorry, this voucher has
          expired';
      return false;
    }
    else
    {
      // check to see there are some vouchers, and customer
      // has enough in their basket (points 3 and 4)
      if( $voucher['num_vouchers'] != 0 )
      {
        if( $this->cost >= $voucher['min_basket_cost'] )
        {
          $this->discountCode = $voucherCode;
          $this->discountCodeId = $voucher['ID'];
          // If the discount operation is a percentage, then
          // the discount value is applied to the basket cost
          // to calculate that percentage. This amount is then
          // deducted from the order cost, giving a "discount
          // value"% discount from the order.
          if( $voucher['discount_operation'] == '%' )
          {
            $this->cost = $this->
        cost - (($this->cost)/100)*$voucher['discount_amount'];
            $this->voucher_notice = 'A '
                . $voucher['discount_amount']
                . '% discount has been applied to your order';
            return true;

            // If the discount operation is a subtraction, then
            // the discount amount from the discount code is
            // deducted from the order cost, and the order cost
            // is updated to reflect this.
         }
          elseif( $voucher['discount_operation'] == '-' )
          {
            $this->cost =
                $this->cost - $voucher['discount_amount'];
            $this->voucher_notice = 'A discount of &pound;'
```

```
                        . $voucher['discount_amount']
                        . ' has been applied to your order';
                    return true;
                    // Finally, if the discount operation is set to s
                    // then, we set the shipping cost to the discount
                    // value. This could allow us to set free shipping,
                    // or just reduce shipping costs.
                  }
                  elseif( $voucher['discount_operation'] == 's' )
                  {
                    $this->shipping_cost = $voucher['discount_amount'];
                    $this->voucher_notice = 'Your orders shipping cost
                        has been reduced to &pound;'
                        . $voucher['discount_amount'];
                    return true;
                  }
                }
                else
                {
                  $this->voucher_notice = 'Sorry, your order total is
                      not enough for your order to qualify for this
                      discount code';
                  return false;
                }
              }
              else
              {
                $this->voucher_notice = 'Sorry, this was a limited
                    edition voucher code, there are no more instances
                    of that code left';
                return false;
              }
            }
          }
          else
          {
            $this->voucher_notice = 'Sorry, the vocuher code you
                entered is no longer active';
            return false;
          }
        }
      }
    }
```

Now, we have the basis for our discount code feature. We can issue discount codes as an incentive to get new customers, or to encourage existing customers to make more purchases.

Reducing the number of codes available

One feature we added to the database structure for our discount codes was the ability to limit the number of times a voucher could be used, effectively allowing a code to be "issued" a set number of times. We need to take this into account, and *reduce* the number of vouchers in circulation, once one has been used.

This won't be done at this stage, but will need to be implemented at the final checkout stage when an order is updated to "paid", or the customer pays online. The following function will do this for us; it checks to see if a code is used, or if it's unlimited or has expired, and if it's not unlimited and hasn't already expired, it updates the code to decrement the number of uses remaining:

```
private function adjustDiscountCodeQuantities( $codeId )
{
  $sql = "SELECT num_vouchers FROM discount_codes WHERE ID=" . $codeId;
  $this->registry->getObject('db')->executeQuery( $sql );
  if( $this->registry->getObject('db')->numRows() > 0 )
  {
    $codeData = $this->registry->getObject('db')->getRows();
    if( $codeData['num_vouchers'] > 0 )
    {
      $sql = "UPDATE discount_codes SET num_vouchers=num_vouchers-1
              WHERE ID=" . $codeId;
      $this->registry->getObject('db')->executeQuery( $sql );
    }
  }
}
```

Purchasable voucher codes

Voucher codes work in the same way as discount codes, except that they are purchased for use by a customer, as opposed to given away for promotional reasons.

Existing functionality

The discount codes and product variation features already give us most of the functionality required for purchasable voucher codes.

Discount codes

As a voucher code has the same functionality as our discount codes, which we built earlier, we have a large portion of the functionality in place. When a voucher is purchased, a new record in the discount codes table will be made, with a `num_vouchers` value of `1`.

Product variations

The product variations feature we built in Chapter 4, *Product Variations and User Uploads*, allows us to create a purchasable voucher code product, with variations that increase the price in increments, perhaps of $5.

Required additional functionality

We only need to add additional functionality to this if we wish to *automate* the process of generating a voucher code when a customer purchases a voucher code. Without additional functionality, we would simply notice a new order, manually create a new discount code, and then e-mail the customer with the code. However, automating this would save us quite a lot of time, so let's look at what would be involved in doing this:

1. First, we must wait until an order has its status updated to "paid". This would either be when a payment is made online by the customer, or when an offline payment is received and we, as the administrator, mark the order as "paid".
2. The order contents must be searched for anything that is a purchasable voucher code.
3. For each of these vouchers purchased, a new record must be automatically inserted into the `voucher_codes` table with the following values:
    - `vouchercode`: A randomly generated string
    - `active`: `1`
    - `min_basket_cost`: The amount of the voucher purchased
    - `discount_operation`: (because we are subtracting an amount from the cost)
    - `discount_amount`: The amount of the voucher purchased
    - `num_vouchers`: `1`
    - `expiry`: A year in the future would be a suitable validity period
4. These vouchers would then be sent through e-mail to the customer.

Referrals

To reward loyal customers, we may wish to offer referral discounts to them. This would work by encouraging our customers to introduce new customers, and giving them a credit based off a percentage of orders placed by customers they refer to our store. Customers referring other customers would be given a referral code, which could either be part of a link used to promote the site, or given to customers to enter somewhere on the order process.

The referral process should work like this:

1. Check for a referral code in the URL.
2. If a code is found in the URL, it should be stored in a cookie, unless a previous referral code is already being stored; in that case, the older code should be used.
3. At the checkout stage, this code should be looked up, to check if it is valid and to see the commission to be applied to the customer who passed the referral. The order number, commission value, and referring customer should be stored in a database table.
4. When the order is updated to "paid" or "paid online", the commission should be applied to the relevant customer's account.

Database changes

This requires a new database table, as well as some database alterations to work.

New table: Referrers

A referrers table is required to link customers to a unique referral code, and a commission percentage. It would require the following fields:

Field	Type	Description
Customer ID	Integer	A reference to the customer's ID number
Referral Code	Varchar (Unique)	The customer's referral code, which he/she would give to friends
Commission percentage	Float	The percentage of commission that would be credited back to the customer's account
Active	Boolean	If the customer is a referrer, or not, as we may wish to disable a customer's referral code, if he/she is abusing it

Changes

Customers should have a credit field, showing how much credit they have with the store based on referred customers. The orders table should also be altered to store a record of any referral code used.

Functionality

The workflow of this feature would work like this:

1. An order is placed, and a referral code is associated with it.
2. The order is marked as "paid".
3. A lookup is performed on the referrers table to find the customer and the percentage.
4. The commission value is calculated.
5. The referring customer's credit value is updated to include new commission earned.

We could extend this to record a list of transactions to a customer's credit, such as dates and amounts their account was credited by, and allow reports to be generated and sent to them. However, that could occupy several chapters itself!

Checkout process consideration

This requires an important consideration when we get to the checkout stage: we must take into account any credit stored on a customer's account. If they have credit, it should be deducted from any of their own orders, to ensure they can spend the commission they earned.

Summary

We now have a number of useful features to encourage new orders and customers to our store, as well as encouraging existing customers to promote our store through affiliate codes.

This included creating a voucher code system which supported vouchers that deducted a percentage from the order total, vouchers that deducted a fixed amount from the order total, and vouchers that altered the cost of shipping to the customer. These vouchers were able to take into account expiry dates, limited-use options, and the current cost of the customer's basket.

Taking this forward, we looked at how to extend our framework to allow customers to purchase vouchers as gifts for other customers, as well as how to extend our store to support referral bonuses and incentives for our customers.

Now we can move onto the most important stage of our store: payments. Without payments, our store would not actually trade.

10
Checkout

We now have only one primary feature left for the checkout and order process, and this is the payment section. Although this is the last primary feature, there are still quite a few loose ends to tidy up. In this chapter, you will learn:

- How to store delivery addresses
- How to manage a default delivery address on a per-customer basis
- How to allow customers to select a payment method
- How to let customers confirm their order

All of the functionality we are working on in this chapter can be contained nicely within its own controller: `checkout`.

Order process review

If we review the order process we discussed in Chapter 7, *The Checkout and Order Process*, we have the following process:

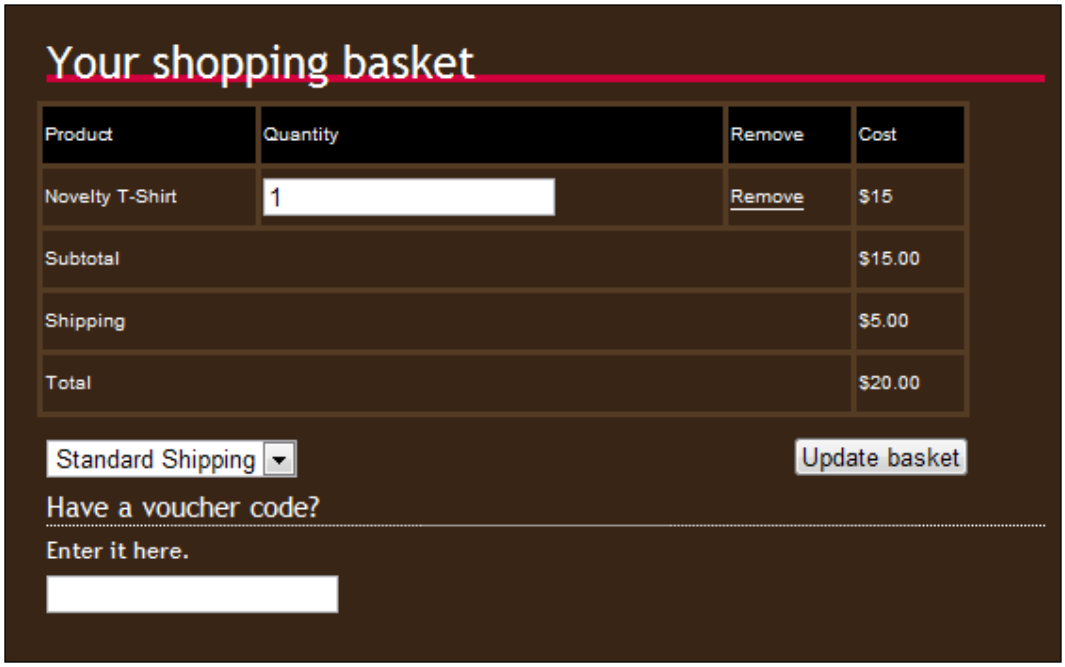
1. View the basket
    - Enter voucher code
    - Select shipping method
    - Review cost based on shipping and voucher code
2. Authentication
    - Log in
    - Register
    - Do nothing (if already logged in)
3. Confirm delivery address
4. Select payment method

5. Order confirmation
6. Display payment details
7. Payment made
8. Order processed

As things currently stand, we have the first section completed:

- Viewing the basket—done in Chapter 6, *The Shopping Basket*
- Entering voucher code—done in Chapter 9, *Discounts, Vouchers, and Referrals*
- Selecting the shipping method—done in Chapter 8, *Shipping and Tax*
- Reviewing cost based on shipping and voucher code—done in Chapters 8 and 9

So let's take a look at our shopping basket:



The final three points—payment details, making payment, and order processing are to come in the next chapter. The four points in the middle, although not major features, still need to be worked into our framework; otherwise, we will not be able to guide our customers to the point where they can actually make the payment for their order.

Authentication

When the customer has reviewed their basket and made any necessary changes, we need to allow them to supply their delivery address. Before they can do that; however, we must consider authentication.

If the customer is logged in, then we can simply send them to the "delivery address" page, where they will be presented with their default delivery address based on their user account details. If the customer is not logged in, we need to allow them to either log in, or enter their details to sign up to the store.

To make this process as seamless as possible, there are a few things we should consider:

- Should the user have to click on a link or a button to view the registration page?
- Should the user have to click on a link or a button to view the login page?
- Should we validate the user's e-mail address by sending them an e-mail and asking them to confirm if it is valid?

The answer to all three of these points is no. If we do any of these things, then we are creating an unnecessary barrier between the customer and them completing their order. Some believe that even requiring the customer to create a user account is a step too far. As we are going to require the customer to make their own account, we should make this less intrusive by allowing them to enter all of their details on one page, with only a few additions as opposed to if they were just supplying their delivery address.

```
private function authenticationCheck()
{
  //First we check that the user is logged in.
  if( $this->registry->getObject('authenticate')->isLoggedIn()
      == true )
  {
    // Then we check to see if they have just logged in.
    if( $this->registry->getObject('authenticate')->justProcessed()
        == true )
    {
      // As the user has just logged in, we transfer their basket to
      //  their account.
      // store the basket in the user account
      $this->basket->transferBasketToUser( $this->registry
          ->getObject('authenticate')->getUserID() );
      // check the basket, to ensure the user has some products
      //  in their basket after logging in
```

```
            $this->basket->checkBasket();
          }

          $this->setDelivery();
          return true;
        }
        else
        {
          //User is not logged in, so we show them the login/register page.
          $this->registry->getObject('template')->
              buildFromTemplates('header.tpl.php',
              'checkout/loginreg.tpl.php','footer.tpl.php');
          $this->registry->getObject('template')->getPage()->
              addTag('pagetitle', 'Login or sign up' );
          return false;
        }
      }
```

Having our customer now able to log in, we now confirm their delivery address as follows:



Delivery address

Firstly, we check to see if the customer is submitting the delivery address form, if he/she is, then we sanitize the address details and set the delivery address with the following code:

```
private function setDelivery()
{
  // Checking to see if the set_delivery_address post field has been
  // set, indicates that the customer has set their delivery address.
  if( isset( $_POST['set_delivery_address'] ) )
  {
    // save delivery address
    // We then call the basket's setDeliveryAddress method, which
    // saves the default delivery address. This method takes a series
    // of strings as its parameters. These strings are inserted
    // directly into the database, so we must sanitize them first,
    // using the databases sanitizeData method.
    $this->basket->setDeliveryAddress(
        $this->registry->getObject('db')->
            sanitizeData( $_POST['address_name'] ),
        $this->registry->getObject('db')->
            sanitizeData( $_POST['address_lineone'] ),
        $this->registry->getObject('db')->
            sanitizeData( $_POST['address_linetwo'] ),
        $this->registry->getObject('db')->
            sanitizeData( $_POST['address_city'] ),
        $this->registry->getObject('db')->
            sanitizeData( $_POST['address_postcode'] ),
        $this->registry->getObject('db')->
            sanitizeData( $_POST['address_country'] ));
    // Once the delivery address is updated, we then redirect the
    // customer to the payment method page.
    $this->registry->redirectUser('checkout/select-payment-method/',
        'Delivery address saved', 'Your delivery address has been
        saved', false );

  }
  else
  {
    // If the customer has not just entered their delivery details,
    // we display the form using the default delivery details.
    $this->registry->getObject('template')->
        buildFromTemplates('header.tpl.php',
        'checkout/delivery.tpl.php','footer.tpl.php');
    $address = $this->basket->getDeliveryAddress();
    if( ! empty( $address ) )
    {
      $this->registry->getObject('template')->getPage()->
```

```
            addTag('address_name', $address['address_name']);
        $this->registry->getObject('template')->getPage()->
            addTag('address_lineone'], $address['address_lineone']);
        $this->registry->getObject('template')->getPage()->
            addTag('address_linetwo'], $address['address_linetwo']);
        $this->registry->getObject('template')->getPage()->
            addTag('address_city'], $address['address_city']);
        $this->registry->getObject('template')->getPage()->
            addTag('address_postcode'],
            $address['address_postcode']);
        $this->registry->getObject('template')->getPage()->
            addTag('address_country'], $address['address_country'] );
      }
    }
  }
```

Here's what our "delivery address confirmation" page will look like:



Payment method

The final configurable option for the customer is the payment method. We simply need to generate a list of available payment methods, and present them to the customer.

```
private function selectPayment()
{// If the customer has set the payment method, we save that and then
redirect the customer.
  if( isset( $_POST['eagle_payment'] ) )
  {
    $method = intval( $_POST['payment_method']);
    $this->basket->setPaymentMethod( $method );
    $this->registry->redirectUser('checkout/confirm/',
        'Payment method saved', 'Your preferred payment method has
        been saved', false );
  }
```

If the customer has not selected a payment method, we show them the list.

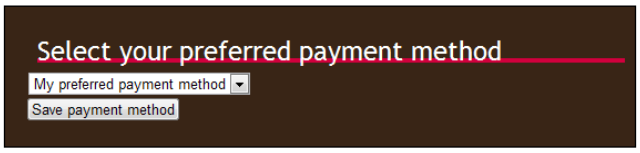
```
    else
    {
      $this->registry->getObject('template')->getPage()->
          addTag('pagetitle', 'Select your payment method');
      $this->registry->getObject('template')->
          buildFromTemplates('header.tpl.php',
          'checkout/payment.tpl.php','footer.tpl.php');
      $methods_sql = "SELECT name as method_name, ID as method_id
          FROM payment_methods";
      $this->registry->getObject('db')->executeQuery( $methods_sql );
          $methods = array();
      $selected = $this->basket->getPaymentMethod();
```

As the customer may have already selected a payment method, and then came back to this page to change it, we need to highlight the currently selected payment method.

```
        while( $method = $this->registry->getObject('db')->getRows() )
        {
          if( $method['method_id'] == $id )
          {
            $method['selected'] = "selected='selected'";
          }
          else
          {
            $method['selected'] = '';
          }
          $methods[] = $method;
        }

        $methodsCache = $this->registry->getObject('db')->
            cacheData( $methods );
        $this->registry->getObject('template')->getPage()->
            addTag( 'payment_methods', array( 'DATA', $$methodsCache ) );
    }
  }
```

So here's what our "payment selection" page looks like this:

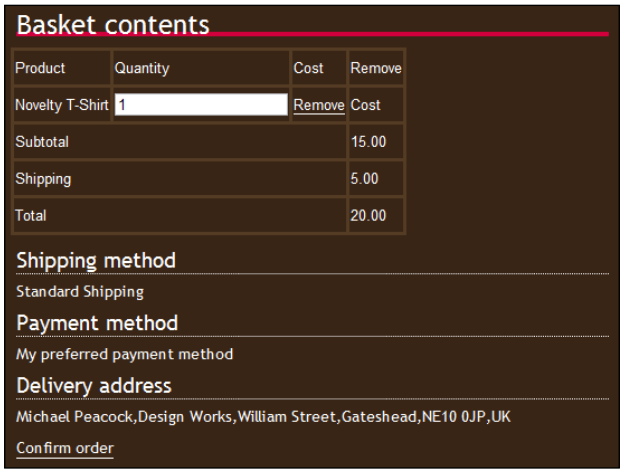


Confirmation

The final stage, before payment, is to allow the customer to confirm their order. This involves displaying the order to the customer along with the selected shipping method, payment method, and delivery address. When the customer is happy with the order, we then must:

- Create a new order
- Create a new order item entry for each product in the basket
- Create a new order item attribute association entry for each attribute selected for each product in the basket
- Delete the contents of the user's basket
- Send the customer to the payment page

Our confirmation page looks like this:



Storing orders in the database

The functionality we have developed in the course of this chapter, requires a number of new database tables to store orders. We need five new tables; these are:

- **Orders**: To store the orders.
- **Order statuses**: To maintain a list of possible order statuses.
- **Order items**: To store items associated with each order.
- **Order item attributes**: To store item attributes associated with each item within an order.
- **Payment methods**: To store a list of payment methods.

Orders table

Our orders table is used to relate all items in an order together, store their cost, status, and other information that is common for all items in the order, such as the delivery address.

It is important that we store the cost and shipping cost in this table; one reason for this is that if a product price changes after an order has been processed, our orders won't add up.

Field	Type	Description
ID	Integer, Primary Key, Auto Increment	A unique reference for the order
User	Integer	The ID of the user placing the order
IP	Varchar	The user's IP address for security reasons
Timestamp	Timestamp	The time the order was placed
Status	Integer	Reference to the order status table, indicating the order status, for example awaiting payment, dispatched, and so on
Comment	Longtext	We may wish to add space for the customer to add a note to their order
Delivery_comment	Longtext	Same as above, but for special delivery instructions
Shipping_method	Integer	The method to be used for shipping the order
Payment_method	Integer	The method by which the customer wishes to pay
Shipping_address_name	Varchar	The name of the recipient of the order
Shipping_address_lineone	Varchar	The delivery address
Shipping_address_linetwo	Varchar	The delivery address
Shipping_address_city	Varchar	The delivery address
Shipping_address_postcode	Varchar	The delivery address
Shipping_address_country	Varchar	The delivery address
Products_cost	Float	The total cost of products in the order
Shipping_cost	Float	The total shipping cost for the order
Voucher_code	Integer	A reference to any voucher code used for the order

Order statuses

As each order will have a status associated with it, we should maintain a list of these statuses in a separate table. We can also store some additional information so that we can detect if the customer needs to do something for the order to progress, for example to make a payment, or if the store owner needs to do something, like dispatch the order.

Field	Type	Description
ID	Integer, Primary Key, Auto Increment	The unique ID for the status
Name	Varchar	The status of the order
Awaiting_customer	Boolean	Indicates if the order needs the customer to perform an action for the order to progress
Awaiting_staff	Boolean	Indicates if the administrator needs to perform an action for the order to progress
Completed	Boolean	Indicates that the order is complete

Order items

For each item in the order, we need to store details in a separate table. This table should essentially be the same as our basket contents table, except that instead of storing user details, we associate each row with an order.

Field	Type	Description
ID	Integer, Auto Increment, Primary Key	A unique reference for the order item
Order_Id	Integer	The order the product is part of
Quantity	Integer	The number of these products within the order
Uploaded_file	Varchar	Path to a file uploaded as part of the order, if appropriate
Custom_text_values	Longtext	A serialized array of custom text values
Standard	Boolean	If the product was standard, or customized

Order item attributes

Because some products can have selectable attributes, we need to also select these and store them in the database.

Field	Type	Description
Order_item_id	Integer	The product item record for an order
Attribute_id	Integer	The attribute of this product which is being ordered

Payment methods

Finally, we have payment methods. This database table stores a list of payment methods. We will come on to this properly in the next chapter; however, for now we do need a table, because we reference payment methods in the order table.

Field	Type	Description
ID	Integer, Primary key, Auto Increment	A unique ID for the payment method
Name	Varchar	The name of the payment method
Key	Varchar	Refers to the payment method code files
Type	Enum(offline, online)	Indicates if the payment method is for online or offline payment.

Summary

In this chapter we have taken our framework and extended the order process, so that the customer is now ready to make their payment.

We now check that the customer is logged into the store, and if they are not we allow them to log in or create an account. The customer can then set a delivery address, and their default delivery address is shown. So, if they choose to use that, they don't need to change anything. The customer can then select the payment method they wish to use to pay for the order. Finally, the customer can review their order, before confirming and placing the order. At any time during this process, the customer can move backwards and forwards between stages if they wish.

The next stage is to actually take payments from our customers!

11
Taking Payment for Orders

Now that our checkout process has been implemented, we are left with the final stage: payment. Up to this stage, our customers can place orders, go right through the order process, including selecting a payment method, but they cannot yet make payment for their order. In this chapter, you will learn:

- How to take payment online
    - Using PayPal
    - Using other methods
- How to take payment offline for our store
- How an order progresses once payment has been made

Taking payment

Taking payment is a fundamental aspect for any business; without payment, the business would not be able to function. With e-commerce stores, one of the benefits is that we can take and process payment instantly online, giving the customer reassurances that their order has been placed and will be processed, and also giving assurance to the store that payment has been made.

Our payment system

For our payment system to work, we will need to separate logic for each payment method out into a different set of files. For instance, if the customer opts to pay online using PayPal, we need to generate a PayPal payment button. If they opt to pay online, we need to display payment details to them. If we want to add new payment methods in the future, we need to be able to easily slot in a new payment method. Different payment gateways have different ways of receiving payments and of sending notification back to us that a payment has been made. This logic needs to be encapsulated in the separate gateway files.

The *best* way to do this would be to create a single payment processor object, and have each payment method extend and inherit from this object. However, this would require more research and investigation into other payment methods to get the optimal method. Instead, we will use the **factory pattern**; this will create a new payment gateway object depending on the payment method the customer chooses. The Factory pattern implementation is highlighted in the following code:

```
// get the order details
$orderId = intval( $urlBits[2] );
// we should really abstract this out into an order object
$sql = "SELECT o.*, p.`key` AS payment
        FROM orders o, payment_methods p
        WHERE p.ID=o.payment_method AND o.ID={$orderId}";
$this->registry->getObject('db')->executeQuery( $sql );
if( $this->registry->getObject('db')->numRows() > 0 )
{
  $data = $this->registry->getObject('db')->getRows();
  $method = $data['payment'];
  require_once FRAMEWORK_PATH . 'controllers/payment/methods/'
      . $method .'.php';
  $this->paymentMethod = new Paymentmethod();
  switch( $urlBits[1] )
  {
    case 'process-payment':
      $this->paymentCallback();
      break;
    case 'make-payment':
      $this->displayMakePayment();
      break;
  }
}
```

To display a payment page, we need to take our payment method object, and either display a form for credit card details, a PayPal payment button, or details for sending a payment through check in the post.

The following code takes out the payment method object, assigns appropriate template tags, and builds the template from a template specific to the payment method:

```
private function displayMakePayment()
{
  $paymentKey = $this->paymentMethod->getKey();
  // template tags please
  $this->paymentMethod->makePaymentScreen();
  $this->registry->getObject('template')->
```

```
        buildFromTemplates('header.tpl.php',
        'payment/'.$paymentKey.'.tpl.php','footer.tpl.php');
    $tags = $this->paymentMethod->getTags();
        $this->registry->getObject('template')->dataToTags( $tags, '');
}
```

When it comes to processing the payment, we simply call a payment processing method for our `paymentMethod` object.

```
$this->paymentMethod->processPayment();
```

Taking payment online

There are a number of different ways in which we can take payment online, using a number of different gateways, which we will discuss later in this chapter. However, for our Juniper Theatricals store, the owner wants to take payment through PayPal, because it is quick and easy to sign up to, you don't need a PayPal account to send payment, and it is well known. So let's look at integrating PayPal functionality into the store.

PayPal

PayPal is one of the most well-known payment gateways available, primarily through use with its parent company eBay. One of the main benefits of PayPal is the low cost barrier of entry to use it, and its simple standard payment system for website integration.

The payment button

PayPal's standard payment options rely on a payment button. We generate a form and a button on our site, which store details such as the order number, seller's PayPal details, and other information as hidden fields. When the customer clicks on the button, the data is posted to PayPal.

Our PayPal payment method object sets many of the variables required for this button to work.

```
private function makePaymentScreen()
{
    $this->registry->getObject('template')->getPage()->
        addTag('payment.email', $this->registry->
        getSetting('payment.paypal.email') );
    $this->registry->getObject('template')->getPage()->
        addTag('payment.currency', $this->registry->
        getSetting('payment.currency') );
```

```
    $test = ( $this->registry->getSetting('payment.testmode') ==
        'ACTIVE' ) ? '.sandbox' : '';
    $this->registry->getObject('template')->getPage()->
        addTag('payment.test', $test );
}
```

Of course, we need a template to display the payment button, where these variables and variables set by the payment system are inserted.

```
<h1>Pay for your order</h1>
```

The payment button image is actually a submit button, and so to submit, it needs to be within a form, with an action of the PayPal processing URL. The `payment.testmode` tag is populated with sandbox if we are running in test mode.

```
<form
    action="https://www{payment.testmode}.paypal.com/cgi-bin/webscr"
    method="post">
    <input type="hidden" name="cmd" value="_xclick">
```

The custom form field is where we store our order ID number—when PayPal notifies us of payment, we need this to process the payment.

```
    <input type="hidden" name="custom" value="{reference}">
```

PayPal needs to know our e-mail address, so it knows who to send the payment to!

```
    <input type="hidden" name="business"
           value="{payment.paypal.email}">
```

We also supply a name for the item and an item number.

```
    <input type="hidden" name="item_name" value="{sitename} Purchase">
    <input type="hidden" name="item_number"
           value="{siteshortname}-{reference}">
```

We supply the cost, so PayPal knows how much to charge the customer, and in which currency.

```
    <input type="hidden" name="amount" value="{cost}">
    <input type="hidden" name="no_shipping" value="1">
    <input type="hidden" name="no_note" value="1">
    <input type="hidden" name="currency_code"
           value="{payment.currency}">
    <input type="hidden" name="src" value="1">
```

Next we have our notification URL, which is where PayPal sends payment notification, the return URL where the customer is returned to once payment is made, and the cancel URL where the customer is returned to if they cancel payment.

```
<input type="hidden" name="notify_url"
       value="{siteurl}payment/process-payment/{reference}">
<input type="hidden" name="return"
       value="{siteurl}payment/payment-received">
<input type="hidden" name="cancel_return"
       value="{siteurl}payment/payment-cancelled">
<input type="hidden" name="lc" value="GB">
<input type="hidden" name="bn" value="PP-BuyNowBF">
```

Finally, we have the PayPal payment image button.

```
<input type="image"
       class="paypal-button"
       src="https://www.paypal.com/en_US/i/btn/x-click-but6.gif"
       border="0" name="submit"
       alt="Make payments with PayPal - it's fast, free and secure!" >
<img alt="" border="0"
     src="https://www{paymen.testmode}.paypal.com
         /en_GB/i/scr/pixel.gif"
     width="1" height="1">
</form>
```

Processing payment to update the order

Although the customer is sent back to a "thanks" or a "cancelled" page when the payment is sent, PayPal sends some POST data to a special callback page. We can develop this page to process the data, verify the transaction is valid, and mark an order as "paid".

First we take POST data, which PayPal has sent to us, and use it to structure our callback request, which we will use to verify the transaction is genuine (and not just someone with a specially formatted script to send these requests to us).

```
$postback = '';
foreach ( $_POST as $key => $value )
{
  $postback .= $key . '=' . urlencode( stripslashes( $value )) . '&';
}
$postback .='cmd=_notify-validate';

$header = "POST /cgi-bin/webscr HTTP/1.0\r\n";
$header .= "Content-Type: application/x-www-form-urlencoded\r\n";
$header .= "Content-Length: " . strlen( $postback ) . "\r\n\r\n";
```

Next we must check if our store is in test payment mode; if it is, we send our callback request to the PayPal sandbox, otherwise we use the live PayPal site.

```
// live payment or test payment?
if(  $this->registry->getSetting('payment.testmode') != 'ACTIVE' )
{
  $fp = fsockopen ('www.paypal.com', 80, $errno, $errstr, 30);
}
else
{
  $fp = fsockopen ('www.sandbox.paypal.com', 80, $errno,
      $errstr, 30);
}
if (!$fp)
{
  // debug point
}
else
{
  $request = $header . $postback;
  fputs( $fp, $request );
  while ( ! feof( $fp ) )
  {
    $response = fgets ( $fp, 1024 );
```

We check that PayPal's response to our callback is that the transaction is verified.

```
    if (strcmp( $respose, "VERIFIED" ) == 0 )
    {
      // transaction is verified!
```

We then check that there is a relevant order in our orders table, based off the custom POST value. This POST value is the same as the custom field we provided in our PayPal payment button, before the customer made payment.

```
      $order = intval( $_POST['custom'] );
      $sql = "SELECT FORMAT( ( o.products_cost + o.shipping_cost ),
                2 ) AS order_cost, u.email AS customer_email
              FROM orders o, users u
              WHERE u.ID=o.user_id AND o.ID={$order} LIMIT 1";
      $this->registry->getObject('db')->executeQuery( $sql );
      if( $this->registry->getObject('db')->numRows() == 1 )
      {
        // we have an order in our database
        $orderData = $this->registry->getObject('db')->getRows();
```

If the order exists, we must then check that the cost of the order matches the amount being sent and that the payment has actually been sent to us, and not sent to pay someone else, yet sending notification to our processing script. We must also check that the currency of the payment is the one we wish to accept—otherwise, someone may send us the correct value of payment, in a currency with a lower value.

```
$currency = $_POST['mc_currency'];
$total = $_POST['mc_gross'];
$email = $_POST['receiver_email'];
if( $orderData['order_cost'] == $total &&
    $currency == $this->registry->
        getSetting('payment.currency') &&
    $email == $this->registry->
        getSetting('payment.paypal.email') )
{
  if( $status == 'Completed' )
  {
    // We then update the order in the database, and can
    // then e-mail the customer and the administrator to
    // inform them of this.
    // update the order
    $changes = array( 'status' => 2 );
    $this->registry->getObject('db')->
        updateRecords('orders', $changes, 'ID=' . $order );
    // email the customer
    // email the administrator
  }
  elseif( $status == 'Reversed' )
  {
    // charge back
    // update the order

    // e-mail the customer
    // e-mail the administrator
  }
  elseif( $status == 'Refunded' )
  {
    // we refunded the payment
    // update the order

    // email the customer
    // email the administrator
  }
  else
  {
```

```
            // ...
          }
        }
        else
        {
          // amount incorrect or wasn't sent to us
        }
      }
      else
      {
        // error
      }
    }
  }
  fclose ($fp);
}
exit();
```

Direct with a credit/debit card

Sometimes a store may not want to use an off-site payment gateway; many high-profile stores such as Amazon, Play.com, most supermarkets, and high street stores, process payments directly on their websites. This generally works in one of two ways:

- Payment details are passed behind the scenes to a gateway to verify them. The site then stores the details until the order is processed, at which point it charges the card and dispatches the order.
- Payment details are passed, behind the scenes, directly to the payment gateway and are never stored on the website itself. This method either returns a response to indicate if the transaction was accepted or not (and if not, why not), or returning a secure token. This token can then be used by the store to charge the card at a later stage (also for recurrent billing), by simply passing details of the charge and the secure token to the gateway, which can then process the transaction.

Storing card details

Storing card details on our own server offers a range of flexibility. As we would be keeping a copy of the card details, we can charge the card when we require it (for example, recurring billing, charging only when we are ready to dispatch, service-based payments—we can charge the card to settle their account, and so on). If the gateway we use changes its pricing structure, or is unavailable, we have the details; so we can use another gateway, or process the payment when the gateway is back online.

Storing card details has one major drawback: security. The security implications of storing card details on a server are vast; if the website was compromised in terms of security, we could leave all of our customers vulnerable, and be liable for the damage. To assist with this, there are some compulsory guidelines imposed by credit card companies (and subsequently required and enforced by the gateways) for storing card details. These guidelines are the **Payment Card Industry Data Security Standards** (**PCI DSS**). The PCI DSS specifies six control objectives, which are:

- Build and maintain a secure network
- Protect cardholder data
- Maintain a vulnerability management program
- Implement strong access-control measures
- Regularly monitor and test networks
- Maintain an information security policy

These objectives and their associated requirements are assessed to validate compliance.

Further information on PCI DSS can often be obtained from payment gateways themselves, and also the PCI website (`https://www.pcisecuritystandards.org/index.shtml`).

Some web hosts have specialist hosting available, which ensures compliance from a server and network infrastructure perspective, and also makes it easy for other aspects to be verified. One example is the A Small Orange business hosting service—`http://asmallorange.com/hosting/business/`.

Not storing card details

If we don't store card details, we don't get as much flexibility as discussed earlier. There are generally two ways this works, we both pass the details and charge the card, or we pass the details to the gateway, obtain a token, and charge the card by passing the token and the amount to the gateway.

If we use this method, with a token we are tied to that gateway, as we can't pass the token to another gateway to charge the card, because they won't have a card associated with our token. However, this method does remove a lot of the concern regarding security, although the stance taken by gateways on if PCI DDS compliance is required (and if so, to what level), varies.

Other payment gateways

There are a number of other payment gateways available, including:

- SagePay
- NoChex
- Authorize.net
- 2Checkout
- Gateway
- WorldPay

Each of these gateways has different costs associated with them, and may have different advantages and disadvantages (for example, customers may be more comfortable using them, their dispute procedure may be too favorable to customers, and so on). More information on them can be found on their respective websites; however, I'd also recommend searching for reviews and details of experiences with them too.

Payment gateway tips

When looking into payment gateways, it is important to consider the following factors:

- Do you also need a special merchant bank account, and what is involved in setting one up (time, paperwork, costs, application process, and so on)?
- Monthly costs or a minimum monthly turnover through the gateway to keep the account active.
- Setup costs; some processors have high setup costs, but this may mean a lower monthly cost.
- Transaction costs; that is, how much of each transaction cost the gateway is going to keep to itself?
- Volume of transactions you are looking to process; some gateways offer reduced rates for higher transaction volume.
- Value of transactions you are looking to process; some gateways offer reduced rates for minimum monthly totals processed, others may not be cost effective when individual transactions are small.

With some gateways, you may be able to negotiate special rates; this is particularly true with bank-based gateways, especially in the UK.

Taking payment offline

Taking online payment is great; it means we can process orders quickly. However, not all customers want to pay online. For smaller, less-known e-commerce sites, customers may not trust supplying their card details. We may wish to enable customers without credit cards to make purchases from our store. This is where offline payment comes in.

When the customer confirms their payment method, and confirms the order, we simply mark the order as "pending payment", and inform the customer of how they can send payment, be this by check, in person, or perhaps through card over the phone, along with a reference number. Then when we receive the payment, we simply mark the appropriate order as "paid".

Summary

In this chapter, we have implemented the final stage of our order process: the payment. We now:

- Can take payment online using PayPal
- Have an understanding of how other online payment methods work
- Have an understanding of how to take payment direct with a credit or debit card
- Can take offline payments
- Have our store update orders automatically when payment is received

Now we can look towards developing the administration area for our store, including managing and fulfilling orders, dealing with customers, creating and managing products, and other settings, such as payment methods, shipping methods, voucher codes, and product filters.

12
User Account Features

Our customers can now view and search our store, place orders, and pay for them. This leaves us with two primary areas to cover: the user account and the administration area, before we have a store to use in a live environment. In this chapter, you will learn:

- How to create a user account area
- How to allow customers to change their details
- How to allow customers to change their password
- How to allow customers to see their orders
- How to allow customers to cancel orders

User account area

A user account area provides a central area for our customers to view and amend their details, apart from an area to see a history of their orders and their status. This is important as it allows customers to check on the status of their orders, which should be automatically updated, so they don't need to keep getting in touch with us to see if their order has been dispatched yet.

Changing details

Most user account areas allow customers to change their details, maybe they have a new e-mail address, wish to change their password, or have a new default delivery address for all future purchases. By allowing the customer to keep these details up to date, not only are we making this easier for them (they only need to change their default delivery address once, and it will remain the same for all future purchases), but we are also ensuring that our contact details for them are up to date. This means if we wish to send out e-mail newsletters, discount vouchers, and so on to our customers, we are more likely to have up-to-date details for them.

Changing password

The process for allowing a customer to change their password should be relatively simple:

1. The customer should enter their *current* password.
2. The customer should enter their *new* password.
3. The customer should enter their new password *again*.
4. We should then check that their current password is valid.
5. Then we should check that their new password and their confirmation of this new password are the same.
6. We should then hash the new password and update the user's password in the database to this hash.

Let's look at the code for processing this. (Steps 1 to 3 are user actions, steps 4 to 6 are system actions, which we must provide in our code.)

```
$oldPassword = md5( $_POST['old_password'] );
// check their new password is confirmed
```

We compare the password entered in the password box, with the password they entered in the confirmation box, to ensure they match—if they don't and we change the password to one of them, the customer may not know what the corrected password is—as it would seem they made a mistake entering in one of the password boxes. This is essentially a precaution for the customer's benefit.

```
if( $_POST['new_password'] == $_POST['confirm_newpassword'] )
{
  $uid = $this->registry->getObject('authenticate')->getUserID();
  $checkPwdSQL = "SELECT * FROM users
                  WHERE ID={$uid} AND password='{$oldPassword}'";
  $this->registry->getObject('db')->executeQuery( $checkPwdSQL );
  // check their old password is valid
```

As it is possible that the customer left their account logged in, and someone else could be at their computer trying to change their password, we need to confirm that their existing password was entered, verifying it is the customer requesting the change.

```
  if( $this->registry->getObject('db')->numRows() == 1 )
  {
    $changes = array();
    $changes['password'] = md5( $_POST['new_password'] );
    // update their current password to the new password
```

We then update the database to save the customer's new password.

```
      $this->registry->getObject('db')->updateRecords( 'users',
                                                $changes, 'ID=' . $uid );
      // output success message here
    }
    else
    {
      // do our error output here
    }
  }
  else
  {
    // error output
  }
```

Changing default delivery address

Allowing customers to change their default delivery address should be very straightforward; we need to check if they submitted some changes, sanitize the data, and update their details.

```
  private function saveChangesToAccount()
  {
    // default delivery address
    $changes = array();
    // We set each array element to be part of the customer's delivery
    // address, and set the value to the one the customer is
    // submitting.
    $changes['default_shipping_name'] = $this->registry->
        getObject('db')->
        sanitizeData( $_POST['default_shipping_name'] );
    $changes['default_shipping_address'] = $this->registry->
        getObject('db')->
        sanitizeData( $_POST['default_shipping_address'] );
    $changes['default_shipping_address2'] = $this->registry->
        getObject('db')->
        sanitizeData( $_POST['default_shipping_address2'] );
    $changes['default_shipping_city'] = $this->registry->
        getObject('db')->
        sanitizeData( $_POST['default_shipping_city'] );
    $changes['default_shipping_postcode'] = $this->registry->
        getObject('db')->
        sanitizeData( $_POST['default_shipping_postcode'] );
    $changes['default_shipping_country'] = $this->registry->
        getObject('db')->
```

```
            sanitizeData( $_POST['default_shipping_country'] );
        // We then update their delivery address, by making changes to the
        // users_extra table.
        $this->registry->getObject('db')->updateRecords( 'users_extra',
            $changes, 'ID='
            . $this->registry->getObject('authenticate')->getUserID() );
        // We then update the users table, to update the customer's e-mail
        // address.
        // e-mail address
        // The format of the e-mail address should be checked ideally
        $changes = array();
        $changes['email'] = $this->registry->getObject('db')->
            sanitizeData( $_POST['email'] );
        $this->registry->getObject('db')->updateRecords( 'users',
            $changes, 'ID=' .  $this->registry->
            getObject('authenticate')->getUserID() );
        $this->registry->redirectUser( 'useraccount',
            'Account details changed', 'Your account details have been
             saved', $admin = false );
    }
```

Viewing orders

The orders section of a customer's account area should be broken into two areas:

- List of orders, so the customer can select the order they wish to see in more detail
- Viewing order-specific details

Listing orders

To list a customer's order, we simply need a single query to lookup their orders and their status, which we can then cache and send to the template. Within the list of orders, we would want to display the following information:

- Order ID or reference
- The total cost of the order
- The status of the order
- The date the order was placed
- We may also wish to display the date the order was last updated, which could indicate when the order was dispatched, or cancelled, or payment was processed

Query

The following query would allow us to list all of the user's orders, detailing the time the order was placed, the ID, the cost of products, the cost of shipping, and the status of the order:

```
"SELECT os.name AS status_name, o.ID AS order_id,
    (o.products_cost + o.shipping_cost) AS cost,
    DATE_FORMAT(o.timestamp, '%D %b %Y') AS order_placed
 FROM orders o, order_statuses os
 WHERE os.ID=o.status AND o.user_id=" . $this->registry->
    getObject('authenticate')->getUserID()
```

Viewing an order

For viewing an order, we should create a model to encapsulate all of the data for the order; this should also make things easier for us in the next chapter when we create our administration interface.

Order model

Our order model needs to allow us to access the following information:

- The customer
- The delivery address
- The items in the order
- The total cost of the order
- The shipping cost of the order
- If a discount voucher was used
- The status of the order
- The date the order was placed

The model requires two queries to get this data: the first to get the order data itself, and the second to get the products in the order.

Query for order details

This simple query needs to get details stored in the orders table, as well as the status of the order, the payment method, and the shipping method:

```
$sql = "SELECT o.timestamp, o.user_id, o.comment, o.delivery_comment,
        o.shipping_name, o.shipping_address, o.shipping_address2,
        o.shipping_city, o.shipping_postcode, o.shipping_country,
        o.products_cost, o.shipping_cost, o.voucher_code, os.name
```

```
        AS order_stauts, p.name AS payment_method, s.name AS
        shipping_method "
        . "FROM orders o, payment_methods p, order_statuses os,
        shipping_methods s "
        ."WHERE o.ID={$id} AND os.ID=o.status AND
        s.ID=o.shipping_method AND p.ID=o.payment_method "
        ."LIMIT 1";
```

Once the query has been executed and if there is a result, then we can set the model to `valid`, and populate its properties with data from the database. If no records are found, we must set the validity of the model to `false`.

```
$this->registry->getObject('db')->executeQuery( $sql );
if( $this->registry->getObject('db')->numRows() > 0 )
{
  $this->valid = true;
  $orderData = $this->registry->getObject('db')->getRows();
  $this->id = $id;
  $this->status = $orderData['status'];
  $this->created = $orderData['timestamp'];
  $this->deliveryAddress = array( $orderData['shipping_name'],
      $orderData['shipping_address'],
      $orderData['shipping_address2'],
      $orderData['shipping_city'],
      $orderData['shipping_postcode'],
      $orderData['shipping_country'] );
  $this->productsCost = $orderData['products_cost'];
  $this->shippingCost = $orderData['shipping_cost'];
  $this->orderCost = $this->productsCost + $this->shippingCost;
  $this->shippingMethod = $orderData['shipping_method'];
  $this->paymentMethod = $orderData['payment_method'];
  $this->user = $orderData['user_id'];
  $this->comment = $orderData['comment'];
  $this->deliveryNote = $orderData['delivery_comment'];
  // items
  // items query to go here
}
else
{
  $this->valid = false;
}
```

Query for order items

The second query for the model is to get the individual items in the order. The following is a basic query which does this, although this does not take into account variants of a product that we may have been purchased:

```
$sql = "SELECT ctp.price, (ctp.price*i.qty) AS cost, i.qty, v.name,
               i.product_id
        FROM content_types_products ctp, orders_items i,
             content c, content_versions v
        WHERE i.order_id={$this->id}
              AND c.ID=i.product_id AND v.ID=current_revision
              AND ctp.content_version=v.ID";
```

The results of this query should be cached, so that any of our controllers can send the results directly to a view by associating the cache ID with a template variable.

Cancelling an order

There may of course be times when a customer wishes to cancel an order, for reasons such as maybe we were too slow to dispatch it, perhaps they changed their mind, or perhaps they ordered mistakenly. While allowing a customer to cancel an order will obviously lose us a sale, it keeps our potential customers happy, and should hopefully help our store's reputation.

Up until the point where we dispatch an order, it should be very easy for a customer to cancel the order. Of course, we may want to expand this later to handle returns and authorizing returns from customers.

For the customer to cancel an order there should be a few simple stages:

1. First they should view the order.
2. Next they should select an option somewhere to cancel the order.
3. They should be able to enter a comment or note about why they are cancelling the order.
4. They should confirm their wish.
5. The order should be cancelled.
6. An e-mail should be sent to them to confirm their order was cancelled (particularly useful if they cancelled it mistakenly).
7. Payment should be refunded (automatically, if the payment method allows it).
8. An e-mail should be sent to the store administrator.

This requires some functionality in our order model to cancel the order, and also some code for our user area controller.

Order model additions

There are a few additions required to our order model, so that we can change the functionality of the cancel order method. Depending on whether the administrator or the customer cancels the order, we can use a parameter to indicate who is cancelling the order.

```
public function cancelOrder( $initiatedBy )
{
  // Only orders that are awaiting payment or awaiting dispatch can
  // be cancelled, so we must check the status of the order first.
  // is the order pending payment or dispatch i.e. cancellable?
  if( $this->status == 1 || $this->status == 2 )
  {
    // We then update the order item in the database to cancelled.
  $changes = array( 'status' => 4 );
  $this->registry->getObject('db')->updateRecords('orders', $changes,
      'ID=' . $this->id );
    // If the order was cancelled by the customer, we then need to
    // e-mail the administrator, e-mail confirmation to the customer,
    // and if we can, refund the payment.
    if( $initiatedBy == 'user' )
    {
      // e-mail the administrator
      // e-mail the customer confirmation
      // refund the payment?
    }
    // If the order was refunded by the administrator we e-mail the
    // customer to inform then, and if possible, refund the payment.
    elseif( $initiatedBy == 'admin' )
    {
      // e-mail the customer
      // refund the payment?
    }
    // We return true, so that the relevant controller can display a
    // message indicating that the order was cancelled.
    return true;
  }
  else
  {
    // order isnt cancallable
```

We return `false` to indicate that the order was not cancelled, allowing our controller to inform the user of this.

```
      return false;
    }
  }
```

Controller code

To facilitate cancelling the order, we need to have two functions in our controller: one to display a confirmation message and another to actually cancel the order.

The following function checks whether the order is valid and belongs to the current user; if it does, it displays a confirmation message:

```
  private function confirmCancelOrder( $orderId )
  {
    $this->registry->getObject('template')->
        buildFromTemplates('header.tpl.php',
        'account/confirm-cancel.tpl.php','footer.tpl.php');
    require_once( FRAMEWORK_PATH . 'models/order/model.php');
    $this->order = new Order( $this->registry, $orderId );
    if( $this->order->isValid() )
    {
      if( $this->order->getUser() == $this->registry->
          getObject('authenticate')->getUserID() )
      {
        $this->registry->getObject('db')->getPage()->addTag('orderid',
            $orderId );
      }
      else
      {
        $this->registry->redirectUser( 'useraccount', 'Invalid order',
        'The order was not cancelled as it was not tied to your
         account', $admin = false );
      }
    }
    else
    {
      $this->registry->redirectUser( 'useraccount', 'Invalid order',
          'The order was not found', $admin = false );
    }
  }
```

When the customer clicks on the confirmation link within the confirmation page, the following function is called; this again validates whether the order belongs to the customer, and then cancels it using the model (which in turn, checks if the order can be cancelled, and sends any relevant e-mail notifications):

```
private function cancelOrder( $orderId )
{
  require_once( FRAMEWORK_PATH . 'models/order/model.php');
  $this->order = new Order( $this->registry, $orderId );
  if( $this->order->isValid() )
  {
    if( $this->order->getUser() == $this->registry->
        getObject('authenticate')->getUserID() )
    {
      $this->order->cancelOrder('user');
      $this->registry->redirectUser( 'useraccount', 'Order
          cancelled', 'The order has been cancelled',
          $admin = false );
    }
    else
    {
      $this->registry->redirectUser( 'useraccount', 'Invalid order',
      'The order was not cancelled as it was not tied to your
       account', $admin = false );
    }
  }
  else
  {
    $this->registry->redirectUser( 'useraccount', 'Invalid order',
        'The order was not found', $admin = false );
  }
}
```

Expansion

As our needs for our store grow, we can expand this area of the framework; perhaps we would like it to offer:

- **Returns handling**: This will let our customers indicate if they wish to return an item. The store administrator can provide a returns authorization number, and the return can be processed, with the customer being updated.
- **Product recommendations**: This will let us display products that are recommended to the customer, based on previous orders.

- **A feedback area**: This will let us improve the framework by collecting the general feedback from customers.
- **Exclusive discounts**: Incentives for customers!
- **Advance notice on pre-releases**: This will let us entice customers to make pre-orders.

Summary

In this chapter, we have created a centralized customer area, which allowed customers to update their password, update their default delivery address, and list their orders and their statuses. We have also created an orders model, which we used to allow customers to view individual orders, as well as cancel existing orders. We also looked into how we might expand the customer area to make it better, providing more value to the customer.

This provides a nice, convenient place for the customer to manage their account, the information stored by our site on them, and see an overview of all their pending and processed orders at a glance, to see what is happening with them.

13
Administration

Although our journey through creating an e-commerce framework is not yet at an end, we are at the last primary feature: the administration area. In this chapter, you will learn how to enable administrators to:

- Create a product, taking into account photographs and shipping
- Edit a product
- Create, edit, and manage categories
- View and process orders
- View customer profiles
- View and create shipping methods and their corresponding rules
- Create voucher codes

A suitable administration area can be broken down into four primary areas:

- **A dashboard**: This area is used to provide administrators with a brief overview of the store at any one time.
- **Products and categories**: This area is used to allow administrators to view, create, edit, and delete products and categories.
- **Orders and customers**: This area is used to allow administrators to view and process orders, as well as view customer profiles.
- **Miscellaneous**: This area could be used as somewhere for administrators to change any settings, manage shipping methods, shipping rules, and voucher codes.

Dashboard

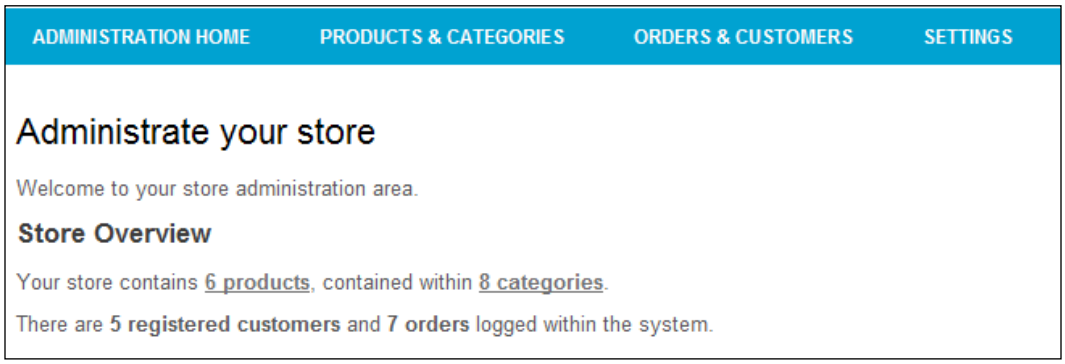
The dashboard should provide the administrator with an overview of the store; this could include statistics such as:

- The number of products and the number of categories they are contained within
- The number of customers
- The number of orders that are awaiting dispatch (that is, orders that the administrator needs to process)
- Number of orders placed within the past 24 hours
- The number of abandoned shopping baskets in the past 24 hours
- The average cost of orders
- Graphs and charts illustrating some of these statistics

Let's start with the basic statistics; we can generate most of the statistics from a query, breaking the individual statistics into subqueries.

```
$ts = date('Y-m-d h:i:s', strtotime('-1 day'));
$sql = "SELECT
  (SELECT COUNT(*) FROM content c, content_types t
   WHERE c.type=t.ID AND t.reference='product') AS num_products,
  (SELECT COUNT(*) FROM users) AS num_customers,
  (SELECT COUNT(*) FROM orders o, order_statuses os
   WHERE o.status=os.ID AND os.name='Awaiting Dispatch')
  AS num_orders_pending_dispatch,
  (SELECT COUNT(*) FROM orders o, order_statuses os
   WHERE o.status=os.ID AND os.name='Awaiting payment')
  AS num_orders_pending_payment,
  (SELECT COUNT(*) FROM orders o WHERE o.timestamp>'{$ts}')
  AS num_orders_placed_24,
  (SELECT FORMAT(AVG(products_cost),2) FROM orders)
  AS avg_order_cost
FROM orders o LIMIT 1";
```

A sample dashboard screen is shown below:



We could extend and enhance this by adding some of the following:

- Support for logging abandoned shopping baskets
- Graphs and charts

Of these, the latter could probably be added using the Google Charts API or an alternative suitable third-party chart service.

Products and categories

For us to be able to sell anything on our site, we need to allow administrators to create products in the store. We also need to be able to edit and delete products, as well as create, edit, and delete categories.

Products

Let's look at how we can allow administrators to create, edit, and delete products within our store.

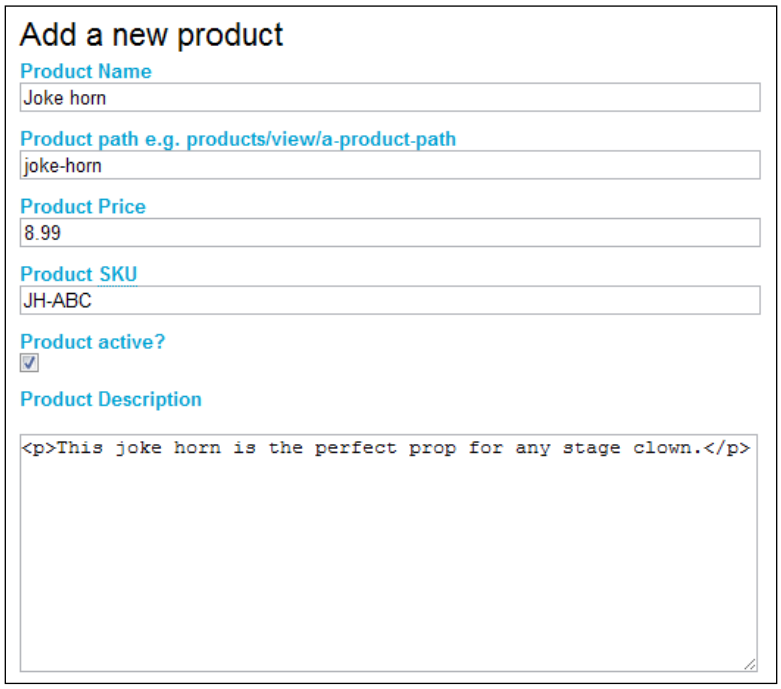
Creating a product

To create a product in our store, we need to:

- Save standard information about the product such as the price, the name, the description, and so on.
- Upload, resize, and save a photograph of the product.
- Upload additional photographs of the product.

- Save a shipping cost to be associated with the product, for each shipping method we have in the store.
- Save the categories the product is part of.
- Save any configuration options, such as if the customer can upload a file when placing the order, if the customer can enter any custom text, if the product has a number of variants, particularly for apparel, for example red, blue, and green t-shirts.

A form such as the partial one below captures the relevant information:



Product photograph

When processing the create product form submission, we need to upload a photograph and resize it to have a thumbnail and to keep the larger image consistently sized.



Photograph resizing with PHP

When resizing images with PHP, we need to use the function `imagecreateresampled` as opposed to `imagecreateresized`. If we use the latter, we will end up with distorted images.

The process of uploading an image involves:

1. Verify if the file was uploaded (that is, verify that someone isn't trying to upload a file already on the server). This could be done using:

```
is_uploaded_file( $_FILES[ 'field' ]['tmp_name'] )
```

2. Verify the extension of the file for that of an image.
3. Check that the type of upload is that of an image. This could be done using:

```
private $uploadTypes = array( 'image/gif', 'image/jpg',
    'image/jpeg', 'image/pjpeg', 'image/png' );
if( in_array( $_FILES[ 'field' ]['type'], $this->uploadTypes ) )
```

4. Move the uploaded file, using:

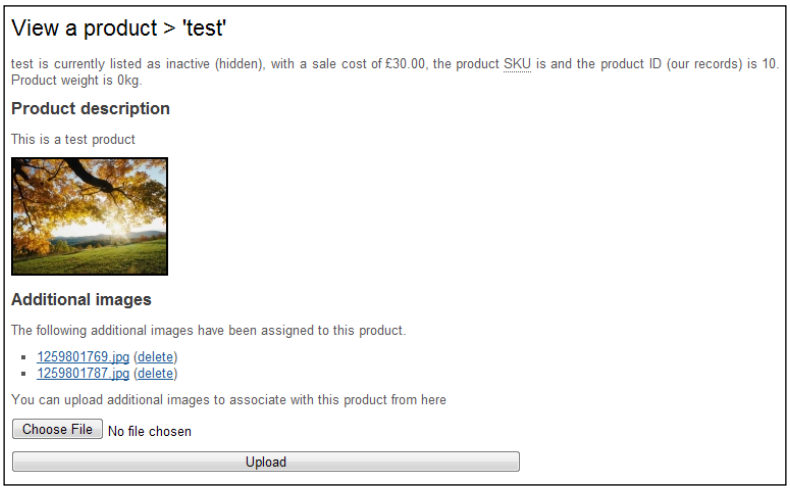
```
move_uploaded_file( $_FILES[ 'field']['tmp_name'] , $path );
```

5. Resize the image. The functions used for this vary depending on the type of the image—we do this by scaling the height based on a defined width.

```
$new = imagecreatetruecolor($x, $y);
imagecopyresampled($new, $theimage, 0, 0, 0, 0, $x, $y,
    $newwidth, $newheight);
imagejpeg( $new, $location, $quality);
```

Additional photographs

Although the product has a primary photograph associated with it, which we have just discussed, often products need to have a number of photographs to fully show off the product to the customer. The product details on the edit page could be shown along with the primary image, a list of additional images, and the additional image upload form as follows:

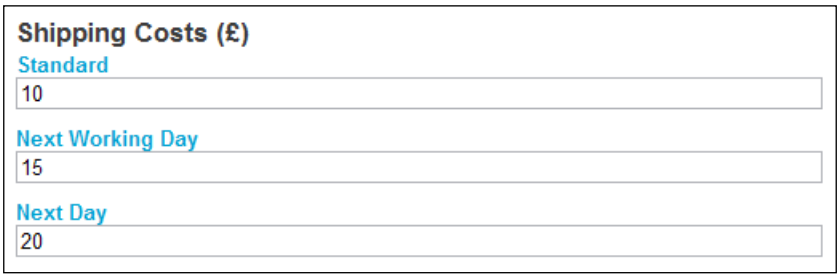


Shipping costs

When creating the product, we must save a shipping cost to be associated with each shipping method in the framework. This stage actually needs to wait until we have created the product record in the database, as we would:

1. Take note of the product ID.
2. Query the shipping methods in the framework and store the results in an array.
3. Iterate through the array of shipping methods:
    - Lookup the value of a shipping cost field, which is suffixed with the ID of the shipping method.
    - Store the shipping cost in the shipping costs table, referencing the product ID, the shipping method ID, and the shipping cost.

As illustrated below, the shipping methods should be listed once for each product, with their corresponding default price:



Categories

This is something else which has to wait until the product has already been created; for each category checkbox that has been checked, we create an associated record in the database table, which relates products to categories.



Customizable products

Finally, as some of our products can be customized by the customer before they place their order, we need to take some options into account; these options include:

- Variations of the product, for example sizes and colors
- If the customer can upload a file
- If the customer can enter any free text, and if so, what the free text fields should be called, and how many of them there should be

Editing a product

Editing a product should be very similar to creating a product. We need to take all of the same aspects into account, and update the relevant database records accordingly, and where appropriate, upload new images.

When editing a product, we would want all of the product information to be pre-populated in the edit form. This includes pre-checked boxes indicating which categories the product belongs to, and textboxes for each shipping method.

Save existing or new variant

One very useful timesaver would be to allow the administrator to create a new product based on an existing product. To facilitate this, we could have an option when saving changes to a product to either save the changes to the existing product, or to create a new variant of the product—in which case, we would then need to actually create a new product from the submitted data.

Categories

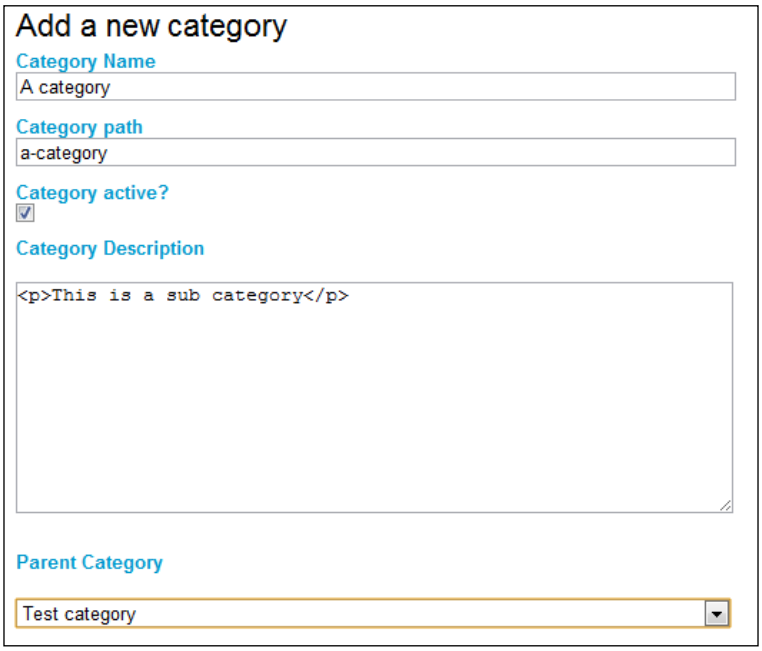
As we associate products with categories and our customers can browse these categories, we also need to be able to administer categories from the administration area.

Creating a category

Creating a category is just a case of:

- Storing the name of the category
- Storing the parent category
- Storing the order the category should display within a hierarchy of other categories
- Generating the search engine-friendly URL for the category

The following form captures this data for a new category:



Editing a category

Again, editing a category is very simple. It is just a case of updating the same details we stored when creating the category.

Deleting a category

Deleting a category requires two stages: first we need to delete the category from the database, and next we need to delete all product category associations with this category.

Orders and customers

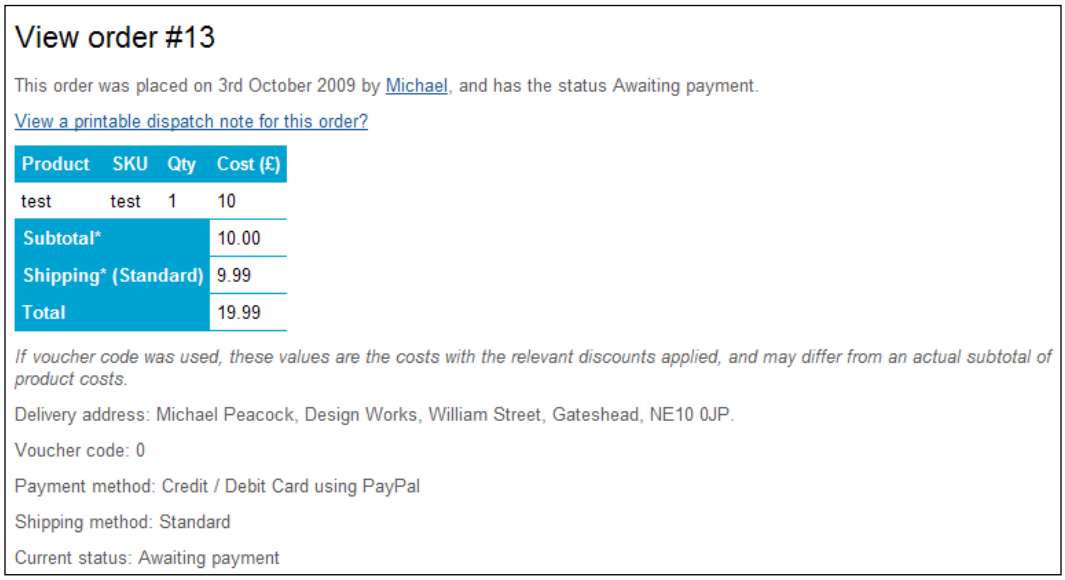
Now that our administration area can create and manage products, we need to be able to view orders and customers so that we can actually fulfill orders.

Orders

First, let's look at orders; we need to be able to:

- View an order
- Update (that is, process) an order, and inform the customer
- Print a dispatch note
- Process refunds where appropriate

The following screenshot illustrates viewing an order, displaying the status, the date it was placed, customer, products, delivery details, payment details, and shipping details:



Updating an order

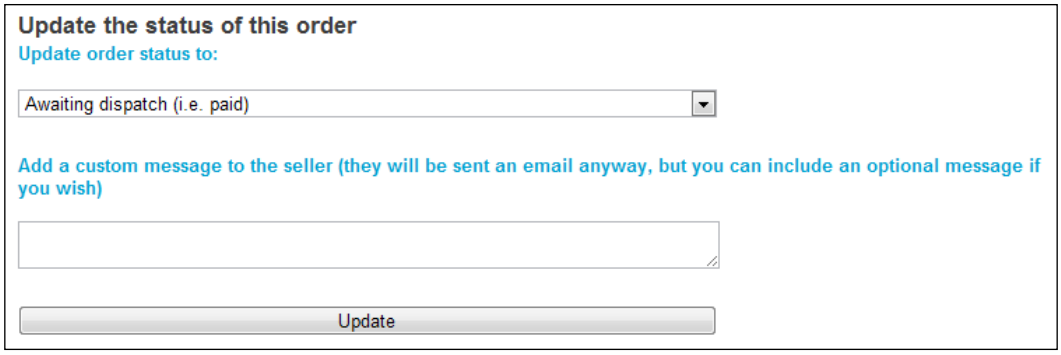
When we update an order, we cannot rely on the customer checking their user account area to see that the status of the order has changed. Instead, we should e-mail the customer automatically to inform them of the change in order status.

When dispatching orders, we may often want to inform the customer of a tracking code, so they can contact the courier to see where their order is, and to get a delivery estimate. To accommodate this, the order update area should have a drop-down list of all of the possible order statuses and a free text area for the administrator to enter some text.

Examples of messages the administrator may need to leave for a customer include:

- Your shipment's tracking number is XXX.
- Your payment was rejected. Please make alternative arrangements.
- As requested, we have cancelled your order and a refund has been issued.
- Your order is currently being processed. Due to a high volume of orders lately, this may take a day or two. Please accept our apologies for the delay.

This simply requires two fields on a form, the order status and a custom message box, as illustrated below:



Dispatch note

When it comes to dispatching customers' orders, a dispatch note would be very useful. This is simply a duplication of the view order screen, with different information in different areas so that the note can be printed and folded to fit within a dispatch note envelope, displaying the delivery address suitably.

Refunds

Depending on our payment method, we may be able to automatically refund a customer with a single click. If this is the case, we could integrate that with the order update screen, so that when we select that an order has been refunded, a refund is automatically issued (if the payment method supports that); or alternatively, if we view an order and click on a refund button, it could then automatically update the order to "refunded".

Customers area

A simple customers area would be useful for listing customers within the store (which can be searched too). This way, if we or the store administrators need to contact a customer, we can simply find the customer from the list, and view their details such as their name, e-mail address, and default delivery address. We could also view all of the orders associated with that customer at one place. If a customer loses their confirmation details and needs to send offline payment, this area would help the store administrator deal with a telephone or e-mail enquiry asking for the customer's order reference number.

Listing customers

The following SQL could be used to list all of our customers:

```
SELECT u.ID, e.default_shipping_name, e.default_shipping_city,
       u.ID, u.email FROM users u, users_extra e
WHERE u.active=1 AND e.user_id=u.ID ORDER BY u.ID ASC
```

We would, of course, wish to paginate the results of this query, limiting the page to 20 or so records, allowing us to navigate through the list, 20 at a time.

A customer's orders

We already have a query which does this for us from the user account feature; all we need to do is change how we detect the selected user's ID.

```
"SELECT os.name AS status_name, o.ID AS order_id,
        (o.products_cost + o.shipping_cost) AS cost,
        DATE_FORMAT(o.timestamp, '%D %b %Y') AS order_placed
 FROM orders o, order_statuses os
 WHERE os.ID=o.status AND o.user_id=" . $customer;
```

Miscellaneous

Finally, we come to the miscellaneous section of the administration area; this can house any number of aspects that can't be as easily classified as the other features.

Shipping

With regards to shipping, we need to be able to create a shipping method, manage existing shipping methods, and create and manage shipping rules.

Creating a shipping method

Creating a shipping method should be quite simple, with one minor complication. Our shipping method just needs to store some basic information:

- Name
- Default shipping cost
- If the method should be the default shipping method

One important aspect we need to take into account when creating a shipping method is that we need to create an associated shipping cost for it for each product in the store. This is where the default cost comes into play—if we know the method will have an average cost of $5 for most products in the store, we create it with that default. This way, each product will automatically get a shipping cost of $5, and we can manually update products where this isn't correct. It also means that when we create new products, this default will be there for the new product.

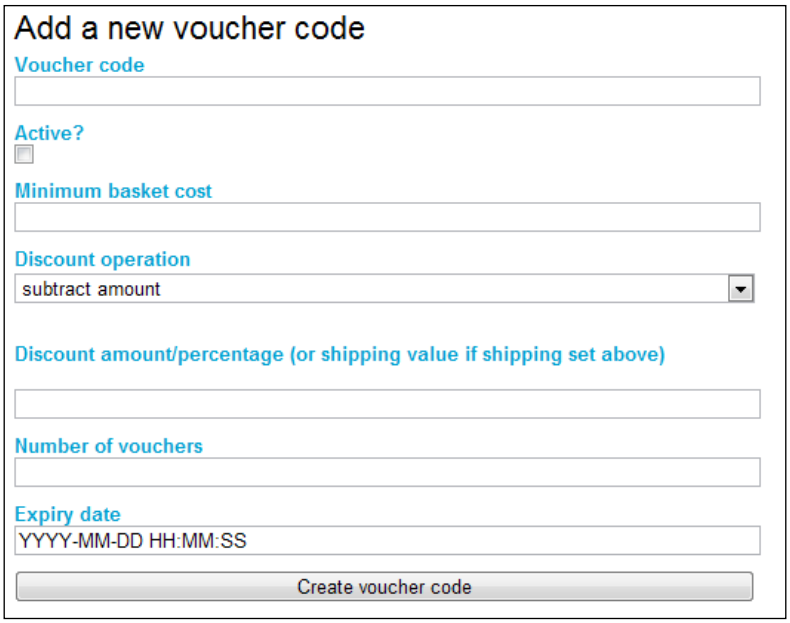
If the administrator selects the option for the shipping method to be the default shipping method, we must update the shipping methods table to update any method already set to be default, to ensure it is no longer the default.

Voucher codes

The other feature we have is voucher codes. We need to be able to create and manage voucher codes from within the administration area.

Creating a voucher code

Creating a voucher code should be straightforward: we simply need to take the information from the create voucher form, and insert it into one record in the voucher codes table. The only complication is that we need to verify and convert the format of the expiry date.



Summary

In this chapter, we have created a centralized administration area that allows administrators to:

- Get an overview of the store
- Create new products
- Edit existing products either by saving changes or creating a product based off another product
- Delete products
- Manage categories within the store
- View and manage orders placed within the store, updating customers as we update the orders
- View customer profiles, their details, and orders associated with their accounts
- Manage other settings such as shipping methods, shipping rules, and voucher codes

In the next chapter, we will look at the security and maintenance of our store, looking at deploying our framework, securing it with SSL certificates, and backing up and restoring our framework in the case of an emergency.

14
Deploying, Security, and Maintenance

We now have a fully functioning e-commerce framework, suitable for use in a live environment. Now it is time for us to look at deploying sites using the framework into a live environment and to examine security and maintenance concerns. In this chapter, you will learn:

- How to deploy the framework into a production environment
- Different ways we can enhance security with our framework
- How to maintain a site running the framework
- How to back up live sites
- How to restore a live site from a backup

Deploying

The process of deploying a site to a production environment generally involves the following steps:

1. **Registering a domain name**: This will allow our customers to access our website through a web address.
2. **Setting up a hosting account**: We need a hosting account so that the files for our website have somewhere to live.
3. **Changing the nameservers on a domain**: We make sure the web address points to our hosting account.
4. **Creating a database on the hosting account**: We need this to store our product catalog, orders, and other content somewhere.

5. **Importing a database to the database on the hosting account**: This is done because our database should have all of the products and content we want.
6. **Uploading our files to the hosting account**: As a consequence, when customers visit the website, they see our store.
7. **Changing the site's configuration files and any relevant database settings**: This sets the framework to use the database on the hosting account.

Hosting accounts and domain names

When it comes to web hosting and domain names, there are a large number of providers and registrars available. This helps to ensure that prices are competitive. When looking for a web host, there are a number of factors that must be taken into account when making a decision, including:

- The amount of web space required
- The amount of bandwidth required (data transferred from the web server to customers and other visitors per month)
- Any service-level agreements in place, such as a guaranteed uptime
- Minimum contract term
- Acceptable usage policy, to ensure they don't prohibit any of the functions of our e-commerce website
- To have software installed on the server, we obviously require PHP, MySQL, and Apache with the `mod_rewrite` module
- Of course, the cost of the hosting

Web-based control panels, such as cPanel or Plesk, are included with most standard web hosting accounts. This makes many administrative tasks easier, including:

- Setting up and managing e-mail accounts
- Setting up and managing databases
- Viewing statistics, access, and error logs
- Performing backups, restoring from backups, and so on

One of the most common control panels is cPanel, and is included with most shared hosting and **Virtual Private Server** (**VPS**) providers. Some aspects of this chapter contain instructions specific for cPanel (manual deployment, and backing up and restoring), along with alternative instructions for power users using the command line (assuming SSH access is enabled on the hosting account).

Packt Publishing has a book available specifically for cPanel, should you be interested in learning more about it: *cPanel User Guide and Tutorial* by *Aric Pedersen* (`www.packtpub.com/cPanel/book`).

Hosting providers

Some popular web hosting providers include:

- 1&1 Internet Inc. (`www.1and1.com`), they provide shared hosting accounts, virtual servers, and dedicated servers for larger websites and web applications. However, be careful as their lower-end shared hosting accounts don't support databases, such as MySQL.
- A Small Orange (`www.asmallorange.com`), who also provide shared hosting accounts, virtual servers, and dedicated servers. They also have a business hosting package, which contains useful features specifically for e-commerce sites.
- MediaTemple (`www.mediatemple.net`) is a provider of scalable virtual servers, with a control panel to make things as simple as with standard shared hosting accounts.
- Slicehost (`www.slicehost.com`) is a Virtual Private Server provider, designed for developers with functionality to easily upgrade and downgrade server capacity.



Research hosting providers

Web Hosting Talk (`www.webhostingtalk.com`) is a popular discussion forum focusing on discussing the web hosting industry, and containing many reviews and comparisons. It is worthwhile taking some time to research for the different providers before signing up with one.

Things to consider when looking for a hosting provider for e-commerce websites include:

- Are websites backed up regularly automatically?
- What security measures are in place?
- Do they offer SSL certificates and additional IP addresses, so we can enable secure areas of the website?

Domain name registrars

In order to get a web address that can point to our website, such as `www.juniperтheatricals.com`, we need to register it through a domain name registrar. Some popular registrars include:

- NameCheap (`www.namecheap.com`)
- GoDaddy (`www.godaddy.com`)
- 123-reg (`www.123-reg.co.uk`)

These registrars make it easy to register a domain name. However, often registrars pre-select several names when making a purchase (for example also selecting `.net` or `.eu` domain names), so be careful to not purchase more than you want. Domains are generally registered for at least one or two years at a time, depending on the type of domain. It is vital to remember to renew domain names, or we risk losing them.

Nameserver changes

Once we have our domain name registered, and a hosting account setup, we need to change the nameservers of our domain to those of our hosting provider. This ensures any traffic to our domain name is directed to our hosting account.

Manual deployment

The most straightforward way to get our files and database set up on our server or hosting account is by manually transferring the data to that hosting account.

Setting up the database



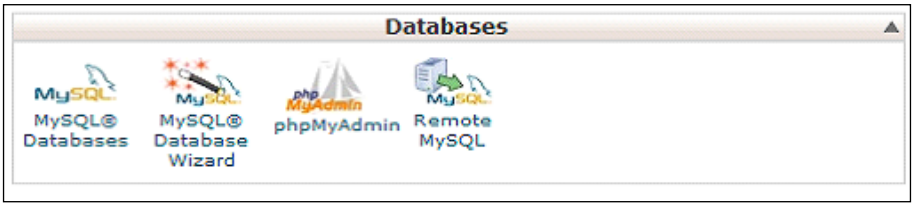
This section assumes a hosting account with cPanel installed.

To set up the database, we need to:

1. Create a MySQL database on the server/hosting account so that we have a database our framework can interact with.
2. Export a copy of the database for our site so we can transfer it to the live site.
3. Import that database into the database on the hosting account so that our framework can interact with the suitably structured and populated database.

Creating a database on the hosting account

The first stage is to log in to our control panel (this is usually, `www.yourdomain.com/cpanel`), and within the **Databases** section click on the **MySQL® Database Wizard** icon.



Next we enter a name for the new database; this is normally then combined with the hosting accounts username, so the database name store would become `junipert_store`. Once we have entered a name, we need to click on **Next Step**, to move on to the next stage of the database wizard.

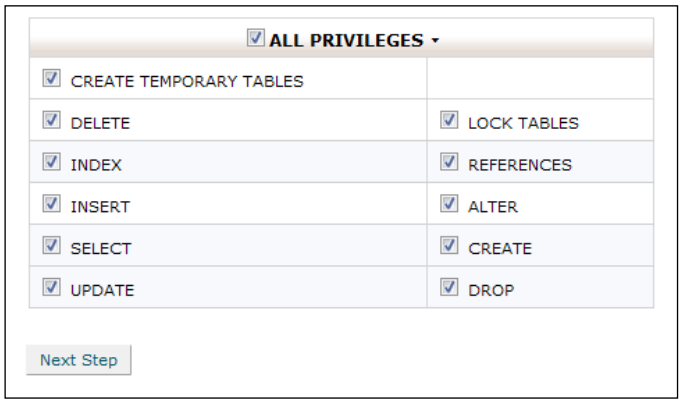


Then we need to create a user within MySQL, which will connect to the database server to access the database we have just created. It is important to use a secure password for this; click on the **Generate Password** button to have cPanel automatically generate a secure password for us.

Once we have entered the username and password, we need to click on the **Next Step** button.



Now that we have a database and a database user, we need to grant permissions for that user to be able to manage the database. Let's check the **ALL PRIVILEGES** check box and click on the **Next Step** button again.

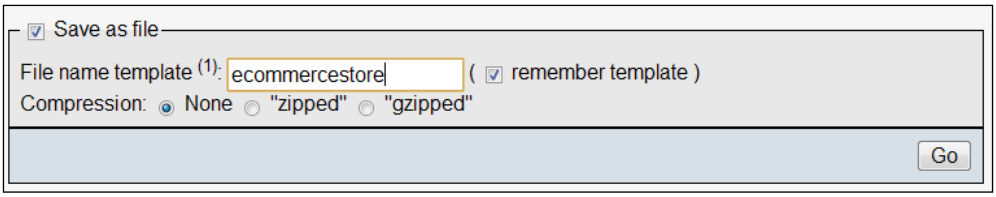


Exporting our local database

As we now have a database set up on the server, we need to get a copy of our local database, which we will import into this. To do this, we need to navigate to phpMyAdmin in our development environment (`http://localhost/phpmyadmin/`), select the database, and then click on the **Export** tab.



From here, we then tick the **Save as file** box, and click on **Go.**



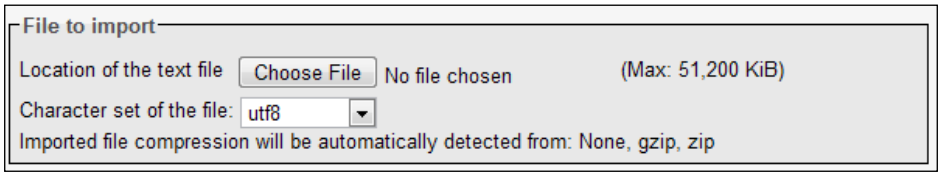
This generates an export file of the database for us to use elsewhere.

Importing the local database to the hosting account

From within phpMyAdmin, we need to select the **Import** tab so that we can import the database.



We can browse from here to the database file on our computer, using the **Choose File** button, and then click on the **Go** button at the bottom of the page to import the database to the hosting account.



We now have our database set up on our hosting account.

Uploading our store

To upload the website files from our development environment to our production environment, we can use an FTP client. One such example of an FTP client is FileZilla, a free FTP client available for download.

Within FileZilla, we simply enter the web address of the site, and our FTP username and password, and then click on **Quickconnect**.

Once the FTP client is connected, we simply drag the files from the relevant folder on our development environment in the **Local** site pane on the left to the relevant folder within the **Remote** site pane on the right. Commonly, the folder on the server would be either `public_html` or `htdocs`, and files within these folders are generally made accessible to the public through a web browser.

Settings

Finally, we need to modify some settings, which involves:

- Editing the configuration file to include the database connection details for our production environment
- Uploading this configuration file onto the server, telling the production site to use that database
- Changing any aspects of the settings table in the database that references our development environment, such as the URL of the site or the path for file uploads

Automated deployment

Automated deployment makes it very easy to deploy code into a production environment. The exact setup of this is beyond the scope of this book, but let's discuss briefly what would be involved in this process:

1. We would make use of version control to store our code.
2. Copies of relevant configuration files would be within the version control, with references to production settings.
3. We would have a script on our production server, which:
    - Checked the code out of version control
    - Moved it into a web accessible environment
    - Removed the development configuration files, and renamed the deployment configuration files
    - Made any necessary changes to file permissions.

This is a topic I've discussed in more detail on my personal blog (`http://www.michaelpeacock.co.uk/blog/entry/svn-deploy-script`), which may be of your interest if you are interested in pursuing an automated deployment system.

Security

Security is a very important aspect with any website, but especially so with e-commerce websites. Let's look into how we can ensure our site and our customers' data can be kept secure.

Server security

The security of the server itself is one aspect of security that needs consideration. This can be broken down into two primary areas:

- Server software
- Firewall and network traffic

Software

Almost all software contain security vulnerabilities; once a vulnerability has been discovered, it is important to ensure that the software is upgraded or patched to prevent malicious users from exploiting these vulnerabilities. With managed hosting, we don't need to concern ourselves with server-installed software, as our hosting provider should keep that up to date. However, if we want to concern ourselves with the software on our server (and check our provider is up to date), or if we are operating on unmanaged virtual or dedicated servers, we need to keep updated on security developments with:

- PHP
- MySQL
- Apache
- The FTP server software
- The SSH server-side software

This could be done by subscribing to any mailing lists found on the sites for those projects.

Any other software we install, such as bulletin board systems, chat rooms, and so on, also need to be regularly checked for available upgrades and security updates.

Securing the site with a firewall

Software and hardware firewalls can help protect our website from attack; these generally work by blocking access to certain parts of the server from certain computers (for example, allow anyone to access the website stored on the server, except users we explicitly banned, but disallow anyone to access aspects such as FTP or SSH unless explicitly permitted). Most web hosts can advise on their firewall setup, and documentation is available for firewalls that can be used on virtual and dedicated servers.

Passwords

As a website owner or administrator of a site, our passwords can provide access to the administration area of the website. Our hosting account password also gives complete access to our website, including areas that are not related to our e-commerce system, such as databases, e-mail, and statistics, so it is important that we use secure passwords.

Passwords that are not secure can be obtained by users' guessing, automated dictionary attacks where a computer goes through a list of words trying them as the password, or by social engineering.

Strong passwords are one of the easiest ways to prevent user accounts from being compromised, or guessed by dictionary or social engineering attacks. These involve either going through a list of common passwords until the system logs the hacker in, or by researching the user and trying to guess passwords based off memorable information, such as dates of birth, names of friends and family, and so on. Some suggestions for making a strong password are as follows:

- Use both letters and numbers
- Make use of special characters, such as @, /, \, #, *, &, and so on
- Make all of your passwords unique; otherwise, if someone guesses your administrator password, they may be able to gain access to your personal e-mail, other websites you are a member of, and so on if the passwords are all the same
- Include spelling mistakes to make the word harder to guess
- Don't include personal information such as dates of birth, names of family, and so on
- Consider using numbers in place of some letters

SSL/TLS

Secure Sockets Layer (**SSL**) is a cryptographic protocol, which provides secure communications on the Internet by using encryption methods to encrypt data that is then transferred between the client and the server over this secure connection. Standard web page requests are not in SSL and data sent from the browser to the server are sent in plain text, which theoretically could be intercepted and read by third parties. SSL connections encrypt this data, preventing it from being read from any person or program other than the server. There is a detailed article on Wikipedia about **Transport Layer Security** (**TLS**), how this works, and the technicalities related to it: `http://en.wikipedia.org/wiki/Secure_Sockets_Layer`. To set this up, we need to purchase and install an SSL certificate.

SSL certificates are used to verify the identity of the server, which is used when encrypting the data sent to and from the server. The company who "signs" the SSL certificate usually determines the cost of such a certificate. This usually involves a trusted company verifying your identity and then issuing the certificate. Once we have a certificate, we need to contact our host to get the certificate set up on the hosting account. This will require a dedicated IP address for the site we are using SSL for; this generally incurs additional charges.

The use of an SSL certificate to secure connections to the website is a good idea; however, the costs and efforts involved in setting this up need to be looked into.

CAPTCHA

SPAM is increasingly common on the Internet. One way to reduce the effect this has on website owners is by implementing CAPTCHA challenges; these are the tests that can normally only be completed by a human, and not a computer, preventing automated bots registering on websites, placing orders, and populating our site's database.

These challenges generally involve something such as entering text from within an image, which a computer can't easily detect. The use of these tests can sometimes be off-putting to users, and should be used sparingly. We will look at integrating CAPTCHA challenges in the appendices.

Maintenance

The final section is maintaining our site; the most important aspect of this is backing up and restoring our site.

Backing up and restoring

It is important that we take regular backups of our sites, in case something were to happen to the website, its hosting account, or even the server the website is stored on. If we were to lose several weeks worth of new product additions, new customer sign-ups, or new orders, this could do some serious damage to our reputation as developers, and the reputation of the business/site in question.

Automated nightly backups should be set up eventually; most hosting providers also have backup procedures in place, so it is also worth investigating what provisions are already there for this. With many non e-commerce sites, if we lost a week's worth of data, the only negative effect would be on our time for any changes made in that past week, or on some contributions from a community. With a business e-commerce site, we could lose order data. If this was for a customer who had paid for their order, we would not know anything about the order to enable us to fulfill it, causing angry customers.

Using cPanel

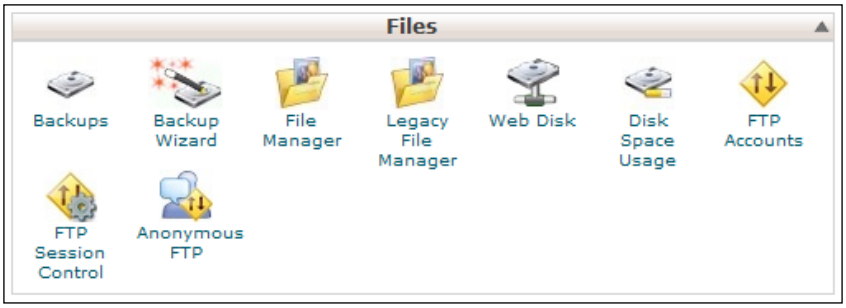
Let's use cPanel, the popular web hosting control panel to backup and restore our site.



This section assumes a hosting account with cPanel installed.

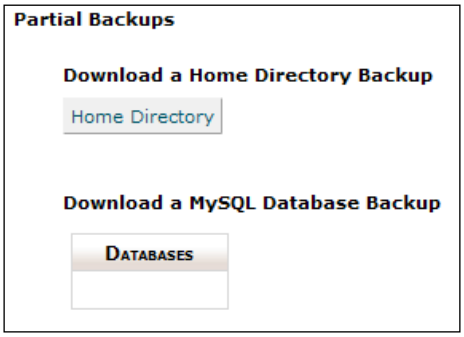
Backing up the site and database

Within the main cPanel interface, in the **Files** section, there is a link to the **Backups** area.



PACKT PUBLISHING
This material is copyright and is licensed for the sole use by jackie tracey on 23rd February 2010
953 Quincy Drive, , Brick, , 08724

We can download a copy of our **Home Directory** (all of the files and most of our settings), and also a copy of the database from this section. Simply clicking on the relevant backup buttons will prompt us to download the backup files from the server.

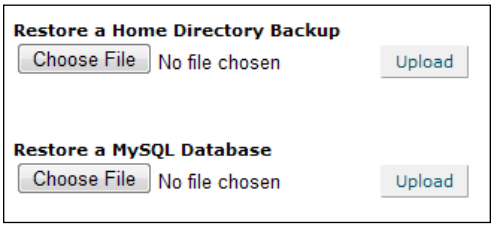


It is essential that we keep these files stored somewhere safe and secure.

Restoring the site and database

To restore from a backup, we need to ensure we are logged into cPanel, and then click on the **Backups** button to go to the backups section, as we did when backing up the site.

On the right-hand side of this screen are the options to **Restore a Home Directory Backup** and to **Restore a MySQL Database**.



To restore from the backups, all we need to do is browse for the file we wish to restore from, and then click on **Upload**.



When restoring, any existing database or home directory content will be removed, so only do this if you really need to. If you need to gain access to a specific file that you need to back up, decompress the home directory backup, look for the file, and upload it to your site using an FTP client.

Using the command line (SSH)

Assuming we have shell access to our server, we can connect to it and issue simple commands to back up and restore our site easily. Programs such as PuTTY can allow us to connect using SSH to our web hosting server.

Backing up the site

Once connected through SSH to the server, we need to navigate to the location of our site.

```
Cd /home/junipert/
```

Then we can compress the `public_html` folder to a single file, using:

```
Tar cvzf backup.tar.gz public_html
```

With the folder compressed, we need to move it to within the `public_html` folder, so we can download it by visiting `oursite.com/backup.tar.gz`.

```
Mv backup.tar.gz public_html/backup.tar.gz
```

Restoring the site

Assuming we upload the `tar.gz` file into our server, we can decompress it with the following command:

```
Tar –xvf backup.tar.gz
```

Backing up the database

The following command exports our database to a web-accessible location on our server, where we can download it using a web browser.

```
Mysqldump –u username –p databasename > /home/junipert/public_html/
backup.sql
```

After executing this command, we will be prompted for our password; then we can download the file from our browser.

Restoring the database

Assuming we upload the SQL file onto our server, we can import it with the following command:

```
Mysql –u username –p databasename < /home/junipert/backup.sql
```

Summary

In this chapter, we looked at the importance of security with our site, and had a primer on SSL, CAPTCHA, password security, and software security. We deployed our website from our development environment to a production environment. We also looked at how we can back up and restore our site on a regular basis to ensure we are covered in case something were to go wrong.

15
Marketing, SEO, and Customer Retention

With our new framework, we are able to take on any e-commerce project that comes up. Even if the framework does not contain all the necessary features, we can expand and extend it to meet those needs. There are, of course, things that should be considered at this post-deployment stage, including marketing techniques, search engine optimization, and potential ways to improve customer retention. In this chapter, you will learn:

- How to market sites/stores you build, using:
    - Online advertising, such as advertising space and **pay-per-click** (**PPC**) adverts
    - Newsletter advertising
- How to avoid being penalized by the search engines
- How to use newsletter systems to market effectively
- Social marketing, including viral marketing, Twitter, and the likes
- On- and off-site search engine optimization
- Customer retention with newsletters and social features

Marketing sites and stores powered by our framework (and other sites for that matter)

There are a number of ways we can market not only online sites and stores created with our framework, but also any sites or stores we are managing or maintaining. This can range from some simple online marketing to advertising, or PPC campaigns. Let's take a look at some of the marketing methods available to us.

Online advertising

There are a number of different online advertising techniques available for us to take advantage of, including:

- Purchasing advertising space
- Search engine advertisements (PPC)
- Various professional/reputable advertising networks
- Newsletter advertising

With both professional advertisement networks and the search engine advertising, their business model typically operates on a PPC basis, whereby we pay for each time someone clicks on one of our adverts and is taken to the site.

Buying advertising space

A number of websites offer advertisement space, generally on a monthly basis, which can often be a great way to generate new traffic and bring new customers to a site. There are a few simple points to take into account when considering renting advertising space from a site:

- Does the site you are looking to advertise on compete directly with your own site? If so, they probably wouldn't accept your advert, nor would it be an ideal place to advertise. The visitors have already gone through to their site, and would probably not be inclined to go elsewhere. Thinking back to our Juniper Theatricals store, this means we wouldn't want to advertise on fictitious sites such as:
    - Global theatre supplier
    - Novelty t-shirt store

- Is the site relevant to ours? If the site is relevant (but non-competing), then we are more likely to get clicks through to our site, as visitors will be interested in the area we work in.
- Is the site we are advertising on reputable? If the site has a bad reputation, that reputation will come to us by association. Visitors will see we are associated with the site, and that will affect their view of our site. It is important to spend some time checking a site's reputation; it may even be worth contacting the owner of the site to find out some background or history about the site and the owner.
- What are the statistics for the site like? If the site does not get many visitors, then it isn't worth us advertising on it. It is important to find out statistics from the website owner, including visitor numbers and preferably some information on the demographics of users. If the site has a small number of visitors, then it would be important to ensure that payment is for a certain number of impressions or clicks, as opposed to a set period of time. Services such as Google Analytics provide this information; however, there are many providers available who can process the raw log files on the hosting server, and generate statistics from that.

Pay-per-click advertisements

Unlike the purchasing of some advertising space, PPC only costs us each time a visitor clicks on an advert and goes through to our site. When looking at or negotiating PPC rates with advertisers, it is important to work out what the conversion rate is likely to be (that is, how many visitors clicking through to our site are converted to customers) and the average purchase amount for them. This way we can work out how much we earn per click, and how much we would be willing to spend on a click through to our site.

Most PPC services allow us to set daily and monthly budgets, so that when a daily maximum is reached our advert is no longer displayed until the next day, when a new daily limit is in effect.

Let us now take a look at how most PPC services work:

1. We sign up to a PPC network.
2. We provide information about our site, and some personal information.
3. We provide billing information, either a credit card number, or we make payments in advance.

4. We select the keywords we wish to target (for example, theatre supplies; these are words which visitors may type into a search engine, or the page may have content related to these keywords for adverts displayed on pages, triggering our adverts), as well as any information on the visitors we want to target (for example UK users).
5. Finally, we set a budget for how much we would be willing to pay for each click, the maximum we would be happy spending in a day, and so on.

Once our campaign is running, we can generally log in to a control panel and see how much of our budget has been spent, and how much we are paying on average per click. The monthly budgets mean if we don't pre-pay, and instead provide credit card information, we are never billed more than we have agreed to.

One thing that advertisers are often concerned about is the possibility of fraudulent clicks. For example, a competitor could perform a search to find our advert, and then repeatedly click our advert. This would cost us our campaign budget, and not give us a return, because the clicking was not done by a potential customer. To prevent this from affecting advertisers, and ruining the reputation of advertising networks, most of them have systems in place, tracking duplicate clicks and crediting the accounts of advertisers when this occurs. It is important to ensure that the PPC network we choose has provisions for detecting fraudulent clicks, so our money isn't wasted!

Advertisement networks provided by search engines

Many search engines also provide their own PPC advertising network, three of which are listed below. The algorithms employed by many of these search engines determine how much a click is likely to cost, based on the site itself, and its position in the natural search engine rankings. So a site that is completely unrelated to theatrical supplies, would probably need to pay more than a theatrical supplier for PPC advertisements with search engines.

- **Google** (`http://www.google.co.uk/intl/en/ads/`)
- **Yahoo**!: (`http://sem.smallbusiness.yahoo.com/searchenginemarketing/index.php`)
- **Microsoft**: (`http://advertising.microsoft.com/search-advertising?s_int=277`)

Most search engines also allow their advertising networks to be used on third-party sites, so apart from appearing as a sponsored link on "search engine results" pages, the site will also display on websites that decide to display adverts from that particular advertisement network, and also contain content relevant to the advertisement. One important thing to remember about competing sites is that most PPC networks allow us to enter sites where we don't want our advert to appear, so if a competitor displays adverts, and ours appears on theirs, we can detect this through their control panel, and add them to the list to prevent our advert displaying, hopefully increasing our return on investment.



Pay per action

A new scheme, being investigated by a number of PPC networks, is pay per action, where you only pay when a visitor performs a certain action on your site. This could involve registering for an account, entering their e-mail address in a newsletter box, or making a purchase. This is still very much at research and development stage for most networks; however, it is worth keeping an eye on the progress in this area.

Newsletter advertising

There are a large number of online newsletters available, many of them targeting specific niche markets. It would be useful to advertise our stores within e-mail newsletters that are relevant to our store, for instance an e-mail newsletter that is sent to all the prop managers at theatre companies.

This method involves quite a lot of research, finding suitable newsletters, and discussing with the owners of the newsletters to negotiate advertising pricing. There are some online management systems designed to help match advertisers with newsletter managers, so it may be worth researching for those too.



Don't forget to consider the points we discussed earlier, with regards to advertising space, when looking at advertising on newsletters. The tips apply to both forms of advertising quite well.

A word of warning: Search engine penalization

Page listings in **Search Engine Results Pages** (**SERPs**) are determined by search engines by a number of different metrics, including age of domain name, content on the site, and also the number of incoming links to a site. With Google, this link factor, along with some other metrics, makes up a page rank. Depending on a site's page rank, the links that the site has to other sites (outbound links) can gain page rank from this. Links from one site to another are classed as a vote, and it assumes that the site owner was happy to display that link, and that they approve of the site, and wish to attribute a vote to it, improving its page rank.

In some cases, paid advertisements are seen as a way to buy increased page rank, which search engines see as a way of "spamming" their search index. Many search engines, including Google, have anonymous online reporting tools, where users can report paid links on websites, which then are investigated and the involved sites are penalized with regards to their rankings in the SERPs.

The sale and purchase of links and adverts on the Internet isn't wrong; it is just the sale or purchase of links to adjust page rank that is, and so, most search engines take into account some additional information within a link that indicates that the site owner does not wish for the link to receive their "vote" when calculating page rank. This attribute should be used for any paid advertisements or links, to ensure neither the site selling nor the site buying the adverts are penalized for this. The solution is to add `rel="nofollow"` to the link, so we would end up with a link such as this:

```
<a href=http://www.packtpub.com rel="nofollow">Packt Publishing</a>
```

This does not mean that we need to add this attribute to all of our outbound links, only links that are paid for.

Tips to stay in the search engines' good books

Here are some useful tips to ensure you stay in the good books of the most popular search engines:

- Don't buy or sell links, only buy advertising space from reputable sites (and ensure the advert has the `rel="nofollow"` attribute).
- Ensure that all adverts on your own site contain the `rel="nofollow"` attribute.
- Be wary of e-mails offering to place advertisements on your site.

Hopefully, by following these tips and taking a common sense approach, you won't jeopardize your search engine rankings.

Newsletters

There are a number of newsletter systems available, which we can use to send newsletters to our customers or interested parties. Visitors to our site could leave their e-mail address to indicate that they are interested in our site, but not yet ready to make a purchase, and we could e-mail them with latest products and news from our site.

One particularly popular newsletter system is Campaign Monitor; this not only makes it easy to manage many lists of subscribers, but also provides advanced tools to track the success and performance of newsletter campaigns, with metrics such as:

- How many users opened the e-mail?
- How many times users opened the e-mail?
- Which links were clicked on, by whom, and how many times?
- Which e-mail clients were used?
- Who, or how many users, unsubscribed from the newsletter, forwarded it to a friend, or reported it as spam?

These metrics are not accurate, as the techniques used to detect how many times an e-mail has been opened rely on images within the newsletter, thus requiring the user to set their e-mail client to display images. However, they are useful as a basic indication of minimum statistics. It is also possible to integrate the newsletters with stats programs such as Google Analytics. One final feature worth mentioning is that, Campaign Monitor, and many other newsletter systems, also allow us to preview the contents of the newsletter in various different e-mail clients to ensure the newsletter will look as intended. For all of our subscribers, along with this, it can also run the e-mails through spam filters to detect if it is likely to be flagged as spam.

Marketing materials

One, often overlooked, aspect is physical marketing materials. Adding a web address to letterheads, business cards, and brochures can help in telling the existing customers about a new website. We could also create dedicated brochures or use business cards as a mini advert for a particular product, or series of products, or alternatively, for voucher codes. We will look again at voucher codes later in this chapter.

Affiliate marketing

We can also look at offering our products and services through other websites, by means of affiliate marketing. This involves allowing other retailers to sell our products, and for us to pay a commission to them for each sale. This can also work the other way round, so if we wanted to sell another product or service on our site, which complemented our current offering, we could use affiliate marketing to sell products of others, and earn a small commission on the sales.

Affiliate marketing can also work by other retailers and marketers simply linking to products or services on our website, helping to promote them, and earning commissions based on those sales.

Social marketing

With the popularity of social networking on the rise, it makes sense to also make use of social networks to promote our store. Most social networks have provisions for user and business information as well as profile data including website addresses. Examples of this include creating a Facebook fan page for our business, adding the site's URL to our Facebook and MySpace profiles, and adding it to our Twitter accounts. These extra links could help with additional promotion, even if they only bring one or two new customers, it is still worthwhile.

Viral marketing

Viral Marketing is a relatively new marketing concept, which revolves around utilizing social networks. One particular example of viral marketing is utilizing video sharing websites such as YouTube and promoting videos within which advertise businesses, websites, products, or services by using them in the video.

This technique is probably more suited to large social networking sites with large marketing budgets who are trying to promote a brand. Information on using YouTube in particular was recently posted on a technology blog called TechCrunch, and can be found at `http://www.techcrunch.com/2007/11/22/the-secret-strategies-behind-many-viral-videos/`.

Twitter

We could use Twitter, a social network that aims to tell your friends and followers what you are doing, to keep up to date with our customers. One potential method is to create a customer service Twitter account to post news, updates, and product releases, in addition to keeping an eye out for comments or feedback from customers on the social network, and responding to them.

RSS with FeedBurner

Many websites offer content to their users through **Really Simple Syndication** (**RSS**), which allows them to read the content, such as blog articles, latest products, recommendations, and reviews and so on, off-site in their favorite RSS reader. Services such as Google's FeedBurner allow us to monitor our customer's usage of RSS feeds, and gather statistics from them.

Search engine optimization

One way to increase traffic to our website is through **search engine optimization** (**SEO**). This involves ensuring the content and the structure of our site are well optimized for search engines, making it easier for them to access our sites, and digest the important content. The other aspect is with regards to inbound links to our site.

Therefore, search engine optimization can be broken down into two primary areas:

- On-site search engine optimization, focusing on changes to the actual website itself
- Off-site search engine optimization, focusing on building up a reputation for the website through reputable, high quality, inbound links

Let us take a brief look at these two methods.

On-site SEO

On-site SEO requires us to ensure that the website itself is suitably structured, and the content is appropriate and up to date, encouraging search engines to index the site, and helping them realize which content is most relevant within the site.

Headings

Properly-structured pages make use of appropriate headings to break down the content of the document into sections. The content within these headings is also considered highly by search engines. It is important that we don't fill them with too much content—three to seven words should be sufficient, keeping with the feel of a heading. The different levels of headings indicate their importance within the page (heading level one is most important, level two less so, and so on). There is much discussion among the web design community about what a first level heading should contain—either the name of the site, or the name of the page. Personally, I find the name of the page more appropriate and more relevant in terms of optimization too.

Links

Links to other pages within the site is a very simple and useful way to improve search engine performance. The trick is to make use of relevant sentences, using the relevant keywords as hyperlinks, and also ensuring that the titles of the link are suitably optimized. Take the example of a "novelty hat category" page.
A poorly-optimized link would be:

```
To view our collection of novelty hats <a href="noveltyhats/">click
here</a>
```

The link has no context to search engines, and contains no meaningful information.
A more meaningful, and therefore, search engine friendly link would be:

```
Why not view our <a href="categories/novelty-hats/"
                  title="Top quality collection of novelty hats">
                        collection of novelty hats</a>
```

All these small changes do make a difference!

Up-to-date content

One of the most important things about a website is its content. Visitors like content to be fresh and up to date.

Meta tags

An older method for search engine optimization was to take advantage of the meta tags within an HTML document. Because this was widely abused, it isn't as effective as it once was; however, it is still a useful technique. Some sites have their description text in "search engine results" pages showing as the text from their `description` meta tags.

The two important meta tags, are `keywords` and `description`. The `keywords` tag allows us to associate a number of keywords with our content, and the `description` tag allows us to associate a friendly, easy-to-read description to the page. Because search engines penalize sites that hide some content from their users (with the purpose of it being shown only to the search engines, to make the search engines think the site was more relevant for certain phrases or keywords), this technique was abused as a legitimate way to have text that was unrelated to the page (or repetitions of related content) to try and boost rankings, and as such the search engines don't put as much emphasis on these now.

The meta tags are contained within the `<head>` section of an HTML document. An example of the `keywords` and `description` tags in use is as follows:

```
<meta name="description"
      content="Juniper Theatricals is a leading supplier of
               theatrical products for the North East of England" />
< meta name="keywords"
       content="theatre, theatricals, supplies, back drops, props,
                scenery, novelty t-shirts. " />
```

While the search engines don't take these into account too much, it is still important not to overuse them, as that indicates to the search engines that the site is trying to abuse the meta tags and their purpose.

Sitemap and webmaster tools

A collection of tools geared toward helping webmasters manage the errors within their site, and see how Google sees their website, has been developed by Google, and is available for use, freely. Webmasters can also create a sitemap in XML format, to tell Google of all of the pages within our site, their importance within the scheme of the site as a whole, and how frequently they are updated, to help them decide when to return to re-index the updated content.

The webmaster tools, in general, outline errors such as duplicate content, duplicate meta data within pages in the same site, as well as broken or forbidden links. More information can be found on the following pages:

- `https://www.google.com/webmasters/tools/home?hl=en`
- `http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=40318`

There is also a very powerful statistics and analytics package called Google Analytics, available from Google, completely free of charge. This is useful for us to see which pages our visitors are using, and which pages are being ignored, allowing us to either promote them more heavily, or to focus on the more popular areas of the site. There are also ways to integrate Google Analytics with e-commerce installations, to try and help us to determine average income per visitor, particularly useful when making use of PPC marketing, as it links in with Google's own PPC network, AdWords. We can sign up for Google Analytics on `http://analytics.google.com/`, where we are supplied with some HTML code to insert into our site's footer template, so that it can begin tracking our statistics.

Off-site SEO

Off-site SEO relies on promoting the website on various other websites through inbound links, which is why it is referred to as off-site SEO. This is a particularly large area, and some companies spend very large amounts of money on this, though of course this is all relative to the amount of return they get on their SEO investment. Off-site SEO is particularly useful for gaining rankings for specific keywords within the search engines.

Inbound links are, as we discussed earlier, an important metric in determining the ranking of websites within the SERPs. One of the easiest ways to generate inbound links, is with existing social networks, or social websites (forums in particular), by adding a link to the website within our personal signatures on discussion forums. This needs to be done carefully and considerately. If we were to sign up just to promote our link, we would be seen as a spammer, and most sites would deactivate our accounts. Posting comments on relevant blog entries or articles with a link back to our site is also useful, provided the comments are appropriate, relevant, and our own site does not compete with the article or blog in question.

Some examples of services that SEO agencies offer as part of an off-site campaign include:

- Writing articles for relevant blogs or article networks with links back to our site
- Guest blog posts on other blogs
- Online distributed press releases
- Link baiting (articles, content, or applications designed to generate many comments, blog trackbacks, forwarding, and linking to; often this is done by posting on controversial topics within a specific niche, or by viral marketing)
- Link building (building high-quality, relevant inbound links)

Customer retention

Another important aspect of marketing is marketing to existing customers, keeping them coming back to the site to make repeat purchases.

Newsletters

We've discussed the option of mailing lists and newsletters earlier. By having a newsletter for customers who have placed orders, or placed orders of certain products we can send them relevant newsletters such as:

- Product line updates
- Related products they may be interested in
- New releases on the store
- General updates, useful for reminding customers that the store still exists

Social features

Many of the social features we have integrated into our store help with customer retention. For example:

- Product ratings
- Product reviews
- Related products

By valuing the input of customers, they feel valued and, in turn, are more likely to contribute to the site, through ratings and reviews. Additionally, these social features encourage them to return after their purchase, to post a review of the product; at this stage, they may be inclined to make another purchase.

Coupons and voucher codes

By providing customers with coupons or voucher codes, we can entice them to make repeat purchases, perhaps by sending a small business card-shaped coupon card, with a voucher for free shipping for a customer's next order. Placing these vouchers when dispatching the customer's first order gives them an immediate incentive to return to the store, to look at the products.

Summary

In this chapter, we looked into effectively marketing and promoting websites and e-commerce stores with online marketing techniques, search engine optimization, and customer retention strategies.

Now, not only do we have a great framework to use for our projects, but we are well-placed to market and promote them effectively, hopefully generating a great return on investment for ourselves with our own projects, and for client projects.



Interacting with Web Services

There are a number of web services available that we as developers can interact with to either help make tasks easier for us, or to help us to target new markets. Now, we will investigate a number of these web services and APIs. In this chapter, you will learn more about:

- Google products
- Google Analytics, and its benefits for e-commerce sites
- Amazon web services
- eBay developer center

Google products

Google has a product search feature, which allows users to search specifically for products using the Google search engine. Products can be added to this search engine using the Google merchant center.

The Google merchant center is an area where online retailers can list (and manage these listings) their products for including in the Google product search. Google merchant center is a specialized section of Google base: a search area that was designed for allowing anything to be added to the Google index—documents, notes, products, events, and so on, essentially anything that generally wasn't in a web page. As this service grew, the products aspect was spun off into the merchant center.

Apart from being able to manually add products, we can also add a feed of products. This feed of products would be tied directly to our store, updating in real time as we added new products, removed older ones, and updated details. By adding a feed, Google can keep an up-to-date copy of our product catalog for inclusion in its search results.

To get started with the Google merchant center, we need to sign up, or sign in at `http://www.google.com/base/`.

Adding the feed to the Google merchant center

Within the merchant center, we can click on the **Data feeds** link on the left-hand side, and then on the **New Data Feed** button to create a new feed. Here we can set:

- The **Target country** to determine who would see results from our feed
- The **Data feed type** (googlebase)
- A name for the feed, that is, the **Data feed filename** (for example `feed.xml`)

After adding the feed to the Google merchant center, we set an update schedule.

Setting an update schedule

The update schedule is where we actually tell the Google merchant center where our feed of products is, and how often it should be updated. When we have added our feed to the center, there is a link next to it under the **Upload schedule** column called **Create**. If we click on this, we will see the **Scheduled Upload** form. This form allows us to select:

- How frequently we wish to upload the feed: **Daily/Monthly/Weekly**
- When we want to update the feed (for example, day 15 of every month)
- Our time zone
- The URL of the feed

Creating the feed

To be able to actually create the update schedule we discussed, we need a product feed. We could use a tab-delimited feed, which would be easy to do using a spreadsheet program. If we did it this way, we would need to create and upload the feed manually—something we don't want to do.

XML is a standard way of representing data, and is particularly useful when interacting with web services.

Product feed controller

We could create a product feed controller, which generates the feed for us. One requirement for XML feeds in the Google merchant center, is that they end in `.xml`, so we would have the controller search the second bit of the URL (for example, `productsfeed/latest.xml`) split the string by the dot, and then depending on the first word, here `latest`, display the relevant feed.

The controller would build a query of products, cache the results, and store them as a template variable, which would go into the XML template for the data feed.

```
<?xml version="1.0"?>
<rss version="2.0"
     xmlns:g="http://base.google.com/ns/1.0"
     xmlns:c="http://base.google.com/cns/1.0">
  <channel>
    <title>Juniper Theatricals Product Feed</title>
    <link>http://www.junipertheatricals.test</link>
    <description>
      Theatrical supplies, props and costumes
    </description>
    <!-- START items -->
    <item>
      <title>{name}</title>
      <link>
        http://www.junipertheatricals.test/products/view/{path}
      </link>
      <description>{description}</description>
      <g:image_link>{image}</g:image_link>
      <g:price>{cost}</g:price>
      <g:condition>new</g:condition>
      <g:id>{ID}</g:id>
      <c:retail_price type="decimal">{cost}</c:retail_price>
      <c:promo_offer type="boolean">false</c:promo_offer>
    </item>
    <!-- END items -->
  </channel>
</rss>
```

This is an XML feed showing some basic information for products. We can define our own custom elements to this too if we wish, such as how the product could be customized, if it is a downloadable product, if we can upload an image, the delivery time, and so on. More information on this is available from Google at `http://www.google.com/support/merchants/bin/answer.py?answer=160603&hl=en`.

Other useful link

For information on data feed specifications, visit
`http://www.google.com/support/merchants/bin/topic.py?topic=24946`

Alternative—Google Base Data API

In addition to adding feeds to the merchant center, we can also add products directly from our framework if we wish. We could do this using the Google Base Data API (`http://code.google.com/apis/base/`). Further details on inserting, updating, and deleting data items using this API are available at `http://code.google.com/apis/base/starting-out.html#insupdel`.

Others

We've discussed Google quite a bit here; they are a very big player in this arena, being one of the most popular search engines around. There are some other options available, and these have their own specifications for data feeds, which are also supported by the Google merchant services center. These feed types include:

- shopzilla
- shopping.com

Google Analytics

Another Google offering is Google Analytics, a useful web-based application for monitoring website analytics, such as visitor numbers, visitor lengths, popular pages, sources of traffic, and so on.

One particularly useful feature within Google Analytics for us is its e-commerce functionality. At minimum, we could add some code to indicate an order has been placed; this would allow us to look at data such as how many visits it took to make a purchase. We can of course go into more detail, supplying other information such as how much the order was for, and so on.

Google Analytics works by having a small piece of JavaScript inserted at the bottom of every page on our site.

Signing up

To sign up for Google Analytics, we simply need to:

1. Visit `http://www.google.com/analytics/` and sign up.
2. Click on **Add Website Profile»**.
3. Enter our web address.
4. Copy the tracking code generated, and put that into our website's footer.
5. View the profiles list, and click on **Edit** for our website profile.
6. Under **Main Website Profile Information**, click on **Edit.**
7. Select **Yes, an e-commerce Site**.

We now have an account set up for e-commerce, and the tracking code is installed; next we need to track e-commerce transactions.

Tracking e-commerce

To track e-commerce sales in our store, we can record transaction details and item details, and then submit this information to Google Analytics.

The information is all captured into a JavaScript function call, which sends the data to the Analytics' server. The following JavaScript needs to go after the `pageTracker._trackPageview();` from our initial tracking code.

Add transaction

To add the transaction, we must at least store:

- The order ID
- The total cost of the order (excluding shipping)

We can also record:

- Affiliation or store name
- Tax costs
- Shipping costs
- Customer's city
- Customer's state or province
- Customer's country

This is reflected in the JavaScript as follows:

```
pageTracker._addTrans(
  "111", // the order ID - this is a required field
  "Props", // affiliation or store name
  "10.50", // total - required
  "0.00", // tax
  "10.00", // shipping
  "Newcastle", // city
  "Tyne and Wear", // state or province
  "UK" // country
);
```

Add item

For each item within the transaction, we must record:

- The order ID
- The product code or **stock keeping unit** (**SKU**)
- The unit price for the item
- The quantity of the item

We can also record:

- The product name
- The category or variation of the product

This is reflected in the JavaScript as follows:

```
pageTracker._addItem(
  "111", // order ID - necessary to associate item with transaction
  "P1", // SKU/code - required
  "Fake Water Jug", // product name
  "Large", // category or variation
  "10.50", // unit price - required
  "1" // quantity - required
);
```

Track transaction

Once the transaction, and all items within the transaction, have all been added, we track the transaction, by issuing the following JavaScript call:

```
pageTracker._trackTrans();
```

Further reading

- Tracking number of sales (but nothing else)—Analytics Talk—`http://www.epikone.com/blog/2008/06/25/google-analytics-e-commerce-tracking-pt-3-why-everyone-should-use-it/`
- **Tracking lead generation forms** (`http://www.epikone.com/blog/2008/07/02/google-analytics-e-commerce-tracking-pt-4-tacking-lead-gen-forms/`)
- **Tracking API: e-commerce** (`http://code.google.com/apis/analytics/docs/gaJS/gaJSApiEcommerce.html`)

Other services

Both Amazon and eBay (along with a number of other retailers) have APIs that allow developers to sell products by listing them using an API.

Amazon

Amazon's **Marketplace Web Service** (**Amazon MWS**) allows sellers to list items for sale on the Amazon marketplace through an API. By using this, we could automate the process of listing products we have in our own e-commerce store, on the Amazon marketplace website. This would extend opportunities for new customers to do business with us. By suitably integrating the API with our framework, we could:

- Automatically list our products on Amazon marketplace
- Automatically remove our products from Amazon marketplace when stock levels are low

A PHP client library is available for the Amazon MWS API, which can be used to integrate with our framework: `http://mws.amazon.com/phpClientLibrary.html`.

eBay

eBay has a developer center for various languages and APIs, including searching, managing feedback, and of course, creating listings. With this API we could:

- Automatically create listings based on new products
- Automatically create repeat listings for existing products that are still in stock (perhaps, generating new ones each week)
- Automatically post good feedback to the buyer once the order is complete

A number of PHP resources are available for working with the API, including ones specific for their trading API, which would be of most use to us. Some of these include:

- `http://developer.ebay.com/developercenter/php/`
- `http://developer.ebay.com/developercenter/php/trading/`

More to come

There are a few more web services, which we will look at interacting with in Appendix C, *Cookbook*.

Summary

In this chapter, we have primarily looked at the Google merchant center and how we would create a data feed of products which would automatically update with our product catalog, to keep our information constantly up to date in Google product search. We also had a brief look at:

- What similar services are available
- Google Analytics:
    - Tacking website statistics
    - Tracking sales
        - Order data
        - Item data
- Amazon web services
- eBay developer center

B
Downloadable Products

In this appendix, we are going to look at how we would rapidly extend our framework to allow downloadable products. In this chapter, you will learn:

- How to extend the types of products available in the framework
- How to extend the payment and administration areas to unlock downloadable products when payment is made
- How to lock access to downloaded products when payment is refunded
- How to create a centralized download area for customers to access downloadable products

This is a very basic implementation of downloadable products functionality. Ideally, we would also look into storing the files in a location that isn't accessible through the Web. We would then either copy the file for each user who purchases the product in a unique location, or create a script, which when an authorized user copies the file, temporarily to a public location, allows a single download, and then removes that copy.

Extending products

When we abstracted out content earlier in the book, we separated product-specific data in a table, `content_types_products`. This is the table we need to extend to enable downloadable products.

We should just need to add two new fields to this table:

- **Downloadable**: This indicates if the product is downloadable.
- **File**: This indicates the name of the file.

We also need to define a setting, which will be the location of files that are downloadable products.

```
ALTER TABLE `content_types_products`
ADD `downloadable` BOOL NOT NULL DEFAULT '0',
ADD `file` VARCHAR( 255 ) NULL
```

Extending the payment and administration areas

To extend the payment and administration areas to allow customers to download the files for their purchases, we should create a centralized record of these types of purchases (we will discuss this later in more detail). This makes providing a nice download area much easier.

At this stage, there are three things we need to do:

- Create a new database table relating customers to their downloadable purchases
- Update the payment methods to automatically provide access to these downloads by creating new records in the new table when payment is received
- Update the payment methods to automatically remove access to these downloads by removing records from the new table when orders are updated to "refunded".

Access database

Our table for providing access to download files needs the following fields:

- An ID
- A reference to the customer's ID
- A reference to the product ID
- The date access was granted
- The location of the file

The location at present will just be copied from the products table. However, it makes things easier for us if we extend the functionality to create a separate copy of the file for each customer.

The following SQL would create the table for us.

```
CREATE TABLE `book4appa`.`download_access` (
  `ID` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
  `user_id` INT NOT NULL ,
  `product` INT NOT NULL ,
  `file` VARCHAR( 255 ) NOT NULL ,
  `access_granted` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
  INDEX ( `user_id` , `product` )
) ENGINE = MYISAM ;
```

Providing access

When a customer's order is updated to "paid", there are a few stages our framework needs to go through to provide access to these downloadable products:

1. Check to see if any of the products in the customer's order are downloadable.
2. Get a list of the IDs and files for these downloadable products.
3. Create a record in the new table for each of these products.

The following code does exactly that!

```
// provide access to downloads
$downloadables = array();
```

First, we lookup the downloadable products.

```
$sql = "SELECT ctp.file, v.name, i.product_id
        FROM content_types_products ctp, orders_items i,
             content c, content_versions v
        WHERE ctp.downloadable=1 AND i.order_id={$order}
              AND c.ID=i.product_id AND v.ID=current_revision
              AND ctp.content_version=v.ID";
$this->registry->getObject('db')->executeQuery( $sql );
```

We then iterate through those downloadable products, and store a copy.

```
if( $this->registry->getObject('db')->numRows() > 0 )
{
  while( $row = $this->registry->getObject('db')->getRows() )
  {
    $downloadables[] = $row;
  }
```

We then iterate through them again, this time inserting them into the access table.

```
    foreach( $downloadables as $data )
    {
      $insert = array();
      $insert['user_id'] = $orderData['userid'];
      $insert['product'] = $downloadables['product_id'];
      $insert['file'] = $downloadables['file'];
      $this->registry->getObject('db')->
          insertRecords('download_access', $insert );
    }
  }
```

Rescinding access

Similar to giving access, when the order is refunded, we need to remove this access.

```
  elseif( $status == 'Refunded' )
  {
    // remove access to downloads
    $downloadables = array();
    $sql = "SELECT ctp.file, v.name, i.product_id
            FROM content_types_products ctp, orders_items i,
                 content c, content_versions v
            WHERE ctp.downloadable=1 AND i.order_id={$order}
                  AND c.ID=i.product_id AND v.ID=current_revision
                  AND ctp.content_version=v.ID";
    $this->registry->getObject('db')->executeQuery( $sql );
    if( $this->registry->getObject('db')->numRows() > 0 )
    {
      while( $row = $this->registry->getObject('db')->getRows() )
      {
        $downloadables[] = $row;
      }
      foreach( $downloadables as $data )
      {
        $p = $downloadables['product_id'];
        $u = $orderData['userid'];
        $this->registry->getObject('db')->
            deleteRecords('download_access', " user_id='{$u}'
            AND product='{$p}' ",1 );
      }
    }
    // we refunded the payment
    // update the order

    // email the customer
    // email the administrator
  }
```

Centralized download area

Finally, we come to the actual download area. This just needs to be a simple controller, which lists the users' entries from the access table, along with links to the corresponding downloads.

Our controller needs to query the database, cache the results, and add them to a template tag. Obviously, the controller also needs to check that the user is logged in; otherwise, an error message should be displayed.

```
private function listDownloads()
{
  $this->registry->getObject('template')->
      buildFromTemplates('header.tpl.php',
                         'downloads.tpl.php',
                         'footer.tpl.php');
  $u = $this->registry->getObject('authenticate')->getUserID();
  $sql = "SELECT p.name, d.file FROM content c,
                 content_types_products p, download_access d
          WHERE c.ID=d.product
                AND p.content_version=c.current_revision
                AND d.user_id={$u}";
  $cache = $this->registry->getObject('db')->cacheQuery( $sql );
  $this->registry->getObject('template')->getPage()->
      addTag('downloads', array( 'SQL', $cache ) );
}
```

The corresponding view file would look like this:

```
<h1>Downloads</h1>
<ul>
  <!-- START downloads -->
    <li><a href="ourdownloadarea/{file}">{name}</a></li>
  <!-- END downloads -->
</ul>
```

What else is needed?

Here we have looked into the bare bones of creating this functionality, but there are still other features and aspects that we would need to implement, including:

- Editing the create product functionality in the administration area
- Editing the edit product functionality in the administration area, so that a standard product could be converted to a downloadable product, and vice versa

- More security provisions, as discussed
- Download logging, to ensure that customers are not making a purchase and then giving their login details to others, so they can access the files
- In most instances, we would only want customers to make a single purchase of these products, as in most cases, duplicate purchases would be redundant

Summary

In this chapter, we looked into extending our framework to allow downloadable products, for which customers are automatically provided access when their purchase is made. We also looked into removing this access when orders are updated to "refunded".

Although this is a very basic implementation, with a number of security issues that would need to be looked into in a live site, it does illustrate how we can quite easily extend the framework to accommodate new features and new types of products. If we wanted, we could easily extend our products' functionality to:

- **Allow subscriptions**: This allows upgrading user accounts based on certain purchases. This could be a "gold membership" product, which upgraded a user's permission rights to the site, allowing them to access more areas.
- **Build dynamic downloads**: If we were selling software, we could integrate the build process for that, so that license data or customer data is automatically inserted into their copy of the software (that is, the software is built direct from version control specifically for that customer, helping to manage licensing and track piracy).
- **E-mail-based products**: This is just as we did with our purchasable voucher codes, where a custom e-mail is sent to the customer.
- **E-mail- or file-based products**: We could combine our newly-created file-based products with an e-mail-based product, and have the files be sent as e-mail attachments.
- **Drop shipping orders**: If we have a supplier who offers a drop shipping service, where they fulfill our orders, we could (assuming they had a suitable API) integrate with their systems to have the order dispatched.

C
Cookbook

Throughout the course of this book, we have developed a very flexible framework, which we have illustrated several times, by rapidly extending the system to fit new needs, or add new features. In this chapter, we will look at some smaller code snippets that can add additional value to our framework. In this chapter, you will learn:

- How to remind customers about forgotten details
- How to integrate Campaign Monitor in order to add new customers to our mailing list
- How to prevent spam signups with reCAPTCHA
- How to tweet about happy customers each time an order is paid

Authentication reminders

One useful feature for our framework would be to allow our customers to easily reset their password or to send them notification of their username.

Help! I forgot my password!

When a customer forgets their password, we can't just e-mail them a copy, because passwords are stored as a hash in the database. We also can't just reset the password, as fraudulent requests for new passwords would become a nuisance for customers.

The solution to this is to generate a password reset key when a customer informs us that they have forgotten their password. We then e-mail the customer a link to a "reset password" page, with the reset key in the URL. The reset key is used to verify the customer resetting the password is the owner of that user account.

Our users table already has a suitable field for this, `pwd_reset_key`; all we need now is the code!

Generate the reset key, update the user record, and e-mail the customer

This section of code simply creates a reset key for the user, and e-mails it to the customer, as part of a special URL the customer can use to reset their password.

```
$email = $this->registry->getObject('db')->
    sanitizeData( $_POST['email'] );
$sql = "SELECT * FROM users WHERE email='{$email}'";
$this->registry->getObject('db')->executeQuery( $sql );
if( $this->registry->getObject('db')->numRows() == 1 )
{
  $changes = array();
  $changes['pwd_reset_key'] = generatePasswordKey(8);
  $this->registry->getObject('db')->updateRecords('users', $changes,
         "email='{$email}'");
  // email the customer a link to
  // user/reset-password/userid-pwd_reset_key
}
function generatePasswordKey( $length = 8 )
{
  $characters = '0123456789abcdefghijklmnopqrstuvwxyz';
  $string = '';
  for ( $i = 0; $i < $length; $i++ )
  {
    $string .= $characters[mt_rand(0, strlen($characters))];
  }
  return $string;
}
```

Reset the password

This code would be part of the "reset password" page (which is accessed using the "reset password" URL). This splits part of the URL to extract the user ID and the password reset key. It then updates the user's password, assuming their password and confirmation match and the reset key matches that of the user ID.

```
$data = explode('-', $urldata[2]);
$userid = intval( $data[0] );
$key = $data[1];
if( $_POST['new_password'] == $_POST['confirm_newpassword'] )
{
  $pwd = md5( $_POST['new_password'] );
  $sql = "SELECT * FROM users
          WHERE ID={$userid} AND pwd_reset_key='{$key}'";
```

```
    $this->registry->getObject('db')->executeQuery( $sql );
    if( $this->registry->getObject('db')->numRows() == 1 )
    {
      $changes = array();
      $changes['password'] = $pwd;
      $this->registry->getObject('db')->
          updateRecords('users', $changes, "ID=" . $userid);
      // e-mail customer confirmation?
    }
  }
```

Help! I forgot my username!

If a customer forgets their username, we will require them to enter their e-mail address into a reminder form. If they can't remember their e-mail address, there is little we can do automatically, but they could still get in contact and inform us of their delivery address or confirm some details from a recent order, should they need to.

```
$email = $this->registry->getObject('db')->
    sanitizeData( $_POST['email'] );
$sql = "SELECT username FROM users WHERE email='{$email}'";
$this->registry->getObject('db')->executeQuery( $sql );
if( $this->registry->getObject('db')->numRows() > 0 )
{
  $data = $this->registry->getObject('db')->getRows();
  // send email to the customer, include their username
}
```

E-mailing customers

Throughout this book, and also in this chapter, we have worked on features that would need to e-mail the customer. We haven't actually implemented any e-mailing functionality to our store. Let's have a brief look at how we could do this.

To make a flexible e-mail system, we should be able to plug in e-mail templates and change how we would deliver the e-mail. We already have a template system in our framework, which can take view templates, and interchange data where template variables are.

We could take this system, and use it for e-mails, building a populated e-mail template. Within this object, we could code functionality to send the e-mail using PHP `mail()` (most-commonly available), SMTP (primarily useful for Windows servers or enterprise setups), or a third-party library (useful for complicated or custom setups, such as Gmail, Exchange, or IMAP where custom settings or non-standard configurations are required, such as Gmail's security requirements). A setting would be required to define which of these methods we would use to send the e-mail, and a switch statement would be used to process the e-mail, and send it using the correct method.

Integrating Campaign Monitor

Campaign Monitor is a really useful e-mail newsletter application. It makes sending e-mail newsletter campaigns really simple, and tracks various statistics including:

- Who opened the e-mails
- Who clicked on links in the e-mails
- Which e-mail programs were used to open the e-mails
- Reports over time

Because the service is hosted, the subscribers are stored in the Campaign Monitor database. Thankfully, it is quite simple to integrate. All we need to do is open up a URL with our API key and list ID (both available from the Campaign Monitor control panel), and the subscriber's name and e-mail address—both of these pieces of information we take when the customer signs up.

```
$newsletter_ping =
    fopen("http://api.createsend.com/api/api.asmx/Subscriber.Add?
    ApiKey=".$csAPIKey."
    ListID=".$csListId."&
    Email=".$_POST['register_email']."&
    Name=" . urlencode( $_POST['register_name']), "r");
```

Integrating reCAPTCHA

reCAPTCHA is a useful tool to prevent automated spam signups. We discussed it in Chapter 14, *Deploying, Security, and Maintenance*. There are a number of advantages and disadvantages to using this—one advantage being that it helps prevent automated signups, a disadvantage being sometimes they can be difficult to read, and thus act as a barrier to sign up.

The reCAPTCHA website has a PHP library available, `http://recaptcha.net/plugins/php/`. We need to download this, and sign up for an API key. When we have done this, we simply need to put some code into the signup process.

On the registration page

On the registration page, we require the following code; this includes the library, sets the API key, and adds the reCAPTCHA HTML to the form.

```
require_once('lib/recaptchalib.php');
$publickey = "APIKEY";
$this->registry->getObject('template')->getPage()->
    addTag('captcha', recaptcha_get_html($publickey) );
```

When processing the registration

When the customer submits their registration, we need the following code to check their response to the CAPTCHA challenge was correct:

```
require_once('lib/recaptchalib.php');
$privatekey = "APIKEY";
$resp = recaptcha_check_answer ($privatekey,
                                $_SERVER["REMOTE_ADDR"],
                                $_POST["recaptcha_challenge_field"],
                                $_POST["recaptcha_response_field"]);
if (!$resp->is_valid)
{
  // the sign up wasn't successful, store this, and display an error
}
```

Tweeting about happy customers

With a sharp rise in social networking, automated tweets can be a nice touch to add to a website. We could have our framework automatically send a tweet each time a customer pays for an order.

There is a simple, easy-to-use PHP Twitter library available, which makes sending Twitter updates a breeze: `http://emmense.com/php-twitter/`.

To use the library, all we need to do is include the library, create a new Twitter object, set our username and password, and then call the update method. To send a tweet on each purchase, we just add the following code into our payment method object, where it updates the status of orders.

```
requier_once('lib/class.twitter.php');

$t = new Twitter;
$t->username = 'TWITTER USERNAME';
$t->password = 'TWITTER PASSWORD';

$t->update('Another happy customer has just completed a purchase
            with us! Visit our store www.ourstore.com');
```

Other uses

There are, of course, a number of other potential uses for this, including:

- Tweeting every time we add a new product to the store
- Tweeting every time a new customer signs up
- Tweeting every time we update a product and reduce the price to inform customers of a special offer
- Tweeting if we enable a sale mode (of course, we would need to implement a sale mode!)

Summary

In this chapter, we have looked at how we can make some really simple but useful improvements to our framework, by utilizing other services and libraries and just adding a few lines of code to our system. The number of improvements we can make are endless; some options include:

- Integrating graphs and charts into our administration area, using
    - Google charts
    - PHP chart libraries
    - JavaScript chart libraries
- Bringing jQuery improvements to the design. jQuery is a great JavaScript library, which can enhance the user interface. It has a number of plugins and code snippets available, including:
    - `autocomplete`: This plugin makes searching for products easier by auto-completing products in the database.
    - `uploadprogress`: When a customer uploads a file for a customizable product, this plugin would show the progress of the upload.
    - Toggle images: For products where we have a number of photographs uploaded, we could use JavaScript to toggle between the different images, swapping a larger image with the larger version of a thumbnail image. This is a code snippet, which can be copied from `http://www.michaelpeacock.co.uk/blog/entry/manual-photo-filmstrip-in-jQuery`.

Index

Symbols

__deconstruct method 37
1&1 Internet Inc 275
eBay 12

A

addProduct method 165-168
administration
  categories 261
  customer area 269
  dashboard 260
  miscellaneous 269
  orders 267
  products 261
administration areas, extending. *See* **payment areas, extending**
affectedRows method 36
Amazon, features
  delivery address choosing flexibility 185
  detailed basket 184
  gift wrapping 184
  payment history, tracking 185
  streamlined authentication 184
Amazon
  about 8, 12
  checkout button 183
  features 12, 184
  limitation 183
  stages 182, 183
Amazon's Market Web Service. *See* **Amazon MWS**
Amazon MWS 309
A Small Orange, hosting providers 275
authentication, order process
  about 225, 226
  delivery address, considering 226
authentication, process
  login 191
  need for 191
  registering 191
authentication reminders
  about 317
  password, recovering 317
  pwd_reset_key 317
  username, recovering 319

B

backing up, site maintenance
  cPanel, using 284
  SSH 286
basket, process
  overview 190
  shipping method 190
  voucher codes, adding 189
Basketcontroller
  empty basket template, inserting 180
  smallBasket method, adding 179
Brick 'N Mortar stores 8

C

cacheData method 33
cacheQuery method 32
campaign monitor
  about 320
  uses 320
categories
  about 83
  controller 89-91
  creating 265
  deleting 266

editing 266
getCategory method 84
images 92
model 84-87
new category, adding 266
routing 92, 93
view, building 87
viewCategory method 89
centralized download area
creating 315
checkBasket method 162, 164
closeConnection method 31
content, structuring
additonal functionality 63
advanced content types 63
pages 63
versioning 64
content_types_products table
extending 311
tables, adding 311
contents, embedding
about 93
featured product, viewing 93
contents, shopping basket
addProduct method 165-168
controller 168-170
customizable products, adding 170
etiquette 170
products, adding 165
product variants, adding 172
quantities, editing 174-176
viewing 162
controller
tasks 74, 76
cost determination, shipping
location-based shipping cost 201
product-based shipping cost 200
shipping methods, using 199
ways 199
weight-based shipping cost 200, 201
cPanel
backup, restoring from 285
database, backing up 284, 285
database, restoring from 285
site, backing up 284, 285
using 284
credit card
details, not storing 243
details, storing 242, 243
CubeCart 10
customer's basket
empty basket 159
main page, viewing 159
viewing 159
customer area
about 269
listing 269
selected user's ID, detecting 269
customer retention
coupons 301
newsletters, sending 301
social features 301
techniques 301
voucher codes 301
customizable products, shopping basket
adding 170
basket, viewing 171
basket templates 111
controller 172
database, modifying 171
model changes, making 171, 172
product customizations 111
product variations 111
purchasing 170
stock control 110
subtotals 111

D

dashboard
sample screen 261
statistics 260
statistics, generating 260
data, discount codes
database fields 215
storing 214
types, fixed amount deducted 214
types, fixed amount set to shipping 214
types, percentage 214
database, manual deployment
creating, on hosting account 277, 278
local database, exporting 278, 279
local database, importing 279
setting up 276
database changes, referrals
credit field 221

referrers table, fields 221
database object
__deconstruct method 37
affectedRows method 36
cacheData method 33
cacheQuery method 32
closeConnection method 31
dataFromCache method 33
deleteRecords method 34
executeQuery method 35
extending 37
getRows method 36
information, debugging 37
inheritance 37
insertRecords method 35
logic abstraction, to queries 37
newConnection method 31
numRowsFromCache method 32
resultsFromCache method 33
sanitizeData method 36
setActiveConnection method 32
updateRecords method 34, 35
database structure, users control
changing 107
products table, allow_upload (Boolean) field 107
products table, changing 107
products table, custom_text_inputs (longtext) field 107
dataFromCache method 33
data management
categories 70
content 65
content, types 67
content, versions 68
database, designing 65
products 69
deleteRecords method 34
delivery address, order process
confirmation page 228
setting 227
delivery address, process 191
directCall parameter 81
discount codes
about 213
data, storing 214
functionality 215, 216
options 213
discount codes functionality
code quantity, reducing 219
codes 216-218
tasks 216
Drupal e-commerce 10

E

e-commerce
about 7, 59
administration 259
applications 7, 10
checkout process 14
CubeCart 10
Drupal e-commerce 10
examples 181
Magento 10
need for 9
overview 7
product-related features 13
registry 59
required key features 13
shipping 197
shopping basket 157
site 10
supplementary features 14
tax 209
users 8
users choice 95
e-commerce, examples
Amazon 182
eBay 185
Play.com 187
reviewing 181
e-commerce, Google Analytics
item, adding 308
tracking 307
transaction, adding 307, 308
transaction, tracking 308
e-commerce, sites
Amazon 12
eBay 12
iStockphoto 11
Play.com 12
WooThemes 11
e-commerce, users
Amazon 8

Brick 'N Mortar stores 8
eBay 8
service-based companies 8
e-mailing customers 319
eBay
about 8, 185, 309, 310
distinguishing feature 186
features 12, 186
working 186
electronic commerce. *See* **e-commerce**
executeQuery method 35, 36
expansion
exclusive discounts 257
feedback area 257
pre-releases advanced notices 257
product, recommending 256
returns, handling 256

F

feed creation, Google products
about 304
product feed controller, creating 305
useful links 306
filterProducts() method 129
framework
authentication reminders 317
building 24
creating 9
deploying, into production environment 273
designing 19
existing package, using 10
fundamental features 14, 15
Juniper Theatricals 16
need for 9
PHP 9
planning 19
security, enhancing 281
site, maintaining 283
tasks 15
framework, building
diagrammatic representation 24
MVC pattern, implementing 25
registry objects 29
registry pattern, implementing 25
requests, routing 54
singleton pattern, implementing 27
framework, designing
patterns 20
structure 23
framework, structure
administration controllers 23
controllers 23
directory structure 23
models 23
registry 23
views 23
framework powered sites, marketing
affiliate marketing 296
marketing materials 295
newsletters 295
online advertising 290
social marketing 296

G

getProperties() method 71
getRows method 36
Google Analytics
about 306
e-commerce, tracking 307
feature 306
signing up 307
Google products
about 303
feed, adding to merchant center 304
feed, creating 304
feed of products, adding 303
Google Base Data API, using 306
merchant center 303
shopping.com 306
shopzilla 306
update schedule, setting 304

H

headings, on-site SEO 297

I

imagecreateresampled function 262
information, building
category information 62
content, structuring 63
production information 62

requirements 61
insertRecords method 35
isActive() method 71
isSecure() method 71
iStockphoto
about 11
features 11
isValid() method 71

J

Juniper Theatricals
about 16, 113
framework 16, 17

K

keyword search, search 104

L

location-based shipping cost
regional shipping costs 202
third-party APIs 202
ways 201

M

Magento 10
manual deployment, site deployment
database, setting up 276
settings, modifying 280
store, uploading 279, 280
MediaTemple 275
miscellaneous
new voucher codes, adding 271
shipping method, creating 270
voucher codes 270
voucher codes, creating 270, 271
Model-View-Controller. *See* **MVC pattern**
MVC pattern
about 20, 21
user interface (view) 20
working 21

N

newConnection method 31
numRowsFromCache method 32

O

off-site SEO
about 300
inbound links 300
services 300
offline payment 245
on-site SEO
about 297
content 298
headings 297
links 298
meta tags 298
meta tags, description 298, 299
meta tags, keywords 298, 299
sitemap 299
webmaster tools 299
online advertising
advantages 290
newsletter advertising 293
PPC services 291
search engine, tips 294
search engine advertisement networks 292
search engine penalization 294
space, purchasing 290, 291
online payment
about 237
credit card, using 242
payment gateways 244
payment gateways, factors 244
PayPal 237
order confirmation
database, using 230
orders storage, database used 230
page, viewing 230
steps 230
order model
information, accessing 251
query, for order details 252
query, for order items 253
order process
authentication 225
payment method 228
reviewing 225
order process, reviewing
authentication 223, 225
basket, viewing 223

delivery address 227
delivery process 223
making payment 224
order confirmation 224, 230
order processed 224
payment details 224
payment method 223, 228
shopping basket, viewing 224
order storage, database used
order item, attributes 233
order items 232
order items, fields 232
orders table 231
orders table, fields 231
order statuses 232
order statuses, fields 232
payment methods 233
orders
cancelling 253
dispatch note, printing 268
listing 250
listing, query 251
refunds, processing 269
updating 267
viewing 251, 267
orders, cancelling
controller code 255, 256
order model additions 254
stages 253

P

pages, enabling within framework
about 70
getProperties() method 71
isActive() method 71
isSecure() method 71
isValid() method 71
methods, using 71, 72
model, adding 70
model, constructor 70
view, template files 73
password, recovering
customer, e-mailing 318
password, resetting 318, 319
reset key, generating 318
user record, updating 318
patterns, framework
MVC pattern 20, 21
registry 21
singleton 22
Pay-Per-Action 293
pay-per-click. *See* **PPC**
payment areas, extending
about 312
access, providing 313
access, rescinding 314
database, accessing 312
Payment Card Industry Data Security Standards. *See* **PCI DSS**
payment collection
about 235
offline payment 245
online process 237
payment page, displaying 236, 237
system 235, 236
payment gateways
factors 244
list 244
payment method, order process
list, generating 228, 229
section, viewing 229
payment method, process
off-site online payment method 192
off-site online payment method, advantages 192
off-site online payment method, disadvantages 192
offline payment method 192
offline payment method, advantages 192
offline payment method, disadvantages 192
on-site online payment method 192, 193
on-site online payment method, advantages 193
on-site online payment method, disadvantages 193
selecting 192
PayPal
about 237
payment, processing 239, 241
payment button 237-239
PCI DSS
control objectives 243

PHP
framework 9
Play.com
about 12, 187, 188
features 13, 188
order process, discussing 187
similarity, with Amazon 188
PPC 289
process
authentication 190
basket 189
confirmation 193
delivery address 191
order processed 194
payment details 193
payment method 192
structure 189
product, users control
customizing 105
custom text, handling 105
custom text, limitation 106
image, uploading 105
product filteration
about 119
attributes 120
database changes 120, 122
filter attribute association 122
filter attribute types 120
filter attribute values 121
filtered products, displaying 129
filtered result, storing 119
filter requests, processing 125-128
improving 130
options 122, 124
price range filtering 119
product ratings, social oriented features
about 148
displaying 151
information, capturing 149
saving 149
updating 150
user interface, improving 151
product reviews, social oriented features
about 152
adding 153, 154
comments, displaying 154
comments, processing 153
representing 152
submission form 153
products
category, relating to 264
changes, saving 265
controller, using 81, 82
creating 261, 262
customizing 265
editing 265
extending 311
functionality, building 76
image, adding 263
image, uploading 262, 263
images 92
model, functionality 77-79
other features 315
photograph, uploading 263
routing 92, 93
shipping costs 264
view 80
product search
constructor changes 115, 116
controlling, within products controller 115
improving 118, 119
results 117, 118
search features, adding 114
search function 116, 117
search box, adding 115
product stock alert
about 148
framework, modifying 143
model, altering 144
stock alerts database table 145-147
stock levels, detecting 144
template bit 144, 145
product variants, shopping basket
controller 174
database table, adding 173
model changes, making 173
purchasable voucher codes
existing functionality 219
existing functionality, product variations 220
existing functionality. discount codes 219
required additional functionality 220

purchase, wish lists
gift purchase 138
self purchase 138

R

Really Simple Syndication. *See* **RSS**
reCAPTCHA integration
about 320
advantages 320
disadvantages 320
on registration page 321
registration, processing 321
recommendation
E-mail recommendations 142
methods 139
related products, controlling 141
related products, displaying 139
related products, viewing 142
referrals
about 220
database changes 221
workflow, working 222
working 221
registry objects, framework
database object 29
database object, working 29-35
e-mail, parsing 53
e-mail, sending 52
filesystem management 54
security management 53
template management 42-51
template management, extending 52
user authentication 38-42
registry pattern, framework
code setting 26, 27
data setting 26
implementing 25
incoming URL, processing 25
need for 21
overview 22
pagination 25
storeObject method 26
tasks 22
URL, building 25
requests, routing
.htaccess file 58
alternative, router used 54
configuration file 58
index.php file 56, 57
URL, processing 55, 56
resultsFromCache method 33
RSS 297
rules, shipping
about 203
capped shipping 204
free shipping 204
representing, SQL used 203

S

sanitizeData method 36
search
filtering method 114
keyword search 114
product, filtering 119
product finding 114
search engine optimization. *See* **SEO**
Search Engine Results Pages. *See* **SERPs**
Secure Sockets Layer. *See* **SSL**
security
about 281
CAPTCHA 283
passwords 282
server security 281
SSL 283
TLS 283
SEO
about 297
off-site SEO 300
on-site SEO 297
primary areas 297
SERPs 294
server security
firewall 282
software 281
setActiveConnection method 32
shipping
about 197
costs, determining 199
costs, integrating into basket 205
rules 202
shipping method 197
tracking 204

shipping cost
determining 199
determining, ways 199
integrating, into basket 205
location-based shipping cost 201
product-based shipping cost 200
weight-based shipping cost 200, 201
shipping cost, integrating into basket
calculating, based on products 205
calculating, based on product weights 206
default shipping method, storing 205
prices, adjusting 207-209
rules, considering 207-209
shipping method
details, storing 198
details representation, SQL used 198
requirements 198
shopping basket
about 157
auctions 158
cleaning 178
contents, controlling 161
creating 160
customizable products, consequences 111
displaying, on each page 178
one-click payment 157
product customizations 160
product variations 160
service subscription payments 158
stock level 159
subtotals 160
templates 160
user's basket, emptying 178
viewing 212
shopping basket, creating
database 160
steps 160
user's basket, building 160
shopping basket, viewing
checkBasket method 162-164
controller 164, 165
stages 162
singleton pattern, framework
implementing 27, 29
using 22
site deployment
accounts, hosting 275
automated deployment 280
domain names 274, 275
manual deployment 276
providers 275
steps 273
web hosting 274
site maintenance
about 284
backing up 284
SKU 308
Slicehost 275
social marketing
about 296
RSS with FeedBurner 297
Twitter 296
viral marketing 296
social oriented features
product ratings 148
product reviews 152
ratings, combining with rating 155
SSH
database, backing up 286
database, restoring 286
site, backing up 286
site, restoring 286
SSL 283
stock keeping unit. *See* **SKU**
storeObject method 26

T

tax
calculating, ways 210, 211
location-based tax costs 211
tackling, ways 209
TLS 283
transferToUser function
about 177
using 177
Transport Layer Security. *See* **TLS**
tweeting
about 322
other uses 322
Twitter 296

PACKT PUBLISHING
This material is copyright and is licensed for the sole use by jackie tracey on 23rd February 2010
953 Quincy Drive, , Brick, , 08724

U

updateRecords method 34, 35
uploads, users control
maintaining 106
security considerations 107
user account area
about 247
details, changing 247
expansion 256
user account area, details
changing 247
default delivery address, changing 249, 250
password, changing 248, 249
user experience
importance 114
improvements 155
username, recovering
steps 319
users choice
about 95
simple variants 96
simple variants, advantage 104
variant combination 96
users control
database structure, changing 107
example 104
product, customizing 105
template, switching 108-110
uploads, maintaining 106

V

variant combination, users choice
attributes 96, 97
attributes table 98
attribute values table 98
database structure 98
high-level overview 97
product attribute value association table 99
template, changing 100-102
templates 103
view, categories
building 87
category template 88
products template 88
subcategories template 88
view, page
404 error template 74
footer template 73
header template 73
other templates 74
page template 74
template files 73
viewCategory method 89
viral marketing 296
Virtual Private Server. *See* **VPS**
VPS 274

W

web hosting, site deployment
domain name registrar 276
nameserver changes 276
providers 275
Web Hosting Talk forum 275
wish lists
about 130
improving, ways 138
product, adding 135
purchase 137
saving 132
saving, controller 132-135
structure, creating 131
viewing 137
viewing, controller changes 135-137
WooThemes
about 11
features 11



Thank you for buying
PHP 5 e-commerce Development

Packt Open Source Project Royalties

When we sell a book written on an Open Source project, we pay a royalty directly to that project. Therefore by purchasing PHP 5 e-commerce Development, Packt will have given some of the money received to the PHP group project.

In the long term, we see ourselves and you—customers and readers of our books—as part of the Open Source ecosystem, providing sustainable revenue for the projects we publish on. Our aim at Packt is to establish publishing royalties as an essential part of the service and support a business model that sustains Open Source.

If you're working with an Open Source project that you would like us to publish on, and subsequently pay royalties to, please get in touch with us.

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.

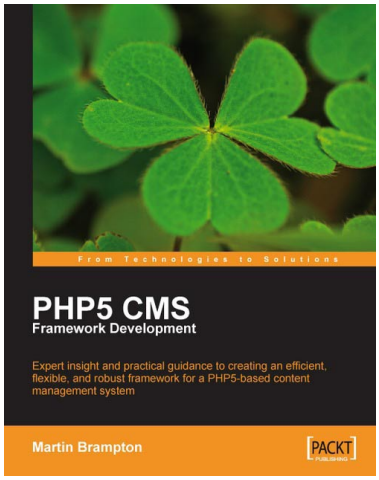
About Packt Publishing

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`.

PACKT PUBLISHING

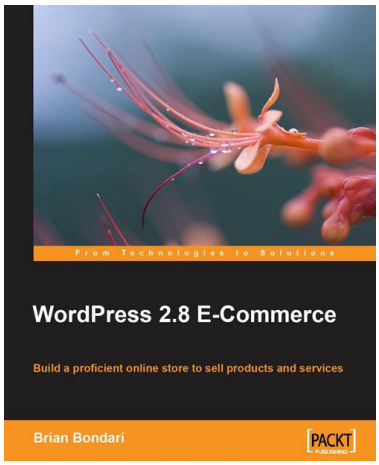


PHP 5 CMS Framework Development

ISBN: 978-1-847193-57-5 Paperback: 348 pages

Expert insight and practical guidance to creating an efficient, flexible, and robust framework for a PHP 5-based content management system

1. Learn how to design, build, and implement a complete CMS framework for your custom requirements
2. Implement a solid architecture with object orientation, MVC
3. Build an infrastructure for custom menus, modules, components, sessions, user tracking, and more
4. Written by a seasoned developer of CMS applications



WordPress 2.8 E-Commerce

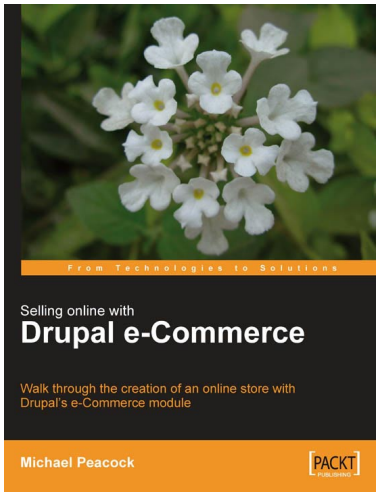
ISBN: 978-1-847198-50-1 Paperback: 292 pages

Build a proficient online store to sell products and services

1. Earn huge profits by transforming WordPress into an intuitive and capable platform for e-Commerce
2. Build and control a vast product catalog to sell physical items and digital downloads
3. Configure and integrate various payment gateways into your store for your customers' convenience
4. Promote and market your store online for increased profits

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

PACKT PUBLISHING

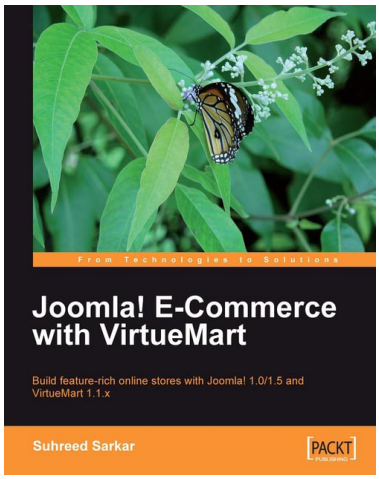


Selling Online with Drupal e-Commerce

ISBN: 978-1-847194-06-0 Paperback: 264 pages

Walk through the creation of an online store with Drupal's e-Commerce module

1. Set up a basic Drupal system and plan your shop
2. Set up your shop, and take payments
3. Optimize your site for selling and better reporting
4. Manage and market your site



Joomla! E-Commerce with VirtueMart

ISBN: 978-1-847196-74-3 Paperback: 476 pages

Build feature-rich online stores with Joomla! 1.0/1.5 and VirtueMart 1.1.x

1. Build your own e-commerce web site from scratch by adding features step-by-step to an example e-commerce web site
2. Configure the shop, build product catalogues, configure user registration settings for VirtueMart to take orders from around the world
3. Manage customers, orders, and a variety of currencies to provide the best customer service
4. Handle shipping in all situations and deal with sales tax rules

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