

THE PHP ANTHOLOGY

101 ESSENTIAL TIPS, TRICKS & HACKS

BY DAVEY SHAFIK
MATTHEW WEIER O'PHINNEY
LIGAYA TURMELLE
HARRY FUECKS
BEN BALBO
2ND EDITION



SOLUTIONS TO THE MOST COMMON PROGRAMMING PROBLEMS

The PHP Anthology: 101 Essential Tips, Tricks and Hacks, 2nd Edition (Chapters 2, 10, and 11)

Thank you for downloading these sample chapters of *The PHP Anthology 101 Essential Tips, Tricks, and Hacks, 2nd Edition*, published by SitePoint.

This excerpt includes the Summary of Contents, Information about the Author, Editors and SitePoint, Table of Contents, Preface, three chapters from the book, and the index.

We hope you find this information useful in evaluating this book.

[For more information, visit sitepoint.com](http://sitepoint.com)

Summary of Contents of this Excerpt

Preface.....	xi
2. Using Databases with PDO.....	39
10. Access Control.....	269
11. Caching.....	363
Index.....	505

Summary of Additional Book Contents

1. Introduction.....	39
3. Strings.....	77
4. Dates and Times.....	95
5. Forms, Tables, and Pretty URLs.....	115
6. Working with Files.....	147
7. Email.....	179
8. Images.....	197
9. Error Handling.....	237
12. XML and Web Services.....	395
13. Best Practices.....	435
A. PHP Configuration.....	473
B. Hosting Provider Checklist.....	483
C. Security Checklist.....	489
D. Working with PEAR.....	497



THE PHP ANTHOLOGY

101 ESSENTIAL TIPS, TRICKS & HACKS

BY DAVEY SHAFIK
MATTHEW WEIER O'PHINNEY
LIGAYA TURMELLE
HARRY FUECKS
BEN BALBO
2ND EDITION

The PHP Anthology: 101 Essential Tips, Tricks & Hacks

by Davey Shafik, Matthew Weier O'Phinney, Ligaya Turmelle, Harry Fuecks, and Ben Balbo

Copyright © 2007 SitePoint Pty. Ltd.

Expert Reviewer: Jason Sweat

Editor: Georgina Laidlaw

Managing Editor: Simon Mackie

Editor: Hilary Reynolds

Technical Editor: Andrew Tetlaw

Index Editor: Fred Brown

Technical Director: Kevin Yank

Cover Design: Alex Walker

Printing History:

First Edition: December, 2003

Second Edition: October, 2007

Notice of Rights

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.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

424 Smith Street Collingwood

VIC Australia 3066

Web: www.sitepoint.com

Email: business@sitepoint.com

ISBN 978-0-9758419-9-0

Printed and bound in the United States of America

Ben Balbo

Ben Balbo was born in Germany, grew up in the UK, lives in Melbourne, and likes Guinness. While he isn't drinking Guinness (which is most of the time in Melbourne, as it just doesn't taste the same), he earns a living as a PHP developer and trainer, security consultant, and Open Source developer. He has been known to talk in public about web development-related topics, which comes as part of the package of being on the committees of both the Melbourne PHP User Group and Open Source Developers' Club. Although he wouldn't admit this, he participates at this level only in order to go to restaurants or pubs after the meetings.

Harry Fuecks

Harry Fuecks¹ is a technical writer, programmer, and system engineer. He has worked in corporate IT since 1994, having completed a Bachelor's degree in Physics. He first came across PHP in 1999, while putting together a small intranet. Today, he's the lead developer of a corporate extranet, where PHP plays an important role in delivering a unified platform for numerous back office systems. In his off hours he writes technical articles for SitePoint and runs phpPatterns,² a site exploring PHP application design. Originally from the United Kingdom, he now lives in Switzerland. Harry is the proud father of a beautiful baby girl who keeps him busy all day (and night!).

Davey Shafik

Davey Shafik is a full-time PHP developer with ten years' experience in PHP and related technologies. An avid magazine writer, book author, and speaker, Davey keeps his mind sharp by trying to tackle problems from a unique perspective from his home in Central Florida where he lives with five cats and more computers.

Ligaya Turmelle

Ligaya Turmelle is a full-time goddess, occasional PHP programmer, and obsessive world traveler. Actively involved with the PHP community as a founding Principal of phpwomen.org, administrator at codewalkers.com, roving reporter for the Developer Zone on Zend.com, and PHP blogger and long-time busybody of #phpc on freenode, she hopes to one day actually meet the people she talks to. When not sitting at her computer staring at the screen, Ligaya can usually be found either playing golf, scuba diving, snorkeling, kayaking, hiking, or just playing with the dogs outside. Ligaya Turmelle is a Zend Certified Engineer.

¹ Harry Fuecks photo credit: Bruno Gerber <http://www.flickr.com/photos/beegee74/231137320/>

² <http://www.phppatterns.com/>

Matthew Weier O'Phinney

Matthew Weier O'Phinney is a full-time father of two and spends his free time developing in PHP. He is a PEAR developer, core contributor to Zend Framework, and all-around PHP 5 proponent—though PHP 6 cannot come soon enough for him.

About the Expert Reviewer

Jason Sweat has used PHP since 2001, where he was searching for a free—as in beer—substitute for IIS/ASP to create an accounting system for a home business. His Unix administrator pointed him towards Linux, Apache, and PHP. He has since adopted PHP as an intranet development standard at work, as well as using PHP in a Unix shell scripting environment. He is the author of *php| architect's Guide to PHP Design Patterns* (Toronto: Marco Tabini & Associates, 2005), and was a co-author of *PHP Graphics Handbook* (Birmingham: Wrox 2003), has published several articles for the Zend web site and for *php| architect* magazine, and has presented numerous talks on PHP at various conferences. Jason is a Zend Certified Engineer, and maintains a blog at <http://blog.casey-sweat.us/>.

About the Technical Editor

Andrew Tetlaw has been tinkering with web sites as a web developer since 1997 and has also worked as a high school English teacher, an English teacher in Japan, a window cleaner, a car washer, a kitchen hand, and a furniture salesman. At SitePoint he is dedicated to making the world a better place through the technical editing of SitePoint books and kits. He is also a busy father of five, enjoys coffee, and often neglects his blog at <http://tetlaw.id.au/>.

About the Technical Director

As Technical Director for SitePoint, Kevin Yank oversees all of its technical publications—books, articles, newsletters, and blogs. He has written over 50 articles for SitePoint, but is best known for his book, *Build Your Own Database Driven Website Using PHP & MySQL*. Kevin lives in Melbourne, Australia, and enjoys performing improvised comedy theatre and flying light aircraft.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our books, newsletters, articles, and community forums.

Table of Contents

Preface	xv
Who Should Read this Book?	xvi
What's Covered in this Book?	xvii
Running the Code Examples	xix
The Book's Web Site	xx
The SitePoint Forums	xxi
The SitePoint Newsletters	xxi
Your Feedback	xxi
Conventions Used in this Book	xxi
Chapter 1 Introduction	1
Where do I get help?	2
What is OOP?	9
How do I write portable PHP code?	33
Summary	38
Chapter 2 Using Databases with PDO	39
What is PDO?	40
How do I access a database?	41
How do I fetch data from a table?	44
How do I resolve errors in my SQL queries?	49
How do I add data to, or modify data in, my database?	53
How do I protect my web site from an SQL injection attack?	55
How do I create flexible SQL statements?	57
How do I find out how many rows I've touched?	59

How do I find out a new INSERT's row number in an autoincrementing field?	62
How do I search my table?	63
How do I work with transactions?	65
How do I use stored procedures with PDO?	67
How do I back up my database?	69
Summary	75
Chapter 3 Strings	77
How do I output strings safely?	79
How do I preserve formatting?	81
How do I strip HTML tags from text?	82
How do I force text to wrap after a certain number of characters?	84
How do I perform advanced search and replace operations?	84
How do I break up text into an array of lines?	86
How do I trim whitespace from text?	88
How do I output formatted text?	88
How do I validate submitted data?	90
Summary	94
Chapter 4 Dates and Times	95
How do I use Unix timestamps?	96
How do I obtain the current date?	98
How do I find a day of the week?	101
How do I find the number of days in a month?	101
How do I create a calendar?	102
How do I store dates in MySQL?	107
How do I format MySQL timestamps?	109
How do I perform date calculations using MySQL?	111
Summary	112

Chapter 5	Forms, Tables, and Pretty URLs . . .	115
	How do I build HTML forms with PHP?	116
	How do I display data in a table?	127
	How do I display data in a sortable table?	130
	How do I create a customized data grid?	134
	How do I make "pretty" URLs in PHP?	139
	Summary	145
Chapter 6	Working with Files	147
	How do I read a local file?	148
	How do I use file handles?	153
	How do I modify a local file?	155
	How do I access information about a local file?	157
	How do I examine directories with PHP?	160
	How do I display PHP source code online?	161
	How do I store configuration information in a file?	163
	How do I access a file on a remote server?	166
	How do I use FTP from PHP?	167
	How do I manage file downloads with PHP?	170
	How do I create compressed ZIP/TAR files with PHP?	172
	How do I work with files using the Standard PHP Library in PHP 5?	174
	Summary	177
Chapter 7	Email	179
	How do I send a simple email?	179
	How do I simplify the generation of complex emails?	182
	How do I add attachments to messages?	184
	How do I send HTML email?	186

How do I mail a message to a group of people?	188
How do I handle incoming mail with PHP?	191
How can I protect my site against email injection attacks?	193
Summary	195
Chapter 8 Images	197
How do I specify the correct image MIME type?	198
How do I create thumbnail images?	199
How do I resize images without stretching them?	202
How can I put together a simple thumbnail gallery?	214
How do I extract EXIF information from images?	217
How do I add a watermark to an image?	220
How do I display charts and graphs with PHP?	223
How do I prevent the hotlinking of images?	230
How do I create images that can be verified by humans only?	234
Summary	235
Chapter 9 Error Handling	237
What error levels does PHP report?	238
What built-in settings does PHP offer for error handling?	239
How can I trigger PHP errors?	241
How do I implement a custom error handler with PHP?	242
How do I log and report errors?	247
How can I use PHP exceptions for error handling?	248
How do I create a custom Exception class?	252
How do I implement a custom exception handler with PHP?	257
How can I handle PHP errors as if they were exceptions?	260
How do I display errors and exceptions gracefully?	261
How do I redirect users to another page following an error condition?	265

Summary	267
Chapter 10 Access Control	269
How do I use HTTP authentication?	271
How do I use sessions?	277
How do I create a session class?	281
How do I create a class to control access to a section of the site?	283
How do I build a registration system?	297
How do I deal with members who forget their passwords?	318
How do I let users change their passwords?	330
How do I build a permissions system?	339
How do I store sessions in a database?	353
Summary	362
Chapter 11 Caching	363
How do I prevent web browsers from caching a page?	365
How do I control client-side caching?	367
How do I examine HTTP headers in my browser?	371
How do I cache file downloads with Internet Explorer?	372
How do I use output buffering for server-side caching?	373
How do I cache just the parts of a page that change infrequently? ..	377
How do I use PEAR::Cache_Lite for server-side caching?	382
What configuration options does Cache_Lite support?	385
How do I purge the Cache_Lite cache?	389
How do I cache function calls?	390
Summary	392
Chapter 12 XML and Web Services	395
Which XML technologies are available in PHP 5?	396

Why should I use PHP's XML extensions instead of PHP string functions?	396
How do I parse an RSS feed?	398
How do I generate an RSS feed?	405
How do I search for a node or content in XML?	409
How can I consume XML-RPC web services?	412
How do I serve my own XML-RPC web services?	416
How can I consume SOAP web services?	420
How do I serve SOAP web services?	423
How can I consume REST services?	425
How can I serve REST services?	431
Summary	433
Chapter 13 Best Practices	435
How do I track revisions to my project's code?	436
How can I maintain multiple versions of a single codebase?	438
How can I write distributable code?	441
How can I document my code for later reference by myself or others?	448
How can I ensure future changes to my code won't break current functionality?	454
How can I determine what remains to be tested?	463
I've reviewed some of my old code, and it's horrible. How can I make it better?	467
How can I deploy code safely?	468
Summary	471
Appendix A PHP Configuration	473
Configuration Mechanisms	473
Key Security and Portability Settings	475

Includes and Execution Settings	475
Error-related Settings	480
Miscellaneous Settings	481
Appendix B Hosting Provider Checklist	483
General Issues	483
PHP-related Issues	485
Appendix C Security Checklist	489
Top Security Vulnerabilities	489
Appendix D Working with PEAR	497
Installing PEAR	498
The PEAR Package Manager	501
Installing Packages Manually	503
Alternatives to PEAR	504
Index	505

Preface

One of the great things about PHP is its vibrant and active community. Developers enjoy many online meeting points, including the SitePoint Forums,¹ where developers get together to help each other out with problems they face on a daily basis, from the basics of how PHP works, to solving design problems like “How do I validate a form?” As a way to get help, these communities are excellent—they’re replete with all sorts of vital fragments you’ll need to make your projects successful. But putting all that knowledge together into a solution that applies to your particular situation can be a challenge. Often, community members assume other posters have some degree of knowledge; frequently, you might spend a considerable amount of time pulling together snippets from various posts, threads, and users (each of whom has a different programming style) to gain a complete picture.

The PHP Anthology: 101 Essential Tips, Tricks & Hacks, 2nd Edition is, first and foremost, a compilation of the best solutions provided to common PHP questions that turn up at the SitePoint Forums on a regular basis, combined with the experiences and insights our authors have gained from their many years of work with PHP.

What makes this book a little different from others on PHP is that it steps away from a tutorial style, and instead focuses on the achievement of practical goals with a minimum of effort. To that extent, you should be able to use many of the solutions provided here in a plug-and-play manner, without having to read this book from cover to cover. To aid you in your endeavours, each section follows a consistent question-and-solution format. You should be able to scan the table of contents and flip straight to the solution to your problem.

That said, threaded throughout these discussions is a hidden agenda. As well as solutions, this book aims to introduce you to techniques that can save you effort, and help you reduce the time it takes to complete and maintain your web-based PHP applications.

Although it was originally conceived as a procedural programming language, in recent years PHP has proven increasingly successful as a language for the develop-

¹ <http://www.sitepoint.com/forums/forumdisplay.php?f=34>

ment of object oriented solutions. With the release of PHP 5, PHP gained a completely rewritten and more capable object model. This has been further reinforced by the fact that on July 13, 2007 the PHP development team made the end-of-life announcement for PHP 4.

The object oriented paradigm seems to scare many PHP developers, and is often regarded as being off limits to all but the PHP gurus. What this book will show you is that you don't need a computer science degree to take advantage of the object oriented features and class libraries available in PHP 5 today.

The PHP Extension and Application Repository, known as PEAR,² provides a growing collection of reusable and well-maintained solutions for architectural problems (such as web form generation and validation) regularly encountered by PHP developers around the world. Wherever possible in the development of the solutions provided in this book, we've made use of freely available libraries that our authors have personally found handy, and which have saved them many hours of development.

The emphasis this book places on taking advantage of reusable components to build your PHP web applications reflects another step away from the focus of many current PHP-related books. Although you won't find extensive discussions of object oriented application design, reading *The PHP Anthology: 101 Essential Tips, Tricks & Hacks, 2nd Edition* from cover to cover will, through a process of osmosis, help you take your PHP coding skills to the next level, setting you well on your way to constructing applications that can stand the test of time.

The PHP Anthology: 101 Essential Tips, Tricks & Hacks, 2nd Edition will equip you with the essentials with which you need to be confident when working the PHP engine, including a fast-paced primer on object oriented programming with PHP (see "What is OOP?" in Chapter 1). With that preparation out of the way, the book looks at solutions that could be applied to almost all PHP-based web applications, the essentials of which you may already know, but have yet to fully grasp.

Who Should Read this Book?

If you've already gotten your feet wet with PHP, perhaps having read Kevin Yank's *Build Your Own Database Driven Website Using PHP & MySQL, 3rd Edition* (Site-

² <http://pear.php.net/>

Point, Melbourne, ISBN 0-9752402-1-8), and completed your first project or two with PHP, then this is the book for you.

If you've been asking questions like "How do I validate a web page form?", "How do I add a watermark to my photos?", or "How do I send automated email messages from my web application?", you'll find the answers to those questions in this book. If you have the drive to progress your skills or improve your web application through concepts such as reusable components, caching performance, or web services, then you will find this book to be an excellent primer.

What's Covered in this Book?

Here's what you'll find in each of the chapters of this book:

Chapter 1: Introduction

This chapter provides a useful guide to finding help through the PHP manual and other resources. It includes an introduction object oriented programming: a run-down of PHP's class syntax, as well as a primer that explains how all the key elements of the object oriented paradigm apply to PHP. It's essential preparatory reading for later chapters in this anthology. This chapter also provides tips for writing portable code, and gives us the chance to take a look at some of the main PHP configuration pitfalls.

Chapter 2: Using Databases with PDO

This chapter provides you with everything you'll need to get up to speed with the PHP Data Objects (PDO) extension. We start with the basics, covering important topics such as how to write flexible SQL statements and avoid SQL injection attacks. We then delve into many lesser-known aspects, such as searching, working with transactions and stored procedures, and how to back up your database.

Chapter 3: Strings

This chapter explores the details of handling content on your site. We'll discuss string functions you can't live without, along with the process for validating and filtering user-submitted content.

Chapter 4: Dates and Times

Here, you'll learn how to use PHP's date functions, and implement an online calendar. You'll also obtain a solid grounding in the storage and manipulation of dates in MySQL.

Chapter 5: Forms, Tables, and Pretty URLs

The essentials of web page forms and tables are covered here. We'll discuss the development of forms with PEAR::HTML_QuickForm, and you'll see how to use PEAR::HTML_Table to implement data grids and paged result sets. We'll also take a look at some tricks you can use with Apache to generate search engine friendly URLs.

Chapter 6: Working with Files

This chapter is a survival guide to working with files in PHP. Here, we'll cover everything from gaining access to the local file system, to fetching files over a network using PHP's FTP client. We'll go on to learn how to create your own zipped archives with PEAR::Archive_Tar, and touch on the use of the Standard PHP Library.

Chapter 7: Email

In this chapter, we deal specifically with email-related solutions, showing you how to take full advantage of email with PHP. We'll learn to successfully send HTML emails and attachments with help from PEAR::Mail and PEAR::Mail_Mime, and to use PHP to easily handle incoming mails delivered to your web server.

Chapter 8: Images

This chapter explores the creation of thumbnails and explains how to watermark images on your site. We'll also discuss how you can prevent hotlinking from other sites, create an image gallery complete with Exif data, and produce a few professional charts and graphs—as well as CAPTCHA images—with JpGraph.

Chapter 9: Error Handling

Understand PHP's error reporting mechanism, how to take advantage of PHP's custom error handling features, and how to handle errors gracefully—with a focus on exception handling and custom exceptions—in this action-packed chapter.

Chapter 10: Access Control

Beginning with basic HTTP authentication, then moving on to application-level authentication, this chapter looks at the ways in which you can control access to your site. Later solutions look at implementing a user registration system, and creating a fine-grained access control system with users, groups, and permissions.

Chapter 11: Caching

This chapter takes the fundamental view that HTML is fastest, and shows you how you can take advantage of caching on both the client and server sides to reduce bandwidth usage and dramatically improve performance. It covers HTTP headers, output buffering, and using PEAR:Cache_Lite.

Chapter 12: XML and Web Services

With XML rapidly becoming a crucial part of almost all web-based applications, this chapter explores the rich XML capabilities of PHP 5. Here, you'll discover how easy it is to produce and consume web services based on RSS, XML-RPC, SOAP, and REST.

Chapter 13: Best Practices

The goal of this chapter is to examine some of the techniques that have proven themselves in helping development projects succeed. The discussion covers code versioning, how to write distributable code, how to add API documentation to your work, how to reduce bugs with unit testing, and how to deploy code safely.

Running the Code Examples

To run the code examples in this book you will need to ensure you have all the required software, libraries, and extensions. Some of the examples make use of additional packages that will need to be installed separately. Where solutions requiring additional packages are introduced you will find a link to the relevant web page; be sure to read the documentation, including the installation instructions.

The following packages are used in the examples in this book:

- PHP 5.21 (including the GD, EXIF, and XML-RPC extensions)
- PEAR: <http://pear.php.net/> (including Archive_Tar, Cache_Lite, HTML_Table, HTML_QuickForm, Mail, Net_FTP, Structures_DataGrid, and Validate)
- Zend Framework: <http://framework.zend.com/>
- JpGraph: <http://www.aditus.nu/jpgraph/>

To run all the examples you will also need a web server, database server, email server and FTP server, although instructions for their installation and configuration are out of scope for this book. If you want to setup a software environment for learning PHP you can't go past the XAMPP (<http://www.apachefriends.org/en/xampp.html>) server package for ease of installation and use. It is also available for a variety of operating systems.

The Windows version of XAMPP has all of the following components (and more) wrapped up in a single package with a convenient web interface for management:

- PHP 5 and PEAR
- Apache HTTP Server: <http://httpd.apache.org/>
- MySQL Database Server: <http://mysql.org/>
- Mercury Mail Transport System: <http://www.pmail.com/>
- Filezilla FTP server: <http://filezilla-project.org/>

Some examples in the book make specific use of the Apache HTTP Server and MySQL Database Server.

The Book's Web Site

Located at <http://www.sitepoint.com/books/phpant2/>, the web site that supports this book will give you access to the following facilities.

The Code Archive

As you progress through this book, you'll note file names above many of the code listings. These refer to files in the code archive, a downloadable ZIP file that contains all of the finished examples presented in this book. Simply click the **Code Archive** link on the book's web site to download it.

Updates and Errata

No book is error-free, and attentive readers will no doubt spot at least one or two mistakes in this one. The Corrections and Typos page on the book's web site³ will provide the latest information about known typographical and code errors, and will offer necessary updates for new releases of browsers and related standards.

The SitePoint Forums

If you'd like to communicate with other web developers about this book, you should join SitePoint's online community.⁴ The PHP forum,⁵ in particular, offers an abundance of information above and beyond the solutions in this book, and a lot of fun and experienced PHP developers hang out there. It's a good way to learn new tricks, get questions answered in a hurry, and just have a good time.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters including *The SitePoint Tribune*, *The SitePoint Tech Times*, and *The SitePoint Design View*. Reading them will keep you up to date on the latest news, product releases, trends, tips, and techniques for all aspects of web development. Sign up to one or more SitePoint newsletters at <http://www.sitepoint.com/newsletter/>.

Your Feedback

If you can't find an answer through the forums, or if you wish to contact us for any other reason, the best place to write is books@sitepoint.com. We have an email support system set up to track your inquiries, and friendly support staff members who can answer your questions. Suggestions for improvements as well as notices of any mistakes you may find are especially welcome.

Conventions Used in this Book

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

³ <http://www.sitepoint.com/books/phpant2/errata.php>

⁴ <http://www.sitepoint.com/forums/>

⁵ <http://www.sitepoint.com/forums/forumdisplay.php?f=34>

Code Samples

Code in this book will be displayed using a fixed-width font like so:

```
<h1>A perfect summer's day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code may be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css (*excerpt*)

```
border-top: 1px solid #333;
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➤ indicates a line break that exists for formatting purposes only, and should be ignored.

```
URL.open("http://www.sitepoint.com/blogs/2007/05/28/user-style-she
➤ets-come-of-age/");
```


Tips, Notes, and Warnings



Hey, You!

Tips will give you helpful little pointers.



Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.



Make Sure you Always ...

... pay attention to these important points.



Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Chapter 2

Using Databases with PDO

In the “old days” of the Internet, most web pages were nothing more than text files containing HTML. When people visited your site, your web server simply made the file available to their browsers. This approach started out fine, but as web sites grew, and issues such as design and navigation became more important, developers found that maintaining consistency across hundreds of HTML files was becoming a massive headache. To solve this problem, it became popular to separate variable content (articles, news items, and so on) from the static elements of the site—its design and layout.

If a database is used as a repository to store variable content, a server-side language such as PHP performs the task of fetching that data and placing it within a uniform layout template. This means that modifying the look and feel of a site can be handled as a separate task from the maintenance of content. And maintaining consistency across all the pages in a web site no longer consumes a developer’s every waking hour.

PHP supports all the relational databases worth mentioning, including those that are commonly used in large companies: Oracle, IBM’s DB2, and Microsoft’s SQL Server, to name a few. The three most noteworthy open source alternatives are

SQLite, PostgreSQL, and MySQL. PostgreSQL is arguably the best database of the three, in that it supports more of the features that are common to relational databases. SQLite is the perfect choice for smaller applications that still require database capability. MySQL is a popular choice among web hosts that provide support for PHP, and for this reason is typically easier to find than PostgreSQL.

This chapter covers all the common operations that PHP developers perform when working with databases: retrieving and modifying data, and searching and backing up the database. To achieve these tasks, we'll use the built-in PDO extension, rather than database-specific extensions. The examples we'll work with will use a single table, so no discussion is made of table relationships here. For a full discussion of that topic, see Kevin Yank's *Build Your Own Database Driven Website Using PHP & MySQL, 3rd Edition* (SitePoint, Melbourne, 2006)¹.

The examples included here work with the MySQL sample database called “world,” though all the interactions we'll work through can be undertaken with any database supported by PDO. The SQL file for the world database is available at <http://dev.mysql.com/doc/#sampledb> and the instructions explaining its use can be found at <http://dev.mysql.com/doc/world-setup/en/world-setup.html>.

What is PDO?

PDO, the PHP Data Objects extension, is a data-access abstraction layer. But what the heck is that? Basically, it's a consistent interface for multiple databases. No longer will you have to use the `mysql_*` functions, the `sqlite_*` functions, or the `pg_*` functions, or write wrappers for them to work with your database. Instead, you can simply use the PDO interface to work with all three functions using the same methods. And, if you change databases, you'll only have to change the **DSN** (or Data Source Name) of the PDO to make your code work.²

PDO uses specific database drivers to interact with various databases, so you can't use PDO by itself. You'll need to enable the drivers you'll use with PDO, so be sure

¹ <http://www.sitepoint.com/books/phpmysql1/>

² That's all you'll have to do so long as you write your SQL in a way that's not database specific. If you try to stick to the ANSI 92 standard [<http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt>], you should generally be okay—most databases support that syntax.

to research how to do it for your specific host operating system on the PDO manual page.³

PDO is shipped with PHP 5.1 and is available from PECL for PHP 5.0. Unfortunately, as PDO requires the new PHP 5 object oriented features, it's not available for PHP 4. In this book, all of our interactions with the database will use PDO to interact with the MySQL back end.

How do I access a database?

Before we can do anything with a database, we need to talk to it. And to talk to it, we must make a database connection. Logical, isn't it?

Solution

Here's how we connect to a MySQL database on the localhost:

mysqlConnect.php (excerpt)

```
<?php
$dsn = 'mysql:host=localhost;dbname=world;';
$user = 'user';
$password = 'secret';
try
{
    $dbh = new PDO($dsn, $user, $password);
}
catch (PDOException $e)
{
    echo 'Connection failed: ' . $e->getMessage();
}
?>
```

We'd use this code to connect to a SQLite database on the localhost:

³ <http://www.php.net/pdo/>

sqliteConnect.php (excerpt)

```
<?php
$dsn = 'sqlite2:"C:\sqlite\world.db"';
try
{
    $dbh = new PDO($dsn);
}
catch (PDOException $e)
{
    echo 'Connection failed: ' . $e->getMessage();
}
?>
```

And this code will let us connect to a PostgreSQL database on the localhost:

postgreConnect.php (excerpt)

```
<?php
$dsn = 'pgsql:host=localhost port=5432 dbname=world user=user ';
$dsn .= 'password=secret';
try
{
    $dbh = new PDO($dsn);
}
catch (PDOException $e)
{
    echo 'Connection failed: ' . $e->getMessage();
}
?>
```

Discussion

Notice that in all three examples above, we simply create a new PDO object. Only the connection data for the PDO constructor differs in each case: for the SQLite and PostgreSQL connections, we need just the DSN; the MySQL connection also requires username and password arguments in order to connect to the database.⁴

⁴ We could have put the username and password information in the MySQL DSN, providing a full DSN, but the average user has no cause to do this when using MySQL. It just adds unnecessary complexity to the DSN.

The DSN in Detail

As we saw above, DSN is an acronym for Data Source Name. The DSN provides the information we need in order to connect to a database. The DSN for PDO has three basic parts: the PDO driver name (such as *mysql*, *sqlite*, or *pgsql*), a colon, and the driver-specific syntax. The only aspect that may be a bit confusing here is the driver-specific syntax, as each driver requires different information. But have no fear—the trusty manual is here, of course!

The manual describes the database driver-specific syntax that’s required in the DSN for each of the PDO drivers. All you need to do is to go to the database driver page,⁵ select your database driver, and follow the link to the DSN information. For example, the MySQL DSN page in the manual is found at <http://www.php.net/manual/en/ref.pdo-mysql.connection.php>; it’s shown in Figure 2.1.

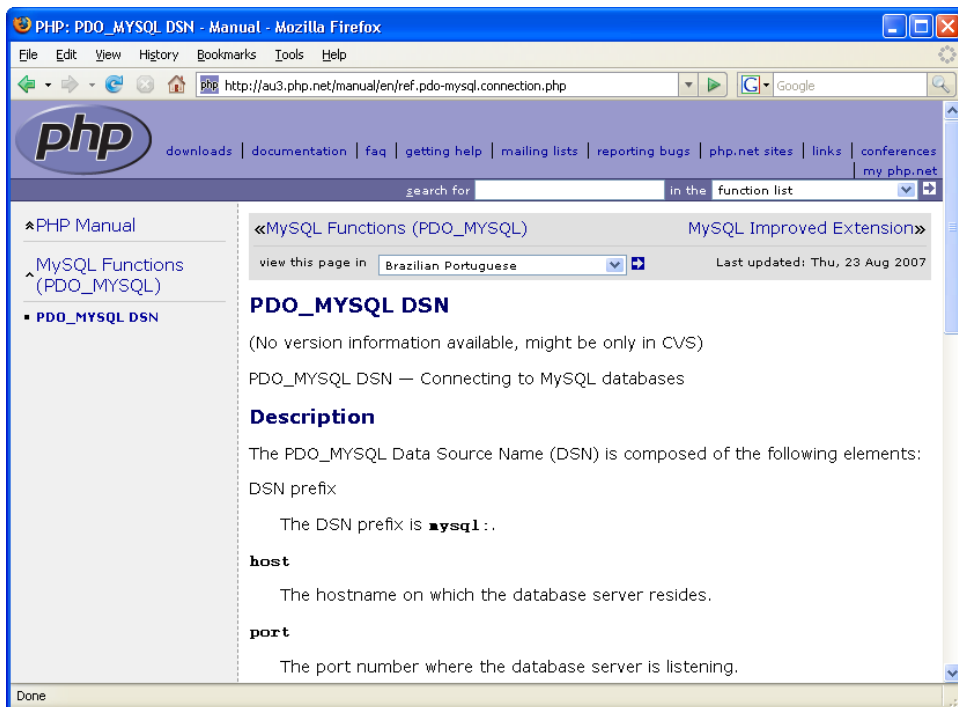


Figure 2.1. The PDO_MySQL DSN manual page

⁵ <http://www.php.net/manual/en/ref.pdo.php#pdo.drivers>

DSN examples are also provided on each manual page to get you started.



Do Not Pass Credentials in the DSN

In the database connection examples we just saw, I included my access credentials within the DSN, or in the `$user` and `$pass` variables, but I did so for illustration purposes *only*. This is not standard—or appropriate—practice, since this information can be misused by malicious parties to access your database.

Other Concepts

There are several concepts that you should understand when working with a database. First, you need to remember that the database server is a completely separate entity from PHP. While in these examples the database server and the web server are the same machine, this is not always the case. So, if your database is on a different machine from your PHP, you'll need to change the host name in the DSN to point to it.

To make things more interesting, database servers only listen for your connection on a specific port number. Each database server has a default port number (MySQL's is 3306, PostgreSQL's is 5432), but that may not be the port that the database administrator chose to set, or the one that PHP knows to look at. When in doubt, include your port number in the DSN.

You also need to be aware that a database server can have more than one database on it, so yours may not be the only one. This is why the database name is commonly included in the DSN—to help you get to *your* data, not some other person's!

Finally, make sure you understand what you'll receive from your PDO connection. Your connection will return a PDO object—not a reference to the database, or any data. It is through the PDO object that we interact with the database, bending it to our will.

How do I fetch data from a table?

Here we are, connected to the database. Woo hoo! But what good is that if we can't get anything out of the database?

Solutions

PDO provides a couple of ways for us to interact with the database. Here, we'll explore both possible solutions.

Using the Query Method

First, let's look at the faster, but not necessarily better, way—using the query method:

`pdoQuery.php` (excerpt)

```
$country = 'USA';
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'Select * from city where CountryCode = ' .
        $dbh->quote($country);
    foreach ($dbh->query($sql) as $row)
    {
        print $row['Name'] . "\t";
        print $row['CountryCode'] . "\t";
        print $row['Population'] . "\n";
    }
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}
```

An excerpt of this code's output can be seen in Figure 2.2.

New York	USA	8008278
Los Angeles	USA	3694820
Chicago	USA	2896016
Houston	USA	1953631
Philadelphia	USA	1517550
Phoenix	USA	1321045
San Diego	USA	1223400
Dallas	USA	1188580
San Antonio	USA	1144646
Detroit	USA	951270
San Jose	USA	894943
Indianapolis	USA	791926
San Francisco	USA	776733
Jacksonville	USA	735167
Columbus	USA	711470
Austin	USA	656562
Baltimore	USA	651154
Memphis	USA	650100
Milwaukee	USA	596974
Boston	USA	589141
Washington	USA	572059
Nashville-Davidson	USA	569891
El Paso	USA	563662
Seattle	USA	563374
Denver	USA	554636

Figure 2.2. Output produced using the PDO query method

Using the Prepare and Execute Methods

Using the prepare and execute methods is generally considered the better way to handle a query to the database. First, we call `PDO->prepare` with our SQL statement as an argument. In return, we receive a `PDOStatement` object, on which we call the `execute` method. Then, within a while loop, we repeatedly call the `PDOStatement->fetch` method to retrieve the data we've selected from our database:

`pdoPrepEx.php` (excerpt)

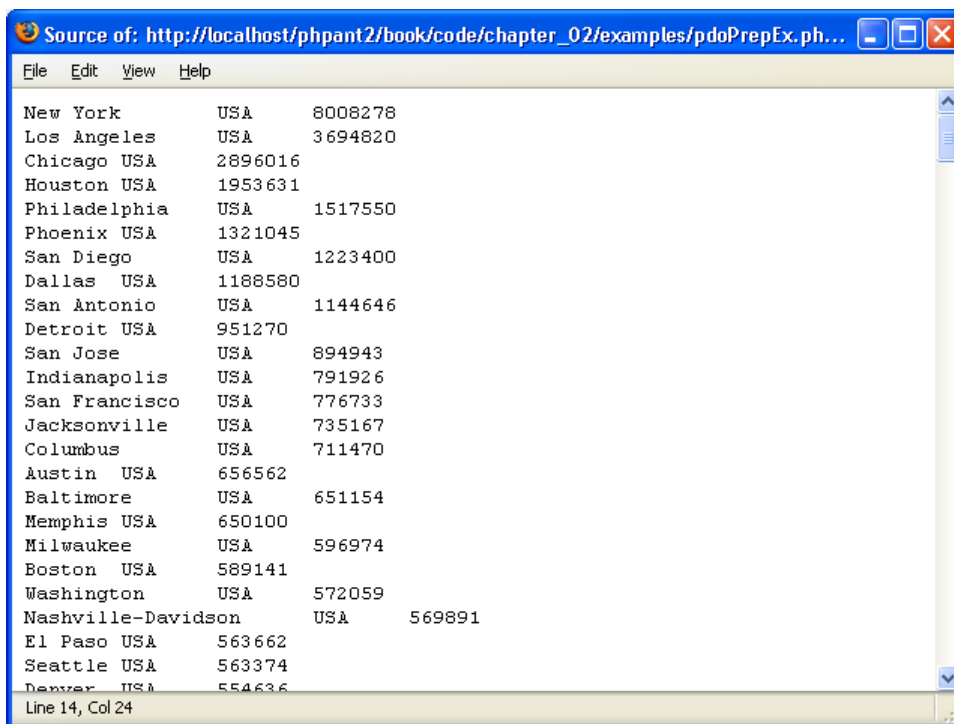
```
$country = 'USA';
try
{
    $dbh = new PDO($dsn, $user, $password);
    $sql = 'Select * from city where CountryCode =:country';
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $stmt = $dbh->prepare($sql);
    $stmt->bindParam(':country', $country, PDO::PARAM_STR);
```

```

$stmt->execute();
while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    print $row['Name'] . "\t";
    print $row['CountryCode'] . "\t";
    print $row['Population'] . "\n";
}
}
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}
}

```

An excerpt of the output of this code can be seen in Figure 2.3.



```

New York      USA      8008278
Los Angeles   USA      3694820
Chicago USA    2896016
Houston USA    1953631
Philadelphia  USA      1517550
Phoenix USA    1321045
San Diego     USA      1223400
Dallas USA    1188580
San Antonio   USA      1144646
Detroit USA    951270
San Jose      USA      894943
Indianapolis  USA      791926
San Francisco USA      776733
Jacksonville  USA      735167
Columbus      USA      711470
Austin USA    656562
Baltimore     USA      651154
Memphis USA    650100
Milwaukee     USA      596974
Boston USA    589141
Washington    USA      572059
Nashville-Davidson USA      569891
El Paso USA    563662
Seattle USA    563374
Denver USA    554636

```

Figure 2.3. Output using the PDO prepare and execute methods

Discussion

You'll have noticed that both these solutions give you the same data, which is as it should be. But there are very specific reasons for choosing one solution over the other.

`PDO->query` is great when you're only executing a query once. While it doesn't automatically escape any data you send it, it does have the very handy ability to iterate over the result set of a successful `SELECT` statement. However, you should take care when using this method. If you don't fetch all the data in the result set, your next call to `PDO->query` might fail.⁶ If you're going to use the SQL statement more than once, your best bet is to use `prepare` and `execute`—the preferred solution. Using `prepare` and `execute` has a couple of advantages over `query`. First, it will help to prevent SQL injection attacks by automatically escaping any argument you give it (this approach is often considered the better practice for this reason alone). Granted, if you build any other part of your query from user input, that will negate this advantage, but you wouldn't ever do that, would you? Second, prepared statements that are used multiple times (for example, to perform multiple inserts or updates to a database) use fewer resources and will run faster than repeated calls to the `query` method.

There are a couple of other ways we can use `prepare` and `execute` on a query, but I feel that the example we discussed here will be the clearest. I used named parameters in this solution, but be aware that PDO also supports question mark (?) parameter markers. In the example we saw here, you could have chosen not to use the `paramBind` method—instead, you could have given the parameters to the `execute` command. See The PHP Manual if you have any questions about the alternative syntaxes.

Using Fetch Choices

When you use `prepare` and `execute`, you have the choice of a number of formats in which data can be returned. The example we saw used the `PDO::FETCH_ASSOC`

⁶ For further information, see The PHP Manual page at <http://www.php.net/manual/en/function.pdo-query.php>.

option with the `fetch` method, because it returns data in a format that will be very familiar for PHP4 users: an associative array.⁷

If you'd rather use only object-oriented code in your application, you could instead employ the `fetchObject` method, which, as the name implies, returns the result set as an object. Here's how the `while` loop will look when the `fetchObject` method is used:

`pdoPrepEx2.php` (excerpt)

```
while ($row = $stmt->fetchObject())
{
    print $row->Name . "\t";
    print $row->CountryCode . "\t";
    print $row->Population . "\n";
}
```

How do I resolve errors in my SQL queries?

Errors are inevitable. They assail all of us and can, at times, be caused by circumstances outside our control—database crashes, database upgrades, downtime for maintenance, and so on. If something goes wrong when you're trying to deal with PHP and SQL together, it's often difficult to find the cause. The trick is to get PHP to tell you where the problem is, bearing in mind that you must be able to hide this information from visitors when the site goes live.



We're Only Looking for Errors—Not Fixing Them!

I won't be explaining error handling in depth here—instead, I'll show you how to find errors. See Chapter 9 for more information on what to do when you've found an error and want to fix it.

Solutions

PDO provides multiple solutions for catching errors. We'll go over all three options in the following examples, where we'll introduce a typo into the `world` database

⁷ For a full listing of the ways in which you can have data returned, see the `fetch` page of the manual at <http://www.php.net/manual/en/function.pdostatement-fetch.php>.

table name, so that it reads *cities* instead of *city*. If you run this code yourself, you can also try commenting out the error-handling code to see what may be displayed to site visitors.

Using Silent Mode

`PDO::ERRMODE_SILENT` is the default mode:

`pdoError1.php (excerpt)`

```
$country = 'USA';
$dbh = new PDO($dsn, $user, $password);
$sql = 'Select * from cities where CountryCode =:country';
$stmt = $dbh->prepare($sql);
$stmt->bindParam(':country', $country, PDO::PARAM_STR);
$stmt->execute();
$code = $stmt->errorCode();
if (empty($code))
{
    : proceed to fetch data
}
else
{
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo '<pre>';
    var_dump($stmt->errorInfo());
    echo '</pre>';
}
```

The default error mode sets the `errorCode` property of the `PDOStatement` object, but does nothing else. As you can see in this example, you need to check the error code manually to ascertain whether or not an error was found—otherwise your script will happily continue on its merry way.

Using Warning Mode

`PDO::ERRMODE_WARNING` generates a PHP warning as well as setting the `errorCode` property:

pdoError2.php (excerpt)

```

$country = 'USA';
$dbh = new PDO($dsn, $user, $password);
$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
$sql = 'Select * from cities where CountryCode =:country';
$stmt = $dbh->prepare($sql);
$stmt->bindParam(':country', $country, PDO::PARAM_STR);
$stmt->execute();
$code = $stmt->errorCode();
if (empty($code))
{
    : proceed to fetch data
}
else
{
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo '<pre>';
    var_dump($stmt->errorInfo());
    echo '</pre>';
}

```

Again, the program will continue on its merry way unless you specifically check for the error code. So, unless you have the Display Errors functionality turned on, use a custom error handler, or check your error logs, you may not notice it.

Using Exception Mode

`PDO::ERRMODE_EXCEPTION` creates a `PDOException` as well as setting the `errorCode` property:

pdoError3.php (excerpt)

```

$country = 'USA';
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'Select * from cities where CountryCode =:country';
    $stmt = $dbh->prepare($sql);
    $stmt->bindParam(':country', $country, PDO::PARAM_STR);
    $stmt->execute();
}

```

```

    : proceed to fetch data
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo '<pre>';
    echo 'Error: ' . $e->getMessage() . '<br />';
    echo 'Code: ' . $e->getCode() . '<br />';
    echo 'File: ' . $e->getFile() . '<br />';
    echo 'Line: ' . $e->getLine() . '<br />';
    echo 'Trace: ' . $e->getTraceAsString();
    echo '</pre>';
}

```

`PDO::ERRMODE_EXCEPTION` allows you to wrap your code in a `try {...} catch {...}` block. An uncaught exception will halt the script and display a stack trace to let you know there's a problem.

The `PDOException` is an extension of the general PHP `Exception` class found in the Standard PHP Library (or **SPL**).⁸

Discussion

Most people will choose to take advantage of PHP's more powerful object oriented model, and use the Exception mode to handle errors, since it follows the object oriented style of error handling—catching and handling different types of exceptions—and is easier to work with.

Regardless of the way you choose to handle your errors, it's a good idea to return the text of the SQL query itself. This allows you to see exactly which query is problematic and will assist you in the error's debugging.

⁸ You can learn more about the SPL and PHP's base `Exception` class in the manual, at <http://www.php.net/spl/> and <http://www.php.net/manual/en/language.exceptions.php>.

How do I add data to, or modify data in, my database?

Being able to fetch data from the database is a start, but how can you put it there in the first place?

Solution

We add data to the database with the SQL `INSERT` command, and modify data that's already in the database with the SQL `UPDATE` command. Both commands can be sent to the database using either the query method or the prepare and execute methods. I'll be using the prepare and execute methods in this solution.

INSERT Data into the Database

First up, let's look at a simple `INSERT`, using the `City` table from the `world` database:

`insert.php` (excerpt)

```
$id = '4080';
$name = 'Guam';
$country = 'GU';
$district = 'Guam';
$population = 171018;
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'INSERT INTO city
        (ID, Name, CountryCode, District, Population)
        VALUES (:id, :name, :country, :district, :pop)';
    $stmt = $dbh->prepare($sql);
    $stmt->bindParam(':id', $id);
    $stmt->bindParam(':name', $name);
    $stmt->bindParam(':country', $country);
    $stmt->bindParam(':district', $district);
    $stmt->bindParam(':pop', $population);
    $stmt->execute();
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
```

```

    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}
?>

```

UPDATE Data in the Database

And here's a simple UPDATE, using the City table from the world database:

update.php (excerpt)

```

$id = '4080';
$name = 'Guam';
$country = 'GU';
$district = 'Guam';
$population = 171019; // data provided by the U.S. Census
                    // Bureau, International Data Base
                    // Mid year 2006

try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'UPDATE city SET Name = :name,
        CountryCode = :country, District = :district,
        Population = :pop WHERE ID = :id';
    $stmt = $dbh->prepare($sql);
    $stmt->bindParam(':id', $id);
    $stmt->bindParam(':name', $name);
    $stmt->bindParam(':country', $country);
    $stmt->bindParam(':district', $district);
    $stmt->bindParam(':pop', $population);
    $stmt->execute();
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}
?>

```

Discussion

Note that other than changing the SQL statement used in the `prepare` method, the code in both examples above is exactly the same. We do like to keep things easy in PHP!

In a practical application, some, if not all of the inputs to the query will be garnered from user-generated content. Because we're using the `prepare` and `execute` methods, we don't have to worry about an SQL injection attack on this query: all the variables will be escaped automatically.



Be Cautious with UPDATE and DELETE

Be very careful when you use `UPDATE` or `DELETE` in your SQL. If you don't have a `WHERE` clause in your SQL statement, you will end up updating or deleting all the rows in the table. Needless to say, either outcome could cause serious problems!

How do I protect my web site from an SQL injection attack?

An SQL injection attack occurs when an attacker exploits a legitimate user input mechanism on your site to send SQL code that your unsuspecting script passes on to the database for execution. The golden rule for avoiding SQL injection attacks is: escape all data from external sources before letting it near your database. That rule doesn't just apply to `INSERT` and `UPDATE` queries, but also to `SELECT` queries.

As we discussed earlier, using prepared statements for all your queries within a script almost eliminates the problem of SQL injection attacks, but if you choose to use the `query` method, you'll have no such protection—you'll have to manually escape any user input that goes into the query. Let's look at an example:

`sqlInject.php` (excerpt)

```

//$city = 'New York';
$city = "' or Name LIKE '%" ;
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,

```

```

        PDO::ERRMODE_EXCEPTION);
    $sql = "Select * from city where Name = '". $city ."'";
    foreach ($dbh->query($sql) as $row)
    {
        print $row['Name'] . "\t";
        print $row['CountryCode'] . "\t";
        print $row['Population'] . "\n";
    }
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}

```

In this example, we'll pretend that the `$city` variable used in the SQL statement comes from a form submitted by the user. A typical user would submit something like **New York**. This would give us the following SQL statement:

```
Select * from city where Name = 'New York'
```

This would cause no problems within the script. A savvy attacker, however, may enter `' OR Name LIKE '%'`, which would give us the following SQL statement:

```
Select * from city where Name = '' OR Name LIKE '%'
```

This input opens the entire table to the attacker. “No big deal,” you say. “It’s only a list of cities.” Yes, but what if instead of our simple city table, this was the authorized users table? The attacker would have access to extremely sensitive data!

Solution

Luckily, this issue is fairly easy to avoid, though the solution will mean more work for you. You can use PDO’s handy `quote` method to escape any data that you’re passing to the SQL string. Simply change the SQL code to this:

```
$sql = "Select * from city where Name = '". $dbh->quote($city) ."'";
```

Remember that you'll need to quote *each individual* piece of data you use in the SQL query—there aren't any shortcuts! That is, unless you consider prepare and execute a shortcut.

Discussion

If you're using the PDO->query method, always quote your input. *Always!*

If you choose to use the prepare and execute approach, you won't have to quote the values that you bind to the prepared SQL (for example, the values to be inserted)—that's all done for you by the driver. However, there may be times when you won't be able to bind a variable to the prepared SQL. In such cases, you'll need to quote any values you use that cannot be bound (for example, a GROUP BY or ORDER BY clause, or the table name) if you're building a dynamic SQL statement.

Remember: a strong defense is a good offense.

How do I create flexible SQL statements?

SQL is a powerful language for manipulating data. With PHP, we can construct SQL statements out of variables—an approach that can be useful for sorting a table by a single column, or displaying a large result set across multiple pages.

Solution

Until the SQL is prepared and executed, it's still just a string that you can manipulate as you'd expect. This solution uses concatenation based on user input to select cities from the specified country and display them in a specified order:

flexSQLConcat.php (excerpt)

```
$validCountries = array ('USA', 'CAN', 'GU', 'ISR');
if (isset($_GET['country']) &&
    in_array($_GET['country'], $validCountries))
{
    $country = $_GET['country'];
}
else
{
    $country = 'USA';
}
```

```

$order = (!isset($_GET['order'])) ? FALSE : $_GET['order'];
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'SELECT * FROM city WHERE CountryCode = :country';
    switch ($order) {
        case 'district':
            // Add to the $sql string
            $sql .= " ORDER BY District";
            break;
        case 'pop':
            $sql .= " ORDER BY Population DESC";
            break;
        default:
            // Default sort by title
            $sql .= " ORDER BY Name";
            break;
    }
    $stmt = $dbh->prepare($sql);
    $stmt->bindParam(':country', $country);
    $stmt->execute();
    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
        print $row['Name'] . "\t";
        print $row['CountryCode'] . "\t";
        print $row['Population'] . "\n";
    }
}
catch (Exception $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}

```

In this code, the user input is read either from a web form that has GET as its method, or a URL with a query string. In the switch statement above, we're generating dynamic SQL using concatenation. The \$order value is read, and an ORDER BY clause is added to the SQL query.

Discussion

An alternative solution involves using `sprintf` to build your dynamic SQL. This approach is similar to binding variables to the prepared SQL:

`flexSQLsprintf.php` (excerpt)

```
switch ($order) {
    case 'district':
        $orderby = " District";
        break;
    case 'pop':
        $orderby = " Population DESC";
        break;
    default:
        $orderby = " Name";
        break;
}
$format = 'SELECT * FROM city
        WHERE CountryCode = :country ORDER BY %s';
$sql = sprintf($format, $orderby);
```

It's a matter of personal style, but either of these approaches can be extended to columns, table names, `WHERE` clauses, `LIMIT` clauses, and anything else you wish to include in your SQL query.

Remember that until the point at which the SQL is prepared and executed, it's just a string that you can manipulate as much as you require.

How do I find out how many rows I've touched?

Often, it's useful to be able to count the number of rows returned or affected by a query before you do anything with them. This capability is particularly handy when you're splitting results across pages, or producing statistical information.

Solutions

The two solutions that follow will enable you to count the number of rows returned, and the number of rows affected, by your operations within the database.

Counting the Rows Returned

PDO doesn't have a magic method that counts the number of rows returned from a `SELECT` call. You can use the `PDOStatement->rowCount` method to return the number of rows returned by a `SELECT` statement with some PDO database drivers. However, as the behavior of this function isn't guaranteed to be consistent with every database driver, I won't cover it here. Feel free to try it yourself with your database driver, but keep in mind that if you need to write portable code, this approach is not reliable. There is, however, a solution that works around this lack of a useful method—it uses the SQL aggregate function `COUNT`.

Here's the code that will count the number of rows returned:

`count.php` (excerpt)

```
$country = 'USA';
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'SELECT COUNT(*) FROM city
        WHERE CountryCode =:country';
    $stmt = $dbh->prepare($sql);
    $stmt->bindParam(':country', $country, PDO::PARAM_STR);
    $result = $stmt->execute();
    echo 'There are ', $stmt->fetchColumn(), ' rows returned.';
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}
```

Discussion

`COUNT` returns the number of rows from a query, or a part of a query, and is commonly used with the `DISTINCT` keyword. SQL's aggregate function `COUNT` is widely supported by the various database systems. For more information on how your database handles `COUNT`, see your database's documentation.

Counting the Rows Affected

We can use the `PDOStatement->rowCount` method to find out how many rows were affected by an `UPDATE`, `INSERT` or `DELETE` query. The use of `rowCount` is not common in typical PHP applications, but it can be a good way to inform users that “Number of records deleted from the Customers table: *n*.”

Here’s the code you’ll need:

affect.php (excerpt)

```
$country = 'AFG';
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'DELETE FROM city WHERE CountryCode = :country';
    $stmt = $dbh->prepare($sql);
    $stmt->bindParam(':country', $country, PDO::PARAM_STR);
    $result = $stmt->execute();
    echo 'Number of records deleted from the city table: ';
    echo $stmt->rowCount();
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}
```

After you call `PDOStatement->execute`, you can call the `PDOStatement->rowCount` method to return the number of rows affected.



Make Sure you Add a WHERE Clause

When you’re using the SQL commands `UPDATE` and `DELETE`, always make sure you add a `WHERE` clause. Without it, you will either be updating an entire column in the database, or deleting all the data in the table, neither of which is what you likely meant to do!

How do I find out a new INSERT's row number in an autoincrementing field?

When you're dealing with autoincrementing columns in database tables, you'll often need to find out the ID of a row you've just inserted, so that you can update other tables with this information. After all, that's how relationships between tables are maintained.

Solution

To accomplish this task, PDO provides the `lastInsertId` method, which returns the ID generated by the last `INSERT` operation if this capability is supported by the driver being used.⁹ Here's how it works:

`lastId.php` (excerpt)

```
$name = 'Dededo';
$country = 'GU';
$district = 'Guam';
$population = 42980; // according to the 2000 US census
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'INSERT INTO city
        (Name, CountryCode, District, Population)
        VALUES (:name, :country, :district, :pop)';
    $stmt = $dbh->prepare($sql);
    $stmt->bindParam(':name', $name);
    $stmt->bindParam(':country', $country);
    $stmt->bindParam(':district', $district);
    $stmt->bindParam(':pop', $population);
    $stmt->execute();
    echo 'ID of last insert: ', $dbh->lastInsertId();
}
catch (PDOException $e)
{
```

⁹ `lastInsertId` may not behave consistently when it's used with different database drivers—some database drivers do not support autoincrementing fields. Read the manual page at <http://www.php.net/manual/en/function.pdo-lastinsertid.php> for more information.

```

echo 'PDO Exception Caught. ';
echo 'Error with the database: <br />';
echo 'SQL Query: ', $sql;
echo 'Error: ' . $e->getMessage();
}

```

Discussion

When you're using the `lastInsertId` method, be sure to use the PDO object (`$dbh` above), not the `PDOStatement` object (that's the object you create when you use `prepare`—`$stmt` above). If you don't, an error will result.

How do I search my table?

Some people are just impatient; rather than exploring your site with the friendly navigation system you've provided, they demand relevant information now! And obliging PHP developers like you and I happily implement search functionality to provide visitors with a shortcut to the information they want.

In the bad old days when all content was stored in the form of HTML files, developing usable search functionality could be quite a problem, but now that we use databases to store content, performing searches becomes much easier.

Solution

The most basic form of search occurs against a single column, with the database `LIKE` operator:

like.php (excerpt)

```

$country = 'A';
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'SELECT * FROM city
        WHERE CountryCode LIKE :country';
    $stmt = $dbh->prepare($sql);
    $country = $country.'%';
    $stmt->bindParam(':country', $country, PDO::PARAM_STR);
}

```

```

$stmt->execute();
while ($row = $stmt->fetchObject()) {
    print $row->Name . "\t";
    print $row->CountryCode . "\t";
    print $row->Population . "\n";
}
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}

```

Discussion

The LIKE search is supported by almost all database systems,¹⁰ and is usually used in conjunction with wildcard characters. The % character I used in the example above matches any number of characters—even zero characters. The wildcard character used in the example allows my query to find any city in a country that starts with the letter A.

The other wildcard character that's typically available is _, which will match any single character. So if, in the example above, I wanted to find only cities in countries that started with A and ended with G, I'd need to change just one line of code:

```

/* $country = $country.'%';    <- remove this */
$country = $country.'_G';    // <- add this

```

If you need a more complicated search method, check your database documentation to see what's available. For example, MySQL has FULLTEXT search capabilities, as explained on the MySQL manual site.¹¹

¹⁰ You should verify the availability of the LIKE keyword, and the wildcard characters you want to use with it, in your database system documentation.

¹¹ <http://dev.mysql.com/doc/refman/5.0/en/fulltext-search.html>

How do I work with transactions?

Let's imagine we're trying to complete a transaction at our local bank—we need to move some money from our savings account to our checking account (to pay for that vacation, of course). Now, if a problem arises in the middle of the transaction (after you withdraw the money from the savings account, but before you deposit it into the checking account), the money will disappear, and you can forget that vacation. Or does it?

If you need to run a group of SQL queries as one operation in order to maintain the integrity of your data, then you need **transactions**. Almost all databases provide transaction support in one form or another, and knowing how to use transactions with PDO can help you secure that well-deserved vacation.

Solution

We start the hypothetical transaction with the PDO->beginTransaction method, and if all goes well, end it with PDO->commit. If a problem occurs, we use the PDO->rollback method to undo everything that's taken place in the transaction:

transaction.php (excerpt)

```
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $dbh->beginTransaction();
    $sql = 'INSERT INTO transactions
        (acctNo, type, value, adjustment)
        VALUES (:acctNo, :type, :value, :adjust)';
    $stmt = $dbh->prepare($sql);
    $stmt->execute(array(':acctNo'=>$acctFrom, ':type'=>$withdrawal,
        ':value'=>$amount, ':adjust'=>'-'));
    $sql = 'INSERT INTO transactions
        (acctNo, type, value, adjustment)
        VALUES (:acctNo, :type, :value, :adjust)';
    $stmt = $dbh->prepare($sql);
    $stmt->execute(array(':acctNo'=>$acctTo,
        ':type'=>$deposit,
        ':value'=>$amount,
        ':adjust'=>'+'));
}
```

```
    $dbh->commit();
}
catch (Exception $e)
{
    $dbh->rollBack();
    : further error handling here
}
```

Discussion

Before we get into the deeper nuances of PDO's transaction handling capabilities, let's look at the official definition of **database transactions** from the PDO manual page¹²:

“If you've never encountered transactions before, they offer 4 major features: *Atomicity, Consistency, Isolation and Durability (ACID)*.¹³ In layman's terms, any work carried out in a transaction, even if it is carried out in stages, is guaranteed to be applied to the database safely, and without interference from other connections, when it is committed. Transactional work can also be automatically undone at your request (provided you haven't already committed it), which makes error handling in your scripts easier.”

“Transactions are typically implemented by “saving-up” your batch of changes to be applied all at once; this has the nice side effect of drastically improving the efficiency of those updates. In other words, transactions can make your scripts faster and potentially more robust (you still need to use them correctly to reap that benefit).”

Unfortunately, not all database systems support transactions. So, by default, PDO will run in auto-commit mode, where each query is treated as its own transaction. If the database does not support transactions, the query is issued without one.

If your database supports transactions, rather than using the auto-commit feature, you can start and stop transactions manually. In the example above, the `PDO->beginTransaction` and `PDO->commit` methods are called in the try block. The

¹² <http://www.php.net/pdo>

¹³ Emphasis added by the author.

PDO->rollback is used in the catch block to roll the database back in case of a problem.

How do I use stored procedures with PDO?

Many databases support **stored procedures**—scripts that are run on your database typically in a database-specific SQL language.¹⁴ Stored procedures allow the manipulation of the data close to the location where the data is held, reducing bandwidth. They maintain the separation of the data from the script logic, and allow multiple systems in potentially different languages to access the data in a uniform manner (saving you valuable coding and debugging time). Finally, stored procedures increase query speeds using predetermined execution plans, and can prevent any direct interaction with the data, thereby protecting it.

Solution

Using PDO to work with stored procedures is fairly easy. In the example below, you'll see the simple stored procedure we'll be interacting with in our code.¹⁵ It does nothing more than generate the quote, "Out, damned spot!" from Shakespeare's *Macbeth*:

getQuote.sql (excerpt)

```
DROP PROCEDURE IF EXISTS getQuote;

DELIMITER //
CREATE PROCEDURE getQuote()
BEGIN
  DECLARE outStr VARCHAR(45);
  SET outStr = "Out, damned spot!";
  SELECT outStr;
END//

DELIMITER ;
```

Here's the code that uses the stored procedure:

¹⁴ Such languages include PL/SQL (Oracle), T-SQL (SQL Server), PL/pgSQL (PostgreSQL), and SQL::2003 (IBM DB2 and MySQL).

¹⁵ This procedure is written in SQL::2003 syntax for MySQL.

```

try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $sql = 'CALL getQuote()';
    $stmt = $dbh->prepare($sql);
    $stmt->execute();
    $return_string = $stmt->fetch();
}
catch (PDOException $e)
{
    echo 'PDO Exception Caught. ';
    echo 'Error with the database: <br />';
    echo 'SQL Query: ', $sql;
    echo 'Error: ' . $e->getMessage();
}

echo 'Called stored procedure. It returned: ', $return_string[0];

```

The example script will produce this output:

```
Called stored procedure. It returned: Out, Damned Spot!
```

Discussion

Each database's stored procedure language is different, so be sure to check your system's documentation to identify the specific syntax you'll need to create a stored procedure. To learn more about MySQL's stored procedures, check out the relevant manual pages.¹⁶

In the example above, which was made for MySQL, you'll notice that the stored procedure includes the `DECLARE`, `SET`, and `SELECT` statements. Generally speaking, these are needed in any MySQL stored procedure to retrieve data. Nothing special is needed to retrieve the data from the stored procedure—we just use the `PDOStatement->fetch` method to grab the value returned from the final `SELECT`

¹⁶ <http://dev.mysql.com/doc/refman/5.0/en/stored-procedures.html>

statement in the stored procedure. (MySQL normally uses an *OUT* parameter for the stored procedure, but this is not necessary with PDO.)

How do I back up my database?

The bigger a database becomes, the more nerve-wracking it can be not to have a backup of the data it contains. It's truly the stuff of nightmares: what happens if your server crashes and everything is lost?

It's common for database software to have some kind of built-in backup utility for just this reason. In this solution, we'll work through an example that uses the `mysqldump` utility for the MySQL database system.

Solution

You can export the contents of a database from the command line using `mysqldump`:

```
mysqldump -uuser -psecret world > world.sql
```

This command will log in to MySQL as user “user” (**-uuser**) with the password “secret” (**-psecret**) and output the contents of the `world` database to a file called **world.sql**. The contents of **world.sql** will be a series of queries that can be run against MySQL. Using the `mysql` utility, we can perform the reverse operation from the command line:

```
mysql -uuser -psecret world < world.sql
```

You can use PHP's `system` function to execute this command from within a PHP script (though you'll need to be logged in and able to execute PHP scripts from the command line). The following example wraps the `mysqldump` command line utility in a handy PHP class that you can use to keep regular backups of your site:

[MySQLDump.class.php \(excerpt\)](#)

```
<?php
class MySQLDump
{
    private $cmd;
    public function __construct($dbUser, $dbPass, $dbName, $dest,
        $zip = 'gz')
```

```

{
    $zip_util = array('gz'=>'gzip','bz2'=>'bzip2');
    if (array_key_exists($zip, $zip_util))
    {
        $fname = $dbName . '.' . date("w") . '.sql.' . $zip;
        $this->cmd = 'mysqldump -u' . $dbUser . ' -p' . $dbPass .
            ' ' . $dbName . '|' . $zip_util[$zip] . ' >' .
                $dest . '/' . $fname;
    }
    else
    {
        $fname = $dbName . '.' . date("w") . '.sql';
        $this->cmd = 'mysqldump -u' . $dbUser . ' -p' . $dbPass .
            ' ' . $dbName . ' >' . $dest . '/' . $fname;
    }
}
public function backup()
{
    system($this->cmd, $error);
    if ($error)
    {
        trigger_error('Backup failed: ' . $error);
    }
}
}
?>

```



MySQLDump Assumptions

The `MySQLDump` class makes some assumptions about your operating system configuration. It assumes that the `mysqldump` utility is available in the path of the user that executes this script. If the `gzip` or `bzip2` utilities are used, they're also expected to be present in the user's path. If you have a choice, use `bzip2`, as it provides better compression than `gzip`, and helps to save disk space.

The following code shows how this class can be used:

`backup.php` (excerpt)

```

<?php
require_once 'MySQLDump.class.php';
$dbUser = 'user';

```

```
$dbPass = 'secret';  
$dbName = 'world';  
$dest   = '/home/user/backups';  
$zip    = 'bz2';  
$mysqldump = new MySQLDump($dbUser, $dbPass, $dbName, $dest, $zip);  
$mysqldump->backup();  
?>
```

This code will create a backup of the `world` database in the `/home/user/backups` directory. If you test this example, make sure to change the variables to suit your setup.

Discussion

The `$dest` variable specifies the path to the directory in which the backup file should be placed. The filename that's created will be in this format:

```
databaseName.dayOfWeek.sql.zipExtension
```

Here's an example:

```
world.1.sql.bz2
```

A number from 0 to 6 that represents the day of the week (0 being Sunday and 6 being Saturday) is inserted into the `dayOfWeek` element. This filename convention can provide a weekly rolling backup, with the files for the current week overwriting those from the previous week. Such an approach should provide adequate backups; it gives you a week to discover any serious problems, and doesn't require excessive disk space for file storage.

The use of a ZIP utility is optional. The default value of the `$zip` parameter is `gz`, which indicates the `gzip` utility should be used. The other option is `bz2`, which indicates the `bzip2` utility should be used. If neither of these values is used, no compression will be made; however, for large databases it's obviously a good idea to use a compression tool to minimize the amount of disk space required.

This class is intended for use with the `crontab` utility, which is a Unix feature that allows you to execute scripts on a regular (for example, daily) basis.

Catering to Platform Differences

You may have noticed that the above MySQLDump class will only work on a *nix server. What if your database server uses a Windows box? I offer the following solution to circumvent this problem. First we define an abstract MySQLDump class, then we extend it to create a class for each platform, and finally we create a factory method to instantiate the correct MySQLDump object needed. Here's our abstract MySQLDump class:

AbstractMySQLDump.class.php (excerpt)

```
require_once 'MySQLDump_ms.class.php';
require_once 'MySQLDump_nix.class.php';

abstract class MySQLDump
{
    public static function factory($dbUser, $dbPass, $dbName, $dest,
        $zip)
    {
        if (strtoupper(substr(PHP_OS, 0, 3)) === 'WIN')
        {
            return new MySQLDump_ms($dbUser, $dbPass, $dbName, $dest,
                $zip);
        }
        else
        {
            return new MySQLDump_nix($dbUser, $dbPass, $dbName, $dest,
                $zip);
        }
    }

    abstract public function __construct($dbUser, $dbPass, $dbName,
        $dest, $zip = 'gz');

    public function backup()
    {
        system($this->cmd, $error);
        if ($error)
        {
            throw new MySQLDumpException(
                'Backup failed: Command = ' . $this->cmd .
                ' Error = ' . $error);
        }
    }
}
```

```

}

class MySQLDumpException extends Exception {}

```

The backup method represents our backup API. Child classes need to implement a custom constructor that sets the `cmd` property. Overriding the backup method is optional. The static method `factory` will instantiate a `MySQLDump` object instance based on the `PHP_OS` constant—representing the host platform. We’ve also added a custom exception class, `MySQLDumpException`, for error handling.

The `*nix` version of our backup class will contain an implementation similar to the solution class above, but we’ll need to change the class definition so that it extends the abstract `MySQLDump` class:

MySQLDump_nix.class.php (excerpt)

```

require_once 'AbstractMySQLDump.class.php';
class MySQLDump_nix extends MySQLDump
{
    protected $cmd;

    public function __construct($dbUser, $dbPass, $dbName, $dest,
        $zip = 'gz')
    {
        $zip_util = array('gz'=>'gzip', 'bz2'=>'bzip2');
        if (array_key_exists($zip, $zip_util))
        {
            $fname = $dbName . '.' . date("w") . '.sql.' . $zip;
            $this->cmd = 'mysqldump -u' . $dbUser . ' -p' . $dbPass .
                ' ' . $dbName . '| ' . $zip_util[$zip] . ' >' .
                $dest . '/' . $fname;
        }
        else
        {
            $fname = $dbName . '.' . date("w") . '.sql';
            $this->cmd = 'mysqldump -u' . $dbUser . ' -p' . $dbPass .
                ' ' . $dbName . ' >' . $dest . '/' . $fname;
        }
    }
}

```

We can then make an implementation for the Windows platform:

MySQLDump_ms.class.php (excerpt)

```
require_once 'AbstractMySQLDump.class.php';
class MySQLDump_ms extends MySQLDump
{
    protected $cmd;

    public function __construct($dbUser, $dbPass, $dbName, $dest,
        $zip = 'none')
    {
        $fname = $dbName . '.' . date("w") . '.sql';
        $this->cmd = 'mysqldump -u' . $dbUser . ' -p' . $dbPass .
            ' ' . $dbName . ' >' . $dest . '\\\' . $fname;
    }
}
```

The Windows version above includes changes to suit the Windows path and ignores the *\$zip* argument due to the lack of *gzip* and *bzip2* on that platform. This class also assumes that the path to the **mysqldump.exe** executable file is in the system *PATH* environment variable.

Here's an example of a backup script that makes use of the above classes on a Windows box:

backup2.php (excerpt)

```
<?php
require_once 'AbstractMySQLDump.class.php';
try
{
    $dbUser = 'user';
    $dbPass = 'secret';
    $dbName = 'world';
    $dest = 'c:\backups';
    $zip = 'none';
    $mysqlDump = MySQLDump::factory($dbUser, $dbPass, $dbName,
        $dest, $zip);
    $mysqlDump->backup();
}
catch (Exception $e)
{
```

```
    echo $e->getMessage();  
}  
?>
```

Since we've used an abstract class to define our API, the use of the class remains the same no matter what platform it's used on, as long as it's one of our supported platforms.

Summary

There you have it—our whirlwind tour of PDO and databases is done! By now, you should have a grasp of the basic workings between PHP's PDO extension and databases. We also covered the topics of searching, stored procedures, protecting your script from SQL injection attacks, writing flexible code, and making database backups.

Being able to work comfortably with a database is part of a strong foundation for PHP, and learning to make the most of PHP's PDO extension only makes it easier. Use the examples and solutions presented here to help build on your existing database skills.

I also hope you'll take the time to learn more about SQL and your database. Learning the nuances and capabilities of your chosen database platform can only help make your code more efficient and elegant over time.

Chapter 10

Access Control

One of the realities of building your site with PHP, as opposed to plain old HTML, is that you build dynamic web pages rather than static web pages. Making the choice to develop your site with PHP will allow you to achieve results that aren't possible with plain HTML. But, as the saying goes, with great power comes great responsibility. How can you ensure that only you, or those to whom you give permission, are able to view and interact with your web site, while it remains safe from the Internet's evil hordes as they run riot, spy on private information, or delete data?

In this chapter, we'll look at the mechanisms you can employ with PHP to build authentication systems and control access to your site. I can't stress enough the importance of a little healthy paranoia in building web-based applications. The SitePoint Forums frequently receive visits from unhappy web site developers who have had their fingers burned when it came to the security of their sites.



Data Transmission Over the Web is Insecure

Before we go any further into discussing any specific site security topics, you must be aware that any system you build that involves the transfer of data from a web page over the Internet will send that information in clear text by default

(unless you're using HTTPS, which encrypts the data). This potentially enables someone to “listen in” on the network between the client's web browser and the web server; with the help of a tool known as a **packet sniffer**, they'll be able to read the username and password sent via your form, for example. The chance of this risk eventuating is fairly small, as typically only trusted organizations like ISPs have the access required to intercept packets; however, it *is* a risk, and it's one you should take seriously.



About the Examples in this Chapter

Before we dive in, I need to let you know about the example solutions discussed in this chapter.

The example classes in some of these solutions require the use of a configuration file: **access_control.ini**. This file is used to store various database table names and column names used in the examples. Since not everyone names their database tables in the same way, configuration values like these are often intended to be customizable. The **access_control.ini** file is read into an array using the PHP `parse_ini_file` function (you can read more about this technique in “How do I store configuration information in a file?” in Chapter 6). The configuration file looks like this:

access_control.ini (excerpt)

```
; Access Control Settings

;web form variables e.g. $_POST['login']
[login_vars]
login=login
password=password
: more settings follow...
```

When an example uses configuration information from this file, that will be indicated within the section.

Similarly, the solutions below assume a certain database configuration. The SQL details relevant to each solution are indicated in the text where appropriate.

If you've downloaded the code archive for this book from the SitePoint web site, you'll find a file called **access_control_dump.sql** in the folder for this chapter. You can use this file to create the database and insert some sample data. Using this

file is identical to using the `world` database in Chapter 2. The instructions found at <http://dev.mysql.com/doc/world-setup/en/world-setup.html> can be used to create the `access_control` database too, like so:

```
command prompt> mysql -u root -p
mysql> CREATE DATABASE access_control;
mysql> USE access_control;
mysql> SOURCE access_control_dump.sql;
```

Of course, you'll have to add the missing path and password information as appropriate for your system.

Finally, all these solutions use the `PDO` class to make the connection to the database. For more information about using the `PDO` class, see Chapter 2. All the solutions involving web page forms use the `PEAR HTML_QuickForm` package. You can read more about using this package in “How do I build HTML forms with PHP?” in Chapter 5.

How do I use HTTP authentication?

Hypertext Transfer Protocol, or HTTP—the transfer protocol used to send web pages over the Internet to your web browser—defines its own authentication mechanisms. These mechanisms, basic and digest authentication, are explained in RFC 2617.¹ If you run PHP on an Apache server, you can take advantage of these mechanisms—digest is available from PHP version 5.1.0—using PHP's `header` function and a couple of predefined variables. A general discussion of these features is provided in the Features section of The PHP Manual.²



HTTP Authentication and Apache

If you wish to use HTTP authentication on your web site, you can set it up using only the Apache configuration settings—PHP is not required. For more information on how to do this, see the Apache documentation for your server version.³

¹ <http://www.ietf.org/rfc/rfc2617>

² <http://www.php.net/manual/en/features.http-auth.php>

³ For example, the documentation for version 2.2 can be found at <http://httpd.apache.org/docs/2.2/howto/auth.html>.

Solution

Let's step through a simple example page that uses the `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']` automatic global variables and the WWW-Authenticate HTTP header to protect itself—if the current user is not in a list of allowed users, access is denied.

First, we need a list of valid usernames and passwords. For the purpose of this simple demonstration, we'll just use an array, but this would not be advisable for a real-world situation where you'd likely use a database (which we'll see in “How do I build a registration system?”). Here's the `$users` array:

`httpAuth.php` (excerpt)

```
<?php
$users = array(
    'jackbenimble' => 'sekret',
    'littlepig' => 'chimmy'
);
```

Next, we test for the presence of the automatic global variable `$_SERVER['PHP_AUTH_USER']`. If the variable is not set, a username hasn't been submitted and we need to make an appropriate response—a HTTP/1.1 401 Unauthorized response code, as well as a second header to indicate that we require basic authentication using the WWW-Authenticate header:

`httpAuth.php` (excerpt)

```
if (!isset($_SERVER['PHP_AUTH_USER']))
{
    header('HTTP/1.1 401 Unauthorized');
    header('WWW-Authenticate: Basic realm="PHP Secured"');
    exit('This page requires authentication');
}
```

If a username has been submitted, we need to check that the username exists in our list of valid usernames, then ensure that the submitted password matches the one associated with the username in our list:

`httpAuth.php (excerpt)`

```

if (!isset($users[$_SERVER['PHP_AUTH_USER']]))
{
    header('HTTP/1.1 401 Unauthorized');
    header('WWW-Authenticate: Basic realm="PHP Secured"');
    exit('Unauthorized!');
}

if ($users[$_SERVER['PHP_AUTH_USER']] != $_SERVER['PHP_AUTH_PW'])
{
    header('HTTP/1.1 401 Unauthorized');
    header('WWW-Authenticate: Basic realm="PHP Secured"');
    exit('Unauthorized!');
}

```

Finally, if all our checks pass muster, we can proceed to display the web page. In this example, we simply display the credentials we've received from the authentication form. Of course, this output is for demonstration purposes only—you'd never do this in a real situation:

`httpAuth.php (excerpt)`

```

echo 'You\'re in ! Your credentials were:<br />';
echo 'Username: ' . $_SERVER['PHP_AUTH_USER'] . '<br />';
echo 'Password: ' . $_SERVER['PHP_AUTH_PW'];
?>

```

Discussion

To understand how HTTP authentication works, you must first understand what actually happens when your browser sends a web page request to a web server. HTTP is the protocol for communication between a browser and a web server. When your browser sends a request to a web server, it uses an HTTP request to tell the server which page it wants. The server then replies with an HTTP response that describes the type and characteristics of the document being sent, then delivers the document itself.

For example, a client might send the following request to a server:

```
GET /subcat/98 HTTP/1.1
Host: www.sitepoint.com
```

Here's what it might receive from the server in return:

```
HTTP/1.1 200 OK Date: Sat, 24 Mar 2007 08:12:44 GMT
Server: Apache/2.0.46 (Red Hat)
X-Powered-By: PHP/4.3.11
Transfer-Encoding: chunked
Content-Type: text/html; charset=ISO-8859-1

<!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" lang="en" xml:lang="en">
  <head>
    <title>PHP & MySQL Tutorials</title>
    : and so on...
```

If you'd like to see this process in action, the next example will give you the chance, as we open a connection to *www.sitepoint.com* and request */subcat/98*.⁴ The example script will read the response from the server and output the complete HTTP response for you:

[seeHeaders.php](#)

```
<?php
// Connect to sitepoint.com
$fp = fsockopen('www.sitepoint.com', '80');

// Send the request
fputs($fp,
      "GET /subcat/98 HTTP/1.1\r\nHost: www.sitepoint.com\r\n\r\n");

// Fetch the response
$response = '';
while (!feof($fp))
{
    $response .= fgets($fp, 128);
}
}
```

⁴ We use sockets in the next example to illustrate the passing of the HTTP headers. You can use any of a multitude of alternative methods to get the contents of the page itself, from `file_get_contents` to `fopen`, `fread`, and `fclose`. For more information, see Chapter 6.

```

fclose($fp);

// Convert HTML to entities
$response = htmlspecialchars($response);

// Display the response
echo nl2br($response);
?>

```

Authentication headers are additional headers sent by a server to instruct the browser that it must send a valid username and password in order to view the page.

In response to a normal request for a page secured with basic HTTP authentication, a server might respond with headers like these:

```

HTTP/1.1 401 Authorization Required
Date: Tue, 25 Feb 2003 15:41:54 GMT
Server: Apache/1.3.27 (Unix) PHP/4.3.1
X-Powered-By: PHP/4.3.1
WWW-Authenticate: Basic realm="PHP Secured"
Connection: close
Content-Type: text/html

```

No further information is sent, but notice the status code HTTP/1.1 401 Authorization Required and the WWW-Authenticate header. Together, these HTTP request elements indicate that the page is protected by HTTP authentication, and isn't available to an unauthorized user. A visitor's browser can convey this information in a variety of ways, but usually the user will see a small popup like that shown in Figure 10.1.

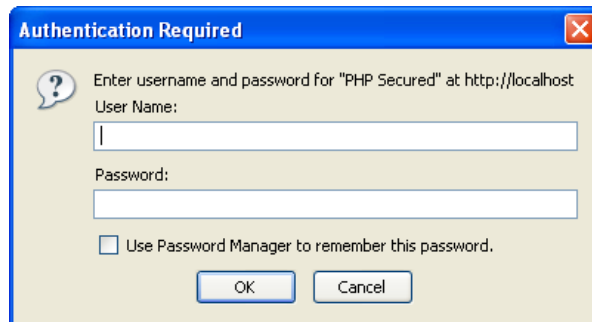


Figure 10.1. The Authentication Required dialog

The dialog prompts site visitors to enter their usernames and passwords. After visitors using Internet Explorer have entered these login details incorrectly three times, the browser displays the “Unauthorized” message instead of displaying the prompt again. In other browsers, such as Opera, users may be able to continue to try to log in indefinitely.

Notice that the `realm` value specified in the `WWW-Authenticate` header is displayed in the dialog. A **realm** is a security space or zone within which a particular set of login details are valid. Upon successful authentication, the browser will remember the correct username and password combination, and automatically resend any future request to that realm. When the user navigates to another realm, however, the browser displays a fresh prompt once again.

In any case, the user must provide a username and password to access the page. The browser sends those credentials with a second page request like this:

```
GET /admin/ HTTP/1.1
Host: www.sitepoint.com
Authorization: Basic jTSAbT766yN0hgjUi
```

The `Authorization` header contains the username and password encoded with base64 encoding which, it’s worth noting, isn’t secure—it’s unreadable for humans, but it’s a trivial task to convert base64-encoded values back to the original text.

The server will check to ensure that the credentials are valid. If they’re not, the server will send the HTTP/1.1 401 Authorization Required response again, as shown previously. If the credentials are valid, the server will send the requested page as normal.

A package you should consider if you expect to use the HTTP Authentication a lot is the `HTTP_Auth` package available from PEAR.⁵ `HTTP_Auth` provides an easy-to-use API so that you don’t have to worry about handling the header calls yourself.



Sending Headers

In PHP, the moment your script outputs anything that’s meant for display, the web server finishes sending the headers and begins to send the content itself. You

⁵ You can view the package’s information at http://pear.php.net/Auth_HTTP/.

cannot send further HTTP headers once the output of the body of the HTTP message—the web page itself—has commenced. If you do use the `header` or `session_start` functions after the rendering of the body has begun, you'll see an error message like this:

```
Warning: Cannot add header information - headers already
sent by (output started at...
```

Remember, any text or whitespace outside the `<?php ... ?>` tags causes output to be sent to the browser. If you have whitespace before a `<?php` tag or after a `?>` tag, you won't be able to send headers to the browser beyond that point.

How do I use sessions?

Sessions are a mechanism that allows PHP to **preserve state** between executions. In simple terms, sessions allow you to store variables from one page—the state of that page—and use them on another. For example, if a visitor submits his first name, Bob, via a form on your site, sessions will allow your site to remember his name, and allow you to place personal messages such as “Where would you like to go today, Bob?” on all the other pages of your site for the duration of his visit. Don't be surprised if Bob leaves rather quickly, though!

The basic mechanism of sessions works like this: first, PHP generates a unique, 32-character string to identify the session. PHP then passes the value to the browser; simultaneously, it creates a file on the server and includes the session ID in the filename. There are two methods by which PHP can keep track of the session ID: it can add the ID to the query string of all relative links on the page, or send the ID as a cookie. Within the file that's stored on the server, PHP saves the names and values of the variables it's been told to store for the session.

When the browser makes a request for another page, it tells PHP which session it was assigned via the URL query string, or by returning the cookie. PHP then looks up the file it created when the session was started, and so has access to the data stored within the session.

Once the session has been established, it'll continue until it's specifically destroyed by PHP (in response to a user clicking **Log out**, for example), or the session has been inactive for longer than a given period of time (as specified in your `php.ini` file under

`session.gc_maxlifetime`). At this point it becomes flagged for garbage collection and will be deleted the next time PHP checks for outdated sessions.

Solution

Here's a very simple demonstration of storing and retrieving a session variable:

```

simpleSession.php
<?php
session_start();
// If session variable doesn't exist, register it
if (!isset($_SESSION['test']))
{
    $_SESSION['test'] = 'Hello World!';
    echo $_SESSION['test'] . ' is registered.<br />' .
        'Please refresh page';
}
else
{
    // It's registered so display it
    echo $_SESSION['test'] . ' . ' . $_SESSION['test'];
}
?>
```

The script registers the session variable `test` the first time the page is displayed. The next time (and all times thereafter, until the session times out through inactivity), the script will display the value of the `test` session variable.

Discussion

In general, sessions are easy to use and powerful—they're an essential tool for building online applications. The first order of business in a script that uses sessions is to call `session_start` to load any existing session variables.

You should always access session variables via the predefined global variable `$_SESSION`, not the functions `session_register` and `session_unregister`. `session_register` and `session_unregister` fail to work correctly when PHP's `register_globals` setting has been disabled, which should always be the case.

In the following HTTP response headers, a server passes a session cookie to a browser as a result of the `session_start` function in a PHP script:

```

HTTP/1.1 200 OK
Date: Wed, 26 Feb 2003 02:23:08 GMT
Server: Apache/1.3.27 (Unix) PHP/4.3.1
X-Powered-By: PHP/4.3.1
Set-Cookie: PHPSESSID=ce558537fb4aefe349bb8d48c5dcc6d3; path=/
Connection: close
Content-Type: text/html

<!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">
: and so on...

```



Storing Sessions Elsewhere

Notice that I've said sessions are stored, by default, on the server as files. It's also possible to store sessions elsewhere, such as in a database or even shared memory. We'll discuss creating a custom session handler for saving the session variables to a database in "How do I store sessions in a database?". Storing sessions in database can be useful for displaying "who's online" information, and for load-balancing multiple web servers using a single-session repository—a mechanism that allows visitors to (unknowingly) swap servers while their session is maintained.



Sessions Aren't Perfect

While sessions are a wonderful tool, they can easily cause headaches if you don't understand their limitations. Take care when you handle data that's relevant to the session state. For example, when users open multiple windows for a site, a script executed in one window may overwrite data saved from another, rolling back a user to an earlier state in the site. Also be aware that resource handles and references are not saved with an object in the session—you need to release and recreate them in the `__sleep` and `__wakeup` methods of your classes.⁶ Also, try to keep the amount of data in the session variables to a minimum, as pulling large chunks of data that aren't used for every page may slow the pages down.

⁶ `__sleep` and `__wakeup` are examples of magic methods, and are explained at <http://www.php.net/manual/en/language.oop5.magic.php>.

Session Security

Sessions are very useful, but there are some important security considerations you should take into account when you use sessions in your applications.

By default, all a browser has to do to gain control of a session is pass a valid session ID to PHP. In an ideal world, you could store the IP address that registered the session, and double-check it against every new request that used the associated session ID. Unfortunately, some ISPs, such as AOL, assign their users a new IP on almost every page request, so this type of security mechanism would soon start to throw valid users out of the system. As such, it's important to design your application in a manner that assumes that one of your users will eventually have his or her session "hijacked."

The user's account is exposed until the session expires, so your aim should be to prevent the hijackers from causing serious damage while the session is active. This means, for example, that when a logged-in user goes to change his or her account password, the old password must be provided—obviously, hijackers won't know that. Also, be careful with the way you handle the users' personal information (such as credit card details). If you give users the opportunity to make significant changes to their account details, such as change a shipping addresses, be sure to send a summary notification of that change to them via email to alert users whose sessions may have been hijacked.

Keep the session ID completely hidden, using SSL (secure sockets layer) to encrypt the conversation. What's more, you should only use the cookie method of passing the session ID. If you pass it in the URL, you might give away the session ID upon referring the visitor to another site, thanks to the `referer` header in the HTTP request.

The files PHP creates for the purpose of storing session information are, by default, stored in the temporary directory of the operating system under which PHP is running. On Unix-based systems such as Linux, this directory will be `/tmp`. And, if you're on a shared server, the session files from all the hosted sites will be stored together, which means that other users on the server can read the files' contents. They might not be able to identify which virtual host and PHP script are the owners of the session but, depending on the information you place there, they might be able to guess.

This possibility is a serious cause for concern on shared PHP systems; the most effective solution is to store your sessions in a database, rather than in the server's temporary directory. We'll look more closely at custom session handlers later in this chapter, but a partial solution is to set the `session.save_path` option in your `php.ini` to a directory that's not available to the public. You'll need to contact your hosting company in order to have the correct permissions set for that directory, so that the `nobody` or `wwwuser` user with which PHP runs has access to read, write, and delete files in that directory.

One final warning: with the help of a common web security exploit, **cross-site scripting**, or XSS, it's possible for an attacker to place JavaScript on your site that will cause visitors to give away their session IDs to a remote web site, thereby allowing their sessions to be hijacked. If you allow your visitors to post any HTML to your site, make sure you check and validate it very carefully. Remember the golden rules: never rely on client-side technologies (such as JavaScript) to handle security, and never trust any content submitted from a browser.

How do I create a session class?

You can make a simple wrapper class to handle your sessions. Doing so ensures that if you ever want to switch to an alternative session-handling mechanism, such as one you've built yourself, you simply need to modify the class rather than rewriting a lot of code. We can provide an interface to the `$_SESSION` variable with a few simple methods.

Solution

Our custom `Session` class begins with the constructor method that simply calls `session_start`:

Session.class.php (excerpt)

```
class Session
{
    public function __construct()
    {
        session_start();
    }
}
```

We can then add the `set` and `get` methods to set a session variable and get a session variable, respectively:

Session.class.php (excerpt)

```
public function set($name, $value)
{
    $_SESSION[$name] = $value;
}

public function get($name)
{
    if (isset($_SESSION[$name]))
    {
        return $_SESSION[$name];
    }
    else
    {
        return false;
    }
}
```

Finally, we add a `del` method to delete a session variable, and the `destroy` method to remove all session variables and reset the session:

Session.class.php (excerpt)

```
public function del($name)
{
    unset($_SESSION[$name]);
}

function destroy()
{
    $_SESSION = array();
    session_destroy();
    session_regenerate_id();
}
}
```

How do I create a class to control access to a section of the site?

Now we reach the business end of access control—let’s look at a class that controls who’s permitted access to those private sections of your site. This class uses a database to hold the access credentials and works with an HTML login form.

Solution

The Auth class wraps login, session storage, and logout functionality in a simple, easy-to-use PHP class.

The Auth Class

The Auth class uses the following configuration settings:

access_control.ini (excerpt)

```
; Access Control Settings

;web form variables e.g. $_POST['login']
[login_vars]
login=login
password=password
hash=login_hash

;user login table details
[users_table]
table=user
col_login=login
col_password=password
```

The first two settings reflect the names of the username and password fields that will appear on the login form we’ll build in a moment. They’ll match the names of the `$_POST` variables submitted by the form: `$_POST['password']`, for example. The next three settings provide details of the table in which user information is stored—the name of the table, and the names of the username and password columns in the table.

The database table `user` will be used in all the solutions in this section. Here's the SQL for the table:

`access_control.sql` (excerpt)

```
CREATE TABLE user (  
  user_id      INT(11)      NOT NULL AUTO_INCREMENT,  
  login       VARCHAR(50) NOT NULL DEFAULT '',  
  password    VARCHAR(50) NOT NULL DEFAULT '',  
  email       VARCHAR(50) DEFAULT NULL,  
  firstName   VARCHAR(50) DEFAULT NULL,  
  lastName    VARCHAR(50) DEFAULT NULL,  
  signature   TEXT        NOT NULL,  
  PRIMARY KEY (user_id),  
  UNIQUE KEY user_login (login)  
);
```

The `Auth` class body begins with the class properties:

`Auth.class.php` (excerpt)

```
class Auth  
{  
  protected $db;  
  protected $cfg;  
  protected $session;  
  protected $redirect;  
  protected $hashKey;
```

The `$db` property will store an instance of our DB connection class, while the `$cfg` property will store the configuration settings. The `$session` property will store an instance of the `Session` class we created in “How do I create a session class?”. The `$redirect` property will store a URL to which visitors will be redirected if they aren't logged in, or if their usernames or passwords are incorrect; this might be a login form, for example. The `$hashKey` property is a seed we provide to double-check the usernames and passwords of users who are already logged in. I'll explain this in more detail later.

Now we can create the constructor method of our `Auth` class:

Auth.class.php (excerpt)

```
function __construct(PDO $db, $redirect, $hashKey)
{
    $this->db      = $db;
    $this->cfg      = parse_ini_file('access_control.ini', TRUE);
    $this->redirect = $redirect;
    $this->hashKey  = $hashKey;
    $this->session  = new Session();
    $this->login();
}
```

The constructor requires a *\$db* parameter that accepts an instance of the PDO class (although you can alter it to a custom class—just be sure to adjust the database interaction areas as required for your class). The *\$redirect* parameter is a URL string and the *\$hashKey* parameter is a string.

In the constructor, we set the Auth instance variables, load the configuration file, and create a new instance of the Session class, which we store in the *\$session* property; finally, we call the *login* method to validate the user against the database.

The *login* method checks the user's login credentials:

Auth.class.php (excerpt)

```
private function login()
{
    $var_login = $this->cfg['login_vars']['login'];
    $var_pass  = $this->cfg['login_vars']['password'];
    $user_table = $this->cfg['users_table']['table'];
    $user_login = $this->cfg['users_table']['col_login'];
    $user_pass  = $this->cfg['users_table']['col_password'];

    if ($this->session->get('login_hash'))
    {
        $this->confirmAuth();
        return;
    }
    if (!isset($_POST[$var_login]) ||
        !isset($_POST[$var_pass]))
    {
        $this->redirect();
    }
}
```

The configuration settings are assigned to local variables for the sake of readability. The `login` method first checks to see whether values for the username and password are currently stored in the session; if they are, it calls the `confirmAuth` method. If username and password values are not stored in the session, the method checks to see whether they're available in the `$_POST` array; if they're not, the method calls the `redirect` method.

Assuming the script *has* found the `$_POST` values, it calls the `md5` function to get a digest for the password:

Auth.class.php (excerpt)

```
$password = md5($_POST[$var_pass]);
```

We use the MD5 algorithm to store the password for security reasons, either in the session or on the database—we don't want to leave plain-text passwords lying around.



The MD5 Algorithm

MD5 is a simple *message digest* algorithm (often referred to as one-way encryption) that translates any string (such as a password) into a short series of ASCII characters called an **MD5 digest**. A particular string will always produce the same digest, but it's practically impossible to guess a string that will produce a given digest. By storing only the MD5 digest of your users' passwords in the database, you can verify their login credentials without actually storing the passwords on your server! The built-in PHP function `md5` lets you calculate the MD5 digest of any string in PHP.

The script then performs a query against the database to see if it can find a record to match the submitted username and password:

Auth.class.php (excerpt)

```
try
{
    $sql = "SELECT COUNT(*) AS num_users " .
        "FROM " . $user_table . " WHERE " .
        $user_login . "=:login AND " .
        $user_pass . "=:pass";
```

```

$stmt = $this->db->prepare($sql);
$stmt->bindParam(':login', $_POST[$var_login]);
$stmt->bindParam(':pass', $password);
$stmt->execute();
$row = $stmt->fetch(PDO::FETCH_ASSOC);
}
catch (PDOException $e)
{
    error_log('Error in '.$e->getFile().
        ' Line: '.$e->getLine().
        ' Error: '.$e->getMessage()
    );
    $this->redirect();
}
if ($row['num_users'] != 1)
{
    $this->redirect();
}
else
{
    $this->storeAuth($_POST[$var_login], $password);
}
}

```

We use the PDO methods `prepare` and `execute` to perform the database query, binding our `$_POST[USER_LOGIN_VAR]` and `$password` variables to the SQL parameters `:login` and `:pass` respectively. We can't authenticate the user reliably if a `PDOException` is thrown, so in that case, we log the error and call the `redirect` method.

After we fetch the result of the query, we test that there is exactly one matching record. If not, we call the `redirect` method. Finally, assuming it has reached this point, the script registers the username and password as session variables by way of the `storeAuth` method (explained below), which makes them available for future page requests.



login and Magic Quotes

One point to note about the `login` method is that it assumes `magic_quotes_gpc` is switched off. In the scripts that utilize this class, we'll need to nullify the effect of magic quotes. You can read more about this task in the section called “Checking for Magic Quotes” in Chapter 1.

The `storeAuth` method is used to add the username and password digest to the session, along with a special hash value:

Auth.class.php (excerpt)

```
public function storeAuth($login, $password)
{
    $this->session->set($this->cfg['login_vars']['login'], $login);
    $this->session->set($this->cfg['login_vars']['password'],
        $password);
    $hashKey = md5($this->hashKey . $login . $password);
    $this->session->set($this->cfg['login_vars']['hash'], $hashKey);
}
```

This special hash value is comprised of a seed value—the `$hashKey` parameter required by the constructor—as well as the username and password values. As we'll see in the `confirmAuth` method below, instead of laboriously checking the database to verify the login credentials whenever a user requests a page, the class simply checks that the current username and password produce a hash value that's the same as that stored in the session. This approach prevents potential attackers from attempting to change the stored username after login if your PHP configuration has `register_globals` enabled.

The `confirmAuth` method is used to double-check credentials stored in the session once a user is logged in:

Auth.class.php (excerpt)

```
private function confirmAuth()
{
    $login = $this->session->get(
        $this->cfg['login_vars']['login']);
    $password = $this->session->get(
```

```

        $this->cfg['login_vars']['password']);
    $hashKey = $this->session->get(
        $this->cfg['login_vars']['hash']);
    if (md5($this->hashKey . $login . $password) != $hashKey)
    {
        $this->logout(true);
    }
}

```

Notice how we reproduce the hash built by the `storeAuth` method—if this fails to match the original hash value, the user is immediately logged out.

The `logout` method is used to remove the login credentials from the session, destroy the session, and return the user to the page URL stored in the `$redirect` property:

Auth.class.php (excerpt)

```

public function logout($from = false)
{
    $this->session->del($this->cfg['login_vars']['login']);
    $this->session->del($this->cfg['login_vars']['password']);
    $this->session->del($this->cfg['login_vars']['hash']);
    $this->session->destroy();
    $this->redirect($from);
}

```

For security reasons, I choose to destroy the session here and start a completely new one. However, you may want to consider whether or not you wish to destroy the session. When the session is destroyed, not only are the Auth credentials removed, but all session data is as well, and a new session ID is created. If you have session data that you don't want to lose upon logout, you may wish to remove or comment out the `session->destroy` method call.

The final piece of our Auth class is the `redirect` method:

Auth.class.php (excerpt)

```

private function redirect($from = true)
{
    if ($from)
    {

```

```

        header('Location: ' . $this->redirect . '?from=' .
            $_SERVER['REQUEST_URI']);
    }
    else
    {
        header('Location: ' . $this->redirect);
    }
    exit();
}
}

```

The `redirect` method is used to return the visitor to the login form (or whichever URL we specified upon instantiating the `Auth` class). By default, this method will send the original page URL, requested in the `from` variable, in the query string to the URL to which the browser is redirected—most likely the login form. This allows the login form to read the query string and return the users to the location from which they came; it saves the users from having to navigate back to that point, which feature might be useful if, for example, a session times out. Note that I specified in the `logout` method that `redirect` should not provide the `from` variable. If it did, the script might return users to the URL they used to log out, trapping them in a loop from which they can't log in.

One important point to note here is that the redirection URL argument passed to the constructor function should be absolute, not relative. According to the HTTP specification, an absolute URL must be provided when a `Location` header is used. Later on, when we put this class into action, I'll break that rule and use a relative URL, because I can't guess the script's location on your server. This trick works because most recent browsers understand the relative location anyway (even though they shouldn't, as this doesn't honour the specification). On a live site, though, make sure you provide a full, absolute URL.

Finally, and most importantly, we use the `exit` function to terminate all further processing. Calling the `exit` function prevents the calling script from sending the protected content that follows the authentication code. Although we've sent a header that should redirect the browser, we can't rely on the browser to do what it's told. If the request were sent by, for instance, a Perl script pretending to be a web browser, whoever was using the script would, no doubt, have total control over

its behavior and could quite easily ignore the instruction to redirect elsewhere. Hence, the `exit` statement is crucial.

The Restricted Area

Now that you've seen the internals of the `Auth` class, let's take a look at some code that makes use of it.

Here's an example of a page we want to protect. First, we list the files we require:

`access.php` (excerpt)

```
<?php
require_once 'strip_quotes.php';
require_once 'Session.class.php';
require_once 'Auth.class.php';
require_once 'dbcred.php';
```

`strip_quotes.php` is a general-purpose script that checks for `magic_quotes_gpc = On` and strips them from incoming requests, if necessary. `classes/Session.class.php` is the `Session` class required by our `Auth` class and `classes/Auth.class.php` is the `Auth` class itself. `dbcred.php` contains our database login credentials for use with PDO. The file contains credentials relevant to our testing environment, so you'll need to change them should you wish to try this on your own web server.

Next, we instantiate the PDO object and authenticate the user. This code needs to go at the top of any page we wish to protect from unauthorized access:

`access.php` (excerpt)

```
try
{
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
}
catch (PDOException $e)
{
    error_log('Error in ' . $e->getFile().
        ' Line: ' . $e->getLine().
        ' Error: ' . $e->getMessage()
    );
}
```

```

    header('Location: error.php?err=Database Error&msg=' .
        $e->getMessage());
    exit();
}

$auth = new Auth($dbh, 'login.php', 'secret');

if (isset($_GET['action']) && $_GET['action'] == 'logout')
{
    $auth->logout();
}
?>

```

First, we attempt to create a PDO instance to connect to our database. If an exception is thrown and we can't connect, we don't want to reveal our protected content. Instead, we simply log the error, and redirect the user to an error page that displays some helpful information. Once we have a PDO instance, we can create a new Auth instance to check the current user's login credentials. We pass our PDO instance, the URL of our login form—**login.php**, and the seed for the login details hashing functionality to the constructor function. Following that, we use an `if` statement to check for a logout request. If a `$_GET['action']` variable is present and it equals the value `logout`, we know the logout link has been clicked and we should log the user out by way of the `Auth->logout` method. All we have to do to make a logout link is append `?action=logout` to any URL on our site.

Finally, here's the HTML of our restricted page, complete with a logout link:

`access.php (excerpt)`

```

<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    : HTML head contents...
  </head>
  <body>
    : restricted content...
    <p><a href="<?php echo $_SERVER['PHP_SELF']; ?>?action=logout">
      Logout</a></p>
  </body>
</html>

```


The only way the user can view this page is to provide a correct username and password. The Auth class performs the security check as soon as it's instantiated. If valid username and password values have been submitted via a form, they're stored by the Auth class in a session variable, which allows the visitor to continue using the sites various sections without having to log in again.

Creating the login form itself isn't complex, but it's made even easier with the PEAR::HTML_QuickForm package. HTML_Quickform allows us to add fields to our form and define the validation requirements easily. I won't launch into an explanation of how this works, but if you'd like to learn more about HTML_Quickform, you can read the documentation online.⁷



PEAR PHP 5 E_STRICT Compliance

It should be noted that most PEAR packages are not PHP 5 E_STRICT compliant. You can expect errors to be generated, but don't forget that you can turn them off with the `error_reporting` function. Submit a bug report to the PEAR bug system for any errors you do come across to help stomp them out in future versions.⁸

Let's begin the login form: we'll start by setting the error reporting level and requiring the PEAR::HTML_QuickForm package:

`login.php` (excerpt)

```
<?php
error_reporting(E_ALL);
require_once 'HTML/QuickForm.php';
```

We set the error reporting level to `E_ALL` with the `error_reporting` function since we're using PEAR packages, which will cause `E_Strict` errors under PHP 5.

Next we check for the presence of a `$_GET['from']` variable:

⁷ <http://pear.php.net/manual/en/package.html.html-quickform.php>

⁸ <http://pear.php.net/bugs/>

login.php (excerpt)

```
if (isset($_GET['from']))
{
    $target = $_GET['from'];
}
else
{
    $target = 'access.php';
}
?>
```

The `$_GET['from']` variable will have been set by our Auth class if it's required. This variable will represent the page to which the user was trying to gain access, and from which they've been redirected to this login form. It's used as the form's action attribute to send the user back to that page once he or she is logged in. Otherwise, for the purposes of this demonstration, the form defaults to **access.php**, our demonstration-restricted content page.

The next step is to construct our form with the `PEAR::HTML_QuickForm` class:

login.php (excerpt)

```
$form = new HTML_QuickForm('loginForm', 'POST', $target);

// Add a header to the form
$form->addElement('header', 'MyHeader', 'Please Login');

// Add a field for the login name
$form->addElement('text', 'login', 'Username');
$form->addRule('login', 'Enter your login', 'required', false,
    'client');

// Add a field for the password
$form->addElement('password', 'password', 'Password');
$form->addRule('password', 'Enter your password', 'required',
    false, 'client');

// Add a submit button
$form->addElement('submit', 'submit', ' Login ');

?>
```

Finally, we have the HTML for the login form page:

```
login.php (excerpt)

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    : HTML head contents...
  </head>
  <body>
    <h1>Please log in</h1>
    <?php echo $form->toHTML(); ?>
  </body>
</html>
```

The finished login form can be see in Figure 10.2.

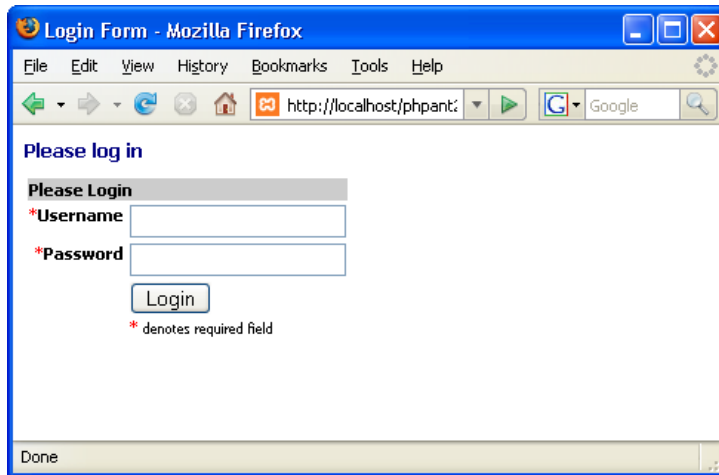


Figure 10.2. The finished login form

Discussion

Access control consists of two main parts, or stages:

Authentication

Authentication is the process by which you determine that users are who they say they are. Our Auth class handles this determination for us in the login method, when we confirm the username and password against the database.

We make the assumption that only the correct user will have these two pieces of information.

Authorization

Authorization is the process by which you determine which permissions must be given to an authenticated user. The Auth class is very limited in this respect, as no levels of access are defined—there’s only global access or no access to the site. Of course, you may want to grant a level of access that lies somewhere between these extremes, in which case you should see “How to do I build a permissions system?”

You may wonder why we handle the users in this class using a database, rather than something similar to the HTTP authentication explained earlier. There are a couple of reasons, actually. First, as a site grows from only a few members to hundreds, thousands, or millions (we hope) of members, HTTP authentication becomes harder to handle and slower. Yes, we can add the member details to the user file, but as this, in turn, grows larger, it takes longer for the server to read and find a given user. Second, what if we want to store more information about the user—as, of course, most of us do—than just the username and password? Where would we keep that information? Well, we’d keep it in the database, of course. Doesn’t that mean we’re storing user information in two places? Yes, that’s right and, as you know, that’s something we’d want to avoid; it just makes the job harder for us if we ever have to go back and change things later.

Room for Improvement

The basic mechanics of the Auth class are solid, but it lacks the more sophisticated elements that will be necessary to halt the efforts of any serious intruders.

It’s a good idea to implement a mechanism that can keep an eye on the number of failed login attempts made from a single client. If your application always responds immediately to any login attempt, it will be possible for a potential intruder to make large numbers of requests—with different username and password combinations—in a very short time, possibly using automated software to do so. The solution is to build a mechanism that counts the number of failed attempts using a session variable. Every time the number of failures is divisible by three (such as when three incorrect passwords are entered), use PHP’s `sleep` function to delay the next attempt by, for example, ten seconds. You may also decide that, after a certain threshold value (15 failed attempts, for example), you block all further access from that IP address for

a given period, such as one hour. Of course, changing an IP address is easy for a determined intruder, but you'll stall would-be intruders, at least, and perhaps make their lives difficult enough to persuade them to pursue their nefarious activities elsewhere.

How do I build a registration system?

Having an authentication system is fine, but how will you fill it with users in the first place? If only yourself and a few friends will access your site, you can probably create accounts for all users through your database administration interface. However, for a site that's intended to become a flourishing community to which anyone and everyone is free to sign up, you'll likely need to automate this process. You'll want to allow visitors to register themselves, but you'll probably conduct some level of "screening" so that you have at least a little information about the people who have signed up, such as a way to confirm their identities. A common and effective screening approach is to have the registrants confirm their email address.

The purpose of the screening mechanism is to give you the ability to make it difficult for those users who have "broken the rules" in some way and lost their account privileges to create new accounts. You have their email addresses, or at least one of their email addresses—if they try to register again with that address, you can deny them access. Be warned, though: a new type of Internet service is becoming popular. Pioneered by Mailinator, these services provide users with temporary email addresses that they can use for registrations. This, of course, means email is not a fool-proof screening mechanism, but it is still a worthwhile addition to a registration system.

Solution

Here, we'll put together a registration system that validates new registrants using their email addresses, and in turn, sends them an email that asks them to confirm their registration via a URL.

A registration system is yet another great opportunity to build more classes! This time, though, it will be even more interesting, as we use the `PEAR::HTML_QuickForm`⁹ package and `PEAR::Mail_Mime`¹⁰ to do some of the work for the registration system.

⁹ http://pear.php.net/package/HTML_QuickForm/

¹⁰ http://pear.php.net/package/Mail_Mime/

The rest will be handled by classes we'll build, but the end result will be easy for you to customize and reuse in your own applications.

First of all, we need to understand the process of signing up a new user:

- The user fills in the registration form.
- Upon the user's completion of the form, the registration system inserts a record into the `signup` table and sends a confirmation email.
- The visitor follows the link in the email and confirms the account.
- We copy the details from the `signup` table to the `user` table. The account is now active.

We use two tables for handling signups: this way, we can separate the “dangerous” or unverified user data from the “safe” or confirmed user data. You'll need a cron job or similar scheduled task to check the `signup` table on a regular basis and delete any entries that are older than, say, 24 hours. Our separation of the tables makes it easier to purge the contents of the `signup` table (and avoid unfortunate errors), and keep the `user` table trim so that there's no unnecessary impact on performance during user authentication.

Our solution uses a specific database structure. Here's the SQL for the `signup` table:

access_control.sql (excerpt)

```
CREATE TABLE signup (
  signup_id      INT(11)      NOT NULL AUTO_INCREMENT,
  login          VARCHAR(50) NOT NULL DEFAULT '',
  password       VARCHAR(50) NOT NULL DEFAULT '',
  email          VARCHAR(50) DEFAULT NULL,
  firstName      VARCHAR(50) DEFAULT NULL,
  lastName       VARCHAR(50) DEFAULT NULL,
  signature      TEXT        NOT NULL,
  confirm_code   VARCHAR(40) NOT NULL DEFAULT '',
  created        INT(11)     NOT NULL DEFAULT '0',
  PRIMARY KEY (signup_id),
  UNIQUE KEY confirm_code (confirm_code),
  UNIQUE KEY user_login (login),
  UNIQUE KEY email (email)
);
```

Here's the SQL for the user table:

access_control.sql (excerpt)

```
CREATE TABLE user (
  user_id      INT(11)      NOT NULL AUTO_INCREMENT,
  login        VARCHAR(50) NOT NULL DEFAULT '',
  password     VARCHAR(50) NOT NULL DEFAULT '',
  email        VARCHAR(50) DEFAULT NULL,
  firstName    VARCHAR(50) DEFAULT NULL,
  lastName     VARCHAR(50) DEFAULT NULL,
  signature    TEXT        NOT NULL,
  PRIMARY KEY (user_id),
  UNIQUE KEY user_login (login)
);
```

The SignUp Class

The first part of our solution is the `SignUp` class, which provides all the functionality for signing up new users, and uses the following configuration settings:

access_control.ini (excerpt)

```
; Access Control Settings

;user login table details
[users_table]
table=user
col_id=user_id
col_login=login
col_password=password
col_email=email
col_name_first=firstName
col_name_last=lastName
col_signature=signature

;signup login table details
[signup_table]
table=signup
col_id=signup_id
col_login=login
col_password=password
col_email=email
col_name_first=firstName
```

```
col_name_last=lastName
col_signature=signature
col_code=confirm_code
col_created=created
```

The first group of settings represent the details of the user table in our database—the name of the database and its columns. The second group represent the database and column names of the signup table.

Let's define some custom exception classes so that we can provide a consistent level of error handling:

Signup.class.php (excerpt)

```
class SignUpException extends Exception
{
    public function __construct($message = null, $code = 0)
    {
        parent::__construct($message, $code);
        error_log('Error in '.$this->getFile().
            ' Line: '.$this->getLine().
            ' Error: '.$this->getMessage()
        );
    }
}
class SignUpDatabaseException extends SignUpException {}
class SignUpNotUniqueException extends SignUpException {}
class SignUpEmailException extends SignUpException {}
class SignUpConfirmationException extends SignUpException {}
```

Our base class, `SignUpException`, is a custom exception that ensures the exception details are logged using the `error_log` function. The subclasses represent different exception situations that might arise during the signup process. This method of error handling implementation ensures that all exceptions are logged consistently, and allows any script that uses our `SignUp` class to implement custom logic to handle the various types of exceptions. We'll see how such logic can be implemented in our script very soon.

We begin our `SignUp` class definition with the class properties:

Signup.class.php (excerpt)

```
class SignUp
{
    protected $db;
    protected $cfg;
    protected $from;
    protected $to;
    protected $subject;
    protected $message;
    protected $html;
    protected $listener;
    protected $confirmCode;
```

`$db` will contain a PDO instance for our database connection, `$cfg` will store our configuration details, `$from` will contain the name and address used in the confirmation email's From field, `$to` will contain the name and address the email is sent to, `$subject` will contain the subject of the email, `$message` will represent the body of the email, and `$html` will contain a true or false value to indicate whether or not the email is an HTML email. The `$listener` property will contain the URL listed as the email confirmation link and `$confirmCode` will contain the unique code needed to confirm this particular user's registration.

The `$to` and `$confirmCode` properties are set and used internally by the class, while the rest of the properties are initialized by the class constructor:

Signup.class.php (excerpt)

```
public function __construct(PDO $db, $listener, $frmName,
                           $frmAddress, $subj, $msg, $html)
{
    $this->db           = $db;
    $this->cfg          = parse_ini_file('access_control.ini',
        TRUE);
    $this->listener     = $listener;
    $this->from[$frmName] = $frmAddress;
    $this->subject      = $subj;
    $this->message      = $msg;
    $this->html         = $html;
}
```

When we instantiate the object in the constructor above, we need to pass it a PDO object instance containing the connection to the database, the URL to which registrants should be directed when they confirm their signups, a Sender name and From address for use in the signup email (for example Your Name <you@yoursite.com>), and the subject and message for the email itself. Finally, we need to identify whether or not this is an HTML email, so that `PEAR::Mail_Mime` can format the message correctly.

Whether it contains HTML or not, the message should contain at least one special tag: `<confirm_url/>`. This acts as a placeholder in the message, identifying the location in the email body at which the confirmation URL, built by the `SignUp` class, should be inserted.

The `createCode` method is called internally within the class, and is used to generate the confirmation code that will be sent via email:

SignUp.class.php (excerpt)

```
private function createCode($login)
{
    srand((double)microtime() * 1000000);
    $this->confirmCode = md5($login . time() . rand(1, 1000000));
}
```

When the registration form is submitted, the `createSignup` method creates a record of the registration request. The `createSignup` method takes the information the user submits via the registration form, checks the database to ensure that the username and email address do not already exist in the user table, and inserts a new record into the signup table. Let's take a look at how this method works:

SignUp.class.php (excerpt)

```
public function createSignup($userDetails)
{
    $user_table = $this->cfg['users_table']['table'];
    $user_login = $this->cfg['users_table']['col_login'];
    $user_pass = $this->cfg['users_table']['col_password'];
    $user_email = $this->cfg['users_table']['col_email'];
    $user_first = $this->cfg['users_table']['col_name_first'];
    $user_last = $this->cfg['users_table']['col_name_last'];
```

```

$user_sig = $this->cfg['users_table']['col_signature'];

$sign_table = $this->cfg['signup_table']['table'];
$sign_login = $this->cfg['signup_table']['col_login'];
$sign_pass = $this->cfg['signup_table']['col_password'];
$sign_email = $this->cfg['signup_table']['col_email'];
$sign_first = $this->cfg['signup_table']['col_name_first'];
$sign_last = $this->cfg['signup_table']['col_name_last'];
$sign_sig = $this->cfg['signup_table']['col_signature'];
$sign_code = $this->cfg['signup_table']['col_code'];
$sign_created = $this->cfg['signup_table']['col_created'];

try
{
    $sql = "SELECT COUNT(*) AS num_row FROM " . $user_table . "
        WHERE
            " . $user_login . "=:login OR
            " . $user_email . "=:email";
    $stmt = $this->db->prepare($sql);
    $stmt->bindParam(':login', $userDetails[$user_login]);
    $stmt->bindParam(':email', $userDetails[$user_email]);
    $stmt->execute();
    $result = $stmt->fetch(PDO::FETCH_ASSOC);
}
catch (PDOException $e)
{
    throw new SignUpDatabaseException('Database error when ' .
        'checking user is unique: '.$e->getMessage());
}

```

First, we assign all the needed configuration settings to local variables to improve the readability of our script. The first action the method performs is to complete a database query: it counts the number of rows in the user table where the submitted username matches the value in the login column in the database, or where the submitted email address is a match to the value in the email column. We wrap this action within a try {...} catch (PDOException \$e) {...} block in case a PDOException is thrown. When we catch the PDOException, we throw one of the custom exceptions we wrote for this class—a SignUpDatabaseException.

The next step for the createSignup method is to check the results of the query and, if it's okay to proceed, to prepare the data for insertion into the signup table:

Signup.class.php (excerpt)

```

if ($result['num_row'] > 0)
{
    throw new SignUpNotUniqueException(
        'username and email address not unique');
}

$this->createCode($userDetails[$user_login]);
$toName = $userDetails[$user_first] . ' ' .
    $userDetails[$user_last];
$this->to[$toName] = $userDetails[$user_email];

```

If, on the other hand, the result is not 0, it indicates that we already have a user with that username or email address, and it's not okay to proceed with the signup. Our reaction is to throw another one of our custom exceptions, this time a `SignUpNotUniqueException`, to indicate that the signup details are not unique.

The final step in the `createSignup` method is to insert the new registration into the `signup` table:

Signup.class.php (excerpt)

```

try
{
    $sql = "INSERT INTO " . $sign_table .
        "(. $sign_login . ", " . $sign_pass .
        ", " . $sign_email . ", " . $sign_first .
        ", " . $sign_last . ", " . $sign_sig .
        ", " . $sign_code . ", " . $sign_created . ") ".
        "VALUES (:login, :password,
        :email, :firstname, :lastname,
        :signature, :confirm, :time)";
    $stmt = $this->db->prepare($sql);
    $stmt->bindParam(':login', $userDetails[$user_login]);
    $stmt->bindParam(':password', $userDetails[$user_pass]);
    $stmt->bindParam(':email', $userDetails[$user_email]);
    $stmt->bindParam(':firstname', $userDetails[$user_first]);
    $stmt->bindParam(':lastname', $userDetails[$user_last]);
    $stmt->bindParam(':signature', $userDetails[$user_sig]);
    $stmt->bindParam(':confirm', $this->confirmCode);
    $stmt->bindParam(':time', time());
    $stmt->execute();
}

```

```

    }
    catch (PDOException $e)
    {
        throw new SignUpDatabaseException('Database error when' .
            ' inserting into signup: '.$e->getMessage());
    }
}

```

All the data in the `$userDetails` variable—the details submitted via the registration form—are inserted into the `signup` table. If a `PDOException` is thrown, we throw a new instance of our `SignUpDatabaseException` class.

The `sendConfirmation` method is used to send a confirmation email to the person who's just signed up:

Signup.class.php (excerpt)

```

public function sendConfirmation()
{
    // Pear Mail_Mime included in the calling script
    $fromName = key($this->from);
    $hdrs = array(
        'From' => $this->from[$fromName],
        'Subject' => $this->subject
    );
    $crlf = "\n";

    if ($this->html)
    {
        $replace = '<a href="' . $this->listener . '?code=' .
            $this->confirmCode . '">' . $this->listener .
            '?code=' . $this->confirmCode . '</a>';
    }
    else
    {
        $replace = $this->listener . '?code=' . $this->confirmCode;
    }
    $this->message = str_replace('<confirm_url/>',
        $replace,
        $this->message
    );

    $mime = new Mail_mime($crlf);

```

```

    $mime->setHTMLBody($this->message);
    $mime->setTXTBody(strip_tags($this->message));
    $body = $mime->get();
    $hdrs = $mime->headers($hdrs);
    $mail = Mail::factory('mail');
    $succ = $mail->send($this->to, $hdrs, $body);
    if (PEAR::isError($succ))
    {
        throw new SignUpEmailException('Error sending confirmation' .
            ' email: ' . $succ->getDebugInfo());
    }
}

```

The `sendConfirmation` method will generate the content of the confirmation email, in HTML or text, by replacing the special text `<confirm_url/>` with the confirmation URL the user will need to click on to confirm the registration. The confirmation URL is generated using the `$listener` property, set by the class constructor method, and the unique code returned by the `confirmCode` method. `sendConfirmation` then uses an instance of the `PEAR::Mail_mime` class to create and send the email. If an error is generated with the sending of the email, another one of our custom exceptions, `SignUpEmailException`, will be thrown. We'll also use the `getDebugInfo` method of the `PEAR_Error` object to obtain some information about the error.

Finally, the `confirm` method is used to examine confirmations via the URL sent in the email:

Signup.class.php (excerpt)

```

public function confirm($confirmCode)
{
    $user_table = $this->cfg['users_table']['table'];
    $user_login = $this->cfg['users_table']['col_login'];
    $user_pass = $this->cfg['users_table']['col_password'];
    $user_email = $this->cfg['users_table']['col_email'];
    $user_first = $this->cfg['users_table']['col_name_first'];
    $user_last = $this->cfg['users_table']['col_name_last'];
    $user_sig = $this->cfg['users_table']['col_signature'];

    $sign_table = $this->cfg['signup_table']['table'];
    $sign_id = $this->cfg['signup_table']['col_id'];
    $sign_login = $this->cfg['signup_table']['col_login'];

```

```

$sign_pass = $this->cfg['signup_table']['col_password'];
$sign_email = $this->cfg['signup_table']['col_email'];
$sign_first = $this->cfg['signup_table']['col_name_first'];
$sign_last = $this->cfg['signup_table']['col_name_last'];
$sign_sig = $this->cfg['signup_table']['col_signature'];
$sign_code = $this->cfg['signup_table']['col_code'];

try
{
    $sql = "SELECT * FROM " . $sign_table . "
          WHERE " . $sign_code . "=:confirmCode";
    $stmt = $this->db->prepare($sql);
    $stmt->bindParam(':confirmCode', $confirmCode);
    $stmt->execute();
    $row = $stmt->fetchAll();
}
catch (PDOException $e)
{
    throw new SignUpDatabaseException('Database error when ' .
        ' inserting user info: '.$e->getMessage());
}

```

Again, we assign configuration settings to local variables to improve the script's readability. First, the `confirm` method selects from the `signup` table all records that have a value in the `confirm_code` column that matches the `$confirmCode` value.

If the number of records returned is anything other than 1, a problem has occurred and a `SignUpConfirmationException` exception is thrown:

Signup.class.php (excerpt)

```

if (count($row) != 1) {
    throw new SignUpConfirmationException(count($row) .
        ' records found for confirmation code: ' .
        $confirmCode
    );
}

```

If only one matching record is found, the method can continue to process the confirmation:

```

try
{
    // Copy the data from Signup to User table
    $sql = "INSERT INTO " . $user_table . " (
        " . $user_login . " , " . $user_pass . " ,
        " . $user_email . " , " . $user_first . " ,
        " . $user_last . " , " . $user_sig . " ) VALUES (
        :login, :pass, :email, :firstname, :lastname, :sign )";
    $stmt = $this->db->prepare($sql);
    $stmt->bindParam(':login', $row[0][$sign_login]);
    $stmt->bindParam(':pass', $row[0][$sign_pass]);
    $stmt->bindParam(':email', $row[0][$sign_email]);
    $stmt->bindParam(':firstname', $row[0][$sign_first]);
    $stmt->bindParam(':lastname', $row[0][$sign_last]);
    $stmt->bindParam(':sign', $row[0][$sign_sig]);
    $stmt->execute();
    // Delete row from signup table
    $sql = "DELETE FROM " . $sign_table . "
        WHERE " . $sign_id . " = :id";
    $stmt = $this->db->prepare($sql);
    $stmt->bindParam(':id', $row[0][$sign_id]);
    $stmt->execute();
}
catch (PDOException $e)
{
    throw new SignUpDatabaseException('Database error when ' .
        ' inserting user info: ' . $e->getMessage());
}
}
}

```

If an account is successfully confirmed, the record is copied to the user table, and the old record is deleted from the `signup` table.

Thus the confirmation process, the user's registration, and our `SignUp` class, is complete!

The Signup Page

Now that our `SignUp` class is done, we need a web page from which to display the registration form and run the process.

The first step is to include the classes we'll use:

signup.php (excerpt)

```
<?php
error_reporting(E_ALL);
require_once 'SignUp.class.php';
require_once 'HTML/QuickForm.php';
require_once 'Mail.php';
require_once 'Mail/mime.php';
require 'dbcred.php';
```

First, because we're using PEAR packages, which will cause `E_Strict` errors under PHP 5, we set the error reporting level to `E_ALL` with the `error_reporting` function.

Of course, we need to include our `SignUp` class file. We'll also be using the PEAR `HTML_Quickform` and `Mail_mime` packages. The `dbcred.php` file contains the database credentials we'll need to connect to our database.

Next, we create the variables we need:

signup.php (excerpt)

```
$reg_messages = array(
    'success' => array(
        'title' => 'Confirmation Successful',
        'content' => '<p>Thank you. Your account has now been ' .
        ' confirmed.<br />You can now <a href="access.php">login ' .
        '</a></p>'
    ),
    'confirm_error' => array(
        'title' => 'Confirmation Problem',
        'content' => '<p>There was a problem confirming your ' .
        ' account.<br />Please try again or contact the site ' .
        ' administrators</p>'
    ),
    'email_sent' => array(
        'title' => 'Check your email',
        'content' => '<p>Thank you. Please check your email to ' .
        ' confirm your account</p>'
    ),
    'email_error' => array(
        'title' => 'Email Problem',
```

```

        'content' => '<p>Unable to send confirmation email.<br />' .
        'Please contact the site administrators.</p>'
    ),
    'signup_not_unique' => array(
        'title' => 'Registration Problem',
        'content' => '<p>There was an error creating your' .
        ' account.<br />The desired username or email address has' .
        ' already been taken.</p>'
    ),
    'signup_error' => array(
        'title' => 'Registration Problem',
        'content' => '<p>There was an error creating your' .
        ' account.<br />Please contact the site administrators.' .
        '</p>'
    )
);
$listener = 'http://localhost/phpant2/chapter_10/examples/' .
    'signup.php';
$frmName = 'Your Name';
$frmAddress = 'noreply@yoursite.com';
$subj = 'Account Confirmation';
$msg = <<<EOD
<html>
<body>
<h2>Thank you for registering!</h2>
<div>The final step is to confirm
your account by clicking on:</div>
<div><confirm_url/></div>
<div>
<b>Your Site Team</b>
</div>
</body>
</html>
EOD;

```

The `$reg_messages` variable contains an array of page titles and messages that will be used in the web page, depending on the stage and status of the registration process. `$listener`, `$frmName`, `$frmAddress`, `$subj`, and `$msg` are required by our `Signup` class. If you have a look at the `$msg` variable, the body of our confirmation email, you'll see the special `<confirm_url/>` code which will be replaced by the confirmation URL later in the process.

The `$listener` variable stores the absolute URL of the script to which the confirmation code should be submitted. It links to itself in our example script. This variable is set to reflect the folder setup of our testing environment, so make sure you change this variable to suit your own setup.

The next step is to set up our database connection and instantiate our `SignUp` object:

`signup.php` (excerpt)

```
try
{
    // Instantiate the PDO object for the database connection
    $db = new PDO($dsn, $user, $password);
    $db->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);

    // Instantiate the signup class
    $signup = new SignUp($db, $listener, $frmName,
        $frmAddress, $subj, $msg, TRUE);
```

Notice also that we're opening a try block to catch any exceptions that may be thrown from the execution of the rest of the code. Any exceptions caught after this point—if the PDO connection fails for example—will display an appropriate message on the web page, instead of showing a PHP error.

The next step is to check whether the page is being requested as part of a confirmation—we'll check for the presence of the `$_GET['code']` variable:

`signup.php` (excerpt)

```
if (isset($_GET['code']))
{
    try
    {
        $signup->confirm($_GET['code']);
        $display = $reg_messages['success'];
    } catch (SignUpException $e){
        $display = $reg_messages['confirm_error'];
    }
}
```

If the confirmation code is present, we call the `SignUp->confirm` method, supplying the code the page received. We then set the `$display` variable, which will contain the page title and message to display on our web page. If no exception was raised from the `confirm` method at this point in the script, we can assume all went well and set the `$display` variable to the success message. If, however, a `SignUpException` exception was thrown, we set the `$display` variable to the `confirmation_error` message. You may remember that the `SignUpException` class was the base class for all our custom exceptions. By catching this class of exception, we'll catch an instance of any of our custom exceptions.

If the confirmation code is not present, we prepare to display the registration form:

signup.php (excerpt)

```
else
{
    function cmpPass($element, $confirmPass)
    {
        $password = $GLOBALS['form']->getElementValue('password');
        return $password == $confirmPass;
    }
    function encryptValue($value)
    {
        return md5($value);
    }
}
```

The above are helper functions that will be used by our `HTML_Quickform` object to validate and filter the registration form contents.

The `HTML_Quickform` object makes it very easy to construct the form and the form validation:

signup.php (excerpt)

```
/* Make the form */
// Instantiate the QuickForm class
$form = new HTML_QuickForm('regForm', 'POST');

// Register the compare function
$form->registerRule('compare', 'function', 'cmpPass');
```

```

// The login field
$form->addElement('text', 'login', 'Desired Username');
$form->addRule('login', 'Please provide a username',
    'required', FALSE, 'client');
$form->addRule('login',
    'Username must be at least 6 characters',
    'minlength', 6, 'client');
$form->addRule('login',
    'Username cannot be more than 50 characters', 'maxlength',
    50, 'client');
$form->addRule('login',
    'Username can only contain letters and numbers',
    'alphanumeric', NULL, 'client');

// The password field
$form->addElement('password', 'password', 'Password');
$form->addRule('password', 'Please provide a password',
    'required', FALSE, 'client');
$form->addRule('password',
    'Password must be at least 6 characters', 'minlength', 6,
    'client');
$form->addRule('password',
    'Password cannot be more than 12 characters', 'maxlength',
    12, 'client');
$form->addRule('password',
    'Password can only contain letters and numbers',
    'alphanumeric', NULL, 'client');

// The field for confirming the password
$form->addElement('password', 'confirmPass',
    'Confirm Password');
$form->addRule('confirmPass', 'Please confirm password',
    'required', FALSE, 'client');
$form->addRule('confirmPass', 'Passwords must match',
    'compare', 'function');

// The email field
$form->addElement('text', 'email', 'Email Address');
$form->addRule('email', 'Please enter an email address',
    'required', FALSE, 'client');
$form->addRule('email', 'Please enter a valid email address',
    'email', FALSE, 'client');
$form->addRule('email',
    'Email cannot be more than 50 characters',
    'maxlength', 50, 'client');

```

```

// The first name field
$form->addElement('text', 'firstName', 'First Name');
$form->addRule('firstName', 'Please enter your first name',
    'required', FALSE, 'client');
$form->addRule('firstName',
    'First name cannot be more than 50 characters', 'maxlength',
    50, 'client');

// The last name field
$form->addElement('text', 'lastName', 'Last Name');
$form->addRule('lastName', 'Please enter your last name',
    'required', FALSE, 'client');
$form->addRule('lastName',
    'Last name cannot be more than 50 characters', 'maxlength',
    50, 'client');

// The signature field
$form->addElement('textarea', 'signature', 'Signature');

// Add a submit button called submit
// and "Send" as the button text
$form->addElement('submit', 'submit', 'Register');
/* End making the form */

```

After we've defined the registration form, we use the `HTML_Quickform->validate` method to check that the form has been submitted and that it validates. If it does validate, we can proceed to build the array of form data our `SignUp` object needs to create a new `signup` record:

`signup.php` (excerpt)

```

if ($form->validate())
{
    // Apply the encryption filter to the password
    $form->applyFilter('password', 'encryptValue');

    // Build an array from the submitted form values
    $submitVars = array(
        'login' => $form->getSubmitValue('login'),
        'password' => $form->getSubmitValue('password'),
        'email' => $form->getSubmitValue('email'),
        'firstName' => $form->getSubmitValue('firstName'),

```

```
'lastName' => $form->getSubmitValue('lastName'),
'signature' => $form->getSubmitValue('signature')
);
```

Since we're using `HTML_QuickForm`, any slashes added by magic quotes are automatically removed from the submitted values; when you're not using `HTML_QuickForm`, be sure to strip out the slashes if `magic_quotes` is enabled.

Next, we call the `createSignup` record and send the confirmation email. We want to wrap this in a `try` block in order to catch any possible exceptions:

`signup.php (excerpt)`

```
try
{
    $signup->createSignup($submitVars);
    $signup->sendConfirmation();
    $display = $reg_messages['email_sent'];
}
catch (SignUpEmailException $e)
{
    $display = $reg_messages['email_error'];
}
catch (SignUpNotUniqueException $e)
{
    $display = $reg_messages['signup_not_unique'];
}
catch (SignUpException $e)
{
    $display = $reg_messages['signup_error'];
}
}
```

If no exceptions are thrown, we can set `$display` to an appropriate message that informs the user to expect the email. If exceptions are thrown, we can set `$display` to a message that's appropriate for each one, thanks to our defining of several custom exception classes.

If the form hasn't been submitted yet, it'll need to be shown to the user; we set `$display` to include the form HTML source:

signup.php (excerpt)

```
else
{
    // If not submitted, display the form
    $display = array(
        'title' => 'New Registration',
        'content' => $form->toHtml()
    );
}
}
```

We've reached the end of the first try block, so we need to catch any remaining exception that may be thrown. If an exception is caught here, it won't be one of our custom exceptions. Therefore, we need to make sure that the exception details are logged using the `error_log` function, and that the web page displays an appropriate message to inform the user that registration cannot be completed:

signup.php (excerpt)

```
catch (Exception $e)
{
    error_log('Error in '.$e->getFile().
        ' Line: '.$e->getLine().
        ' Error: '.$e->getMessage()
    );
    $display = $reg_messages['signup_error'];
}
?>
```

Now, the only task left to do is to produce the HTML source for the web page. Our `$display` variable has been set to an array value containing two elements—one for the page title and one for the page contents. This setting will display the registration form and a confirmation message, or an error message if something has gone wrong. These displays are inserted into the source code where appropriate:

signup.php (excerpt)

```

<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    : HTML Head contents...
  </head>
  <body>
    <h1><?php echo $display['title']; ?></h1>
    <?php echo $display['content']; ?>
  </body>
</html>

```

The finished registration form should look like the one shown in Figure 10.3.

The screenshot shows a Mozilla Firefox browser window titled "New Registration - Mozilla Firefox". The address bar shows "http://localhost/phpant:". The page content is titled "New Registration" and contains the following form fields:

- *Desired Username
- *Password
- *Confirm Password
- *Email Address
- *First Name
- *Last Name
- Signature

Below the fields is a "Register" button and a note: "* denotes required field".

Figure 10.3. The finished registration form

And there we have it—a simple but fully functioning user registration system with email confirmation facility!

Discussion

So that you don't grow bored, I've left a couple of pieces of the jigsaw puzzle for you to fill in yourself. If a registered user exists who has the same username or email address as the one entered by the new registrant, the `createSignup` method throws an exception and the procedure is halted. If you're happy using `HTML_QuickForm`, you might want to split this check into a separate method that `HTML_QuickForm` can apply as a validation rule for each field in the form. This approach should reduce frustration when users find that the account name they chose already exists—`HTML_QuickForm` will generate a message to inform them of this fact, preserve the rest of the values they entered, and allow them to try again with a different username.

If you plan to let users change their email addresses once their accounts are created, you'll also need to confirm the new addresses before you store them in the `user` table. You should be able to reuse the methods provided by the `SignUp` class for this purpose. You might even consider reusing the `signup` table to handle this task. Some modifications will be required—you'll want the `confirm` method to be able to update an existing record in the `user` table, for example. Be very careful that you don't create a hole in your security, though. If you're not checking for existing records in the `user` table, a user could sign up for a new account with details that match an existing row in the `user` table. You'll then end up changing the email address of an existing user to that of a new user, which will cause you some embarrassment, at the very least.

How do I deal with members who forget their passwords?

Unfortunately, humans have a tendency to forget important information such as passwords, so a feature that allows users to retrieve forgotten passwords is an essential time saver. Overlook this necessity, and you can expect to waste a lot of time manually changing passwords for people who have forgotten them.

If you encrypt the passwords in your database, you'll need a mechanism to generate a new password that, preferably, is easy to remember.



Be Careful with Password Hints

A common tactic used in web site registration is to use simple questions as memory joggers should users forget their password. These questions can include “Where were you born?” and “What’s your pet’s name?” Yet details like this may well be common knowledge or easy for other users to guess.

Solution

Since we already have a valid email address for each account, as confirmed through our signup procedure in “How do I build a registration system?”, we just need to send the new password to that address. Our solution uses the user table from the previous sections:

access_control.sql (excerpt)

```
CREATE TABLE user (
  user_id      INT(11)      NOT NULL AUTO_INCREMENT,
  login        VARCHAR(50) NOT NULL DEFAULT '',
  password     VARCHAR(50) NOT NULL DEFAULT '',
  email        VARCHAR(50) DEFAULT NULL,
  firstName    VARCHAR(50) DEFAULT NULL,
  lastName     VARCHAR(50) DEFAULT NULL,
  signature    TEXT        NOT NULL,
  PRIMARY KEY (user_id),
  UNIQUE KEY user_login (login)
);
```

The AccountMaintenance Class

The AccountMaintenance class is a utility class that, among other things, will reset the password for a user’s account and generate an email to send the user the new password. Our class uses the following configuration settings:

access_control.ini (excerpt)

```
; Access Control Settings

;web form variables e.g. $_POST['login']
[login_vars]
login=login
```

```

;user login table details
[users_table]
table=user
col_id=user_id
col_login=login
col_password=password
col_email=email
col_name_first=firstName
col_name_last=lastName

```

To provide a consistent level of error handling, we define some custom exception classes:

AccountMaintenance.class.php (excerpt)

```

class AccountException extends Exception
{
    public function __construct($message = null, $code = 0)
    {
        parent::__construct($message, $code);
        error_log('Error in '.$this->getFile().
            ' Line: '.$this->getLine().
            ' Error: '.$this->getMessage()
        );
    }
}
class AccountDatabaseException extends AccountException {}
class AccountUnknownException extends AccountException {}
class AccountPasswordException extends AccountException {}
class AccountPasswordResetException extends AccountException {}

```

Our base class, `AccountException`, is a custom exception that ensures the exception details are logged using the `error_log` function. The subclasses represent different exception situations that might arise during account maintenance.

We begin our `AccountMaintenance` class definition with the class properties:

AccountMaintenance.class.php (excerpt)

```
class AccountMaintenance
{
    protected $db;
    protected $cfg;
    private $words;
```

`$db` will contain a PDO instance for our database connection, `$cfg` will store our configuration details, and `$words` will store the path to the random words file that's used in password generation.

The constructor simply stores the database object for future use by the class and loads the configuration file:

AccountMaintenance.class.php (excerpt)

```
public function __construct(PDO $db)
{
    $this->db = $db;
    $this->cfg = parse_ini_file('access_control.ini', TRUE);
}
```

Since we save the user's password in the database as an MD5 hash (a form of one-way encryption), we can no longer find out what the original password was. If members forget their passwords in such cases, you'll have to make new ones for them. You could simply generate a random string of characters, but it's important to remember that if you make your security systems too unfriendly, you'll put off legitimate users. The `resetPassword` method generates a more human-friendly randomized password:

AccountMaintenance.class.php (excerpt)

```
function resetPassword($login, $email)
{
    //Put the cfg vars into local vars for readability
    $user_table = $this->cfg['users_table']['table'];
    $user_id = $this->cfg['users_table']['col_id'];
    $user_login = $this->cfg['users_table']['col_login'];
    $user_pass = $this->cfg['users_table']['col_password'];
    $user_email = $this->cfg['users_table']['col_email'];
```

```

$user_first = $this->cfg['users_table']['col_name_first'];
$user_last = $this->cfg['users_table']['col_name_last'];
$user_sig = $this->cfg['users_table']['col_signature'];

try
{
    $sql = "SELECT " . $user_id . ",
            " . $user_login . ", " . $user_pass . ",
            " . $user_first . ", " . $user_last . "
        FROM
            " . $user_table . "
        WHERE
            " . $user_login . " =:login
        AND
            " . $user_email . " =:email";
    $stmt = $this->db->prepare($sql);
    $stmt->bindParam(':login', $login);
    $stmt->bindParam(':email', $email);
    $stmt->execute();
    $row = $stmt->fetchAll(PDO::FETCH_ASSOC);
}
catch (PDOException $e)
{
    throw new AccountDatabaseException('Database error when ' .
        ' finding user: ' . $e->getMessage());
}

```

First, we assign the configuration settings to local variables to make the code a little more readable. Next, we deal with the `resetPassword` method, which, when given a combination of a username and an email address, attempts to identify the corresponding row in the user table.

We use both the username and email to identify the row, so it's a little more difficult for other people to reset your members' passwords. Although there's no risk of individuals stealing the new password (unless they have control over a member's email account), it will certainly irritate people if their passwords are continually being reset. Requiring both the username and email address of the user makes the process a little more complex.

If we can't find a single matching row, we throw an exception:

AccountMaintenance.class.php (excerpt)

```

if (count($row) != 1)
{
    throw new AccountUnknownException('Could not find account');
}

```

Next, we call the `generatePassword` method (which we'll discuss in a moment) to create a new password:

AccountMaintenance.class.php (excerpt)

```

try
{
    $password = $this->generatePassword();
}

```

This method call is placed within a try block to catch the exception thrown by `generatePassword` if a new password cannot be generated.

`generatePassword` then updates the user table with the new password (using `md5` to encrypt it), and returns the new password in an array containing the user details:

AccountMaintenance.class.php (excerpt)

```

$sql = "UPDATE " . $user_table . "
SET
    " . $user_pass . " =:pass
WHERE
    " . $user_id . " =:id";
$stmt = $this->db->prepare($sql);
$stmt->bindParam(':pass', md5($password));
$stmt->bindParam(':id', $row[0][$user_id]);
$stmt->execute();
}
catch (AccountPasswordException $e)
{
    throw new AccountResetPasswordException('Error when ' .
        ' generating password: ' . $e->getMessage());
}
catch (PDOException $e)
{
    throw new AccountDatabaseException('Database error when ' .

```

```

        ' resetting password: ' . $e->getMessage());
    }
    $row[0][$user_pass] = $password;
    return $row;
}

```

The `addWords` method is used to supply the class with an indexed array of words with which to build memorable passwords:

AccountMaintenance.class.php (excerpt)

```

function addWords($words)
{
    $this->words = $words;
}

```

I've used a list of over one thousand words, stored in a text file, to build memorable passwords. Be aware that if anyone knows the list of words you're using, cracking the new password will be significantly easier, so you should create your own list.

`generatePassword` constructs a random password from the `AccountMaintenance->words` array, adding separators that can include any number from 0 to 9, or an underscore character:

AccountMaintenance.class.php (excerpt)

```

protected function generatePassword()
{
    $count = count($this->words);
    if ($count == 0)
    {
        throw new AccountPasswordException('No words to use!');
    }
    mt_srand((double)microtime() * 1000000);
    $seperators = range(0,9);
    $seperators[] = '_';
    $password = array();
    for ($i = 0; $i < 4; $i++) {
        if ($i % 2 == 0) {
            shuffle($this->words);
            $password[$i] = trim($this->words[0]);

```



```

    } else {
        shuffle($seperators);
        $password[$i] = $seperators[0];
    }
}
shuffle($password);
return implode('', $password);
}
}

```

The password itself will contain two words chosen at random from the list, as well as two random separators. The order in which these elements appear in the password is also random. The passwords this system generates might look something like 7correct9computer and 48courtclothes, which follow a format that's relatively easy for users to remember.

The Reset Password Page

There's one thing we need to finish our web site's account maintenance feature: we need a web form that our users can fill in to request a password change or reset. First, we include all the packages we need:

newpass.php (excerpt)

```

<?php
error_reporting(E_ALL);
require_once 'Session.class.php';
require_once 'AccountMaintenance.class.php';
require_once 'HTML/QuickForm.php';
require_once 'Mail.php';
require_once 'Mail/mime.php';
require_once 'dbcred.php';

```

We then set the error reporting level to `E_ALL` with the `error_reporting` function, since we're using PEAR packages that will cause `E_Strict` errors under PHP 5.

Of course, we need to include our `AccountMaintenance` class file. We'll also be using the PEAR `HTML_Quickform` and `Mail_mime` packages. The `dbcred.php` file contains the database credentials we'll need to connect to our database.

Next, we create the variables we need:

newpass.php (excerpt)

```

$reg_messages = array(
    'email_sent' => array(
        'title' => 'Check your email',
        'content' => '<p>Thank you. An email has been sent to:</p>'
    ),
    'email_error' => array(
        'title' => 'Email Problem',
        'content' => '<p>Unable to send your details.<br />' .
        'Please contact the site administrators.</p>'
    ),
    'no_account' => array(
        'title' => 'Account Problem',
        'content' => '<p>We could not find your account.<br />' .
        'Please contact the site administrators.</p>'
    ),
    'reset_error' => array(
        'title' => 'Password Reset Problem',
        'content' => '<p>There was an error resetting your ' .
        'password.<br />Please contact the site administrators.' .
        '</p>'
    )
);
$yourEmail = 'you@yourdomain.com';
$subject = 'Your password';
$msg = 'Here are your login details. Please change your password.';

```

The `$reg_messages` variable contains an array of page titles and messages that will be used in the web page at various stages of the registration process. `$yourEmail`, `$subject`, and `$msg` are used in the creation of the email notification.

Next, we build our form with `PEAR::HTML_Quickform`:

newpass.php (excerpt)

```

try
{
    // Instantiate the QuickForm class
    $form = new HTML_QuickForm('passwordForm', 'POST');

    // Add a header to the form
    $form->addElement('header', 'MyHeader',
        'Forgotten Your Password?');
}

```

```

// Add a field for the email address
$form->addElement('text', 'email', 'Enter your email address');
$form->addRule('email', 'Enter your email', 'required', FALSE,
    'client');
$form->addRule('email', 'Enter a valid email address', 'email',
    FALSE, 'client');
// Add a field for the login
$form->addElement('text', 'login', 'Enter your login name');
$form->addRule('login', 'Enter your login', 'required', FALSE,
    'client');

// Add a submit button called submit with label "Send"
$form->addElement('submit', 'submit', 'Get Password');

```

Notice also that we're opening a try block: we want to catch any exceptions that may be thrown from the execution of the rest of the code. This precaution will allow us to display an appropriate message on the web page instead of a PHP error.

If the form has been submitted, we can begin the password changing process:

newpass.php (excerpt)

```

if ($form->validate())
{
    $db = new PDO($dsn, $user, $password);
    $db->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $aMaint = new AccountMaintenance($db);
    $rawWords = file('words.txt');
    $word = array_map('trim', $rawWords);
    $aMaint->addWords($word);
}

```

We instantiate the PDO and AccountMaintenance classes and load our words file (I also trimmed off any whitespace that may appear before or after each word—just in case) so we can pass it to the addWords method.

Next, we call the resetPassword method, passing the login and email values from the form as arguments:

newpass.php (excerpt)

```
$details = $aMaint->resetPassword(
    $form->getSubmitValue('login'),
    $form->getSubmitValue('email'));
```

If all goes well, an email is sent via `PEAR::Mail_Mime` to inform the user of the new password:

newpass.php (excerpt)

```
$CrLf = "\n";
$text = $msg . "\n\nLogin: " . $details[0]['login'] .
    "\nPassword: " . $details[0]['password'];

$headers = array(
    'From'      => $yourEmail,
    'Subject'   => $subject
);

$mime = new Mail_mime($CrLf);
$mime->setTXTBody($text);
$body = $mime->get();
$headers = $mime->headers($headers);
$mail = Mail::factory('mail');
// Send the message
$succ = $mail->send($form->getSubmitValue('email'), $headers,
    $body);
if (PEAR::isError($succ))
{
    $display = $reg_messages['email_error'];
}
else
{
    $display = $reg_messages['email_sent'];
    $display['content'] .= '<p>' .
        $form->getSubmitValue('email') . '</p>';
}
}
```

The page `$display` variable is set to a helpful message when the email is sent successfully; if it's not, the `$display` variable displays an error message.

If the form hasn't yet been submitted, we just display the form HTML:

`newpass.php` (excerpt)

```
else
{
    $display = array(
        'title' => 'Reset Password',
        'content' => $form->toHtml()
    );
}
}
```

Finally, we catch any exceptions that may have occurred and display an appropriate message:

`newpass.php` (excerpt)

```
catch (AccountUnknownException $e)
{
    $display = $reg_messages['no_account'];
}
catch (Exception $e)
{
    error_log('Error in '.$e->getFile().
        ' Line: '.$e->getLine().
        ' Error: '.$e->getMessage()
    );
    $display = $reg_messages['reset_error'];
}
?>
```

The HTML of the Reset Password page looks like this:

`newpass.php` (excerpt)

```
<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        : HTML Head contents...
    </head>
    <body>
```

```

<h1><?php echo $display['title']; ?></h1>
<?php echo $display['content']; ?>
</body>
</html>

```

Figure 10.4 shows the page's display.

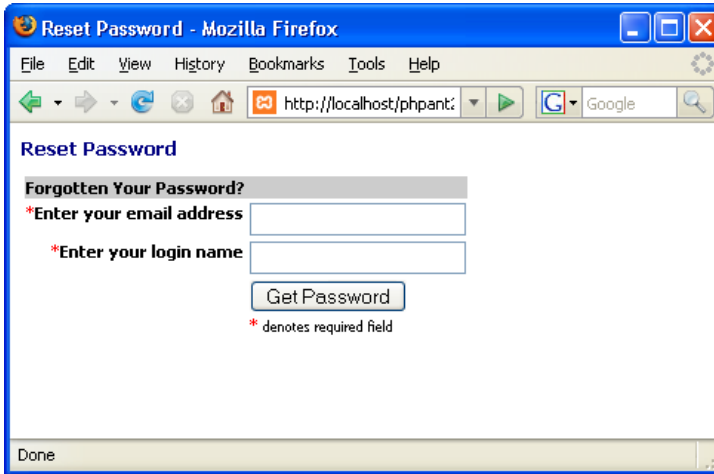


Figure 10.4. The Reset Password page

You can add a link to the bottom of your login form so that the user is able to access the Reset Password page. Here's an example:

```

<a href="newpass.php">Forgotten your password?</a>

```

How do I let users change their passwords?

A good design test for many PHP applications is whether users can change their passwords without needing to log back into the application afterwards. Provided you construct your application carefully, your users should be able to go about their business without further ado after changing their passwords. It's important to be considerate to your site's users if you want them to stick around!

Solution

If we return for a minute to the session-based authentication mechanism we discussed earlier in this chapter, you'll remember that the login and md5 encrypted password are stored in session variables and rechecked on every new page by the Auth class. The trick is to change the value of the password in both the session variable and the database when users change their passwords. We can perform this trick with a small modification to the AccountMaintenance class—found in “How do I deal with members who forget their passwords?”—and the addition of a new form.

Modifying AccountMaintenance

With a little tweaking of the AccountMaintenance class to add a method for changing passwords, we should be able to handle the job fairly easily. The changePassword method requires an instance of the Auth class (found in “How do I create a class to control access to a section of the site?”), the old password, and the new password as arguments:

AccountMaintenance.class.php (excerpt)

```
public function changePassword($auth, $oldPassword, $newPassword)
{
    $var_login = $this->cfg['login_vars']['login'];
    $user_table = $this->cfg['users_table']['table'];
    $user_login = $this->cfg['users_table']['col_login'];
    $user_pass = $this->cfg['users_table']['col_password'];
```

At the beginning of the method, we store some of the configuration settings in local variables to help the readability of the rest of the method.

The method then instantiates a new Session object (which we saw in “How do I create a session class?”) and attempts to find the user record in the database:

AccountMaintenance.class.php (excerpt)

```
$session = new Session();
try
{
    $sql = "SELECT *
          FROM " . $user_table . "
```

```

        WHERE
        " . $user_login . " = :login
        AND
        " . $user_pass . " = :pass";
    $stmt = $this->db->prepare($sql);
    $stmt->bindParam(':login', $session->get($var_login));
    $stmt->bindParam(':pass', md5($oldPassword));
    $stmt->execute();
    $result = $stmt->fetchAll(PDO::FETCH_ASSOC);
}
catch (PDOException $e)
{
    throw new AccountDatabaseException('Database error when ' .
        ' finding user: ' . $e->getMessage());
}

```

The method first performs a database lookup to find the record of the user who's using the current login details—obtained from the session information—and the old password. If a `PDOException` is thrown, the method throws one of our custom exceptions, `AccountDatabaseException`.

The results of the database lookup are checked—if anything but a single matching record is returned, the method will throw an `AccountUnknownException`:

AccountMaintenance.class.php (excerpt)

```

if (count($result) != 1)
{
    throw new AccountUnknownException('Could not find account');
}

```

Finally, if no exceptions have been thrown, the method updates the password information in the database with the new password:

AccountMaintenance.class.php (excerpt)

```

try
{
    $sql = "UPDATE " . $user_table . "
        SET
        " . $user_pass . " = :pass

```



```

        WHERE
            " . $user_login . " = :login";
$stmt = $this->db->prepare($sql);
$stmt->bindParam(':login', $session->get($var_login));
$stmt->bindParam(':pass', md5($newPassword));
$stmt->execute();
$auth->storeAuth($session->get($var_login),
    md5($newPassword));
    }
    catch (PDOException $e)
    {
        throw new AccountDatabaseException('Database error when' .
            ' updating password: '.$e->getMessage());
    }
}

```

After we update the information in the user table, the current session information is also updated via the `Auth->storeAuth` method. Again, if the operation throws a `PDOException`, we throw an `AccountDatabaseException`.

It's a good idea to ask the user to enter the old password before changing it over and giving them access with a new one. Perhaps the user logged in at an Internet café and then left, forgetting to log out, or worse, his or her session was hijacked electronically. The process of ascertaining that the user can provide the old password can preclude some of the potential for damage, as it prevents anyone who “takes over” the session from being able to change the password and thus assume total control. Instead, the newcomer's only logged in as long as the session continues. (You may also wish to ask a user to reenter the password before completing any major actions—like making a credit card purchase—for this very reason.)

The Change Password Form

This web page form will show you how the `changePassword` method can easily be used in your registration system. We start by including all the classes and other files we'll need:

`changepass.php` (excerpt)

```

<?php
error_reporting(E_ALL);
require_once 'Session.class.php';

```

```
require_once 'Auth.class.php';
require_once 'AccountMaintenance.class.php';
require_once 'HTML/QuickForm.php';
require_once 'dbcred.php';
```

We set the error reporting level to `E_ALL` with the `error_reporting` function, as we're using PEAR packages, which will cause `E_Strict` errors under PHP 5. We then include our custom classes for session, authorization, and account management, the `PEAR::HTML_QuickForm` package, and our database credentials file.

Next, we set the `$reg_messages` array to hold the page content for the different form outcomes:

changepass.php (excerpt)

```
$reg_messages = array(
    'success' => array(
        'title' => 'Password Changed',
        'content' => '<p>Your password has been changed ' .
            ' successfully.</p>'
    ),
    'no_account' => array(
        'title' => 'Account Problem',
        'content' => '<p>We could not find your account.<br />' .
            'Please contact the site administrators.</p>'
    ),
    'change_error' => array(
        'title' => 'Change Password Problem',
        'content' => '<p>There was an error changing your ' .
            ' password. Please contact the site administrators,' .
            ' or click ' .
            '<a href="' . $_SERVER['PHP_SELF'] . '">here</a> to ' .
            ' try again.</p>'
    )
);
```

We then test to find out whether the user is currently authorized to see the Change Password form, with the assistance of the `Auth` class:

changepass.php (excerpt)

```

try
{
    $db = new PDO($dsn, $user, $password);
    $db->setAttribute(PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION);
    $auth = new Auth($db, 'login.php', 'secret');

```

At this point, we open a try block; we want to catch any exceptions that may be thrown from the execution of the rest of the code. Catching any exceptions from this point will allow us to display an appropriate message on the web page instead of a PHP error.

We instantiate the PDO and Auth classes; if the user isn't authorized, he or she will be redirected to the login form. And if all's well, we start building the Change Password form with `PEAR::HTML_QuickForm`:

changepass.php (excerpt)

```

$form = new HTML_QuickForm('changePass', 'POST');

function cmpPass($element, $confirm)
{
    $password = $GLOBALS['form']->getElementValue('newPassword');
    return $password == $confirm;
}
$form->registerRule('compare', 'function', 'cmpPass');

```

After instantiating the `HTML_QuickForm` object, we define and register the function `cmpPass` that will be used to validate the password fields, to ensure that the password and password confirmation fields match.

Then we add the form:

changepass.php (excerpt)

```

$form->addElement('header', 'MyHeader', 'Change your password');

// Add a field for the old password
$form->addElement('password', 'oldPassword',

```

```

        'Current Password');
$form->addRule('oldPassword', 'Enter your current password',
    'required', false, 'client');

// Add a field for the new password
$form->addElement('password', 'newPassword', 'New Password');
$form->addRule('newPassword', 'Please provide a password',
    'required', false, 'client');
$form->addRule('newPassword',
    'Password must be at least 6 characters',
    'minlength', 6, 'client');
$form->addRule('newPassword',
    'Password cannot be more than 12 chars',
    'maxlength', 50, 'client');
$form->addRule('newPassword',
    'Password can only contain letters and ' .
    'numbers', 'alphanumeric', NULL, 'client');

// Add a field for password confirmation
$form->addElement('password', 'confirm', 'Confirm Password');
$form->addRule('confirm', 'Please confirm your password',
    'required', false, 'client');
$form->addRule('confirm', 'Your passwords do not match',
    'compare', false, 'client');

// Add a submit button
$form->addElement('submit', 'submit', 'Change Password');

```

If the form has been submitted, we can attempt to change the password:

changepass.php (excerpt)

```

if ($form->validate())
{
    $aMaint = new AccountMaintenance($db);
    $aMaint->changePassword($auth,
        $form->getSubmitValue('oldPassword'),
        $form->getSubmitValue('newPassword')
    );
    $display = $reg_messages['success'];
}

```

On validation of the form, we instantiate an `AccountMaintenance` object and call the `changePassword` method. If no exceptions are thrown, we set the `$display` variable to the success message.

If the form has not yet been submitted and validated, we display the form contents:

`changepass.php` (excerpt)

```
else
{
    // If not submitted, display the form
    $display = array(
        'title' => 'Change Password',
        'content' => $form->toHtml()
    );
}
}
```

The final task of our main script is to catch any possible exceptions and display appropriate page content:

`changepass.php` (excerpt)

```
catch (AccountUnknownException $e)
{
    $display = $reg_messages['no_account'];
}
catch (Exception $e)
{
    error_log('Error in '.$e->getFile().
        ' Line: '.$e->getLine().
        ' Error: '.$e->getMessage()
    );
    $display = $reg_messages['change_error'];
}
?>
```

The HTML content of the Change Password page is as follows:

changepass.php (excerpt)

```

<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    : HTML Head contents...
  </head>
  <body>
    <h1><?php echo $display['title']; ?></h1>
    <?php echo $display['content']; ?>
  </body>
</html>

```

Finally, the new Change Password page can be seen in Figure 10.5.

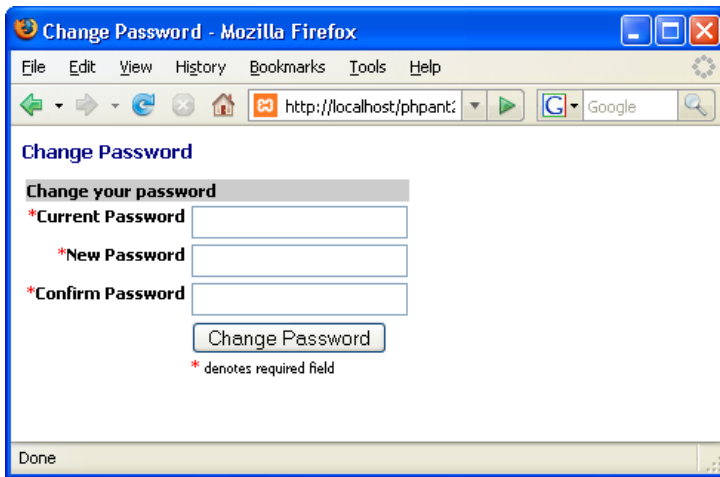


Figure 10.5. The new Change Password page

Discussion

Now that you know how to allow users to change their passwords, it should be no problem for you to change other account settings, such as the first and last names and the signature—simply add the details to the `AccountMaintenance` class. If you want to allow users to change their email addresses, you’ll need to examine the registration procedure used earlier in “How do I build a registration system?”, and modify the `SignUp` class. You should make sure that users confirm a new email address before you allow them to change it.

How to do I build a permissions system?

In the previous sections, we built an authentication system that provided global security for your web site. But, consider this: are all the members of your site equal? You probably don't want all of your users to have access to edit and delete articles, for example. To deal with this issue, you need to add to the security system further functionality that allows you to assign permissions to groups of members, permitting only these users to perform specific actions.

Rather than assign permissions to single accounts, which would quickly become a nightmare to administer, we'll build a permissions system in terms of *users*, *groups*, and *permissions*. Users (login accounts) will be assigned to groups, which will have names like Administrators, Authors, Managers, and so on. Permissions reflect actions that users will be allowed to perform within the site, and they will also be assigned to groups. >From an administration perspective, this system will be easy to manage, as it'll be a simple matter to see which Permissions a particular group has, and which users are assigned to that group.

This kind of access control is known as **role-based access control**. If you'd like to read more on the theory of role-based access control, the web site of the US Government National Institute of Standards and Technology has a complete section on it.¹¹

Solution

Let's leap in and build our permission system.

Setting Up the Database

Building the permissions system initially requires the construction of many-to-many relationships between database tables. This is explained as follows:

- A user can belong to many groups.
- A group may have many users.
- A permission can be assigned to many groups.
- A group may have many permissions.

¹¹ <http://csrc.nist.gov/rbac/>

In practical terms, the way to build many-to-many relationships in MySQL is to use a **bridge table**, which relates to two other tables. The bridge table stores a two-column index, each column being the key of one of the two related tables. For example, we have a user table and a collection table in our database. Here's the SQL for those tables:

access_control.sql (excerpt)

```
CREATE TABLE user (
  user_id      INT(11)      NOT NULL AUTO_INCREMENT,
  login        VARCHAR(50) NOT NULL DEFAULT '',
  password     VARCHAR(50) NOT NULL DEFAULT '',
  email        VARCHAR(50) DEFAULT NULL,
  firstName    VARCHAR(50) DEFAULT NULL,
  lastName     VARCHAR(50) DEFAULT NULL,
  signature    TEXT         NOT NULL,
  PRIMARY KEY (user_id),
  UNIQUE KEY user_login (login)
);

CREATE TABLE collection (
  collection_id INT(11)      NOT NULL auto_increment,
  name          VARCHAR(50) NOT NULL default '',
  description   TEXT         NOT NULL,
  PRIMARY KEY (collection_id)
);
```

Each user has a unique ID and login name, and several other pieces of information associated with his or her record. Each group has a unique ID, a name, and a description. We'll use a bridge table to link users to their groups, and groups to their users. Here's the definition of the user2collection lookup table:

access_control.sql (excerpt)

```
CREATE TABLE user2collection (
  user_id      INT(11)      NOT NULL default '0',
  collection_id INT(11)      NOT NULL default '0',
  PRIMARY KEY (user_id, collection_id)
);
```


Notice that the primary key for the table uses *both* columns: this ensures that no combination of `user_id` and `collection_id` can appear more than once.



Be Aware of Reserved Words

I use “collection” to refer to “group” in MySQL. “Group” is a reserved word in SQL, so it shouldn’t be used as a table name. Technically, it *can* be used with the proper quoting, but why run the risk of confusing ourselves—and possibly MySQL? You can find more about SQL reserved words at the MySQL web site.¹²

Here’s some hypothetical data that shows how the bridge table can be used:

```
mysql> select * from user2collection;
+-----+-----+
| user_id | collection_id |
+-----+-----+
|      1 |             1 |
|      2 |             1 |
|      2 |             2 |
|      3 |             1 |
|      4 |             1 |
+-----+-----+
5 rows in set (0.00 sec)
```

This data tells us that user 1 is a member of group 1, user 2 is a member of groups 1 and 2, user 3 is a member of group 1, and so on.

We’ll also need a permission table for the purpose of keeping track of permissions:

`access_control.sql` (excerpt)

```
CREATE TABLE permission (
  permission_id INT(11) NOT NULL AUTO_INCREMENT,
  name          VARCHAR(50) NOT NULL DEFAULT '',
  description   TEXT      NOT NULL,
  PRIMARY KEY (permission_id)
);
```

¹² <http://dev.mysql.com/doc/refman/4.1/en/reserved-words.html>

Each permission has a unique ID, a name, and a description. Permission names will represent actions; view, create, edit and delete, for example. We'll need a bridge table to link groups to permissions—here's the `collection2permission` table:

`access_control.sql` (excerpt)

```
CREATE TABLE collection2permission (
  collection_id INT(11) NOT NULL DEFAULT '0',
  permission_id INT(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (collection_id, permission_id)
);
```

With the lookup tables defined, we can now perform queries across the tables to identify the permissions a particular user has been allowed. For example, the following query returns all the permission names for the user with `user_id` 1:

```
SELECT p.name as permission
FROM
  user2collection uc,
  INNER JOIN collection2permission cp
    ON uc.collection_id = cp.collection_id
  INNER JOIN permission p
    ON cp.collection_id = p.collection_id
WHERE uc.user_id = 1;
```

Note that I've used aliases for table names, such as `user2collection uc`, to make writing the query easier.

If you've downloaded and installed the sample `access_control` database mentioned in the introduction to this chapter, you'll find it contains three sample user accounts with the details shown in Table 10.1.

Table 10.1. Sample User Accounts

Login	Password	Group
jackblack	password	Users
jackwhite	password	Editors
siteadmin	password	Administrators

The `access_control` database also contains three sample groups, as shown in Table 10.2.

Table 10.2. Sample Groups

Group	Permissions
Users	view
Editors	view, create, edit
Administrators	view, create, edit, delete

The User Class

The `User` class will encapsulate all the functionality for checking a user's permissions. Our class uses the following configuration settings:

`access_control.ini` (excerpt)

```
; Access Control Settings

;web form variables e.g. $_POST['login']
[login_vars]
login=login

;user login table details
[users_table]
table=user
col_id=user_id
col_login=login
col_password=password
col_email=email
col_name_first=firstName
col_name_last=lastName
col_signature=signature

;Permission table details
[permission_table]
table=signup
col_id=permission_id
col_name=name

;Collection table details
[collection_table]
table=collection
```

```

col_id=collection_id
col_name=name

;User to Collection lookup table details
[user_to_collection_table]
table=user2collection
col_id=user_id
col_collection_id=collection_id

;Collection to Permission lookup table details
[collection_to_permission_table]
table=collection2permission
col_id=collection_id
col_permission_id=permission_id

```

We define some custom exception classes to provide a consistent level of error handling:

User.class.php (excerpt)

```

class UserException extends Exception
{
    public function __construct($message = null, $code = 0)
    {
        parent::__construct($message, $code);
        error_log('Error in '.$this->getFile().
            ' Line: '.$this->getLine().
            ' Error: '.$this->getMessage()
        );
    }
}
class UserDatabaseException extends UserException {}

```

Our base class, `UserException`, is a custom exception that ensures the exception details are logged using the `error_log` function. The subclass `UserDatabaseException` represents a database problem. If you were to add further functionality to the `User` class, you could create further custom exceptions based on the `UserException` class to cover all possible exception situations.

We begin to create the class by defining some class properties:

User.class.php (excerpt)

```
class User
{
    protected $db;
    protected $cfg;
    protected $userId;
    protected $firstName;
    protected $lastName;
    protected $email;
    protected $permissions;
```

`$db` will contain a PDO instance for our database connection, `$cfg` will store our configuration details, and the remaining properties will contain information from the user's account details.

The constructor takes an instance of the PDO class, loads the configuration file, and calls the `populate` method:

User.class.php (excerpt)

```
public function __construct(PDO $db)
{
    $this->db = $db;
    $this->cfg = parse_ini_file('access_control.ini', TRUE);
    $this->populate();
}
```

Next comes the `populate` method:

User.class.php (excerpt)

```
private function populate()
{
    $var_login = $this->cfg['login_vars']['login'];
    $user_table = $this->cfg['users_table']['table'];
    $user_id = $this->cfg['users_table']['col_id'];
    $user_login = $this->cfg['users_table']['col_login'];
    $user_email = $this->cfg['users_table']['col_email'];
    $user_first = $this->cfg['users_table']['col_name_first'];
    $user_last = $this->cfg['users_table']['col_name_last'];
```

We load some configuration values into local variables to aid the readability of the code.

Next, we attempt to look up the user's details in the database:

User.class.php (excerpt)

```
$session = new Session();
try
{
    $sql = "SELECT
        " . $user_id . ", " . $user_email . ",
        " . $user_first . ", " . $user_last . "
    FROM
        " . $user_table . "
    WHERE
        " . $user_login . " = :login";
    $stmt = $this->db->prepare($sql);
    $login = $session->get($var_login);
    $stmt->bindParam(':login', $login);
    $stmt->execute();
    $row = $stmt->fetch(PDO::FETCH_ASSOC);
}
catch(PDOException $e)
{
    throw new UserDatabaseException('Database error when ' .
        'populating user details: '.$e->getMessage());
}
```

We first need to instantiate a new session object (which we built in “How do I create a session class?”). The session login variable is then used as the key to find the user's details in the user table. If a `PDOException` is thrown, we throw our custom `UserDatabaseException`.

Once we've retrieved the user's record from the database, we store all the detail in the `User` object properties:

User.class.php (excerpt)

```

$this->userId = $row[$user_id];
$this->email = $row[$user_email];
$this->firstName = $row[$user_first];
$this->lastName = $row[$user_last];
}

```

Populate pulls this user’s record from the database and stores various useful pieces of information from that record in the object’s variables so that we can access them easily; for example, when we want to display the user’s name on the page. The most important aspect is to gather the `user_id` value from the database, for the purpose of checking permissions.

We also add a few **accessor methods**. Accessor methods allow public access to otherwise protected object properties—they allow the properties to be read without granting public access to users of the class to write to them:

User.class.php (excerpt)

```

public function getId()
{
    return $this->userId;
}

public function getFirstName()
{
    return $this->firstName;
}

public function getLastName()
{
    return $this->lastName;
}

public function getEmail()
{
    return $this->email;
}

```

Finally, we add the `checkPermission` method. This method takes a named permission as an argument and checks that the user has that permission:

User.class.php (excerpt)

```

public function checkPermission($permission)
{
    if (!isset($this->permissions))
    {
        $perm_table = $this->cfg['permission_table']['table'];
        $perm_id = $this->cfg['permission_table']['col_id'];
        $perm_name = $this->cfg['permission_table']['col_name'];
        $u2c_table = $this->cfg['user_to_collection_table']['table'];
        $u2c_id = $this->cfg['user_to_collection_table']['col_id'];
        $c2p_table = $this->cfg['collection_to_permission_table']
            ➤ ['table'];
        $c2p_id = $this->cfg['collection_to_permission_table']
            ➤ ['col_id'];
        $c2p_pid = $this->cfg['collection_to_permission_table']
            ➤ ['col_permission_id'];
    }
}

```

The first step we take is to check that the permissions array for this user has been set. If not, we proceed with the database lookup. Before we perform the lookup, though, we assign some configuration settings to local variables to help improve our code's readability.

Next, we assemble the SQL query and perform the lookup using the `User->userId` property as the key:

User.class.php (excerpt)

```

try
{
    $this->permissions = array();
    $sql = 'SELECT p.' . $perm_name . ' as perm
        FROM
            ' . $u2c_table . ' uc
        INNER JOIN ' . $c2p_table . ' cp
        ON uc.' . $u2c_id . ' = cp.' . $c2p_id . '
        INNER JOIN ' . $perm_table . ' p
        ON cp.' . $c2p_pid . ' = p.' . $perm_id . '
        WHERE uc.user_id =:user';
    $stmt = $this->db->prepare($sql);
    $stmt->bindParam(':user', $this->userId);
    $stmt->execute();
}

```



```

        while ($row = $stmt->fetch(PDO::FETCH_ASSOC))
        {
            $this->permissions[] = $row['permission'];
        }
    }
    catch(PDOException $e)
    {
        throw new UserDatabaseException('Database error when' .
            ' checking permissions: '.$e->getMessage());
    }
}

```

If the lookup has returned database rows, we store them in object `User->permissions` property array. This means that if we need to check permissions more than once on a page, that check will only come at the cost of a single query. And, as usual, if a `PDOException` is thrown, we in turn throw our custom `UserDatabaseException`.

Finally, we check that the permission passed into the method as an argument in the `$permission` variable is included in the user's permissions array:

User.class.php (excerpt)

```

    if (in_array($permission, $this->permissions))
    {
        return true;
    }
    else
    {
        return false;
    }
}
}

```

The `checkPermission` simply returns true if the user has the permission, and false if not.

The Permissions Test Page

Now, to test our permissions system, we can build a permissions testing page. This testing page will require you to log in using the details of one of the accounts in the

user table, and will simulate an attempt to access one of four defined permission levels in the permission table—view, create, edit, and delete.

First, we need to include all the required classes and the database credentials file:

permissions.php (excerpt)

```
<?php
require_once 'Session.class.php';
require_once 'Auth.class.php';
require_once 'User.class.php';
require_once 'dbcred.php';
```

Next, we instantiate our PDO, Auth (which we met in “How do I create a class to control access to a section of the site?”), and User objects:

permissions.php (excerpt)

```
try
{
    $db = new PDO($dsn, $user, $password);
    $auth = new Auth($db, 'login.php', 'secret');
    $authuser = new User($db);
```

The Auth object will make sure the current user is authorized, and redirect them to the login form if not. If the user is authorized, we create a User object in order to be able to check the user’s permissions.

We’re simulating permissions through a query string variable—\$_GET['view']:

permissions.php (excerpt)

```
switch (@$_GET['view']) {
    case 'create':
        $permission = 'create';
        $msg = 'You are able to create new content.';
        break;
    case 'edit':
        $permission = 'edit';
        $msg = 'You are able to edit existing content.';
        break;
    case 'delete':
```

```

    $permission = 'delete';
    $msg = 'You are able to delete existing content.';
    break;
default:
    $permission = 'view';
    $msg = 'You are able to read existing content.';
}

```

We set the permission level and the `$msg` variable—the message that appears on the page—to reflect the value of `$_GET['view']`.

Next, we test the user’s permissions:

permissions.php (excerpt)

```

if (!$authuser->checkPermission($permission)) {
    $msg = 'You do not have permission to do this.';
}

```

If the user doesn’t have the required permission, we take appropriate action. Since this demonstration is merely a test, we simply set the page message to indicate that the user does not have the required permission level. In a production web application, you’d redirect the user to the login form, adding a message to indicate that they’re not authorized to obtain that level of access.

Finally, we make sure to catch any exceptions and take appropriate action:

permissions.php (excerpt)

```

}
catch (Exception $e)
{
    $msg = 'An error has occurred: ' . $e->getMessage();
}
?>

```

The only task left is to create the HTML for our permissions testing page:

```

<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    : HTML Head contents...
  </head>
  <body>
    <h1>Permissions Test</h1>
    <p>
      <a href="<?php echo $_SERVER['PHP_SELF']; ?>">View</a> |
      <a href="<?php echo $_SERVER['PHP_SELF'];
        ?>?view=create">Create</a> |
      <a href="<?php echo $_SERVER['PHP_SELF'];
        ?>?view=edit">Edit</a> |
      <a href="<?php echo $_SERVER['PHP_SELF'];
        ?>?view=delete">Delete</a>
    </p>
    <h2><?php echo $authuser->getFirstName() . ' ' .
      $authuser->getLastName(); ?></h2>
    <p>Permission Level: '<?php echo $permission ?>'</p>
    <p><?php echo $msg; ?></p>
  </body>
</html>

```

The testing page is very simple. First, we have a menu of links that test each permission level by appending the appropriate query string to the link URLs. Then, we have a simple page body that consists of the current user’s name, the current permission level, and the message set by the permissions test.

Discussion

The `User` class fetches data on a “need to know” basis. That is, despite the fact that some user data is retrieved on instantiation using the `populate` method, the data pertaining to permissions may not be needed every time the `User` class is instantiated. It’s likely that we’ll only check permissions on a restricted number of pages, so we can save ourselves a database query when the user views public pages, and leave the `checkPermission` method to be called only when needed. This approach of only fetching data from the database at the moment it is needed—as opposed to performing all the queries at the beginning—is known as **lazy fetching**, and can be a useful approach to reducing unnecessary queries and performance overhead.

The permissions testing page was a simple example, of course, but you could use the `checkPermission` method any way you like—perhaps within `if` statements to decide what a user is allowed to do and see. Another approach would be to use a variable, such as the `$msg` variable we’ve used here, to store the name of a PHP script, which contained the restricted content, for use with an `include` statement.

Otherwise, that’s all there is to it. Now, all you need to do is build an administration interface to control Users, Groups, and Permissions. Well, what are you waiting for?

How do I store sessions in a database?

As discussed earlier, in “How do I use sessions?”, the default behavior of sessions in PHP on the server side is to create a temporary file in which session data is stored. This file is usually kept in the temporary directory of the operating system and, as such, presents a security risk to your applications, especially if you’re using a shared server.

Solution

Use the PHP function `session_set_save_handler` to specify a custom session handler that provides an alternative data store that’s fully under your control. The `session_set_save_handler` function definition is as follows:

```
bool session_set_save_handler (callback $open,
    callback $close,
    callback $read,
    callback $write,
    callback $destroy,
    callback $gc
);
```

Each callback argument is a function that must conform to the PHP session’s API. You can read more about the function on The PHP Manual page.¹³ You can simply implement a separate function for each callback; however, in this solution we create a new class—the `DatabaseSession` class—to encapsulate all our session handling needs, and use a PDO object to connect to a database and store session information there.

¹³ http://www.php.net/session_set_save_handler/

Before we delve deep into the details of the class, I'll show you the `create` statement for the session table we use. This statement provides a minimal amount of information for you to keep track of, so feel free to add more if you wish—for example, you might like to store the IP address or the last page visited. Just remember to add the new columns and values to the queries that are used throughout the class's methods below:

```
CREATE TABLE session (
  sess_id      VARCHAR(255),
  sess_start   DATETIME,
  sess_last_acc DATETIME,
  sess_data    VARCHAR(255),
  PRIMARY KEY (sess_id)
);
```

The DatabaseSession Class

Now, let's look at the class. We begin by defining the class properties:

DatabaseSession.class.php (excerpt)

```
class DatabaseSession
{
  private $sess_table;
  private $sess_db;
  private $sess_db_host;
  private $sess_db_usr;
  private $sess_db_pass;
  private $db;
```

`$sess_table` will store the database table name, `$sess_db` will store the database name, `$sess_db_host` will store the database server hostname, `$sess_db_usr` will store the database username, and `$sess_db_pass` will store the database password. The `$db` property will store the PDO object used for all the database queries.

Next, we define the constructor method:

DatabaseSession.class.php (excerpt)

```

public function __construct($sess_db_usr = 'user',
    $sess_db_pass = 'passwd',
    $sess_table = 'session',
    $sess_db = 'dbname',
    $sess_db_host = 'localhost')
{
    $this->sess_db_usr = $sess_db_usr;
    $this->sess_db_pass = $sess_db_pass;
    $this->sess_table = $sess_table;
    $this->sess_db = $sess_db;
    $this->sess_db_host = $sess_db_host;
}

```

The constructor simply stores the database information passed to the method within the object's properties.

The first function callback that we must pass to the `session_set_save_handler` function is an open function, which is called when a session is started. The open method of the DatabaseSession class will handle that job:

DatabaseSession.class.php (excerpt)

```

public function open($path, $name)
{
    try
    {
        $dsn = "mysql:host={$this->sess_db_host};".
            "dbname={$this->sess_db}";
        $this->db = new PDO($dsn, $this->sess_db_usr,
            $this->sess_db_pass );
        $this->db->setAttribute(PDO::ATTR_ERRMODE,
            PDO::ERRMODE_EXCEPTION);
    }
    catch (PDOException $e)
    {
        error_log('Error connecting to the session database. ');
        error_log('Reason given: '.$e->getMessage()."\n");
        return false;
    }
    return true;
}

```

This method is called with two string arguments—the path of the session file and the name of the file—and must return either true or false. The path and filename information is irrelevant to us as we’re using a database, so we do nothing with it. In the method, we make the connection to the database that will hold the session data. If there’s an error, we return false; if the database connection is successful, we return true.

The next function callback we need to implement is the `close` function, so we add a `close` method to our class:

DatabaseSession.class.php (excerpt)

```
public function close()
{
    $this->db = null;
    return true;
}
```

The `close` method is called when we end a session, and must return either true or false. It isn’t uncommon to manually call the garbage collection (`gc`) method here, though it isn’t strictly necessary—PHP will do its own garbage collection throughout. We remove our database connection by setting the `close` method to `null`.

`session_set_save_handler` also requires that a `read` function be implemented. The `read` function needs to take the session ID as an argument and return a string—even an empty one, if that’s appropriate. We implement a `read` method in our class:

DatabaseSession.class.php (excerpt)

```
public function read($sess_id)
{
    try
    {
        $sql = "SELECT sess_data FROM {$this->sess_table} WHERE " .
            "sess_id = :id";
        $stmt = $this->db->prepare($sql);
        $stmt->execute(array(':id'=>$sess_id));
        $res = $stmt->fetchAll(PDO::FETCH_ASSOC);
    }
    catch (PDOException $e)
```



```

{
    error_log('Error reading the session data table in the ' .
        ' session reading method.');
```

```

    error_log(' Query with error: ' . $sql);
    error_log(' Reason given: ' . $e->getMessage() . "\n");
    return '';
}
if (count($res) > 0)
{
    return isset($res[0]['sess_data']) ?
        $res[0]['sess_data'] : '';
}
else
{
    return '';
}
}

```

The read method retrieves the session data from the database, using the session ID as the key, and returns the data as a string. If no data is found or there's a database error, an empty string is returned.

After the read function, the next function callback we need to implement is the write function. This function, as the name implies, handles the writing of the session data. The function is required to take two arguments—the session ID and the session data—and the return value must be either true or false. We implement a write method in our class-based solution. In our method, we first see if the session ID is already in the database:

DatabaseSession.class.php (excerpt)

```

public function write($sess_id, $data)
{
    try
    {
        $sql = "SELECT sess_data FROM {$this->sess_table} WHERE " .
            "sess_id = :id";
        $stmt = $this->db->prepare($sql);
        $stmt->execute(array(':id'=>$sess_id));
        $res = $stmt->fetchAll(PDO::FETCH_ASSOC);
    }
    catch (PDOException $e)

```

```

{
    error_log('Error reading the session data table in the' .
        ' session writing method.');
```

error_log(' Query with error: '.\$sql);

error_log(' Reason given:'.\$e->getMessage()."\n");

return false;

```

}
```

The `$res` variable contains the result of our database lookup. Based upon this result, we either update the existing session record with an SQL UPDATE query or insert a new one with an SQL INSERT query:

DatabaseSession.class.php (excerpt)

```

try
{
    if (count($res) > 0)
    {
        $sql = "UPDATE {$this->sess_table} SET" .
            " sess_last_acc = NOW(), sess_data = :data" .
            " WHERE sess_id = :id";
        $stmt = $this->db->prepare($sql);
        $stmt->bindParam(':data', $data);
        $stmt->bindParam(':id', $sess_id);

    }
    else
    {
        $sql = "INSERT INTO {$this->sess_table}(sess_id," .
            " sess_start, sess_last_acc," .
            " sess_data) VALUES (:id, NOW(), NOW(), :data)";
        $stmt = $this->db->prepare($sql);
        $stmt->bindParam(':id', $sess_id);
        $stmt->bindParam(':data', $data);
    }
    $res = $stmt->execute();
}
}
```

If you know you'll only be using MySQL as your database, consider using the `REPLACE` syntax instead.¹⁴ Since we don't want to limit our class to MySQL, we use the longer but more compatible method above.

Finally, we need to catch any `PDOException`s and return true or false:

`DatabaseSession.class.php` (excerpt)

```
catch (PDOException $e)
{
    error_log('Error writing to the session data table.');
```

```
    error_log('Query with error: '.$sql);
    error_log('Reason given: '.$e->getMessage()."\n");
    return false;
}
return true;
}
```

Our next task is to implement a `destroy` function, which, as the name suggests, is called when the session is destroyed. It receives the session ID as an argument and must return either true or false. In our class method `destroy`, we simply delete the session from the database using the session ID as the key, and return false if an error occurs or true if the operation succeeds:

`DatabaseSession.class.php` (excerpt)

```
public function destroy($sess_id)
{
    try
    {
        $sql = "DELETE FROM {$this->sess_table} WHERE sess_id = :id";
        $stmt = $this->db->prepare($sql);
        $stmt->execute(array(':id'=>$sess_id));
    }
    catch (PDOException $e)
    {
        error_log('Error destroying the session.');
```

```
        error_log('Query with error: '.$sql);
    }
}
```

¹⁴ `REPLACE` is a MySQL extension to the SQL standard that either inserts a new row, or deletes an old row and inserts the new row if the old row had the same value as the new row for a `PRIMARY KEY` or `UNIQUE` index. You can read more about it at <http://dev.mysql.com/doc/refman/5.1/en/replace.html>.

```

        error_log('Reason given: '.$e->errorMessage()."\n");
        return false;
    }
    return true;
}

```

The final function we are required to implement is the `gc`, or garbage collection, function, which is used to clean out any old sessions that were never closed properly. It receives an integer argument for the “time to live” (TTL) value for a session. In our class method, `gc`, we delete any session record where the last access time is less than the current time, minus the TTL value:

DatabaseSession.class.php (excerpt)

```

public function gc($ttl)
{
    $end = time() - $ttl;
    try
    {
        $sql = "DELETE FROM {$this->sess_table} WHERE " .
            " sess_last_acc <:end";
        $stmt = $this->db->prepare($sql);
        $stmt->execute(array(':id'=>$end));
    }
    catch (PDOException $e)
    {
        error_log('Error with the garbage collection method of the ' .
            ' session class. ');
        error_log('Query with error: '.$sql);
        error_log('Reason given: '.$e->getMessage());
        return false;
    }
    return true;
}

```

The garbage collection method is called by PHP as dictated by the `php.ini` settings `session.gc_probability` and `session.gc_divisor`, and is checked every time a new session is started. Again, you can call it manually in the session `close` method if you wish.



MySQL MyISAM Engine Performance

If your session table sees high rates of insertions and deletions, you should consider adding an `OPTIMIZE TABLE` query to the garbage collection function to regain memory and help increase performance. For more information on `OPTIMIZE TABLE`, see the MySQL manual.¹⁵

Finally, we implement a class `__destruct` method. This step is necessitated by the changes that were made in how PHP sessions are closed after version 5.0.5. Basically, we just have to make sure the session is explicitly written and closed by calling the `session_write_close` function. You can read more about this task on the manual page.¹⁶ Here's our `__destruct` method and the end of our class definition:

DatabaseSession.class.php (excerpt)

```
public function __destruct()
{
    session_write_close();
}
}
```

Using the DatabaseSession Class

Here's a simple script to test our new DatabaseSession class:

dbsession.php (excerpt)

```
<?php
require_once 'DatabaseSession.class.php';

$session = new DatabaseSession('user', 'secret', 'session',
    'access_control', 'localhost');
session_set_save_handler(array($session, 'open'),
    array($session, 'close'),
    array($session, 'read'),
    array($session, 'write'),
    array($session, 'destroy'),
    array($session, 'gc')
);
```

¹⁵ <http://dev.mysql.com/doc/refman/5.1/en/optimize-table.html>

¹⁶ http://www.php.net/session_set_save_handler/

```

session_start();

$name = (isset($_SESSION['name']))? $_SESSION['name'] : '';

if ($name !== '')
{
    echo 'Welcome ', $name, ' to your session!';
}
else
{
    echo 'Lets start the session!';
    $_SESSION['name'] = 'PHP';
}
?>

```

We include our `DatabaseSession` class, then instantiate the `DatabaseSession` object. Next, we use `session_set_save_handler` to register our custom PHP session-handling methods. Then we have a quick little demonstration to show us that the session is working—the first time you load the web page you should see the message “Let’s start the session!” We then set the `$_SESSION['name']` to PHP. When you refresh the web page, the message should change to “Welcome PHP to your session!” which indicates that our session data is being stored and retrieved correctly in the database.

Welcome to database-saved sessions!

Summary

In this chapter we’ve investigated HTTP authentication and PHP sessions, and created a complete access control system that can manage user registrations, password resets, and changes, including authorization, groups, and multiple permission levels.

Phew! Well, there you have it—total access control over your site! Now you have the power to bark “Denied” at those that shouldn’t be in restricted areas, and roll out the red carpet for those that should. Can you feel the warm glow of power gathering within you? Will you use it for good—or evil? Either way, I hope you’ve enjoyed it and learned a bit along the way.

Chapter 11

Caching

In the good old days when building web sites was as easy as knocking up a few HTML pages, the delivery of a web page to a browser was a simple matter of having the web server fetch a file. A site's visitors would see its small, text-only pages almost immediately, unless they were using particularly slow modems. Once the page was downloaded, the browser would cache it somewhere on the local computer so that, should the page be requested again, after performing a quick check with the server to ensure the page hadn't been updated, the browser could display the locally cached version. Pages were served as quickly and efficiently as possible, and everyone was happy.

Then dynamic web pages came along and spoiled the party by introducing two problems:

- When a request for a dynamic web page is received by the server, some intermediate processing must be completed, such as the execution of scripts by the PHP engine. This processing introduces a delay before the web server begins to deliver the output to the browser. This may not be a significant delay where simple PHP scripts are concerned, but for a more complex application, the PHP engine may have a lot of work to do before the page is finally ready for delivery. This extra

work results in a noticeable time lag between the user's requests and the actual display of pages in the browser.

- A typical web server, such as Apache, uses the time of file modification to inform a web browser of a requested page's age, allowing the browser to take appropriate caching action. With dynamic web pages, the actual PHP script may change only occasionally; meanwhile, the content it displays, which is often fetched from a database, will change frequently. The web server has no way of discerning updates to the database, so it doesn't send a last modified date. If the client (that is, the user's browser) has no indication of how long the data will remain valid, it will take a guess. This is problematic if the browser decides to use a locally cached version of the page which is now out of date, or if the browser decides to request from the server a fresh copy of the page, which actually has no new content, making the request redundant. The web server will always respond with a freshly constructed version of the page, regardless of whether or not the data in the database has actually changed.

To avoid the possibility of a web site visitor viewing out-of-date content, most web developers use a meta tag or HTTP headers to tell the browser never to use a cached version of the page. However, this negates the web browser's natural ability to cache web pages, and entails some serious disadvantages. For example, the content delivered by a dynamic page may only change once a day, so there's certainly a benefit to be gained by having the browser cache a page—even if only for 24 hours.

If you're working with a small PHP application, it's usually possible to live with both issues. But as your site increases in complexity—and attracts more traffic—you'll begin to run into performance problems. Both these issues can be solved, however: the first with server-side caching; the second, by taking control of client-side caching from within your application. The exact approach you use to solve these problems will depend on your application, but in this chapter, we'll consider both PHP and a number of class libraries from PEAR as possible panaceas for your web page woes.

Note that in this chapter's discussions of caching, we'll look at only those solutions that can be implemented in PHP. For a more general introduction, the definitive

discussion of web caching is represented by Mark Nottingham's tutorial.¹ Furthermore, the solutions in this chapter should not be confused with some of the script caching solutions that work on the basis of optimizing and caching compiled PHP scripts, such as Zend Accelerator² and ionCube PHP Accelerator.³

How do I prevent web browsers from caching a page?

If timely information is crucial to your web site and you wish to prevent out-of-date content from ever being visible, you need to understand how to prevent web browsers—and proxy servers—from caching pages in the first place.

Solutions

There are two possible approaches we could take to solving this problem: using HTML meta tags, and using HTTP headers.

Using HTML Meta Tags

The most basic approach to the prevention of page caching is one that utilizes HTML meta tags:

```
<meta http-equiv="expires" content="Mon, 26 Jul 1997 05:00:00 GMT" />
<meta http-equiv="pragma" content="no-cache" />
```

The insertion of a date that's already passed into the Expires meta tag tells the browser that the cached copy of the page is always out of date. Upon encountering this tag, the browser usually won't cache the page. Although the Pragma: no-cache meta tag isn't guaranteed, it's a fairly well-supported convention that most web browsers follow. However, the two issues associated with this approach, which we'll discuss below, may prompt you to look at the alternative solution.

Using HTTP Headers

A better approach is to use the HTTP protocol itself, with the help of PHP's header function, to produce the equivalent of the two HTML meta tags above:

¹ http://www.mnot.net/cache_docs/

² <http://www.zend.com/>

³ <http://www.php-accelerator.co.uk/>

```
<?php
  header('Expires: Mon, 26 Jul 1997 05:00:00 GMT');
  header('Pragma: no-cache');
?>
```

We can go one step further than this, using the `Cache-Control` header that's supported by HTTP 1.1-capable browsers:

```
<?php
  header('Expires: Mon, 26 Jul 1997 05:00:00 GMT');
  header('Cache-Control: no-store, no-cache, must-revalidate');
  header('Cache-Control: post-check=0, pre-check=0', FALSE);
  header('Pragma: no-cache');
?>
```

For a precise description of HTTP 1.1 `Cache-Control` headers, have a look at the W3C's HTTP 1.1 RFC.⁴ Another great source of information about HTTP headers, which can be applied readily to PHP, is `mod_perl`'s documentation on issuing correct headers.⁵

Discussion

Using the `Expires` meta tag sounds like a good approach, but two problems are associated with it:

- The browser first has to download the page in order to read the meta tags. If a tag wasn't present when the page was first requested by a browser, the browser will remain blissfully ignorant and keep its cached copy of the original.
- Proxy servers that cache web pages, such as those common to ISPs, generally won't read the HTML documents themselves. A web browser might know that it shouldn't cache the page, but the proxy server between the browser and the web server probably doesn't—it will continue to deliver the same out-of-date page to the client.

On the other hand, using the HTTP protocol to prevent page caching essentially guarantees that no web browser or intervening proxy server will cache the page, so

⁴ <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9>

⁵ http://perl.apache.org/docs/general/correct_headers/correct_headers.html

visitors will always receive the latest content. In fact, the first header should accomplish this on its own; this is the best way to ensure a page is not cached. The `Cache-Control` and `Pragma` headers are added for some degree of insurance. Although they don't work on all browsers or proxies, the `Cache-Control` and `Pragma` headers will catch some cases in which the `Expires` header doesn't work as intended—if the client computer's date is set incorrectly, for example.

Of course, to disallow caching entirely introduces the problems we discussed at the start of this chapter: it negates the web browser's natural ability to cache pages, and can create unnecessary overhead, as new versions of pages are always requested, even though those pages may not have been updated since the browser's last request. We'll look at the solution to these issues in just a moment.

How do I control client-side caching?

We addressed the task of disabling client-side caching in “How do I prevent web browsers from caching a page?”, but disabling the cache is rarely the only (or best) option.

Here we'll look at a mechanism that allows us to take advantage of client-side caches in a way that can be controlled from within a PHP script.



Apache Required!

This approach will only work if you're running PHP as an Apache web server module, because it requires use of the function `getallheaders`—which only works with Apache—to fetch the HTTP headers sent by a web browser.

Solutions

In controlling client-side caching you have two alternatives. You can set a date on which the page will expire, or respond to the browser's request headers. Let's see how each of these tactics is executed.

Setting a Page Expiry Header

The header that's easiest to implement is the `Expires` header—we use it to set a date on which the page will expire, and until that time, web browsers are allowed to use a cached version of the page. Here's an example of this header at work:

```
<?php
function setExpires($expires) {
    header(
        'Expires: ' . gmdate('D, d M Y H:i:s', time()+$expires) . 'GMT');
    }
setExpires(10);
echo ( 'This page will self destruct in 10 seconds<br />' );
echo ( 'The GMT is now ' . gmdate('H:i:s') . '<br />' );
echo ( '<a href="' . $_SERVER['PHP_SELF'] . '">View Again</a><br />' );
?>
```

In this example, we created a custom function called `setExpires` that sets the HTTP Expires header to a point in the future, defined in seconds. The output of the above example shows the current time in GMT, and provides a link that allows us to view the page again. If we follow this link, we'll notice the time updates only once every ten seconds. If you like, you can also experiment by using your browser's Refresh button to tell the browser to refresh the cache, and watching what happens to the displayed date.

Acting on the Browser's Request Headers

A more useful approach to client-side cache control is to make use of the Last-Modified and If-Modified-Since headers, both of which are available in HTTP 1.0. This action is known technically as performing a conditional GET request; whether your script returns any content depends on the value of the incoming If-Modified-Since request header.

If you use PHP version 4.3.0 and above on Apache, the HTTP headers are accessible with the functions `apache_request_headers` and `apache_response_headers`. Note that the function `getAllheaders` has become an alias for the new `apache_request_headers` function.

This approach requires that you send a Last-Modified header every time your PHP script is accessed. The next time the browser requests the page, it sends an If-Modified-Since header containing a time; your script can then identify whether the page has been updated since that time. If it hasn't, your script sends an HTTP 304 status code to indicate that the page hasn't been modified, and exits before sending the body of the page.

Let's see these headers in action. The example below uses the modification date of a text file. To simulate updates, we first need to create a way to randomly write to the file:

ifmodified.php (excerpt)

```
<?php
$file = 'ifmodified.txt';
$random = array (0,1,1);
shuffle($random);
if ( $random[0] == 0 ) {
    $fp = fopen($file, 'w');
    fwrite($fp, 'x');
    fclose($fp);
}
$lastModified = filemtime($file);
```

Our simple randomizer provides a one-in-three chance that the file will be updated each time the page is requested. We also use the `filemtime` function to obtain the last modified time of the file.

Next, we send a `Last-Modified` header that uses the modification time of the text file. We need to send this header for every page we render, to cause visiting browsers to send us the `If-Modified-Since` header upon every request:

ifmodified.php (excerpt)

```
header('Last-Modified: ' .
    gmdate('D, d M Y H:i:s', $lastModified) . ' GMT');
```

Our use of the `getallheaders` function ensures that PHP gives us all the incoming request headers as an array. We then need to check that the `If-Modified-Since` header actually exists; if it does, we have to deal with a special case caused by older Mozilla browsers (earlier than version 6), which appended an illegal extra field to their `If-Modified-Since` headers. We use PHP's `strtotime` function to generate a timestamp from the date the browser sent us. If there's no such header, we set this timestamp to zero, which forces PHP to give the visitor an up-to-date copy of the page:

ifmodified.php (excerpt)

```

$request = getallheaders();
if (isset($request['If-Modified-Since']))
{
    $modifiedSince = explode(';', $request['If-Modified-Since']);
    $modifiedSince = strtotime($modifiedSince[0]);
}
else
{
    $modifiedSince = 0;
}

```

Finally, we check to see whether or not the cache has been modified since the last time the visitor received this page. If it hasn't, we simply send a 304 Not Modified response header and exit the script, saving bandwidth and processing time by prompting the browser to display its cached copy of the page:

ifmodified.php (excerpt)

```

if ($lastModified <= $modifiedSince)
{
    header('HTTP/1.1 304 Not Modified');
    exit();
}

echo ( 'The GMT is now ' . gmdate('H:i:s') . '<br />' );
echo ( '<a href="' . $_SERVER['PHP_SELF'] . '">View Again</a><br />' );
?>

```

Remember to use the “View Again” link when you run this example (clicking the Refresh button usually clears your browser's cache). If you click on the link repeatedly, the cache will eventually be updated; your browser will throw out its cached version and fetch a new page from the server.

If you combine the Last-Modified header approach with time values that are already available in your application—for example, the time of the most recent news article—you should be able to take advantage of web browser caches, saving bandwidth and improving your application's perceived performance in the process.

Be very careful to test any caching performed in this manner, though; if you get it wrong, you may cause your visitors to consistently see out-of-date copies of your site.

Discussion

HTTP dates are always calculated relative to Greenwich Mean Time (GMT). The PHP function `gmdate` is exactly the same as the `date` function, except that it automatically offsets the time to GMT based on your server's system clock and regional settings.

When a browser encounters an `Expires` header, it caches the page. All further requests for the page that are made before the specified expiry time use the cached version of the page—no request is sent to the web server. Of course, client-side caching is only truly effective if the system time on the computer is accurate. If the computer's time is out of sync with that of the web server, you run the risk of pages either being cached improperly, or never being updated.

The `Expires` header has the advantage that it's easy to implement; in most cases, however, unless you're a highly organized person, you won't know exactly when a given page on your site will be updated. Since the browser will only contact the server after the page has expired, there's no way to tell browsers that the page they've cached is out of date. In addition, you also lose some knowledge of the traffic visiting your web site, since the browser will not make contact with the server when it requests a page that's been cached.

How do I examine HTTP headers in my browser?

How can you actually check that your application is running as expected, or debug your code, if you can't actually see the HTTP headers? It's worth knowing exactly which headers your script is sending, particularly when you're dealing with HTTP cache headers.

Solution

Several worthy tools are available to help you get a closer look at your HTTP headers:

LiveHTTPHeaders (<http://livehttpheaders.mozdev.org/>)

This add-on to the Firefox browser is a simple but very handy tool for examining request and response headers while you're browsing.

Firebug (<http://getfirebug.org/>)

Another useful Firefox add-on, Firebug is a tool whose interface offers a dedicated tab for examining HTTP request information.

HTTPWatch (<http://www.httpwatch.com/>)

This add-on to Internet Explorer for HTTP viewing and debugging is similar to LiveHTTPHeaders above.

Charles Web Debugging Proxy (<http://getcharles.com/>)

Available for Windows, Mac OS X, and Linux or Unix, the Charles Web Debugging Proxy is a proxy server that allows developers to see all the HTTP traffic between their browsers and the web servers to which they connect.

Any of these tools will allow you to inspect the communication between the server and browser.

How do I cache file downloads with Internet Explorer?

If you're developing file download scripts for Internet Explorer users, you might notice a few issues with the download process. In particular, when you're serving a file download through a PHP script that uses headers such as `Content-Disposition: attachment, filename=myFile.pdf` or `Content-Disposition: inline, filename=myFile.pdf`, and that tells the browser not to cache pages, Internet Explorer won't deliver that file to the user.

Solutions

Internet Explorer handles downloads in a rather unusual manner: it makes two requests to the web site. The first request downloads the file and stores it in the cache before making a second request, the response to which is not stored. The second request invokes the process of delivering the file to the end user in accordance with the file's type—for instance, it starts Acrobat Reader if the file is a PDF document. Therefore, if you send the cache headers that instruct the browser not to cache the

page, Internet Explorer will delete the file between the first and second requests, with the unfortunate result that the end user receives nothing!

If the file you're serving through the PHP script won't change, one solution to this problem is simply to disable the “don't cache” headers, `pragma` and `cache-control`, which we discussed in “How do I prevent web browsers from caching a page?”, for the download script.

If the file download will change regularly, and you want the browser to download an up-to-date version of it, you'll need to use the `Last-Modified` header that we met in “How do I control client-side caching?”, and ensure that the time of modification remains the same across the two consecutive requests. You should be able to achieve this goal without affecting users of browsers that handle downloads correctly.

One final solution is to write the file to the file system of your web server and simply provide a link to it, leaving it to the web server to report the cache headers for you. Of course, this may not be a viable option if the file is supposed to be secured.

How do I use output buffering for server-side caching?

Server-side processing delay is one of the biggest bugbears of dynamic web pages. We can reduce server-side delay by caching output. The page is generated normally, performing database queries and so on with PHP; however, before sending it to the browser, we capture and store the finished page somewhere—in a file, for instance. The next time the page is requested, the PHP script first checks to see whether a cached version of the page exists. If it does, the script sends the cached version straight to the browser, avoiding the delay involved in rebuilding the page.

Solution

Here, we'll look at PHP's in-built caching mechanism, the output buffer, which can be used with whatever page rendering system you prefer (templates or no templates). Consider situations in which your script displays results using, for example, `echo` or `print`, rather than sending the data directly to the browser. In such cases, you can use PHP's output control functions to store the data in an in-memory buffer, which your PHP script has both access to and control over.

Here's a simple example that demonstrates how the output buffer works:

buffer.php (excerpt)

```
<?php
ob_start();
echo '1. Place this in the buffer<br />';
$buffer = ob_get_contents();
ob_end_clean();
echo '2. A normal echo<br />';
echo $buffer;
?>
```

The buffer itself stores the output as a string. So, in the above script, we commence buffering with the `ob_start` function, and use `echo` to display a piece of text which is stored in the output buffer automatically. We then use the `ob_get_contents` function to fetch the data the `echo` statement placed in the buffer, and store it in the `$buffer` variable. The `ob_end_clean` function stops the output buffer and empties the contents; the alternative approach is to use the `ob_end_flush` function, which displays the contents of the buffer.

The above script displays the following output:

```
2. A normal echo
1. Place this in the buffer
```

In other words, we captured the output of the first `echo`, then sent it to the browser after the second `echo`. As this simple example suggests, output buffering can be a very powerful tool when it comes to building your site; it provides a solution for caching, as we'll see in a moment, and is also an excellent way to hide errors from your site's visitors, as is discussed in Chapter 9. Output buffering even provides a possible alternative to browser redirection in situations such as user authentication.

In order to improve the performance of our site, we can store the output buffer contents in a file. We can then call on this file for the next request, rather than having to rebuild the output from scratch again. Let's look at a quick example of this technique. First, our example script checks for the presence of a cache file:

sscache.php (excerpt)

```
<?php
if (file_exists('./cache/page.cache'))
{
    readfile('./cache/page.cache');
    exit();
}
```

If the script finds the cache file, we simply output its contents and we're done!

If the cache file is not found, we proceed to output the page using the output buffer:

sscache.php (excerpt)

```
ob_start();
?>
<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Cached Page</title>
  </head>
  <body>
    This page was cached with PHP's
    <a href="http://www.php.net/outcontrol"
      >Output Control Functions</a>
  </body>
</html>
<?php
$buffer = ob_get_contents();
ob_end_flush();
```

Before we flush the output buffer to display our page, we make sure to store the buffer contents in the `$buffer` variable.

The final step is to store the saved buffer contents in a text file:

sscache.php (excerpt)

```
$fp = fopen('./cache/page.cache','w');
fwrite($fp,$buffer);
fclose($fp);
?>
```

The `page.cache` file contents are exactly same as the HTML that was rendered by the script:

cache/page.cache (excerpt)

```
<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Cached Page</title>
  </head>
  <body>
    This page was cached with PHP's
    <a href="http://www.php.net/outputcontrol"
    >Output Control Functions</a>
  </body>
</html>
```

Discussion

For an example that shows how to use PHP's output buffering capabilities to handle errors more elegantly, have a look at the *PHP Freaks* article “Introduction to Output Buffering,” by Derek Ford.⁶

What About Template Caching?

Template engines often include template caching features—Smarty is a case in point.⁷ Usually, these engines offer a built-in mechanism for storing a compiled version of a template (that is, the native PHP generated from the template), which prevents us developers from having to recompile the template every time a page is requested.

⁶ <http://www.phpfreaks.com/tutorials/59/0.php>

⁷ <http://smarty.php.net/>

This process should not be confused with output—or content—caching, which refers to the caching of the rendered HTML (or other output) that PHP sends to the browser. In addition to the content cache mechanisms discussed in this chapter, Smarty can cache the contents of the HTML page. Whether you use Smarty’s content cache or one of the alternatives discussed in this chapter, you can successfully use both template and content caching together on the same site.

HTTP Headers and Output Buffering

Output buffering can help solve the most common problem associated with the `header` function, not to mention the issues surrounding `session_start` and `set_cookie`. Normally, if you call any of these functions after page output has begun, you’ll get a nasty error message. When output buffering’s turned on, the only output types that can escape the buffer are HTTP headers. If you use `ob_start` at the very beginning of your application’s execution, you can send headers at whichever point you like, without encountering the usual errors. You can then write out the buffered page content all at once, when you’re sure that no more HTTP headers are required.



Use Output Buffering Responsibly

While output buffering can helpfully solve all our `header` problems, it should not be used solely for that reason. By ensuring that all output is generated after all the headers are sent, you’ll save the time and resource overheads involved in using output buffers.

How do I cache just the parts of a page that change infrequently?

Caching an entire page is a simplistic approach to output buffering. While it’s easy to implement, that approach negates the real benefits presented by PHP’s output control functions to improve your site’s performance in a manner that’s relevant to the varying lifetimes of your content.

No doubt, some parts of the page that you send to visitors will change very rarely, such as the page’s header, menus, and footer. But other parts—for example, the list of comments on your blog posts—may change quite often. Fortunately, PHP allows you to cache sections of the page separately.

Solution

Output buffering can be used to cache sections of a page in separate files. The page can then be rebuilt for output from these files.

This technique eliminates the need to repeat database queries, while loops, and so on. You might consider assigning each block of the page an expiry date after which the cache file is recreated; alternatively, you may build into your application a mechanism that deletes the cache file every time the content it stores is changed.

Let's work through an example that demonstrates the principle. Firstly, we'll create two helper functions, `writeCache` and `readCache`. Here's the `writeCache` function:

smartcache.php (excerpt)

```
<?php
function writeCache($content, $filename)
{
    $fp = fopen('./cache/' . $filename, 'w');
    fwrite($fp, $content);
    fclose($fp);
}
```

The `writeCache` function is quite simple; it just writes the content of the first argument to a file with the name specified in the second argument, and saves that file to a location in the **cache** directory. We'll use this function to write our HTML to the cache files.

The `readCache` function will return the contents of the cache file specified in the first argument if it has not expired—that is, the file's last modified time is not older than the current time minus the number of seconds specified in the second argument. If it has expired or the file does not exist, the function returns false:

smartcache.php (excerpt)

```
function readCache($filename, $expiry)
{
    if (file_exists('./cache/' . $filename))
    {
        if ((time() - $expiry) > filemtime('./cache/' . $filename))
        {
```

```

        return false;
    }
    $cache = file('./cache/' . $filename);
    return implode('', $cache);
}
return false;
}

```

For the purposes of demonstrating this concept, I've used a procedural approach. However, I wouldn't recommend doing this in practice, as it will result in very messy code and is likely to cause issues with file locking. For example, what happens when someone accesses the cache at the exact moment it's being updated? Better solutions will be explained later on in the chapter.

Let's continue this example. After the output buffer is started, processing begins. First, the script calls `readCache` to see whether the file header `.cache` exists; this contains the top of the page—the HTML `<head>` tag and the start `<body>` tag. We've used PHP's `date` function to display the time at which the page was actually rendered, so you'll be able to see the different cache files at work when the page is displayed:

`smartcache.php` (excerpt)

```

ob_start();
if (!$header = readCache('header.cache', 604800))
{
?>
<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Chunked Cached Page</title>
    <meta http-equiv="Content-Type"
      content="text/html; charset=iso-8859-1"/>
  </head>
  <body>
    <p>The header time is now: <?php echo date('H:i:s'); ?></p>
<?php
  $header = ob_get_contents();

```

```

    ob_clean();
    writeCache($header, 'header.cache');
}

```

Note what happens when a cache file isn't found: the header content is output and assigned to a variable, `$header`, with `ob_get_contents`, after which the `ob_clean` function is called to empty the buffer. This allows us to capture the output in "chunks" and assign them to individual cache files with the `writeCache` function. The header of the page is now stored as a file, which can be reused without our needing to re-render the page. Look back to the start of the `if` condition for a moment. When we called `readCache`, we gave it an expiry time of 604800 seconds (one week); `readCache` uses the file modification time of the cache file to determine whether the cache is still valid.

For the body of the page, we'll use the same process as before. However, this time, when we call `readCache`, we'll use an expiry time of five seconds; the cache file will be updated whenever it's more than five seconds old:

smartcache.php (excerpt)

```

if (!$body = readCache('body.cache', 5))
{
    echo 'The body time is now: ' . date('H:i:s') . '<br />';
    $body = ob_get_contents();
    ob_clean();
    writeCache($body, 'body.cache');
}

```

The page footer is effectively the same as the header. After the footer, the output buffering is stopped and the contents of the three variables that hold the page data are displayed:

smartcache.php (excerpt)

```

if (!$footer = readCache('footer.cache', 604800)) {
?>
    <p>The footer time is now: <?php echo date('H:i:s'); ?></p>
</body>
</html>

```



```

<?php
    $footer = ob_get_contents();
    ob_clean();
    writeCache($footer, 'footer.cache');
}
ob_end_clean();

echo $header . $body . $footer;
?>

```

The end result looks like this:

```

The header time is now: 17:10:42
The body time is now: 18:07:40
The footer time is now: 17:10:42

```

The header and footer are updated on a weekly basis, while the body is updated whenever it is more than five seconds old. If you keep refreshing the page, you'll see the body time updating.

Discussion

Note that if you have a page that builds content dynamically, based on a number of variables, you'll need to make adjustments to the way you handle your cache files. For example, you might have an online shopping catalog whose listing pages are defined by a URL such as:

<http://example.com/catalogue/view.php?category=1&page=2>

This URL should show page two of all items in category one; let's say this is the category for socks. But if we were to use the caching code above, the results of the first page of the first category we looked at would be cached, and shown for any request for any other page or category, until the cache expiry time elapsed. This would certainly confuse the next visitor who wanted to browse the category for shoes—that person would see the cached content for socks!

To avoid this issue, you'll need to incorporate the category ID and page number in to the cache file name like so:

```

$cache_filename = 'catalogue_' . $category_id . '_' .
    $page . '.cache';
if (!$catalogue = readCache($cache_filename, 604800))
{
    : display the category HTML...
}

```

This way, the correct cached content can be retrieved for every request.



Nesting Buffers

You can nest one buffer within another practically *ad infinitum* simply by calling `ob_start` more than once. This can be useful if you have multiple operations that use the output buffer, such as one that catches the PHP error messages, and another that deals with caching. Care needs to be taken to make sure that `ob_end_flush` or `ob_end_clean` is called every time `ob_start` is used.

How do I use `PEAR::Cache_Lite` for server-side caching?

The previous solution explored the ideas behind output buffering using the PHP `ob_*` functions. Although we mentioned at the time, that approach probably isn't the best way to meet to dual goals of keeping your code maintainable and having a reliable caching mechanism. It's time to see how we can put a caching system into action in a manner that will be reliable and easy to maintain.

Solution

In the interests of keeping your code maintainable and having a reliable caching mechanism, it's a good idea to delegate the responsibility of caching logic to classes you trust. In this case, we'll use a little help from `PEAR::Cache_Lite` (version 1.7.2 is used in the examples here).⁸ `Cache_Lite` provides a solid yet easy-to-use library for caching, and handles issues such as: file locking; creating, checking for, and deleting cache files; controlling the output buffer; and directly caching the results from function and class method calls. More to the point, `Cache_Lite` should be rel-

⁸ http://pear.php.net/package/Cache_Lite/

atively easy to apply to an existing application, requiring only minor code modifications.

`Cache_Lite` has four main classes. First is the base class, `Cache_Lite`, which deals purely with creating and fetching cache files, but makes no use of output buffering. This class can be used alone for caching operations in which you have no need for output buffering, such as storing the contents of a template you've parsed with PHP.

The examples here will not use `Cache_Lite` directly, but will instead focus on the three subclasses. `Cache_Lite_Function` can be used to call a function or class method and cache the result, which might prove useful for storing a MySQL query result set, for example. The `Cache_Lite_Output` class uses PHP's output control functions to catch the output generated by your script and store it in cache files; it allows you to perform tasks such as those we completed in "How do I cache just the parts of a page that change infrequently?". The `Cache_Lite_File` class bases cache expiry on the timestamp of a master file, with any cache file being deemed to have expired if it is older than the timestamp.

Let's work through an example that shows how you might use `Cache_Lite` to create a simple caching solution. When we're instantiating any child classes of `Cache_Lite`, we must first provide an array of options that determine the behavior of `Cache_Lite` itself. We'll look at these options in detail in a moment. Note that the `cacheDir` directory we specify must be one to which the script has read and write access:

`cachelite.php` (excerpt)

```
<?php
require_once 'Cache/Lite/Output.php';
$options = array(
    'cacheDir' => './cache/',
    'writeControl' => 'true',
    'readControl' => 'true',
    'fileNameProtection' => false,
    'readControlType' => 'md5'
);
$cache = new Cache_Lite_Output($options);
```

For each chunk of content that we want to cache, we need to set a lifetime (in seconds) for which the cache should live before it's refreshed. Next, we use the `start` method, available only in the `Cache_Lite_Output` class, to turn on output

buffering. The two arguments passed to the `start` method are an identifying value for this particular cache file, and a cache group. The group is an identifier that allows a collection of cache files to be acted upon; it's possible to delete all cache files in a given group, for example (more on this in a moment). The `start` method will check to see if a valid cache file is available and, if so, it will begin outputting the cache contents. If a cache file is not available, `start` will return false and begin caching the following output.

Once the output for this chunk has finished, we use the `end` method to stop buffering and store the content as a file:

cachelite.php (excerpt)

```
$cache->setLifeTime(604800);

if (!$cache->start('header', 'Static')) {
?>
<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>PEAR::Cache_Lite example</title>
  <meta http-equiv="Content-Type"
    content="text/html; charset=iso-8859-1"/>
</head>
<body>
  <h2>PEAR::Cache_Lite example</h2>
  <p>The header time is now: <?php echo date('H:i:s'); ?></p>
<?php
  $cache->end();
}
```

To cache the body and footer, we follow the same procedure we used for the header. Note that, again, we specify a five-second lifetime when caching the body:

cachelite.php (excerpt)

```
$cache->setLifeTime(5);
if (!$cache->start('body', 'Dynamic')) {
  echo 'The body time is now: ' . date('H:i:s') . '<br />';
  $cache->end();
}
```

```

$cache->setLifeTime(604800);
if (!$cache->start('footer', 'Static')) {
?>
    <p>The footer time is now: <?php echo date('H:i:s'); ?></p>
    </body>
</html>
<?php
    $cache->end();
}
?>

```

On viewing the page, `Cache_Lite` creates cache files in the cache directory. Because we've set the `fileNameProtection` option to `false`, `Cache_Lite` creates the files with these names:

- `./cache/cache_Static_header`
- `./cache/cache_Dynamic_body`
- `./cache/cache_Static_footer`

You can read about the `fileNameProtection` option—and many more—in “What configuration options does `Cache_Lite` support?”. When the same page is requested later, the code above will use the cached file if it is valid and has not expired.



Protect your Cache Files

Make sure that the directory in which you place the cache files is not publicly available, or you may be offering your site's visitors access to more than you realize.

What configuration options does `Cache_Lite` support?

When instantiating `Cache_Lite` (or any of its subclasses, such as `Cache_Lite_Output`), you can use any of a number of approaches to controlling its behavior. These options should be placed in an array and passed to the constructor as shown below (and in the previous section):

```
$options = array(
    'cacheDir' => './cache/',
    'writeControl' => true,
    'readControl' => true,
    'fileNameProtection' => false,
    'readControlType' => 'md5'
);
$cache = new Cache_Lite_Output($options);
```

Solution

The options available in the current version of `Cache_Lite` (1.7.2) are:

cacheDir

This is the directory in which the cache files will be placed. It defaults to `/tmp/`.

caching

This option switches on and off the caching behavior of `Cache_Lite`. If you have numerous `Cache_Lite` calls in your code and want to disable the cache for debugging, for example, this option will be important. The default value is `true` (caching enabled).

lifeTime

This option represents the default lifetime (in seconds) of cache files. It can be changed using the `setLifeTime` method. The default value is `3600` (one hour), and if it's set to `null`, the cache files will never expire.

fileNameProtection

With this option activated, `Cache_Lite` uses an MD5 encryption hash to generate the filename for the cache file. This option protects you from error when you try to use IDs or group names containing characters that aren't valid for filenames; `fileNameProtection` must be turned on when you use `Cache_Lite_Function`. The default is `true` (enabled).

fileLocking

This option is used to switch the file locking mechanisms on and off. The default is `true` (enabled).

writeControl

This option checks that a cache file has been written correctly immediately after it has been created, and throws a `PEAR::Error` if it finds a problem. Obviously, this facility would allow your code to attempt to rewrite a cache file that was created incorrectly, but it comes at a cost in terms of performance. The default value is `true` (enabled).

readControl

This option checks any cache files that are being read to ensure they're not corrupt. `Cache_Lite` is able to place inside the file a value, such as the string length of the file, which can be used to confirm that the cache file isn't corrupt. There are three alternative mechanisms for checking that a file is valid, and they're specified using the `readControlType` option. These mechanisms come at the cost of performance, but should help to guarantee that your visitors aren't seeing scrambled pages. The default value is `true` (enabled).

readControlType

This option lets you specify the type of read control mechanism you want to use. The available mechanisms are a cyclic redundancy check (`crc32`, the default value) using PHP's `crc32` function, an MD5 hash using PHP's `md5` function (`md5`), or a simple and fast string length check (`strlen`). Note that this mechanism is not intended to provide security from people tampering with your cache files; it's just a way to spot corrupt files.

pearErrorMode

This option tells `Cache_Lite` how it should return PEAR errors to the calling script. The default is `CACHE_LITE_ERROR_RETURN`, which means `Cache_Lite` will return a `PEAR::Error` object.

memoryCaching

With memory caching enabled, every time a file is written to the cache, it is stored in an array in `Cache_Lite`. The `saveMemoryCachingState` and `getMemoryCachingState` methods can be used to store and access the memory cache data between requests. The advantage of this facility is that the complete set of cache files can be stored in a single file, reducing the number of disk read/write operations by reconstructing the cache files straight into an array to which your code has access. The `memoryCaching` option may be worth further investigation if you run a large site. The default value is `false` (disabled).

onlyMemoryCaching

If this option is enabled, only the memory caching mechanism will be used. The default value is `false` (disabled).

memoryCachingLimit

This option places a limit on the number of cache files that will be stored in the memory caching array. The more cache files you have, the more memory will be used up by memory caching, so it may be a good idea to enforce a limit that prevents your server from having to work too hard. Of course, this option places no restriction on the size of each cache file, so just one or two massive files may cause a problem. The default value is `1000`.

automaticSerialization

If enabled, this option will automatically serialize all data types. While this approach will slow down the caching system, it is useful for caching nonscalar data types such as objects and arrays. For higher performance, you might consider serializing nonscalar data types yourself. The default value is `false` (disabled).

automaticCleaningFactor

This option will automatically clean old cache entries—on average, one in x cache writes, where x is the value set for this option. Therefore, setting this value to `0` will indicate no automatic cleaning, and a value of `1` will cause cache clearing on every cache write. A value of `20` to `200` is the recommended starting point if you wish to enable this facility; it causes cache cleaning to happen, on average, 0.5% to 5% of the time. The default value is `0` (disabled).

hashedDirectoryLevel

When set to a nonzero value, this option will enable a hashed directory structure. A hashed directory structure will improve the performance of sites that have thousands of cache files. If you choose to use hashed directories, start by setting this value to `1`, and increasing it as you test for performance improvements. The default value is `0` (disabled).

errorHandlingAPIBreak

This option was added to enable backwards compatibility with code that uses the old API. When the old API was run in `CACHE_LITE_ERROR_RETURN` mode (see the `pearErrorMode` option earlier in this list), some functions would return

a Boolean value to indicate success, rather than returning a `PEAR_Error` object. By setting this value to `true`, the `PEAR_Error` object will be returned instead. The default value is `false` (disable).

How do I purge the `Cache_Lite` cache?

The built-in lifetime mechanism for `Cache_Lite` cache files provides a good foundation for keeping your cache files up to date, but there will be some circumstances in which you need the files to be updated immediately.

Solution

In cases in which you need immediate updates, the methods `remove` and `clean` come in handy. The `remove` method is designed to delete a specific cache file; it takes as arguments the cache ID and group name of the file. To delete the page body cache file we created in “How do I use `PEAR::Cache_Lite` for server-side caching?”, we’d use this code:

```
$cache->remove('body', 'Dynamic');
```

If we use the `clean` method, we can delete all the files in our cache directory simply by calling the method with no arguments; alternatively, we can specify a group of cache files to delete. If we wanted to delete both the header and footer cache files we created in “How do I use `PEAR::Cache_Lite` for server-side caching?”, we could do so like this:

```
$cache->clean('Static');
```

Discussion

The `remove` and `clean` methods should obviously be called in response to events that arise within an application. For example, if you have a discussion forum application, you probably want to remove the relevant cache files when a visitor posts a new message.

Although it may seem like this solution entails a lot of code modifications, with some care it can be applied to your application in a global manner. If you have a central script that’s included in every page, your script can simply watch for incoming events—for example, a variable like `$_GET['newPost']`—and respond by deleting

the required cache files. This keeps the cache file removal mechanism central and easier to maintain. You might also consider using the `php.ini` setting `auto_prepend_file` to include this code in every PHP script.

How do I cache function calls?

Many web sites provide access to their data via web services such as SOAP and XML-RPC.⁹ As web services are accessed over a network, it's often a very good idea to cache results so that they can be fetched locally, rather than repeating the same slow request to the server multiple times. A simple approach might be to use PHP sessions, but as that solution operates on a per-visitor basis, the opening requests for each visitor will still be slow.

Solution

Let's assume you wish to create a web page that lists all the SitePoint books available on Amazon. The actual list is not likely to change from moment to moment, so why would we make the request to the Amazon web service every time the web page is displayed? We won't! Instead, we can take advantage of `Cache_Lite` by caching the results of the XML-RPC request.



Requires **PEAR::SOAP Version 0.11.0**

The following solution uses the `PEAR::SOAP` library version 0.11.0 to access the Amazon web service. You can find this package on the PEAR web site.¹⁰

Here's some hypothetical code that fetches the data from the remote Amazon server:

```
$results = $amazonClient->ManufacturerSearchRequest($params);
```

Using `Cache_Lite_Function`, we can cache the results so the data returned from the service can be reused; this will avoid unnecessary network calls and significantly improve performance.

The following example code focuses on the caching aspect to prevent us from getting bogged down in the details of using the Amazon web service. You can see the

⁹ You can read all about web services in Chapter 12.

¹⁰ <http://pear.php.net/package/soap/>

complete script if you download this book's code archive from the SitePoint web site.

The `Cache_Lite_Function` requires the inclusion of the following file:

`cachefunction.php` (excerpt)

```
require_once 'Cache/Lite/Function.php';
```

We instantiate the `Cache_Lite_Function` class with some options:

`cachefunction.php` (excerpt)

```
$options = array(
    'cacheDir' => './cache/',
    'fileNameProtection' => true,
    'writeControl' => true,
    'readControl' => true,
    'readControlType' => 'strlen',
    'defaultGroup' => 'SOAP'
);
$cache = new Cache_Lite_Function($options);
```

It's important that the `fileNameProtection` option is set to `true` (this is in fact the default value, but in this case I've set it manually to emphasize the point). If it were set to `false`, the filename would be invalid, so the data will not be cached.

Here's how we make the calls to our SOAP client class:

`cachefunction.php` (excerpt)

```
$results = $cache->call('amazonClient->ManufacturerSearchRequest',
    $params);
```

If the request is being made for the first time, `Cache_Lite_Function` will store the results as a serialized array or object in a cache file (not that you need to worry about this), and this file will be used for future requests until it expires. The `setLifetime` method can again be used to specify how long the cache files should survive before they're refreshed; currently, the default value of 3600 seconds (one hour) is being used. You can then use the `$results` variable exactly as if you were

calling the web service method directly. The output of our example script can be seen in Figure 11.1.

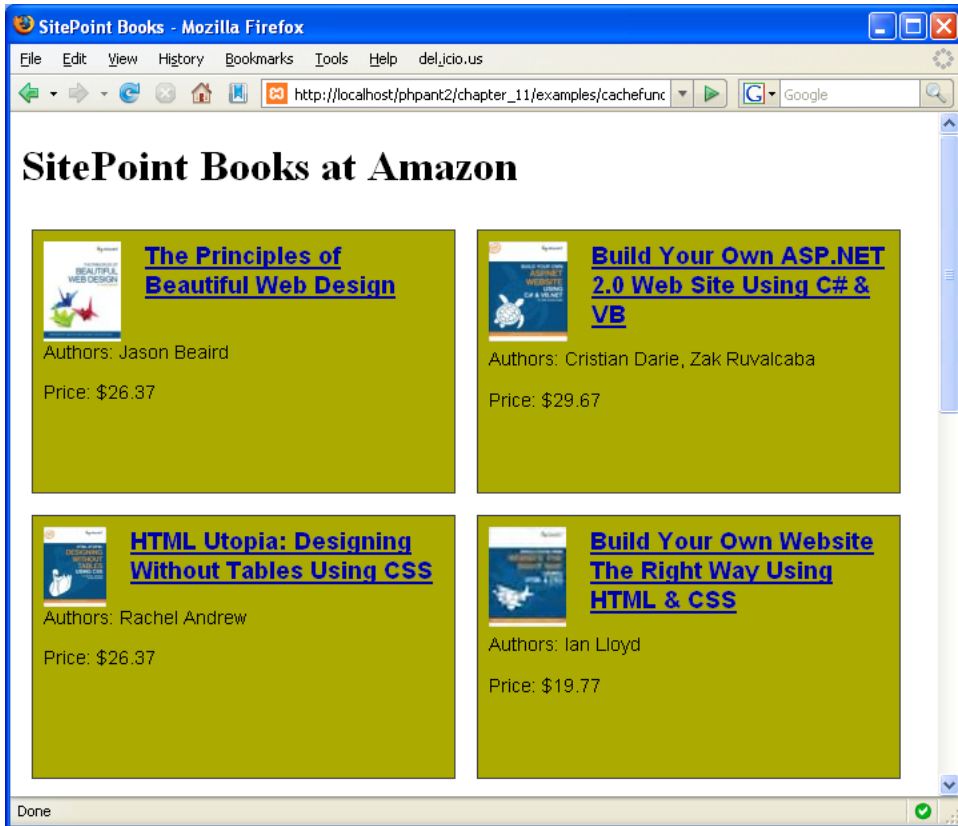


Figure 11.1. SitePoint books at Amazon

Summary

Caching is an important and often overlooked aspect of web site development. Many factors that affect the performance of today's web sites weren't a problem for their predecessors—from complex, dynamic page generation, to a reliance on third-party data over the network. In this chapter, we've examined HTML meta tags, HTTP headers, PHP output buffering and `PEAR::Cache_Lite`, and we've seen how you can use them to control the caching of your web site content and improve the site's reliability and performance.

Implementing a caching system for your site might be simple, but ultimately, it depends on your requirements. If you have a busy and predominantly static web site—such as a blog—that’s managed through a content management system, it will likely require little alteration, yet may benefit from huge performance improvements resulting from a small investment of your time. Setting up caching for a more complex site that generates content on a per-user basis, such as a portal or shopping cart system, will prove a little more tricky and time consuming, but the benefits are still clear. Regardless, I hope the information in this chapter has given you a good grasp of the options available, and will help you determine which techniques are most suitable for your application.

What's Next?

If you've enjoyed these chapters from *The PHP Anthology 101 Essential Tips, Tricks, and Hacks, 2nd Edition*, why not order yourself a copy?

You've only seen a few examples of the most complete question-and-answer book on PHP. It contains over 100 tutorial-style solutions to complex PHP problems using the very latest programming techniques.

Its question-and-answer format lets you learn by example as you work through its impressive PHP solutions, and the copy-and-paste code takes the stress out of getting your web applications off the ground.

You'll get a healthy dose of object oriented PHP as well as learn how to:

- Cut page load times with caching.
- Manage errors gracefully.
- Secure your site with access control systems.
- Create web services with XML.
- Easily work with files, emails and images.
- Build functional forms, tables, and SEO-friendly URLs.
- And so much more!

If you have the drive to progress your skills or and improve your web applications through concepts such as reusable components, caching performance, or web services, then you'll never let this book out of your sight...

[Buy the full version now!](#)



Index

Symbols

- `$_SESSION`, 278, 281
- `$this` variable, 15, 32
- .forward file, 191
- .htaccess file, 473
- .ini files
 - storing configuration information, 164

A

- abstract classes
 - about, 27
- abstract methods
 - about, 28
- AcceptPathInfo
 - "pretty" URLs, 140
- access
 - to cron utility, 485
 - files on remote servers, 166–167
 - to SSH, 484
 - URLs, 495
- access control, 269–362
 - changing passwords, 330–338
 - forgotten passwords, 318–330
 - HTTP authentication, 271–277
 - permission systems, 339–353
 - private sections of web sites, 283–297
 - registration systems, 297–318
 - session classes, 281–282
 - storing sessions in databases, 353–362
 - using sessions, 277–281
- AccountMaintenance class, 319, 331
- adding
 - data in databases, 53–55
- aggregation
 - about, 23
- agile documentation
 - about, 459
- allow_url_fopen, 477
- anti-spam (*see* spam legislation)
- Apache web server
 - caching, 367
 - hosting support, 483
 - HTTP authentication, 271
 - PHP installation, 485
- APIs
 - about, 13
 - callback arguments, 353
 - documenting, 448
 - REST web services, 429
- arguments
 - overriding properties, 21
- arrays
 - of lines, 86–88
 - reading files as, 149
 - strings, 78
- asp_tags, 477
- assertions
 - testing framework, 461
- attachments
 - adding to email messages, 184–186
- Auth class, 283
- authentication
 - (*see also* HTTP authentication)
 - defined, 295
 - security, 494
- authentication headers, 275
- authorization
 - defined, 296

- authorization header, 276
- auto_append_file, 478
- auto_prepend_file, 478
- auto-commit mode
 - default mode, 66
- autoincrementing field
 - determining INSERT's row number, 62–63

B

- back-ups
 - database, 69–75
- bandwidth
 - reading files, 152
- bar graph
 - creating, 224
- batch jobs
 - scheduling, 485
- behavioral testing
 - about, 459
- branches
 - revision control software, 438
- bridge tables
 - about, 340
- browsers (*see* Internet Explorer; web browsers)
- buffering (*see* output buffering)
- build systems
 - developing code, 470

C

- cache files
 - protecting, 385
- Cache_Lite (*see* PEAR::Cache_Lite)
- Cache_Lite cache
 - purging, 389–390

- Cache_Lite_Function class, 391
- caching, 363–393
 - client-side, 367–371
 - examining HTTP headers in web browsers, 371–372
 - file downloads with Internet Explorer, 372–373
 - function calls, 390–392
 - output buffering for server-side caching, 373–377
 - parts of pages, 377–381
 - PEAR::Cache_Lite configuration options, 385–389
 - purging Cache_Lite cache, 389–390
 - using PEAR::Cache_Lite for server-side caching, 382–385
- calendars
 - creating, 102–107
- call_time_pass_reference, 476
- callbacks
 - arguments, 353
 - error handler prototype, 242
- CAPTCHA (Completely Public Turing Test to Tell Computers and Humans)
 - image verification, 234
- CGI mode, 485
- change password form
 - example of, 333
- changesets
 - revision control software, 437
- characters (*see* escape characters; wildcard characters)
- charts
 - displaying, 223–230

- classes
 - (*see also* abstract classes; PEAR; specific classes)
 - about, 10
 - access control for sections of web sites, 283–297
 - repositories of reusable PHP classes, 504
- client-side caching
 - controlling, 367–371
- code
 - (*see also* source code)
 - coding standards, 446
 - deploying, 468–471
 - documenting, 448–453
 - maintaining multiple versions, 438–441
 - reuse, 9
 - revising old code, 467–468
 - testing framework, 454–462
 - tracking revisions, 436–438
 - writing distributable code, 441–448
 - writing portable code, 33–38
- code coverage reports, 463
- command line
 - use of, 485, 486
- communications
 - security, 495
- composition
 - about, 25
- compressing
 - database data, 174
 - files, 172–174
- concatenation
 - strings, 78
- Concurrent Versioning System (CVS), 437
- configuration, 473–481
 - documentation about, 3
 - PEAR::Cache_Lite, 385–389
 - for portable code, 33
 - settings, 286
 - storing configuration information in files, 163–165
- configuration files
 - used in access control examples, 270
- constructors
 - Auth class, 284
 - defined, 14
 - overriding, 22
- content
 - searching for in XML, 409–412
- content-disposition header, 170
- content-length header, 171
- Content-Type header, 170, 198
- control (*see* access control; error handling)
- COUNT function
 - rows returned from a SELECT call, 60
- credentials
 - passing in DSN, 44
- cron utility
 - access to, 485
- cross-site request forgery (CSRF)
 - about, 493
- cross-site scripting (XSS), 83
 - about, 489–491
 - session security, 281
- cryptographic storage
 - security, 495
- CURRENT_TIMESTAMP function, 108

D

data

- compressing in databases, 174
- outputting in table, 127–129, 130–133

data grids

- customized, 134–139

data types

- strings, 77

database connections

- setting up, 311

database servers

- port numbers, 44

database transactions

- defined, 66

databases, 39–75

- accessing, 41–44
- adding or modifying data, 53–55
- backing-up, 69–75
- compressing data, 174
- errors in SQL queries, 49–52
- escape characters, 80
- fetching data from tables, 44–49
- flexible SQL statements, 57–59
- INSERT's row number using autoincrementing field, 62–63
- PDO, 40–41
- permission systems, 339
- rows affected by SQL queries, 59–61
- searching tables, 63–64
- SQL injection attacks, 55–57
- storing sessions, 353–362
- transactions, 65–67

DatabaseSession class, 354, 361

date function, 99, 101, 371

DATE_ADD function, 111

date_default_timezone_set function, 99

DATE_FORMAT function, 109

DATE_SUB function, 111

dates and times, 95–113

(*see also* HTTP dates)

calendars, 102–107

current, 98–99

date calculations using MySQL, 111–112

days of the week, 101

formatting MySQL timestamps, 109–110

number of days in month, 101–102

storing dates in MySQL, 107–109

Unix timestamps, 96–98

days

in a month, 101–102

of week, 101

defaults

error mode, 50

mode in PDO, 66

deploying

code, 468–471

destructors

defined, 14

dir pseudo-class, 161

direct object references

security, 491

directories

examining, 160–161

DirectoryIterator class, 174

display_errors directive, 240, 261, 477

displaying (*see* outputting)

distributed systems

revision control software, 437

docblocks

about, 449

- documentation
 - (*see also* agile documentation)
 - code, 448–453
 - for PHP, 2–9
 - test suites as, 453
- DOM
 - generating XML, 407
- DOM functions
 - navigating XML, 405
- DOM XML extension, 397
- downloads
 - caching files with Internet Explorer, 372–373
- DSN (Data Source Name)
 - about, 43
- dynamic SQL
 - sprintf function, 59
- dynamic web pages
 - caching, 363
- E**
- E_ERROR error level, 240
- E_NOTICE error level, 240
- E_STRICT error level, 173, 181, 240, 293
- E_USER_ERROR error level, 240, 242
- E_USER_NOTICE error level, 240, 241
- E_USER_WARNING error level, 240, 242
- E_WARNING error level, 240
- email, 179–196
 - adding attachments, 184–186
 - email injection attacks, 193–195
 - generating complex emails, 182–184
 - groups, 188–191
 - incoming email, 191–193
 - sending files, 171
 - sending HTML email, 186–188
 - sending simple email, 179–182
- email injection attacks, 193–195
- encapsulation
 - about, 13
- environmental errors
 - defined, 237
- environmental PHP errors
 - handling as exceptions, 260
- ERRMODE_EXCEPTION, 51
- ERRMODE_SILENT, 50
- ERRMODE_WARNING, 50
- error handling, 237–268
 - custom error handlers, 242–247
 - custom exception class, 252–257
 - custom exception handler, 257–260
 - displaying errors and exceptions, 261–265
 - E_STRICT constant, 173
 - error levels reported, 238–240
 - handling as if they were exceptions, 260–261
 - logging and reporting, 247–248
 - redirecting to another page, 265–267
 - security, 493
 - settings, 239–241, 480
 - SQL queries, 49–52
 - triggering errors, 241–242
 - using exceptions for, 248–252
- error notices
 - JpGraph, 224
- error_log, 480
- error_log directive, 241
- error_log function, 263
- error_reporting directive, 239, 477
- escape characters
 - in databases, 80
- exception class, 252–257

- exception classes
 - defining, 300
- exception handlers
 - implementing, 257–260
- exception mode
 - errors in SQL queries, 51
- exceptions
 - displaying, 261–265
 - handling errors as if they were exceptions, 260–261
 - using for error handling, 248–252
- execution
 - settings, 475–479
- EXIF information
 - extracting, 217–220
- exif_read_data function, 218
- Expires header, 371
- Expires meta tag, 366
- explode function, 86
- extension, 481
- extension_dir, 481
- extensions
 - available from hosting service, 486
 - documentation about, 5
 - XML, 396–398
- extracting
 - files, 173

F

- fatal errors
 - handling as exceptions, 260
- features
 - documentation about, 4
- fetchObject method
 - prepare and execute, 49

- fields (*see* auto-incrementing field; form fields)
- file execution attacks, 491
- file handles
 - using, 153–155
- file pointers
 - using, 153
- file_get_contents function, 150
- file_put_contents function, 156
- files, 147–177
 - (*see also* specific files; ZIP utility)
 - accessing information about local files, 157–159
 - accessing on remote servers, 166–167
 - caching downloads with Internet Explorer, 372–373
 - creating compressed ZIP/TAR files, 172–174
 - examining directories, 160–161
 - FTP, 167–169
 - managing downloads, 170–172
 - modifying local files, 155–156
 - outputting source code online, 161–163
 - reading local files, 148–152
 - SPL, 174–177
 - storing configuration information in, 163–165
- fonts (*see* TrueType fonts)
- form fields
 - prepopulating, 80
- formatting
 - dates, 96
 - MySQL timestamps, 109–110
 - output text, 88–90
 - strings, 81–82

forms (*see* HTML forms)

FTP (File Transfer Protocol)

using, 167–169

function calls

caching, 390–392

functions

(*see also* specific functions)

file information, 157

fwrite function, 156

G

galleries (*see* thumbnail galleries)

generating

(*see also* sending)

complex emails, 182–184

GNU Make, 470

graphical watermarks

displaying, 221

graphs

displaying, 223–230

grids (*see* data grids)

groups

email, 188–191

H

handles (*see* file handles; file pointers)

header lines

email injection attacks, 193

headers (*see* authentication headers; au-

thorization header; Expires header;

HTTP headers; request headers;

page expiry headers)

help (*see* documentation)

highlight_file function, 162

highlight_string function, 161

hinting (*see* type hinting)

hints

passwords, 319

hosting

checklist, 483–487

HTML

meta tags, 365

HTML email

sending, 186–188

HTML forms

building, 116–127

HTML tags

stripping from text, 82–83

HTML_QuickForm class, 117

HTML_Quickform package, 309, 325

HTML_Table class, 127

HTML_Table_Matrix class, 102

htmlentities function, 80

HTTP authentication

about, 271–277

HTTP Authentication package, 276

HTTP dates

calculation of, 371

HTTP headers

caching, 365

examining in web browsers, 371–372

file downloads, 170

output buffering, 377

HTTP response headers, 278

http.conf file

hotlinking images, 231

I

ignore_repeated_errors, 480

ignore_repeated_source, 480

imagecopyresampled function, 201

images, 197–236

- charts and graphs, 223–230
 - EXIF information, 217–220
 - hotlinking, 230–234
 - human verification, 234–235
 - MIME type, 198–199
 - resizing, 202–213
 - thumbnail galleries, 214–217
 - thumbnails, 199–202
 - watermarks, 220–223
 - implode function, 87
 - include_path, 478
 - includes
 - settings, 475–479
 - incoming email
 - handling, 191–193
 - information leakage
 - security, 493
 - inheritance
 - about, 17
 - ini_alter, 474
 - ini_set, 474
 - injection flaws, 491
 - INSERT function
 - data into databases, 53
 - determining row number with
 - autoincrementing field, 62–63
 - installation
 - documentation about, 3
 - PEAR, 498–504
 - PHP on Apache web server, 485
 - Zend Framework, 395
 - interfaces
 - (*see also* object interfaces)
 - defined, 27
 - Internet Explorer
 - caching file downloads, 372–373
 - interpolation
 - (*see also* variable interpolation)
 - strings, 77
 - INTERVAL keyword, 111
- ## J
- jobs (*see* batch jobs)
 - JpGraph library, 223
- ## L
- LAMP
 - hosting support, 483
 - levels
 - errors, 238–240
 - LIKE operator
 - searching tables, 63
 - lines
 - arrays of, 86–88
 - Linux
 - dates, 97
 - hosting support, 483
 - session security, 280
 - listInsertId method
 - using, 62
 - local files
 - accessing information about, 157–159
 - modifying, 155–156
 - reading, 148–152
 - localhost
 - connecting to MySQL databases, 41
 - log_errors, 241, 480
 - logging
 - errors, 247–248
 - logic errors
 - defined, 238

login
 magic quotes, 288

M

magic methods
 about, 14

magic quotes
 checking for, 37

magic_quotes_gpc, 288, 476

mail function, 180

Mail_mime package, 309, 325

max_execution_time, 478

MD5 algorithm
 passwords, 286
 security, 495

member variables (*see* properties)

memory_limit, 479

meta tags
 caching, 365

methods
 (*see also* abstract methods; magic methods; static methods)
 about, 11
 overriding, 20

Microsoft Windows (*see* Windows)

MIME type
 specifying, 198–199

mktime function, 97

mod_rewrite
 "pretty" URLs, 141
 hotlinking images, 231

modes (*see* exception mode; silent mode; warning mode)

modifying
 data in databases, 53–55
 local files, 155–156

multi-processing module (MPM)
 hosting support, 484

MultiViews

 "pretty" URLs, 140

MyISAM engine

 performance, 361

MySQL

 calculating dates, 111–112

 MyISAM engine performance, 361

 stored procedure example, 68–69

 storing dates, 107–109

MySQL databases

 connecting to on localhost, 41

MySQL timestamps

 formatting, 109–110

mysql_real_escape_string function, 80

MySQLDump class

 operating system configuration, 70

 using, 72

N

namespaces

 choosing, 445

 defined, 413

nodes

 searching for in XML, 409–412

non-distributed systems

 revision control software, 437

"notice" error messages, 186

NOW function, 108

O

object interfaces

 about, 29

object oriented programming (OOP)

 about, 9–33

- using, 442
 - object type hinting, 249
 - objects
 - creating, 14
 - treating as strings, 16
 - open source
 - revision control software, 437
 - Open Web Application Security Project (OWASP), 489
 - open_basedir, 477
 - operating systems
 - MySQLDump class, 70
 - output buffering
 - caching parts of pages, 378
 - displaying errors and exceptions, 261
 - server-side caching, 373–377
 - outputting
 - charts and graphs, 223–230
 - data in table, 127–129, 130–133
 - errors and exceptions, 261–265
 - formatted text, 88–90
 - source code online, 161–163
 - strings, 79–81
 - overloading
 - servers, 484
 - overriding
 - constructors, 22
 - methods and properties, 20
- P**
- packet sniffers
 - data security, 270
 - page expiry headers
 - setting, 367
 - pages
 - caching parts of, 377–381
 - preventing web browsers from caching, 365–367
 - parsing
 - RSS feeds, 398–405
 - XML with XMLReader, 399
 - passing
 - credentials in DSN, 44
 - passwords
 - changing, 330–338
 - forgotten, 318–330
 - MD5 algorithm, 286
 - security, 494
 - PDO (PHP Data Object)
 - about, 40–41
 - auto-commit mode, 66
 - PEAR, 497–504
 - alternatives to, 504
 - installing, 498–504
 - PEAR Coding Standards, 446
 - PEAR package manager, 501–503
 - PEAR packages
 - PHP 5 E_STRICT compliance, 293
 - PEAR::Cache_Lite
 - configuration options, 385–389
 - server-side caching, 382–385
 - PEAR::HTML_QuickForm package, 297
 - PEAR::Mail class, 182
 - PEAR::Mail package, 180
 - PEAR::Mail_Mime class, 182, 184, 186, 188, 297
 - PEAR::Net_FTP class, 168
 - PEAR::Validate class, 90
 - performance
 - MyISAM engine, 361
 - permissions
 - files on Unix-based Web servers, 156

- permissions systems
 - building, 339–353
 - Phing, 470
 - php.ini file
 - configuration, 473
 - date.timezone setting, 99
 - error handling settings, 242
 - error logging settings, 247
 - safe_mode, 486
 - phpDocumentor, 449
 - phpinfo function, 485
 - phpt testing framework, 456, 460
 - PHPUnit, 456, 461
 - pie charts
 - creating, 227
 - placeholders
 - date function, 100
 - pointers (*see* file pointers)
 - polymorphism
 - about, 27
 - port numbers
 - database servers, 44
 - portability
 - settings, 475–477
 - post_max_size, 479
 - prepare and execute methods
 - PDO database access, 46
 - SQL injection attack, 55
 - prepopulating
 - form fields, 80
 - preserve state, 277
 - "pretty" URLs, 139–145
 - printf function, 89
 - printing (*see* outputting)
 - private implementation
 - defined, 13
 - programming errors
 - defined, 237
 - properties
 - (*see also* static properties)
 - about, 11
 - overriding, 20
 - protected visibility
 - defined, 13
 - protecting
 - cache files, 385
 - prototypes
 - error handlers, 242
 - proxy servers
 - caching, 366
 - public interfaces (*see* APIs)
 - public visibility
 - defined, 13
 - purging
 - Cache_Lite cache, 389–390
- ## Q
- Query method
 - PDO database access, 45
- ## R
- read function, 356
 - readCache function, 378
 - readdir function, 160
 - readfile function, 152, 170
 - reading
 - local files, 148–152
 - realm
 - HTTP authentication, 276
 - redirecting
 - to another page, 265–267

- refactoring
 - about, 467
 - register_globals, 36, 476
 - registration forms
 - example of, 317
 - registration systems
 - building, 297–318
 - relational databases
 - PHP support for, 39
 - remote servers
 - accessing files on, 166–167
 - replace operations
 - advanced, 84–86
 - report_memleaks, 480
 - reporting
 - (*see also* error handling)
 - errors, 247–248
 - repositories
 - layout, 438
 - reusable PHP classes, 504
 - request headers, 368
 - resellers
 - hosting services, 484
 - reserved words, 341
 - resetting
 - passwords, 325
 - resizing
 - images, 202–213
 - REST web services
 - consuming, 425–431
 - serving, 431–433
 - revision control software (RCS)
 - about, 436–438
 - RSS feeds
 - generating, 405–409
 - parsing, 398–405
 - rules (*see* validation rules)
- ## S
- safe_mode, 486
 - SAX
 - parsing RSS feeds, 404
 - XML extension, 397
 - scalar
 - strings, 77
 - scheduling
 - batch jobs, 485
 - screening
 - web site visitors, 297
 - scripts (*see* stored procedures)
 - handling incoming email, 191
 - hosting policy, 485
 - search operations
 - advanced, 84–86
 - searching
 - for nodes or content in XML, 409–412
 - tables, 63–64
 - security, 489–496
 - data transmission, 269
 - documentation about, 4
 - email injection attacks, 193–195
 - files, 148, 156, 165
 - hiding code, 163
 - sessions, 280
 - settings, 475–477
 - SELECT call
 - number of rows returned, 60
 - sending
 - (*see also* generating)
 - email to groups, 188–191
 - HTML email, 186–188
 - simple email, 179–182

- servers
 - (*see also* Apache web server; database servers; proxy servers; remote servers; web servers)
 - displaying errors, 261
 - overloading, 484
 - session files, 280
 - swapping, 279
- server-side caching
 - output buffering, 373–377
 - using PEAR::Cache_Lite for, 382–385
- services (*see* web services; XML)
- session class, 281–282
- session management
 - security, 494
- session.save_path, 481
- session.use_cookies, 481
- session_regenerate_id function, 494
- sessions
 - storing, 279
 - using, 231, 277–281
- set_error_handler function, 243, 257, 260
- set_exception_handler function, 257
- settingAllowOverride, 486
- settings, 475–481
 - configuration, 286
 - error handling, 239–241, 242, 480
 - includes and execution, 475–479
 - miscellaneous, 481
 - security and portability, 475–477
- short_open_tag, 476
- SignUp class, 299
- signup page
 - creating, 308
- silent mode
 - errors in SQL queries, 50
- SimpleTest, 456, 461
- SimpleXML
 - parsing RSS feeds, 398
 - REST web services, 426
 - XML extension, 397
- SOAP web services
 - consuming, 420–422
 - serving, 423–425
- SOAP XML extension, 398
- SoapClient class, 421
- source code
 - outputting online, 161–163
- spam legislation
 - about, 190
- SPL (Standard PHP Library)
 - using, 174–177
- sprintf function, 89
 - dynamic SQL, 59
- SQL, 44–61
 - adding or modifying data in databases, 53–55
 - errors, 49–52
 - fetching data from tables, 44–49
 - flexible SQL statements, 57–59
 - rows affected by a query, 59–61
 - stored procedures, 67–69
- SQL injection attacks
 - about, 491
 - prepare and execute methods, 55
 - protecting from, 55–57
- SSH
 - access to, 484
- SSL
 - security, 495
- standards
 - coding, 446
- static methods
 - about, 31

- validating strings, 90
- static properties
 - about, 31
- stored procedures
 - cross-site scripting security exploit, 83
 - using with PDO, 67–69
- storing
 - configuration information in files, 163–165
 - cryptographic data, 495
 - dates in MySQL, 107–109
 - sessions elsewhere from server, 279
 - sessions in databases, 353–362
- str_replace function, 85
- streams
 - accessing files, 166
- string functions
 - using XML extension instead of, 396–398
- strings, 77–94
 - breaking up text into arrays of lines, 86–88
 - formatting, 81–82
 - outputting formatted text, 88–90
 - outputting safely, 79–81
 - reading files as, 150
 - search and replace, 84–86
 - stripping HTML tags from text, 82–83
 - treating objects as, 16
 - trimming white space, 88
 - validating submitted data, 90–94
 - wrapping text, 84
- strip_quotes.php file, 91
- stripping
 - HTML tags from text, 82–83
- strtotime function, 101, 102

- Structures_DataGrid class, 134
- Subversion (SVN), 436, 437
- swapping
 - servers, 279
- symlinks
 - deploying code, 468
- syntax errors
 - defined, 237

T

- tables
 - fetching data, 44–49
 - outputting data, 127–129, 130–133
 - searching, 63–64
- tags
 - deploying code, 468
 - revision control software, 438
- TAR files
 - creating, 172–174
- template caching
 - about, 376
- ternary operators
 - reading files as arrays, 149
- Test Driven Development (TDD)
 - defined, 462
- test environments, 461
- test pages
 - permission systems, 349
- test suites
 - as documentation, 453
- testing
 - code coverage, 463–467
- text
 - arrays of lines, 86–88
 - outputting, 88–90
 - trimming white space, 88
 - wrapping, 84

text watermarks
 displaying, 220

threaded multi-processing module
 (MPM)
 hosting support, 484

thumbnail galleries
 creating, 214–217

thumbnail images
 creating, 199–202

times (*see* dates and times)

timestamps (*see* MySQL timestamps;
 Unix timestamps)

tracking
 code revisions, 436–438

transactions
 databases, 65–67

trigger_error function, 241

triggering
 errors, 241–242

trimming
 white space from text, 88

TrueType fonts
 JpGraph, 225

type hinting
 about, 25

types
 data, 77

U

Unix
 session security, 280

Unix timestamps
 using, 96–98

UPDATE function
 data into databases, 54

upgrades
 hosting service policy, 486

uptime command, 484

URLs
 "pretty", 139–145
 access, 495
 direct object reference attacks, 491

User class, 343

utilities (*see* cron utility; ZIP utility)

V

validation rules
 forms, 120

variable interpolation
 strings, 77

variables
 constructing SQL statements, 57

verification
 of images by humans, 234

versions
 multiple code, 438–441

visibility
 defined, 13

W

warning mode
 errors in SQL queries, 50

watermarks
 adding to images, 220–223

web browsers
 (*see also* Internet Explorer)
 examining HTTP headers, 371–372
 preventing from caching pages, 365–
 367

web hosting (*see* hosting)

web pages (*see* pages)

web servers
 (*see also* Apache web server)

- caching, 364
 - preventing web browsers from caching pages, 365–367
- web services, 412–434
 - (*see also* XML)
 - consuming REST, 425–431
 - consuming SOAP, 420–422
 - consuming XML-RPC services, 412–416
 - serving REST, 431–433
 - serving SOAP, 423–425
 - serving XML-RPC, 416–420
- week
 - day of, 101
- WHERE clause
 - UPDATE and DELETE SQL commands, 61
- whitespace
 - trimming, 88
- wildcard characters
 - about, 64
- Windows
 - dates, 97
 - MySQLDump class, 72
- wordwrap function, 84
- wrapper class, 281
- wrapping
 - text, 84
- write function, 357
- writeCache function, 378
- WSDL
 - SOAP web services, 421, 424

X

- XDebug, 463
- XML, 395–412
 - extensions, 396–398

- generating RSS feeds, 405–409
 - parsing RSS feeds, 398–405
 - REST web services, 426
 - searching for nodes or content, 409–412
- XMLReader
 - parsing XML, 399
- XMLReader class, 397
- XML-RPC web services
 - consuming, 412–416
 - serving, 416–420
- XML-RPC XML extension, 398
- xmlrpc_encode_request function, 414
- XMLWriter class, 397
 - generating XML, 408
- XPath
 - searching XML, 410
- XPath XML extension, 397
- XSL XML extension, 397
- XXS (*see* cross-site scripting)

Z

- Zend Framework
 - coding standards, 447
 - installing, 395
 - REST web service, 430
 - XML-RPC, 413
- Zend_Feed class
 - SimpleXML, 403
- Zend_XmlRpc_Server class, 416, 419
- ZIP utility
 - backing up databases, 71
 - creating files, 172–174