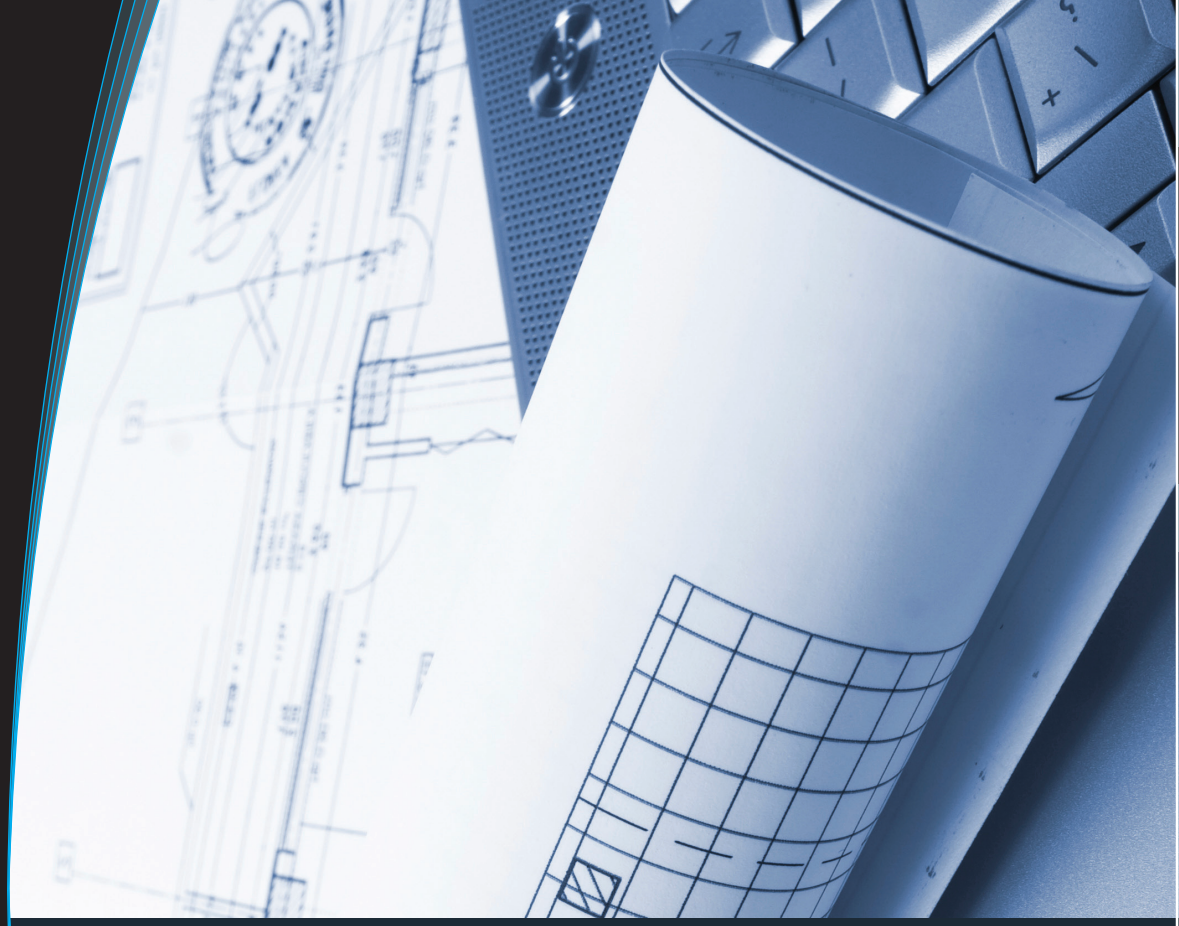
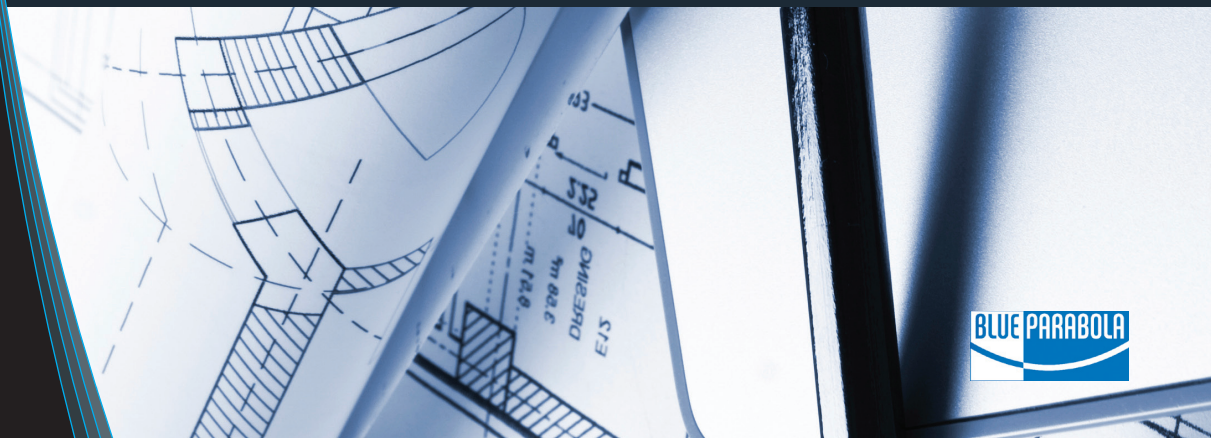


a php|architect guide

PHP Playbook

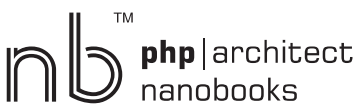
Brandon Savage



BLUE PARABOLA

The PHP Playbook
A php|architect Guide

by Brandon Savage



The PHP Playbook

First Edition: October 2011
ISBN: **978-0-98-103454-6**
Produced in Canada
Printed in the United States

Disclaimer

Although every effort has been made in the preparation of this book to ensure the accuracy of the information contained therein, this book is provided ”as-is” and the publisher, the author(s), their distributors and retailers, as well as all affiliated, related or subsidiary parties take no responsibility for any inaccuracy and any and all damages caused, either directly or indirectly, by the use of such information. We have endeavoured to properly provide trademark information on all companies and products mentioned in the book by the appropriate use of capitals. However, we cannot guarantee the accuracy of such information.

Written by	Brandon Savage
Published by	Blue Parabola, LLC. 28 Bombay Ave. Toronto, ON M3H 1B7 Canada (416) 630-6202 / (877) 630-6202 info@phparch.com / www.phparch.com
Publisher	Marco Tabini
Technical Reviewer	Simon Harris
Copy Editor	Lori Ann Pannier
Layout and Design	Arbi Arzoumani
Managing Editor	Elizabeth Tucker Long
Finance and Resource Management	Emanuela Corso

Contents

Acknowledgments

Thanks to my mother for her love and support, and the belief that I can accomplish that which I set my mind to doing.

Thanks also to my loving wife, Debbie, who endured as I wrote and rewrote various parts of this book.

For their mentorship and involvement, special thanks belong to Keith Casey, Eli White, Cal Evans, Matthew Turland and many others.

Thank you to Travis Swicegood for his review and contribution to the Version Control chapter: it is better for his recommendations.

Thanks to Elizabeth Tucker Long for her patience while this book was written and for the exceptional job editing and managing the project, from conception to completion.

Introduction

The finest schools and best programming books in the world offer a plethora of information about how to string bits of code together into elegant applications and programs. They discuss grand theories of operation, talk about design patterns on a conceptual and enterprise-level field, and teach us the syntax, idiosyncrasies and, in the case of PHP, some of the odd and unusual design choices made by the language creators.

But they do not teach us how to program.

There is a distinct and unusual disconnect between the science of application development and the art of application development. Fresh graduates from the finest computer science courses have no idea how a development shop actually **works**, from a practical standpoint. And those who pick up programming to solve a particular problem, much like I did, do not have much of a guidepost when it comes to how programming is done in the real world.

This book hopes to help answer some of those questions. It is designed not as a book about the ins and outs of writing code, but about the details and concepts critical to working with a team, working within a software development company, and strategies for building winning applications.

Throughout the book important concepts are discussed: refactoring, test-driven development, bug tracking and debugging. We also focus on some PHP-specific concepts, from several "worst practices" to caching for web applications and web application performance. Finally, the book concludes with a chapter on how to make developers happier at what they do.

This book, written for beginners and old hands alike, is designed to approach the development of PHP applications in a new way. It focuses on the tips and tricks of

development, not necessarily the particular code snippets that make up a software application. It is designed to be used as a reference guide, rather than as a novel or other technical book. It does not build on itself; each chapter stands alone, though taken together they form a complete understanding of modern development in the PHP economy.

While many development books are code heavy, this book is code light and contains only a few snippets of code here and there. Since this book is about the method used, rather than the means by which PHP developers go about their jobs, there will be code snippets where appropriate; but for the most part the book will contain advice, ideas and realities gleaned from years of experience.

It is important to remember as we delve into the topics covered here, that this is by no means a "Silver Bullet" that will solve all problems, but instead a guidepost in the long road to improving our software development process. In truth, there is no "nirvana" for software development, no "state of enlightenment" where we have reached perfection. But if the continuous reach for perfection continues to drive us, software development will become consistently better.

Chapter 1

Debugging PHP Projects

Introduction to Debugging

Every developer makes mistakes. The reality is that no developer is perfect; in fact, some statistics say that there is one bug for every seven lines of code in some projects. This astronomically high rate of bugs should not give developers permission to be sloppy, but should inform them that bugs in fact exist, and are something that they will never get away from completely.

Bugs are introduced when mistaken assumptions are used to write code, or an incomplete understanding of the task at hand causes development of a feature that is incompatible with its intended use. Bugs can also be syntax errors or incorrect logic. Every developer experiences bugs of all shapes and sizes in the development process, from small syntax errors to large sections of code that require refactoring to resolve.

Having a solid toolkit, then, is essential to every developer. Being able to resolve bugs, and debug their own code is the difference between a mediocre programmer and an excellent one.

PHP developers have an advantage over traditional software developers in that PHP developers need not wait for their code to be run through a compiler before discovering a programming error. In fact, PHP developers have the ability to compile their code on demand, test it instantly, and resolve bugs as they work. This

combined with the Xdebug extension for PHP, developers of PHP applications are well-equipped to identify and squash bugs before they make it to production.

Introduction to Xdebug

The main tool of choice for this chapter is an extension called Xdebug[1], and it is freely available and maintained by Derick Rethans. Why did we choose Xdebug as our debugger of choice?

Simply put, Xdebug is far superior to any debugging product on the market today. Xdebug offers a great collection of features, including the ability to output variables, trace functions, profile an application and provide useful stack traces along with error messages.

Xdebug has a vibrant community and is actively maintained, which provides it credibility and ensures that it remains useful and compatible with future releases of PHP. Its comprehensive configuration options also help developers customize it to meet their individual needs.

Installing Xdebug

In order to make use of Xdebug, it must be installed. For most users, Xdebug can be compiled from source, or it can be installed via the PECL package manager (the recommended and preferred method for installing Xdebug). Here are installation instructions for various operating systems:

Windows Users

Windows users can install pre-packaged binaries[2] and following the installation instructions located on the Xdebug website[3].

Linux and Mac Users

Users of Linux and Mac computers should install Xdebug through the PECL package manager.

[1] `http://www.xdebug.org`
[2] `http://xdebug.org/download.php`
[3] `http://xdebug.org/find binary.php`

To use the PECL package manager, make sure you are root (or execute the commands as root). The following command will successfully install Xdebug:

```
# pecl install xdebug
```

Once PECL has finished installing it, you must add the extension to the `php.ini` file. Ignore PECL's prompts to add `xdebug.so` to the INI file, as doing so will create problems later on; instead add the following line to your `php.ini` file:

```
zend_extension="/usr/local/php/modules/xdebug.so"
```

Be sure to correct the path to the module with the path for your system.

Installing Xdebug from Source

For those who want to configure Xdebug from source, instructions are available on the Xdebug website[4].

Xdebug Configuration Options

Xdebug offers a great number of configuration options, all of which are documented on the Xdebug website[5]. These options allow for the control of just about everything, from whether or not Xdebug overloads the `var dump()` function to how many nested levels of an array are printed.

Variable Output Configuration Options

`xdebug.overload var dump` By default, Xdebug overloads the `var dump()` function. If you determine that you want to turn off the Xdebug overloading, it is still possible to print the Xdebug `var dump()` with a special function called `xdebug var dump()`.

`xdebug.var display max children` This setting controls how many keys or properties are shown during a `var dump()` or similar function usage. The default value is

[4] `http://xdebug.org/docs/install`

[5] `http://xdebug.org/docs/`

128; this means that by default, 128 keys in an array would be displayed to the user unless this value was changed.

`xdebug.var display max depth` By default, Xdebug only shows 3 levels of depth in an array or list of properties. This means that if you have a multi-dimensional array with several levels of depth, Xdebug must be configured to show a higher amount. The recommended setting is 20 levels of depth; this should be sufficient for almost all developers.

Function Trace Configuration Options

`xdebug.collect params` This setting tells Xdebug whether or not to collect the arguments passed to functions, and whether or not to display them as part of function traces or stack traces. Because this will affect stack traces, it is recommended that you leave this to the default of 0 unless you have a need to see the arguments (for example, to see how arguments travel through an application).

`xdebug.collect return` Setting this to 1 will automatically collect the return value of functions and write it to the function trace file before it records the next called function. While this creates an exceptionally long file, it also allows you to see how and where items are being returned in an application.

`xdebug.trace options` This poorly-named option allows you to append a function trace file, rather than overwrite it. This is useful when you wish to trace multiple files.

`xdebug.trace output name` You can dictate the name of the trace file with a variety of options. You should refer to the documentation on the Xdebug website for the most recent documentation options, as they may change from release to release.

`xdebug.auto trace` Occasionally you will want to have Xdebug trace from the beginning of a file (for example, if you are using `auto prepend`). This will automatically begin the function trace. It is recommended that you use this option in conjunction with `xdebug.trace options`.

`xdebug start trace()` This Xdebug function will start a function trace (as opposed to using `xdebug.trace options` as an INI setting).

`xdebug stop trace()` If you wish to stop a function trace prior to the end of a file, you can use this function to do so. This will stop Xdebug from tracing the file any further, allowing you to identify the path of an application in a specific part of a file.

```
TRACE START [2009-11-10 11:55:32]
    0.0004     623224   -> {main}()  /Users/brandon/Sites/skel/branches/wedding/webapp/htdoc
    0.0006     623656     -> require_once(/Users/brandon/Sites/skel/branches/wedding/webapp/
    0.0008     624096       -> require_once(/Users/brandon/Sites/skel/branches/wedding/webap
    0.0008     624336         -> define() /Users/brandon/Sites/skel/branches/wedding/webpapp
    0.0008     624424         -> define() /Users/brandon/Sites/skel/branches/wedding/webpapp
    0.0009     624496         -> define() /Users/brandon/Sites/skel/branches/wedding/webpapp
    0.0010     634120       -> require_once(/Users/brandon/Sites/skel/branches/wedding/webap
    0.0011     634240       -> Modus::setIncludePath() /Users/brandon/Sites/skel/branches/we
    0.0011     634472         -> get_include_path() /Users/brandon/Sites/skel/branches/weddi
    0.0011     634536         -> set_include_path() /Users/brandon/Sites/skel/branches/weddi
    0.0011     634344       -> Modus::setAutoloader() /Users/brandon/Sites/skel/branches/wed
    0.0012     634944         -> spl_autoload_register() /Users/brandon/Sites/skel/branches/
    0.0012     635152       -> Modus::autoload() /Users/brandon/Sites/skel/branches/wedding/
    0.0012     636128         -> include(/Users/brandon/Sites/skel/branches/wedding/lib/mani
    0.0024    1094000         -> include(/Users/brandon/Sites/skel/branches/wedding/webapp/m
    0.0024    1105104         -> array_merge() /Users/brandon/Sites/skel/branches/wedding/li
    0.0033    1394160         -> require(/Users/brandon/Sites/skel/branches/wedding/lib/modu
    0.0035    1402664           -> require(/Users/brandon/Sites/skel/branches/wedding/lib/mo
    0.0036    1416080           -> require(/Users/brandon/Sites/skel/branches/wedding/lib/mo
    0.0038    1449280           -> require(/Users/brandon/Sites/skel/branches/wedding/lib/mo
    0.0038    1449632           -> spl_autoload_register() /Users/brandon/Sites/skel/branche
    0.0040    1191960       -> Propel::init() /Users/brandon/Sites/skel/branches/wedding/web
    0.0040    1191960         -> Propel::configure() /Users/brandon/Sites/skel/branches/wedd
    0.0041    1192784           -> include(/Users/brandon/Sites/skel/branches/wedding/webapp
    0.0041    1199488           -> Propel::setConfiguration() /Users/brandon/Sites/skel/bran
    0.0042    1199536             -> is_array() /Users/brandon/Sites/skel/branches/wedding/l
    0.0042    1199848             -> Modus::autoload() /Users/brandon/Sites/skel/branches/we
    0.0042    1199848             -> Propel::autoload() /Users/brandon/Sites/skel/branches/w
    0.0044    1207656               -> require(/Users/brandon/Sites/skel/branches/wedding/li
    0.0045    1207336             -> PropelConfiguration->__construct() /Users/brandon/Sites
    0.0045    1207080         -> Propel::initialize()  /Users/brandon/Sites/skel/branches/we
    0.0045    1207080           -> Propel::configureLogging() /Users/brandon/Sites/skel/bran
    0.0046    1207160             -> PropelConfiguration->offsetExists() /Users/brandon/Site
    0.0046    1207208               -> array_key_exists() /Users/brandon/Sites/skel/branches
    0.0046    1207160           -> PropelConfiguration->offsetExists() /Users/brandon/Sites/
    0.0047    1207208             -> array_key_exists() /Users/brandon/Sites/skel/branches/w
    0.0047    1207160           -> PropelConfiguration->offsetGet() /Users/brandon/Sites/ske
    0.0047    1207128           -> is_array() /Users/brandon/Sites/skel/branches/wedding/lib
    0.0048    1207160           -> PropelConfiguration->offsetGet() /Users/brandon/Sites/ske
    0.0048    1207128           -> array_merge() /Users/brandon/Sites/skel/branches/wedding/
    0.0048    1208096     -> Modus::autoload() /Users/brandon/Sites/skel/branches/wedding/li
    0.0049    1215640       -> require(/Users/brandon/Sites/skel/branches/wedding/lib/modus/
    0.0050    1214864     -> ModusSession::getSession() /Users/brandon/Sites/skel/branches/w
```

Figure 1.1

Xdebug Profiler Options

Xdebug comes with a profiler that will produce a cachegrind-compatible file. This file can be used with any number of profilers, like Kcachegrind for Mac or Windows.

Profiling is a bit different, because unlike function traces, profiling must be done over an entire PHP script, and cannot be started in the script (it must be done as an INI setting or directive). There are still a number of configuration options, though, that you should be aware of and utilize.

In order to utilize the profiler you must set `xdebug.profiler enable` to one (1).

Occasionally you may wish to append the cachegrind file, rather than overwrite it altogether. `xdebug.profiler append` will allow for the cachegrind file to be appended.

You can change the name of the profiler's output file with `xdebug.profiler output name` to help differentiate between a variety of different scripts that you are profiling.

Using Function Traces

It is often useful to be able to see the path an application takes to rendering a result. Since it is not possible to see the stack at any given time (without raising an error or throwing an exception), making use of Xdebug's function tracing abilities is an excellent way to figure out tricky problems involving what is being run, and when.

Function traces can be run in two different ways: by explicitly invoking the function trace to run and stop, or by having Xdebug start the trace as soon as PHP is invoked.

By running the function trace manually, and setting where it begins and ends, a developer has the ability to see snippets of the code that they might not otherwise see. Wonder what a particular method does? A function trace provides a quick snapshot of the levels of depth through which the method runs before getting to a return value. A function trace can be used to see if certain conditionals are being met (often in tandem with the option to see the arguments being passed to various functions).

In order to invoke a function trace, developers use the `xdebug start trace()` function; the trace can be stopped just as easily with the `xdebug stop trace()` function. These two functions can be invoked at any point in time during the execution of a script.

Occasionally, developers might like to see a script from top to bottom. The ability to have a trace begin automatically is very useful in cases where a developer might want to see what files are included (and at what stages); or they want to see startup functions that are executed before the main body of the application is invoked. The automatic trace options are also great if a developer is using the `auto prepend` INI directive, since there is no easy way to start a trace in this file.

Function traces are useful for many things, but can also be very confusing. It is best if they are used in small increments for specific purposes; developers may find that using them excessively will give them information overload.

Developers should also be careful to make sure the trace files are being written to a directory that is writable by the web server. The web server is often run as root, and may not share the same username or group with the current user.

Outputting Variables

PHP provides several variable output options, including `print r()` and `var dump()`. By default, `var dump()` provides an output that looks something like this:

```
array(6) {
  [0]=>
  string(3) "abc"
  ["mkey"]=>
  bool(true)
  ["auth"]=>
  bool(false)
  [1]=>
  array(3) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "3"
  }
  [2]=>
  array(2) {
    ["key1"]=>
    string(3) "abc"
    ["key2"]=>
    string(3) "def"
```

```
  }
  [3]=>
  string(8) "finalval"
}
```

Xdebug automatically makes this look more attractive and useful, by laying it out differently, adding colors, and making the type markers more obvious:

```
**array**
  0 => string 'abc' //(length=3)//
  'mkey' => boolean true
  'auth' => boolean false
  1 =>
    **array**
      0 => string '1' //(length=1)//
      1 => string '2' //(length=1)//
      2 => string '3' //(length=1)//
  2 =>
    **array**
      'key1' => string 'abc' //(length=3)//
      'key2' => string 'def' //(length=3)//
  3 => string 'finalval' //(length=8)//''
```

This is a vast improvement over the default `var dump()` in several ways: first and foremost, it makes the string length information clearer. It also provides easier-to-read spacing, adds colors (on the web), and provides easy-to-see information about arrays and objects.

Of course, `var dump()` certainly has its limitations: it cannot return its output and instead prints it immediately, which makes it difficult if not impossible to log it to a file. For situations where this is necessary, `print r()` is excellent.

The `print r()` function takes two arguments: a required variable as the first argument, and a Boolean argument that determines whether or not `print r()` returns the variable output, or prints it right away. This ability to return the data makes it possible to write the data to a log file:

```
$var = array('a', 'b', 'c', 'd');
$vardata = print_r($var, true);
file_put_contents('/path/to/a/file.txt', $vardata);
```

What we are left with is a file that contains formatted data:

```
Array
(
   [0] => a
   [1] => b
   [2] => c
   [3] => d
)
```

This data is easy to read as well, but does not contain quite as much information as `var dump()` gives us. Still, it provides sufficient information such that we are able to understand the output and make sense of it.

Xdebug also offers an advanced feature that allows developers to see the zval data stored in a variable. The zval data is data that is associated with the variable's structure in PHP itself: its refcount, whether it is a reference, and other data.

Using our last code sample, here is the use of the `xdebug debug zval()` function:

```
$var = array('a', 'b', 'c', 'd');
xdebug_debug_zval('var');



var:

(refcount=1, is_ref=0),
array
  0 => (refcount=1, is_ref=0),string 'a' (length=1)
  1 => (refcount=1, is_ref=0),string 'b' (length=1)
  2 => (refcount=1, is_ref=0),string 'c' (length=1)
  3 => (refcount=1, is_ref=0),string 'd' (length=1)''
```

This can be useful for determining if a variable is a reference, its refcount and other information. This function has a sister function, which sends the information to `stdout`; it can be accessed by calling `xdebug debug zval stdout()`.

Xdebug also provides the ability to capture and output superglobal information. Xdebug contains a function called `xdebug dump superglobals()` and this function will dump any superglobals that have been set in the INI file for collection. You can

also use `var dump($ GET)` to dump a particular superglobal, without adding it to the `php.ini` file.

Handling Errors with PHP

One of the biggest mistakes any PHP developer can make is not developing with `display errors` turned on, and without `error reporting` set to `E ALL | E STRICT`. These two settings, in tandem, help developers to identify and resolve errors before they make it into production or are discovered in testing.

Each of these settings can be set in the `php.ini` file, or in an `.htaccess` file or even on a file basis. However, for consistency and good development practice, developers should set these globally in their `php.ini` files.

Another mistake that developers often make is not logging errors, either in development or in production. Error logs are vital sources of information when something goes wrong; they can often be the difference between a quick diagnosis of a problem and hours of headache and downtime. Descriptive error messages are an invaluable source of information.

There are three settings that developers should focus on in their `php.ini` files: `display errors`, `error reporting` and `error log`. The `display errors` directive should be set to ON during development and OFF in production. Developers should have `error reporting` set to `E ALL | E STRICT` in order to see all errors, including notices and coding standard violations. Finally, `error log` should be directed to a location where there is a writable PHP error log, which will track errors.

Many developers turn on `E ALL | E STRICT`, realize the vast number of notices and coding violations in their applications, and quickly turn it off, figuring their code works, so clearly these issues need not be corrected. But this is a huge mistake. While the majority of notices are benign, from time to time crucial clues about larger bugs pop up first in notices, before resulting in fatal errors. Turning off this feature and not resolving notices in code leads to bigger problems down the road, and may even be contributing to existing bugs in existing applications.

Once code enters production, developers should turn `display errors` to OFF, so that attackers cannot glean crucial server and code information from displayed errors. However, this makes logging the errors even more important; I often use a separate error log for each virtual host. Finally, developers can relax their `error reporting`

value to `E_ALL | ~E_DEPRECATED` for PHP 5.3, or simply `E_ALL` for PHP 5.2; it is not necessary to log coding standards violations or deprecated warnings for code that is presently in production.

Finding and Squashing Bugs

No developer is perfect. This means that every developer will inadvertently make mistakes in their code, leading to bugs that must be tracked down and squashed. When debugging, I follow five strategies that help alleviate obvious bugs, which make tracking more difficult bugs easier.

Resolve All Errors Right Away

The first way to help find and squash bugs is to fix all errors, big and small, as soon as they show up. While this might seem obvious, in fact many developers live with warnings and notices for some time, allowing small bugs to manifest themselves as large ones.

Whenever a bug is discovered, be it an errant exception, an unexpected notice or a situation where something fails half-way through, patch it right away. Resolve the issue. Reproduce it, eliminate it, and move on. Resolving small bugs now reduces the chances that large bugs will crop up later on, and when the large bugs do crop up, there are not a host of small bugs running around to clutter the view.

Visualize Everything

My favorite function in PHP is `var_dump()` because it helps me to visualize variables, from objects to arrays. As a visual person, I need to see what is going on in order to understand it. But visualizing helps in more ways than that: it gives me a quick, easy, obvious way to see and understand what data is available, how the code changes that data, and the process through which the data travels.

There are many ways to visualize data. Depending on the service or the code I'm working on, I may output the data to the browser or to stdout. I may opt to write it to file using the `file_put_contents()` function. There are a variety of ways that data can be visualized, and a developer can see the way an application is working.

Visualization does have its limits though: obviously it does not help resolve issues between the database and the application, and sometimes it can be hard to identify the specific point where a bug is introduced, even when visualizing the data. And visualization can be very hard to do when output buffering is enabled.

Use Break Points

Most modern IDEs allow for the inclusion of "break points" - special points in an application that, when a debugger is run along with the IDE's parsing functionality, will stop the application at a predetermined spot and allow for the inspection of the runtime environment at that given point.

Whether or not you use an IDE, breakpoints can be built into your application with ease simply by using one of the developer's best tools: the `die()` construct. This construct allows for a developer to end processing of a script and inspect the environment as it was prior to the execution of that `die()` call.

Together with the variable output options, this concept of creating "break points" helps developers to track where a bug is introduced in their code. Obviously, this gets even easier when using an IDE, and finding the location of a bug becomes a cinch.

Develop Incrementally

There are three things I always have open when I develop: my shell window, my IDE, and my browser. Why the browser? Because I develop incrementally - after implementing a feature or some questionable logic, I execute the code (either on the command line or in the browser) to see that, in fact, it is running the way I want it to work.

PHP is a compile-at-runtime language. This presents an awesome opportunity because developers have the ability to run their applications frequently. They need not wait to compile their applications or wait until they have finished writing a particular aspect of their application before they can test it. Instead, they can knock up a test very quickly and see if the logic works right away.

Developers should use the abilities of PHP to their advantage, and the ability to run the program quickly and easily is one advantage that PHP has over compiled languages. It makes developing incrementally easy, and helps quickly find and squash bugs.

Take Breaks When Nothing Makes Sense

Every developer experiences situations in which nothing makes sense at all, and the bug just seems to be coming from nowhere. This scenario will show up from time to time. Instead of beating your head against a wall, take a step back for a few minutes.

That's right: get some coffee, take a walk, chat with a coworker, listen to music, surf the web: whatever it takes to get your mind off the problem and onto something else. Even start working on another ticket if you can: whatever it takes to get into a different headspace.

Chances are, when you come back to the problem it will be that much clearer - perhaps even so clear you wonder why you couldn't solve it in the first place! This is because our minds tend to work while we are away, and sometimes the most difficult and protracted problems have the easiest solutions that just take a fresh set of eyes to see.

Fighting with a problem does not help anyone. Taking a step back helps clear the mind, improve the focus and reduce the stress of difficult bugs; all while the mind keeps working on the problem in the background.

Chapter 2

Test-Driven Development

Most developers who have been in software development for any length of time have heard the words "unit testing" used to describe a particular method of testing software that is in development. They may even have heard of a principle known as "Test-Driven Development", which in a nutshell is the philosophy that a test should be written before any code is written for the actual project.

The problem with testing is that it is very much like following the speed limit: everyone knows they should be doing it, but very few people actually do (and the ones who insist on doing it annoy the rest of us). However, testing is not just an obnoxious roadblock to software development: it is a crucial component to the development and implementation of software.

Why are unit tests so crucial? As systems get larger and larger, two things happen: first, it becomes impossible for any single developer to fully and completely grasp the intricacies of the system at hand. Secondly, as this takes place, more developers are working on and refactoring existing code, making it almost a certainty that somewhere down the line, someone will make a change that renders the application unusable in some fashion (a regression).

Unit tests can help prevent this by showing you where and why code is failing. While unit tests cannot fix every bug or prevent every regression, they can point out flaws in the application logic or refactoring strategies before the flaws make it into production.

This chapter discusses the various types of unit testing that you can do, as well as strategies for convincing others that unit tests are necessary, and writing them anyway even if those who manage you do not allot the time.

We will focus on one particular testing suite, PHPUnit (version 3.4 as of the writing of this book). There are a number of other testing suites for PHP (the PHPT format used for PHP core tests, as well as SimpleTest), but our focus here will be on what I believe to be the most robust and easiest to use, which is PHPUnit.

For starters, what is PHPUnit? In short, PHPUnit is a framework for writing unit tests. This framework provides a number of methods that are used to run and evaluate unit tests. Every unit test has the logic for testing a particular piece of code, and an assertion of fact (e.g. that the result is a string, an integer, an object of type X, etc.) A unit test passes when the assertion is true: it fails when the assertion is not true.

It is important to note that unit tests are not functional tests - that is to say that we do not test more than one function with our unit tests, most of the time. When developers first begin unit testing it is tempting to try to test the whole system - but this is not what unit testing is about. Unit testing is about identifying which methods or functions are failing and correcting that failure - meaning that we need to identify which methods or functions are failing as closely to the failure as possible, and this means testing individual units of code.

Now that we have gotten through that brief introduction to unit testing, let's dig into some of the "plays" involving unit testing and PHP.

PHPUnit Quick and Dirty

For the developer who wants to dive right into unit testing, the process can appear daunting at first. There are lots of things to keep in mind and think about, which can make a developer feel as though there is far too much to understand, and thus testing gets put off for the future.

However, the PHPUnit framework is so easy and accessible that any developer with a basic understanding of object oriented development can make use of it. It implements a good amount of functionality so that we do not have to: for example, it contains assertions, automatically catches exceptions, and allows for the development of test suites. Let's get started writing our first test.

Before we can get started on this section, we must install PHPUnit. The recommended installation method is PEAR; installing from PEAR means that PHPUnit will automatically install its command line application as well as ensuring that PHPUnit is in the include path (unless we have modified the include path to drop the PEAR directory). Additionally, installing from PEAR means that we can update PHPUnit to the latest version easily, and Sebastian Bergmann (the author of PHPUnit) is regularly releasing upgrades and new features to PHPUnit, making upgrading well worth it.

Follow the instructions for installing PHPUnit[1] from PEAR. After installing PHPUnit, there are some terms that we should familiarize ourselves with. A unit test is a test that runs against a specific block of code - usually a function or a method. This unit test runs the block of code, and then makes assertions as to what the block of code should return - that is, the value, type, or behavior that should occur as a result of running the block of code.

Here is a complete unit test written for PHPUnit:

```
require_once 'PHPUnit/Framework.php';

class ArrayTest extends PHPUnit_Framework_TestCase {
   public function testIsArray() {
      $item  = array();
      $item2 = false;
      $item2 = (array) $item2;
      $this >assertType("array", $item);
      $this >assertType("array", $item2);
   }
}
```

There are some things we should take note of in this test. First, the class name is `ArrayTest`; this employs the convention of naming test classes as `*Test`, and if you are testing a class (for example, a class called `Login`) you should name the test `LoginTest`.

All the test methods in our class are named `test*` (for example, we have `testIsArray ()`). This is how PHPUnit finds the tests and executes them. Without this information, PHPUnit would not run our unit tests. PHPUnit uses reflection to determine which methods to execute.

[1] `http://www.phpunit.de/manual/current/en/installation.html`

Also note that the test class extends from `PHPUnit Framework TestCase`. This is the test case class, and contains all the methods necessary to properly generate and run test cases.

As previously mentioned, when beginning to write tests, it is tempting to write only a single method that tests your single point of entry and then determines if the whole class functioned properly. It is important that you avoid doing this; that is more akin to a "functional test" - that is, a test of the functionality. We want to test individual units of code - the methods themselves - meaning that we want to avoid writing functional tests and instead test "units", typically methods.

This brief introduction should be a guide to writing tests; the PHPUnit manual is much more in depth and is up to date. PHPUnit is still in development; though Version 3.4 of PHPUnit was used in the writing of this book, changes may be made that could affect the accuracy of these guidelines.

How to Test When Time is Not Allotted for It

Because testing takes extra time, many managers feel it is unimportant, and worse, may believe that it is "red tape" preventing them from getting the project shipped on time. This is not unusual; the more non-technical the managers get, the more they fail to see the benefits of unit testing.

Good developers can still find time to write these tests, however. Since developers often have a good bit of influence in the estimation process and bug fixing process, there are strategies that developers can employ to improve their chances of getting good unit tests written.

Controlling the Time Estimate

Most developers play a part in the estimation process. Even in the worst development shops, the project manager must rely on the discretion of the developer in order to properly devise a schedule, and this requires estimation on the part of the developer.

Good developers use this to their advantage. They understand how long things will take, and how long things will take in the event of catastrophic failure, and usually pick a number in between in order to satisfy the project manager. Good estimation

means that you can satisfy your manager while still getting your work done in a high quality way.

If estimation is a core component of your job, and you have 100% leverage in making the estimate, you can in fact build in time for testing. When it comes time to calculate the estimate, figure in the time to write tests, and simply leave that time in the final estimate but do not mark down that it is for unit testing.

Some might find this to be a bit dishonest; in truth it can appear that way. However, proper unit testing is an essential part of completing the project, whether or not a project manager will try and cut it out. Unit testing is part of the job.

By properly estimating and providing an estimate that builds in time for unit testing, you automatically build unit testing into your schedule without sacrificing the quality of the finished product.

Writing Unit Tests for Every Bug

One strategy that the guys over at web2project use to improve their test coverage is that every bug that gets fixed must be accompanied by a unit test that tests the code. This accomplishes two things: first, it allows them to show the bug in a programmatic way (meaning they have a before-and-after view of the fix), and secondly, it forces them to write test coverage for lots of their product.

It is pretty easy to argue for writing a test that will show when you have the bug fixed: and since a unit test is essentially a test of the logic that asserts a particular outcome, your unit test is pretty good evidence that the bug has been resolved. The end result here is that you end up with massive amounts of unit tests that address specific bugs in the code, but also test to make sure that any refactoring you do later on does not reintroduce bugs that would otherwise harm your application.

Most of the time, you do not even need a whole lot of time to write unit tests with this method. Showing that a bug exists in code is pretty easy - if you assert what the outcome should be, and you get a different outcome, the unit test fails. Usually these tests take less than 30 minutes to write (and any test that takes more than 30 minutes to write by itself might be too large anyway).

This strategy is also excellent when you have a large existing codebase that lacks unit tests. By writing them as you fix bugs, you will slowly improve your coverage. Over time you will improve your coverage until you are at or near 100%.

Combining Both Strategies

It is possible to combine both strategies we've discussed into one larger strategy: incorporating unit test writing time into your new features, and forcing unit tests to be written for older features that have bugs.

Incorporating a hybrid strategy will help the total coverage of your application, and improve development in the long run. Because you will have unit tests for the new features, you can simply add unit tests to highlight the bugs; you will also improve your unit test writing abilities because you will have a series of "aha" moments when you see how your unit tests were wrong, just like your code, due to failed assumptions or mistaken choices.

Never Ask for Permission to do your Job

Developers love to blame managers for failings in test coverage, or say that they simply were not given the time to put together a good test suite. Both of these arguments fall flat, however.

The reality is that as developers, it is our job to tell the non-technical managers and supervisors we work with how long something will take, what must be done to accomplish it, and then to implement it.

Professionally.

That is an important word that bears repeating: professionally. Developers are professionals; it would be silly to think that a patient could dictate to a doctor what tests *must* be run, or a client to a lawyer how the briefs will be written. Developers cannot and should not allow themselves to be pushed around when good sense and professional standards dictate a particular course of action. Developers have an ethical responsibility, not unlike other professionals, to provide good, solid advice to their customers and managers.

Convincing a Manager that Unit Testing Matters

The easiest way to ensure that solid unit tests are written by a team is to convince the management that they should be written. This is not often an easy task, though: managers can see unit tests as a waste of time, money and precious resources.

The goal here is to convince management that unit tests are not only important but necessary. By convincing them that unit tests are necessary, it makes it that much easier to request resources for writing them. Here are some strategies for convincing managers:

Unit Testing Saves Money

Every manager wants to save money, and unit testing can help accomplish this goal.

While unit tests represent an up-front cost in the time and effort required to write them, they will save money in the long run. Each unit test written reduces the amount of time necessary for testing in the future.

For example, if you have a testing suite that detects a regression before a feature goes into production, that regression can be fixed before the client ever finds out about it, improving the relationship with the client and encouraging repeat business.

Also, when unit tests catch a mistake, you can generally figure out pretty quickly where the mistake originated. This saves developer time and, ultimately, saves the company money, because it takes less time to resolve bugs and debug the software.

If you can convince a manager to use test-driven development, you can also point out that testing saves money by both improving the quality of the design and by saving time. Because the tests are written before the code is written, the code must be written in a fashion that is "testable" - that is, it must be written in such a way that unit tests can be applied. This will improve the abstraction of the application, improving its design. And because the unit tests are written during the architecture phase, the company saves time due to the fact that two phases (test writing and architecture) are combined into one.

Unit testing also saves money because it saves a developer from having to take the time to actually go through a testing regimen personally. Under ordinary circumstances the developer would have to write the code, then go through and make sure it works - a time-consuming process that can take many extra hours. But having a unit test suite ready to go, a developer can write the code and let the computer go through the testing process automatically, dramatically improving the overall testing time. While the developer will still need to employ some variation of the older testing scheme, they can be more confident that everything has been thought about.

Finally, unit testing can save money by reducing the number of critical bugs that emerge. By having developers write tests, especially beforehand, they can build in security tests, meaning that their code must comply with the security requirements before the product ships. If new security bugs are introduced, the code will fail the unit tests, alerting you to potentially seriously dangerous bugs before they are discovered by accident or in the wild.

By employing the strategies in the last section, you can also point out that unit testing does not necessarily require a significant outlay of time or capital to accomplish. Instead, by writing unit tests incrementally against the code base you already have, and writing them against new functionality that you are incorporating, you can show that unit testing can start right away, and incorporate future products and releases, without needing to write tests for your existing application's code.

Unit Testing Improves the Quality of Software

While we briefly touched on this in the section on saving money, unit testing improves software quality. Managers typically care about quality almost as much as they care about saving money; the end result is that you should be able to make a solid case for the quality of software being improved through the use of unit testing.

Software quality is essentially a function of two things: first, how well the software meets the needs of the user, and secondly, how many bugs that user reports to the developer. There are some bugs that a user reports that will never be caught by unit tests. For example, user interface bugs or usability bugs will always be filed. However, bugs like "submitting the form doesn't change the values" are things that you can test and resolve before the software ships.

It is well known that users hate buggy software products; they refuse to use them, or mock them mercilessly. Microsoft learned this with the release of Microsoft Word; the release was terrible, full of bugs, and users hated it.

When your product is full of bugs, your users will hate that product. The end result is that they will replace it, if they are able. Do not let that happen to you. By writing unit tests as bug reports come in and ahead of the bug reports, you will have the best opportunity to prevent bugs from being built into your system. You will be able to test the data storage class for that form and detect that the database did not actually

store the data because you forgot to save it, rather than the client discovering it in production.

Most unit tests take less than ten minutes by themselves but will result in huge improvements in code quality and customer retention. One retained customer is well worth the cost of every unit test written; thus, they are immensely valuable to retaining customers and improving the quality of your software.

Unit tests also improve quality by helping prevent regressions. There is nothing more frustrating than reporting a bug, which is fixed, only to reappear in the next version of the product. Users begin to think you are asleep at the switch or don't know what you're doing; typically, neither is true, but the situation could have been prevented if you had only written a unit test for the previous bug. Why? Because if you had a unit test in place, it would have failed before the product shipped, facilitating a fix and preventing the bug from shipping in the first place.

Unit Tests Show Higher-Ups that Work Is Being Done

Unit tests make a great way to show higher level managers the amount of work being completed. While bugs closed make for good metrics, sometimes you come across a large bug that has multiple components. If you have a feature request that is open for two weeks, and it is the only ticket closed in those two weeks, it can appear that the team hasn't done anything. This is where your second metric comes in: the unit testing report.

Because unit tests produce a report, you can show this report to higher level managers and show them the progress being made. Unit tests show which tests have passed and failed; if you write them ahead of time, you will know how far you have gotten because the unit tests will start to pass (they should all fail if you write them before you have written any other code). This means that even if the ticket stays open for two weeks, you can show steady progress towards completion.

Unit tests also make a great way to show a product manager that something is not ready to ship yet - or prove that something is, in fact, ready to place into production. Because you can print out a unit test report that shows failing tests, your product manager can immediately know that the product is not perfect, in an unambiguous way. It is hard to compete with the output of a unit test report, and the product

manager cannot blame the developers for "delaying" the project if the unit tests are clearly not passing.

Unit tests provide another report that a manager can use to know how far along a project is and how much progress is being made. Managers like to be able to measure things (it gives them the perception that they can control how fast a product moves along), and will be sure to be pleased with the idea of having another metric. Meanwhile you, as a developer, have another tool to ensure the quality of your work.

Knowing When Not to Use Unit Testing

Much of this chapter has been dedicated to the wonders of using unit testing, and the benefits that it can bring. However, there are some scenarios in which you should not use unit testing; it may be due to the fact that functional testing is more effective, or due to the fact that the code is simply not testable - though you should surely strive to avoid this.

The goal of unit testing is to test independent units of code that are not dependent on other components - methods, frameworks, databases. While you can write unit tests for components that utilize these options, this is often one area where unit testing is impossible. A personal example: I utilize Propel as my model, and one of the most difficult things to do is have a database which could potentially break my unit tests. Thus, I abstract as many components as possible to ensure that I have plenty of material for unit tests; however, the database tests are often left to themselves.

Another difficult thing to test is singletons and static classes. A singleton is a particular design pattern that ensures only one of itself is returned. The problem with this is that by ensuring that only one of something is returned, the unit test runs against the changed object returned by the singleton, rather than a fresh copy. This means that a unit test might fail due to unexpected circumstances that are beyond the control of the developer, even though the developer wrote the code properly.

Unit testing is one particular strategy for resolving issues in software development; it is nowhere near a Silver Bullet. By implementing the strategies of unit testing, you can dramatically improve the quality of your software products, but occasionally it is acceptable to deviate from the practices of unit testing, if only for a little while.

Chapter 3

Application Optimization

Optimizing applications is a popular topic. Whenever an application is slow, or whenever an application does not perform up to a developer's standards, the call is always "optimize that for performance."

But what is optimization? And more importantly, how do we know when and what to optimize? Optimization is often seen as a miracle fix that will rid us of all performance problems, but often overlooked in that desire for the miracle fix is that some things and some situations are inappropriate for optimization. Developers often rush into optimization not even knowing what bottlenecks exist and what should be fixed.

This chapter focuses on the steps to optimization, when and where to optimize, and conditions to look for when deciding that optimization is right for your application.

The One Thing You Must Do Before Optimizing

When developers get an optimization bug, they automatically rush into refactoring code, examining the database, and thinking about ways to improve overall performance. But many developers never think to do the most important first steps, the most basic of things required before optimizing.

They never think to profile the application.

Profiling is an extremely critical component of application development. Without profiling, an application developer who is trying to optimize is throwing darts in the dark with no ability to see where they are landing. In other words, optimizing without profiling is pretty much useless.

We covered profiling in Chapter 1, when we talked about Xdebug. Once you have run the profiler on your application, you are in a much better position to understand what is happening and how to improve upon it. But any developer who has ever looked at a `cachegrind.out` file knows they are impossible for humans to read.

In order to understand the cachegrind file, you will need to install an application that interprets it for you. Many applications exist, including KCachegrind for Linux and MacCallGrind for Mac (not free). These applications often create a visual representation of the cachegrind file, showing you the bottlenecks and explaining in vivid, visual terms what took the most time to execute. See Figure 3.1, which is a cachegrind output for WordPress.

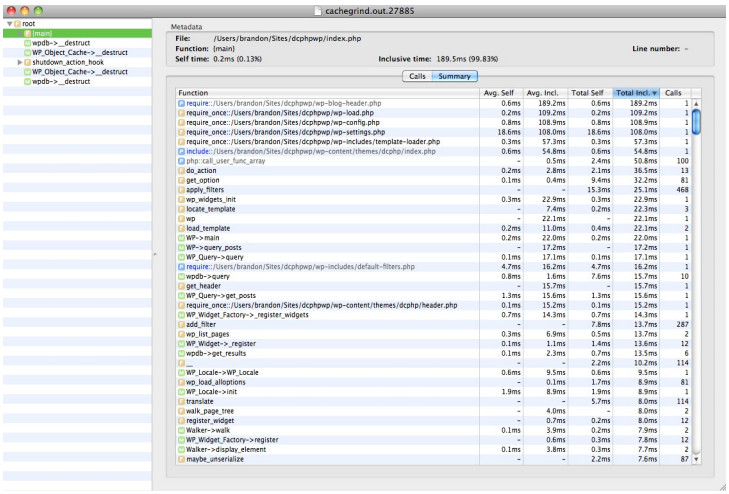


Figure 3.1

This file shows the items in WordPress that took the longest to execute. Depending on your own cachegrind file, you will be able to see where the bottlenecks are, and be able to form a better picture about how to solve them.

Profiling allows us to know exactly what areas of our application are bottlenecks. Without profiling first, we have no idea what areas of our application actually need to be optimized. This can lead to a lot of frustration, lots of refactoring that did not need to be done in the first place, and worst case scenario, the introduction of bugs at the hands of those trying to optimize. Profiling is a crucial component of developing an optimization strategy, and it helps us to know exactly which areas to focus on, in much the same way a surgeon uses diagnostic tests to know where to focus medical treatment.

As you begin to profile your applications, three areas will probably pop up as the largest bottlenecks: the database, code executed inside repeated blocks (like loops), and general code inefficiencies. Let's examine how to take care of all three.

Optimizing Database Queries

Many developers face an ever-present challenge: their applications need to be dynamic, but more often than not the database presents a challenge in the form of a bottleneck. Databases can only handle so much information so quickly. The more information the database contains, the longer it takes to execute even simple queries. But developers can do some things to improve database performance.

A developer's most powerful weapon against database slowness is the `EXPLAIN` SQL command. The most popular databases for PHP developers (Postgres and MySQL) implement `EXPLAIN`; this means that regardless of platform, developers can understand how their SQL queries work.

`EXPLAIN` works by displaying how the database engine intends to attack a particular problem. It will show information about joins, indices, logic, inefficiencies and other important items that can help improve performance.

Imagine that you are building an application with a simple `LEFT JOIN`, which is running inefficiently. Your SQL query is as follows:

```
SELECT * FROM guests LEFT JOIN record ON record.id = guests.id WHERE guests.
    firstname = "Pete";
```

This query runs slowly against your large table, and you need to figure out why. In order to determine how the query is being executed, we add the word `EXPLAIN` on the front of the query:

```
EXPLAIN SELECT * FROM guests LEFT JOIN record ON record.id = guests.id WHERE
    guests.firstname = "Pete";
```

In our command line instance of MySQL, we get a result set back that does not contain matching records, but instead contains the way MySQL intends to process this query. See Figure 3.2.

```
mysql> EXPLAIN SELECT * FROM guests LEFT JOIN record ON record.id = guests.id WHERE guests.first_name = "Pete";
ERROR 1054 (42S22): Unknown column 'guests.first_name' in 'where clause'
mysql> EXPLAIN SELECT * FROM guests LEFT JOIN record ON record.id = guests.id WHERE guests.firstname = "Pete";
+----+-------------+--------+--------+---------------+---------+---------+-------------------+------+-------------+
| id | select_type | table  | type   | possible_keys | key     | key_len | ref               | rows | Extra       |
+----+-------------+--------+--------+---------------+---------+---------+-------------------+------+-------------+
|  1 | SIMPLE      | guests | ALL    | NULL          | NULL    | NULL    | NULL              |    4 | Using where |
|  1 | SIMPLE      | record | eq_ref | PRIMARY       | PRIMARY | 4       | wedding.guests.id |    1 |             |
+----+-------------+--------+--------+---------------+---------+---------+-------------------+------+-------------+
2 rows in set (0.00 sec)
```

Figure 3.2

Straight away we have valuable information about how MySQL is executing this query. Our `LEFT JOIN` looks pretty good: we have the `possible keys` of `PRIMARY` and it is actually using key `PRIMARY`; it is only selecting a single row, because it has the ability to make use of an index.

On the other hand, in our first query (the `SELECT * FROM guests` part), we see that there are no possible keys, and thus it is not using any. We see that it is selecting four records as well. This is a problem, because without any keys, MySQL is not making use of its indices.

The `EXPLAIN` has shown us the problem: if we are making frequent requests where we select based on the first name, we need to add an index on that column. Adding an index on that column yields the following `EXPLAIN` results in Figure 3.3:

```
mysql> EXPLAIN SELECT * FROM guests LEFT JOIN record ON record.id = guests.id WHERE guests.firstname = "Pete";
+----+-------------+--------+--------+---------------+-----------+---------+-------------------+------+-------------+
| id | select_type | table  | type   | possible_keys | key       | key_len | ref               | rows | Extra       |
+----+-------------+--------+--------+---------------+-----------+---------+-------------------+------+-------------+
|  1 | SIMPLE      | guests | ref    | firstName     | firstName | 258     | const             |    1 | Using where |
|  1 | SIMPLE      | record | eq_ref | PRIMARY       | PRIMARY   | 4       | wedding.guests.id |    1 |             |
+----+-------------+--------+--------+---------------+-----------+---------+-------------------+------+-------------+
2 rows in set (0.00 sec)
```

Figure 3.3

In this example, we see that `EXPLAIN` shows us MySQL only needs to request one record, because it knows exactly where the record is that we are hunting for. This is

much more efficient than searching through each and every record for a match; this will dramatically improve our performance.

Without knowing it, we have inadvertently stumbled on another database optimization technique: using indices where appropriate. The database expects that we will define a primary key - a unique key that identifies a record - but the general rule of thumb is that columns we use `WHERE` clauses against, frequently should also be indexed.

And we ought to be using `WHERE` clauses when applicable. Why? `WHERE` clauses help limit the results set, which reduces the data transfer and disk read time of the database. It helps our performance considerably, because the database needs only to find the records we have requested and that match our criteria; it does not need to retrieve and return every single row. Additionally, being explicit in our arguments (using `SELECT column1, column2, ... FROM table` rather than `SELECT * FROM table`) helps reduce the size of the results set.

One other area that is often not considered by database developers is the appropriate use of table types, also known as "storage engines". MySQL (and other database engines) support a number of table types. MySQL, for example, supports `MEMORY` tables, `ARCHIVE` tables, and `TEMPORARY` tables. `MEMORY` tables, as their name implies, are stored in memory. These tables are extremely fast, but are lost once the server shuts down. `ARCHIVE` tables are tables designed for the storage of vast amounts of information. These tables support `INSERT` and `SELECT` statements, but not `DELETE`, `UPDATE` or `REPLACE`. Their design is intended to provide fast retrieval of information that needs to be archived but never updated. The data is compressed on `INSERT` and uncompressed only when retrieved. `TEMPORARY` tables are useful for storing data that does not need to exist for long, or for doing lots of data operations. Selecting the right table type can improve database performance tremendously.

Database optimization is something that every developer must learn, because they will face it with regularity. Databases are a persistent part of the PHP development world, and provide the backend for most applications. Though the movement towards NoSQL continues, relational databases remain the favorites of the business world, which is where most developers are employed.

Function Calls In Loops

Too often, developers write well-intentioned loops that look something like this:

```
$a = array(); //Imagine several keys and pairs.

for($i = 0; $i < count($a); $i++) {
   // Do something
}
```

What is wrong with this loop? It is highly inefficient! Every time the loop starts over, it executes the `count()` function, and determines the value of the count for `$a`. Yet unless we are acting on `$a` in our loop (and if we are, we may get an infinite loop), the value will never change. This means that we are wasting processing time.

This is an easy trap to fall into, which is the reason I am highlighting it in its own section. Developers often make the mistake of calculating "static" information repeatedly in loops, either in the loop's definition or in the code executed inside the loop.

Functions that are executed inside the loop's definition are much better executed outside the loop. This reduces the number of times the loop has to run, and thus the number of times the function is called. This gives us an instantaneous optimization.

If we were to rewrite this code to be better, our code sample would look as follows:

```
$a = array(); //Imagine several keys and pairs.
$count = count($a);
for($i = 0; $i < $count $i++) {
   // Do something
}
```

This code is more efficient, because we are not executing the `count()` function repeatedly for a value that will not change; instead, we are only executing it once, right before we enter the loop.

Developers tend to forget this step either when writing code or trying to optimize it. Reducing the number of functions called in loop declarations will improve the performance of code in an easy, efficient way.

Spotting Code Inefficiencies

Once a developer has profiled their application and determined that code adjustments are needed, there is a real conundrum: how to go about resolving the issues at hand.

Since every application is different, it is impossible to say with certainty "these seven steps will work." Instead, this section is designed to offer five strategies for optimization that improve performance while reducing lines of code and inefficiencies.

Find and eliminate duplicated information. Developers often make the mistake of duplicating the same information. Perhaps it is copying the value of a function and then recalculating that value a second time; it could be storing the same results set in multiple places; it could be that there are two variables that are calculated independently of one another but contain the same value.

This often happens when two or more developers work on the same code. One developer may not be aware of the operations of another developer, or be aware of the full API, and thus re-implements something that does essentially the same thing as something the first developer already implemented. This leads to duplicated information, which creates inefficiencies.

Reduce what is run on every request. It is often possible to reduce the amount of code that is run on each and every request. PHP's nature is a "share nothing" architecture, so every request means initializing and configuring the website; however, it is possible that not every request needs every resource.

For example, is it possible to eliminate the database connection on static pages? Perhaps it is possible to reduce the number of plug-ins that are loaded on requests that do not need them. Optimize the code to the point where only the resources necessary are loaded.

A lot of times this can be accomplished by replacing `require()` and `include()` statements with an autoloader. An autoloader loads classes "on demand" - that is, when they are requested, they are located, included and invoked. Classes that are not needed are never included, reducing the time it takes to read from disk and implement your request.

Intimately learn the way PHP works - and do not re-implement native functionality. Ever written a function to sort an array alphabetically? Lots of developers have, but what many do not know is that PHP implements the `sort()` function that does

this for us. This is one of many examples where developers do not investigate what PHP can do for them natively, and so they rely on their own implementations.

The fact of the matter is that compiled code will run faster than PHP code every single time. The `sort()` function will never be outperformed by anything a developer can implement on their own. Developers who believe they are being clever by implementing some algorithm need only compare their PHP-based algorithm against the compiled PHP functions to see that PHP's compiled functions run faster every single time.

Learning about the items PHP has built into it is the single greatest investment any PHP developer can make. The manual contains up-to-date information about functions, classes, methods, properties, constants, and other detailed information about PHP's inner workings and built-in components. Many PECL extensions exist to augment the existing PHP language, some of which will ultimately make it into core, but many that extend PHP's functionality in ways that solve real-world problems.

It is not always easy to find a native PHP solution, but it is always well worth it. The ability to eliminate duplicated functionality that is ultimately built into PHP will optimize your code far better than removing a thousand function calls.

Share something. This is covered in greater detail in Chapter 5, but share something. PHP's architecture is a "share nothing" architecture. Each request is processed in full, from build up to tear down. PHP shares nothing with future, or even simultaneous, requests by default. This is because PHP was designed to work primarily over HTTP, which is a stateless protocol in itself. But that does not mean you cannot share something.

Caching objects, configurations, and other items can greatly improve performance. Complex operations that do not change from request to request are great candidates for caching. Refer to Chapter 5 for more information about how to accomplish excellent caching.

Learn to abstract classes, functions and methods. Smaller functions have a tendency to perform better than larger ones. Well-abstracted functions, methods and classes help improve performance. This might seem counter-intuitive: add more and performance improves? But there is a simple, if not obvious reason this happens.

Smaller blocks of code mean more specialization, which means the right block gets called for the right job.

A large block of code may contain a number of conditionals, switches, statements and behaviors that do not apply to each and every request. But those components must still be evaluated, considered, and adjudicated by the parser on each request. Breaking large functions, classes and methods into smaller ones reduces the number of items that must be evaluated, which will make your code more optimized overall.

This behavior has an unintended side effect, too: it makes code easier to maintain. By having it in smaller pieces, it is easier to get at the heart of a problem and resolve that particular problem easily and without disturbing larger sections of your code. Abstraction also allows you to refactor (covered in Chapter 7) effectively without changing the results to components that depend on your function or method.

Optimizations to Avoid

During their careers, most developers get caught up in so-called "micro optimizations" - items that are designed to improve performance ever so slightly. The theory is that if a developer can improve performance ever so slightly, they can benefit from the aggregates of these improvements.

This results in lots of developers running around turning double quotes into single quotes and writing static methods when they do not need to do so. It results in developers swearing by `echo()` instead of `print()`, and lots of fights over whether the `print()` return value of 1 actually makes it slower (it does, but you would need 10 million requests to see an appreciable difference in speed).

Do not fall for these attempts to outsmart the PHP engine. Chances are good that the engine is already smarter than you are. Ultimately, developers who try these techniques will be disappointed in the outcome.

A good rule of thumb is that developers should avoid optimizations that do not relate directly to the problem they are trying to solve. For example, replacing all double quotes with single quotes is not going to appreciably improve database performance; running `EXPLAIN` on the queries and adding indices where appropriate is. Developers who try to add optimizations to areas that are not affected by performance problems will be disappointed with the outcome, and are likely to introduce unexpected behavior (and bugs).

Developers should also avoid optimizations that require them to "hack" the way PHP works natively. For example, some benchmarks claim that static methods

are faster than instantiated object methods. However, converting all objects into static classes and static methods would be a fool's errand; it violates the tenets of object-oriented development and introduces a testing and maintenance nightmare for whomever comes after. Similarly, there is evidence that language constructs like `isset()` run faster than functions like `array key exists()`. However, if `array key exists()` is the appropriate tool for the job, by all means use it.

Finally, developers should avoid optimizations that do not actually solve their issue. Sometimes, code cannot be optimized. Sometimes the "optimization" is worse than the original implementation. Sometimes there is a big ugly method in the middle of a class and there is nothing a developer can do to fix it. Simply put, sometimes the optimization is worse than the bottleneck, and in those cases, it should be avoided.

When Not to Optimize

So when should a developer optimize, and when should a developer consider other alternatives? There are four cases when a developer should consider alternatives to optimization.

Do not optimize when the issue is performance. Optimization is a fantastic tool, but is unlikely to solve every single performance issue a developer encounters. Instead, developers who experience performance problems should implement solutions that improve performance considerably, rather than the bit by bit that optimization promises.

Too often developers think that all performance problems can be solved through optimization. This is not necessarily true. Developers can certainly build a well-optimized product, but at some point the traffic and load necessitates identifying and implementing performance-enhancing solutions that optimization simply cannot deliver.

Avoid optimizing when code needs heavy refactoring. Optimization should be viewed as fine tuning. Code that is in serious need of refactoring should be refactored (see Chapter 7); you should not optimize it as a solution.

This really comes down to picking the right tool for the job. If the code is a mess, has performance issues, and needs to be heavily rewritten, optimizing it will ultimately be a waste of time. It is better for the developer to refactor the code (presum-

ably in the correct way) than to optimize poor-quality code. This will be a better use of time, resources and effort.

A lot of times an optimization project turns into a refactoring project. A developer will realize that in order to implement the efforts they are hoping to implement, they must redesign the implementation of the code they are working with. This is ok, as developers can abandon an optimization and move into a refactoring with ease. Good leaders understand this, and give developers the ability to determine when and where to employ optimization versus refactoring.

Do not optimize until you know exactly what the bottlenecks are. This was the introduction to the chapter, but it bears repeating: do not optimize unless you know exactly what areas are your problem areas.

The key here is that the profiler should be rerun after the optimization is complete to re-identify new bottlenecks, compare the old bottleneck to the optimized area, and to identify if any unexpected new bottlenecks were introduced. The worst optimizations are those that introduce new problems, so make sure that the original problem has been resolved without the introduction of new issues before moving on to additional optimizations.

Do not optimize if you do not have unit tests to check your work. Unit testing is important whenever you optimize, refactor, or otherwise reengineer components of your application's internals. This is because it is possible to introduce bugs during an optimization; bugs that are hard to detect with functional tests and may crop up in inconvenient places (like production).

A solid suite of unit tests can help you to check your work and make sure that no unexpected bugs have been introduced. Writing unit tests (covered in Chapter 2) should be a familiar and regular endeavor for developers, and provide invaluable feedback on the efficacy of optimization changes.

Be sure when optimizing that the unit tests are modified to reflect the changes the optimization brings about. For example, if one large method is broken into four smaller methods, be sure to rewrite the unit test for the large method so that the four smaller methods are tested instead. Otherwise, the single test is liable to fail, and there is no testing on the new methods that have been written.

The caveat here is that, as Wez Furlong says, "Your unit tests are only as good as the bugs you've already found." Unit tests will not prevent the introduction of bugs, but can reduce the instance of stupid mistakes that make it into the finished product.

Optimizing has a higher likelihood of introducing unexpected behavior, and having a solid set of unit tests is an essential component of a successful optimization.

Chapter 4

Improving Performance

As websites grow, there is often a need to boost the performance of the site, or its ability to handle large amounts of traffic. While many sites never get to this point, once a developer has reached this point, it is important to understand the options that are available.

This chapter discusses a number of ways to improve the performance of a site and scale it for additional users. While the title of this chapter is "Improving Performance", many of the items here will focus on scalability and scaling as well. Semantics aside, the point of this chapter is to identify and highlight strategies for boosting the number of visitors that can use a given server, and improving the load time for visitors as a site's needs grow.

There is an important caveat to be noted before reading this section, and it is this: when considering performance and scalability, it is crucial that the developer decide what the best options are for them based on known quantities. That is to say, if traffic warrants adding additional servers, Memcached is a poor choice; likewise when one server can handle the traffic when tuned more efficiently, adding additional servers will only increase the cost without solving the problem.

Adding Opcode Caching

One of the fastest and easiest ways to improve the performance of a PHP-based application is to add what is known as "opcode caching". In fact, this step is so incred-

ibly easy, I recommend it for every single PHP application available. There has been talk about adding APC (the Alternative PHP Cache) to PHP 6 as well; in short, the future is opcode caching.

Opcode caching is a simple principle to understand. Without it, every time a PHP script is called by Apache (or on the command line), it must be compiled and then executed. This compilation results in opcodes - low-level instructions which are executed by the PHP engine.

An opcode cache takes the opcodes that are generated by the compiler and stores them, usually in memory, for retrieval at a later date. Based on the configuration, the opcode cache will, on the next request for that particular script, evaluate whether or not the script has changed on the disk. If the script is the same (as it most often will be), the opcode cache will provide the opcodes directly to the PHP engine, rather than allowing the script to be recompiled. This process saves time, since already-compiled code does not need to be recompiled.

There are a number of opcode caches available for PHP: APC, eAccelerator and Wincache, to name a few. Many of them are open source and can be added to PHP with little or no effort.

Adding Memcache

At some point, opcode caching is not enough, and developers need to cache additional information. Many developers will experiment with the user caches built into many opcode caches (APC has a user cache that can be used to store data), but run into a significant limitation: these caches are often limited to a single server, and limited to the memory on that server.

Developers have long desired to solve this problem, and to do so they invented Memcache. Memcache is a memory-based cache with a really important component: the ability to run it on a different server, or several different servers in a pool.

Memcache gives developers the best of two worlds: a cache that is as fast as RAM-based information, and the ability to store that cache on a different machine.

When developers begin working on caching, and particularly begin moving into larger-scale systems, Memcache is pretty much the option they decide use. PHP has two extensions for Memcache: `ext/memcache` and `ext/memcached`.

Adding Database Servers

As web applications grow, the point where strain often becomes evident is in the database layer. This is for a number of reasons, including the fact that many web applications are heavily data-driven, meaning that they are heavy users of the database. Most of the time, the database is the first thing that has to be scaled.

In fact, it is always a good idea for a web application to have two database servers, which can preserve data from loss and help protect against downtime. It also helps developers determine the best way to write their applications for more than a single database server, which in turn makes scaling easier.

The most common approach to adding database servers is to create a master-slave configuration. In this kind of setup, a single master can support a number of slaves. The master is the only server against which writes are performed. The changes are then synchronized to the slaves, which are used for the purposes of reading data. This setup is common and fairly easy to implement.

There are, however, several caveats that developers should be careful about. First, it takes effort to determine which server should be written to and which should be read from. Developers therefore should use a common API for this task. Additionally, developers should be careful to write their applications in such a way as to not rely on the database to instantaneously make the written data available to them; as slaves will take a small amount of time (usually unnoticeable) to receive the replicated data. If there is any kind of a lag in this, the data may be unavailable.

Both of these caveats require careful programming, but provide benefits during subsequent attempts to add database hardware. Once the initial work of adding a master-slave setup is accomplished, developers can freely add additional slaves to handle the load of reading from the database (which is typically the most common operation).

Adding Web Servers

Sites that have reached the limits of what they can serve effectively using a single web server may find the need to add additional web servers, as well as a load balancer. Typically this happens after the need for additional database hardware becomes relevant, because most of the time it is not the code that is eating the majority of the processing time. Still, as sites grow, this need increases.

Adding web servers increases the challenges faced by a developer, for several reasons. First, they must identify a way to balance traffic coming into each web server, usually with a third machine known as a load balancer. This machine's sole purpose is to divert traffic to a particular machine, and can be configured in a number of ways.

However, when implementing a load balancer, developers have to be careful, especially with local machine file system or memory operations. For example, if the web server also has a Memcached server on it that is storing sessions in memory, in a load balanced environment there is no guarantee that the user will be redirected back to that particular server on the next request. This also applies to file uploads and other activities that affect the memory or disk of the local machine; thus, it becomes increasingly important that developers use a CDN or other method for sharing common files across servers.

Adding web servers - especially a lot of them - increases the difficulty involved in deploying new code as well. Since it is possible that users can be directed to a different web server on each request, having only half updated at the time of a user's request could result in them seeing very different code bases (or their session containing incorrect information for the particular environment). It is thus important to solve deployment issues right away.

Still, when adding web servers, there is a definite mark of accomplishment, as this means a site has grown to the point where additional resources are necessary. Unlike database servers, where it is common to have more than one, web servers are not always needed, and adding a second web server is a sure sign of progress for any website.

The Benefits of Expanding Hardware

Adding hardware has some additional benefits. With more than one server, there is failover in the event that a particular system goes down. This redundancy is built in to the idea of adding additional hardware and so long as the master database server does not go down, the application can continue operating without a hitch.

When larger organizations add hardware, they may have the ability to specify the technical requirements of that hardware. Being able to require that hardware have certain technical specifications allows an organization to tailor their hardware pur-

chases to their needs. For example, database servers should be equipped with large amounts of storage space and very fast disks, while Memcached servers will have large amounts of RAM and relatively little disk space. Web servers should be able to handle high numbers of concurrent connections and have processing power to serve many threads of the web server.

Tailoring the servers to their specific roles can improve the performance of each server in the completion of that task, which in turn helps reduce the need for additional machines (because each machine already in use is more efficient).

The Drawbacks of Expanding Hardware

Adding hardware does present some significant challenges, and is not without drawbacks.

When adding web servers, the first reality is that the file system is no longer an adequate place to store things, from media (images, uploads, etc.) to caches. Because the file system may be different on each machine, and depending on the load balancing setup, a user may not end up on the same machine on each request; sessions are an obvious casualty and must be moved to a more stable medium, such as Memcached.

Additional hardware also increases maintenance costs and reduces the ease with which new products can be rolled out. Since they must be rolled out to multiple machines, the planning process gets longer, and since there are multiple machines to maintain, failed disks and necessary upgrades cost that much more.

Finally, adding additional hardware adds complexity to the process of developing the application. With the file system gone as a resource and the possibility that any one of several database servers might be serving requests, issues with concurrency, latency, slave lag, missing cache data, multiple user tabs on the same session and a whole host of other issues come into play; that are not issues on a single server, or on single web server plus database master setups. These issues are challenging, and add to the overall complexity of managing an application.

Chapter 5

Caching Techniques

What is Caching?

As primarily a language for the web, PHP's design is that of a "shared nothing" architecture. The HTTP protocol is stateless, meaning that no information is retained about the state of an application when a page is served. These two things together mean that for data-driven sites, there is no built-in ability to preserve information about logins, data retrieved, user information or other information that should persist between requests.

Since the beginning of the internet, then, developers have worked to solve this "stateless" problem. Most developers already solve much of it, by using sessions (which write information to the server and use a key to access it), cookies (which is how the session key is usually stored on a user's system), and databases (which preserve application data in physical, permanent storage). Working in concert, these items help preserve the "state" of an application for future requests; helping users to enjoy a more seamless and meaningful experience on the web.

Of course, these are fairly primitive solutions that are not always the most efficient and effective. Querying the database on each request, for example, can be a difficult and strenuous task, especially for high traffic sites. In the last chapter we talked about boosting performance, but did not focus on how we can reduce the load of each request. This chapter focuses on reducing that load through caching.

Caching is the storage of often needed, but infrequently changing data in its final or near-final state for easy retrieval and use within an application. Examining each part of this definition we find three parts:

Part 1: "...often needed, but infrequently changing data..." Many applications have large pieces of information that are needed on each request, but do not change very often. This ranges from the user's personal information to a list of blog posts. If the information changes frequently, it must be refreshed frequently, which makes caching ineffective for reducing the load on each request.

Part 2: "...data in its final or near-final state..." A big emphasis in caching the data is that we reduce the load on the server as much as possible on subsequent requests. Therefore, storing the data in an unfinished or processing-needed state reduces the efficacy of our cache. Often, data is preserved in a cache in a state that requires no processing at all, other than display formatting. This is relatively simple for the server to send back to the user.

Part 3: "...for easy retrieval and use within an application." Caching should simply be easy to use. Complicated methods for obtaining cached data are often slow, and certainly cumbersome, which will prevent a team from making use of them. When using a cache, developers owe it to themselves to make it easy to get the information, serve it, and move on to the next request.

Rules for Caching

Before moving into the types of caches and their individual benefits, it is important to establish some ground rules for caching. There are five crucial rules that must be followed when considering caching, in order for caching to be an effective tool in a developer's toolkit.

Cache only infrequently changing information

While at first glance this rule might appear to be obvious, it reminds us of an important point: cached data is supposed to stick around for a while. Whether that time period is ten minutes, an hour or a week, cached data is meant to be preserved.

A good guideline is that if the data will be invalid after only a few requests, it is better to load it rather than caching it. Data that is going to be the same for many requests or even for an entire user's session can safely be cached.

Do not cache information that could lead to a security problem

When caching data, particularly user data, it is important to consider the security implications of that caching choice. For example, if a user's permissions are cached until the end of the session, it is possible that the user's rights might be revoked but that the change may not be reflected immediately. This can create a security problem.

This is not to say that no user data should ever be cached; that would be an impractical and foolish assertion. Instead, when using caching it is critical that each developer determine the security policy they wish to employ and, with regards to protecting the integrity of the application, know the tradeoffs they are making when caching certain pieces of data. Developers can employ a variety of techniques to secure their applications (displaying the "Delete User" page, but confirming that the user deleting another user still has those rights when the delete behavior is invoked, for example).

Remember that caching is a bit of black magic

Luke Welling put it best in a tweet when he said "Any sufficiently complex caching infrastructure is indistinguishable from black magic." And he is right: any caching system is designed to provide data almost "magically", inserting information without processing it or checking to see if it is still valid.

Debugging caching problems can be a hairy process for this reason. Developers should therefore build their caching infrastructures carefully, with debugging tools designed in (e.g. a key that indicates the results set was cached) in such a way that fixing cache-related bugs can be resolved easily.

Developers should also work to develop with caching turned off, for the most part, to help ensure that their code actually functions properly (instead of them getting lucky and caching the results). Developing with caching on (APC included) can create difficulties down the road.

Always Have a Backup Plan

Sometimes caches make development too easy, so easy that developers forget that the cache data might not be there one day. This can lead to catastrophic application failure, strange bugs, and lots of difficult debugging.

Cached data is transient. This means that developers should never *expect* it to be there. Instead, they should *prefer* that it be there, but always have a backup mechanism in place to retrieve that data from whatever source it originally was stored in.

On that note, due to a cache's transient nature, they are not to be considered permanent storage. Developers should always use a backup data storage method - the file system, a database, or some other tried-and-true storage engine. A generic rule of thumb is that data in a cache should never exist only in the cache; it should always have come from somewhere stable.

Make Sure the Cache Has a Built-in Expiration Date

Cache data should be transient and refreshed at regular intervals. Obviously, certain data is more persistent than other pieces of data, but in the end, all cached data expires at some point: therefore, so should the cache.

Old data should automatically be removed from a cache. Many caches like Memcached and APC allow for the inclusion of a "Time To Live" (TTL) value, which sets how long the cache data should be considered valid before it is purged. If that TTL has been exceeded, the cache automatically clears the data and reports that it does not have the information requested.

Uncleared caches lead to stale data (more on this later), which creates an inconsistent and undesirable user experience.

File-Based Caches

One of the easiest ways to store data for later retrieval is to place it on the file system. In fact, sessions work this way by default: the user is issued a session cookie, which contains a key, which is then used to load the session data, which is stored on the file system. Usually, files stored on the file system are faster to read than it is to process the data all over again.

Caches on the file system have a number of advantages: PHP comes with built-in file system functions requiring no additional extensions or modules. All developers have a tangential understanding of how the file system works, which means that most developers will be able to effectively use a file system cache. The file system is easy to write to and read from. And PHP's `var export()` function and other tools make it easy to simply include file-system-based cache files.

File-based caches also have some drawbacks, the most obvious being that if the application uses more than one server, files stored on one server cannot be read from another server very easily. File system caches are also limited by the I/O speeds and capabilities of the machine itself. Also, if the file-based cache is to be written to and read from regularly, some sort of file locking mechanism must be used to prevent two concurrent PHP threads from trying to write to the same file. Finally, file-based caches do not automatically expire (they must be purged from the file system).

The following listing contains a sample file-based cache that can easily be used to cache information (note that it does not implement file locking):

```
class FileCache {
   protected $ttl;
   protected $path;

   public function __construct($path, $ttl = 3600) {
      $this >setPath($path);
      $this >setTTL($ttl);
   }

   public function write($data, $append = false) {
      if(!is_string($data) || strlen($data) < 0) {
        throw new Exception('Data must be a string in FileCache::writeCache()');
      }

      if($append) {
         $arg = FILE_APPEND;
      } else {
         $arg = null;
      }

      $result = file_put_contents($this >path, $data, $arg);
      //Test for (bool)false, which is returned on failure.
      if($result === false) {
        throw new Exception('Writing to cache failed FileCache::writeCache()');
      }
```

```
        return $result;
    }

    public function read() {
        if(!file_exists($path)) {
            throw new Exception('The cache file does not exist');
        }

        //Check to see if cache has expired.
        if (filemtime($this >path)+$this >ttl < time()) {
            unlink($this >path);
            return false;
        } else {
            $string = file_get_contents($this >path);
            if($string === false) {
              throw new Exception('Unable to read the contents of ' . $this >path);
            }
            return $string;
        }
    }

    public function remove() {
        if(file_exists($this >path)) {
            unlink($this >path);
        }

        return true;
    }

    public function getPath() {
        return $this >path;
    }

    public function getTTL() {
        return $this >ttl;
    }

    public function setPath($path) {
        $path = realpath($path);
        if(!is_writable($path) || !is_readable($path)) {
            throw new Exception('Path provided is not readable and/or writable');
        }

        $this >path = $path;
        return true;
```

```
    }

    public function setTTL($ttl) {
       if($ttl < 0 || !is_int($ttl)) {
          throw new Exception('TTL must be a positive integer greater than 0');
       }

       $this >ttl = $ttl;
       return true;
    }
}
```

Memory-based Caches

Memory-based caches are caching systems that make use of the available memory on a system. There are two core distinctions between file-based caches and memory-based caches: first, memory-based caches always require an intermediate component to access the RAM on the system; secondly, memory-based caches are lost if power is lost to the RAM, or PHP is reloaded (as it is through a web server restart).

Memory-based caches offer some advantages over other caches: in particular, they are usually very fast (because RAM is very fast) which improves performance. In fact their speed is such an advantage Memcached was discussed in Chapter 4. Additionally, memory-based caches do not typically have issues with locking, or file system permissions problems.

The disadvantage of a memory-based cache is that it is limited to the RAM on the system. This is not significantly different from a file-based cache (which is limited by hard disk space), except that most machines have hundreds of gigabytes of disk space, while they have only a few dozen gigabytes of RAM available. Memory-based caches will make logical choices as to which components to invalidate rather than allowing the system to swap, but this does not necessarily prevent the web server or other machine processes from swapping themselves, which can lead to a performance degradation; thus, when using a memory-based cache, developers must consider the implications of using a memory-based cache.

There are two popular memory caches that we will discuss: APC and Memcached.

Alternative PHP Cache (APC)

In Chapter 4, APC was highlighted for its ability to cache the compiled "opcodes" of PHP for later retrieval, thus avoiding the compile cycle on each individual request. This is considered the "opcode cache" component of APC. In this chapter, we will discuss the "user cache" feature of APC.

Simply put, the "user cache" function of APC allows for the storage of developer-designated information in memory on the machine. This data can be retrieved quickly on subsequent requests, which, when used correctly, has the potential to greatly improve performance. APC also benefits from being one of the easiest memory-based caches to use.

The following contains an example of an APC-based memory cache, similar to the interface used for our file-based cache. Note that the code sample is much shorter, since we do not need to test for write or read permissions, and if the item is not in the cache we simply return false:

```
class APCache {
   public function write($key, $data, $ttl = 0, $overwrite = false) {
      if (!$overwrite) {
         return apc_add($key, $data, $ttl);
      }
      else {
         return apc_store($key, $data, $ttl);
      }
   }

   public function read($key) {
      return apc_fetch($key);
   }

   public function remove($key) {
      return apc_delete($key);
   }
}
```

Memcached

As discussed in Chapter 4, Memcached is the de-facto standard for large web applications and performance enhancement. Memcached offers several advantages

over APC, namely that Memcached can be configured with a group of servers, Memcached can be served from a completely different server from the application (unlike APC which only writes to the memory on the same machine as the application), and Memcached is designed for high level use (it is used by Facebook, Yahoo, Digg, etc.).

Memcached can also be used to store sessions in PHP. To do this, one need only change the following lines in the php.ini file (assuming that PHP Memcached support is installed):

```
session.save_handler = memcache
session.save_path = "tcp://127.0.0.1:11211"
```

It's also very easy to create a Memcached caching class, which will do simple caching operations. Memcached has been used to do some extremely complicated caching, up to and including serving the entire webpage, bypassing PHP altogether.

Note in the listing below that the Memcached extension in PHP is already object-oriented. The wrapper is provided to demonstrate the functionality, but the Memcached object can be used directly if the developer prefers:

```
class Memcache_Cache {
   protected $memcacheObj;

   public function __construct($host, $port) {
      $this >memcacheObj = new Memcache();
      $this >addConnection($host, $port);
   }

   public function addConnection($host, $port) {
      $this >memcacheObj >connect($host, $port);
   }

   public function write($key, $data, $expires = 0, $flag = false) {
      return $this >memcacheObj >set($key, $data, $flag, $expires);
   }

   public function read($key, $flags = false) {
      return $this >memcacheObj >get($key, $flags);
   }
}
```

Avoiding the Pitfalls of Caching

Caching is not all sunshine and roses: while it has its benefits and certainly improves application performance, it also looks an awful lot like black magic.

A developer's most important task when using caching is making sure that the cache is not required for the application to function. Caching is an optional add-on that we bolt to our applications, not a persistent storage mechanism that is always available (like a database). Even if caching is deeply ingrained into the structure of an application, making sure that the data is available from somewhere else is a crucial component of development.

Developers also need to make sure that the data they cache is current and reflects the application's state as best they can. Stale data, or data that has outlived its usefulness and should be discarded, takes away from an application's overall experience. While some level of stale data is acceptable (and each project makes a determination as to what that level of acceptability is), for many developers having an application that displays patently false and obviously wrong information is not an option.

Avoiding these pitfalls can be a difficult task if developers fail to carefully plan their caching systems. But a carefully planned caching system will help alleviate or eliminate these problems.

There are three things each developer can do to avoid the pitfalls of caching.

Centralize the Method and API for Caching and Retrieving Information

By standardizing the method for caching and retrieving information, developers gain two distinct advantages. First, they gain the advantage of having a single, testable spot for storing, retrieving and expiring information. Secondly, developers can switch out the backend with ease, knowing that so long as the API stays consistent, the code accessing the API will be able to make a seamless transition.

Developers also benefit from the fact that caching code is not scattered throughout their applications, meaning that developers have a greater sense of control over the application's caching activities.

Establish Written Rules for Data Fidelity

Without a hard-and-fast set of rules for data fidelity, there is no way to consistently establish what level of staleness is acceptable. Establishing written rules makes every developer aware of the rules concerning data fidelity, and reduces the chances that some data will be stale and other data will not.

Consider Caching Data Up Front, Before It Is Requested

Using cronjobs or other methods, data can be automatically cached by the application for delivery to the end user before that data is even requested. Facebook makes use of such a caching behavior: they place a large amount of their information into memory automatically, making it a simple matter of retrieving it for the end user when it is actually needed.

Summary

Caching is a complicated topic but one deserving of examination and consideration. When done carefully, caching can be a boon to the implementation of an application, improving overall performance and reducing server load at the same time. Developers need to carefully plan and contemplate their caching systems; but once implemented, caches are a great source of performance improvement.

Chapter 6

Harnessing Version Control

Most developers today make use of version control in their application development process. Many of us cannot imagine developing in a world that did not have version control available for us. Some developers have been around long enough to use the old standbys of CVS (Concurrent Versioning System) and seen the migration to Subversion; others became users after Subversion or Git became popular.

Regardless of when a developer begins using version control, all quickly discover its magical power to organize and improve development workflow, almost as though it were manna from heaven.

This chapter discusses what version control is and how to use Subversion and Git (two of the most popular VCS or Version Control Systems), as well as how to use version control even if your manager will not.

What Is Version Control?

Version control is, simply put, a way of keeping track of different versions of a file automatically. While each VCS is different, they all share a similar purpose: to track how a file changes over time and allow developers to view a file in any given state from any point in history.

Version control has a secondary advantage, in that it allows different developers to work on the same files without overriding the changes made by the other. Version control is smart enough to know which parts of a file were changed, and apply those

changes to both versions in a process called "merging". From time to time, version control encounters a "conflict", where the same file was changed in the same location by two different developers, but is still smart enough to inform the developers and encourage them to merge their changes.

Why Does Version Control Matter?

Version control is important for several reasons. First, it offers protection of the source code, either from accidental or intentional damage. Secondly, it offers developers the ability to identify when, and who, changed certain aspects of an application. It allows developers to work together without the stress of managing files or manually merging them together. Finally, it allows developers the ability to experiment, without impacting the development line adversely.

Version control matters because it helps keep code safe. From time to time, accidents happen: for example, developers have hard disks that crash. It is also possible that a disgruntled developer might attempt to damage the company by deleting or altering source code in such a way that it causes it to fail. Version control helps prevent this. By creating a system of essentially backed up files from various points in history, a developer can easily check out a new copy or restore damaged code from the known good code at a particular revision.

Version control matters for other reasons too. In large projects the API, components, modules and structure often changes, and does so quickly. A developer can sometimes be caught off guard by a change in architecture or the API, and might want to know when a change took place. Version control lets that developer look through the code's history and see when and how something changed. Version control also tracks each line to determine who committed it and when, helping identify who introduced bugs or when a bug was introduced.

Because version control offers the ability to merge changes from two or more developers, it helps facilitate collaboration. Developers can work on similar, related but different components of an application at the same time without discussing who has that particular file open for writing or who is doing what. In the event of a conflict, no developer has to worry that their changes will be overwritten: version control allows for manual merging of these conflicts and does its best to merge the files together when the work does not overlap.

Version control also allows users the ability to create a "branch" - a complete copy of the code - into a separate "tree" or area of the repository for development. This branch can be used for the purposes of experimenting, or can be part of the development cycle. At any point, developers have the option of merging the branch together with the main line of development (often called "trunk") without the hassle of copying files by hand and making sure files are not overwritten. A branch can also be removed if a developer decides their work was incorrect or should have been in a different area of the application.

Selling A Manager on Version Control

Unfortunately, just because something makes development easier doesn't mean it's liable to be accepted by non-technical managers. Many developers still work without version control, which is a common problem in "creative" environments that focus less on development and more on the creative element of web design. While this is not true everywhere, it is true in some places.

However, there are some things that developers can do to convince managers that version control is both important and necessary.

First and foremost, version control protects the huge investment that managers are making on development. Because version control protects from one developer overwriting another, and protects against malicious users attempting to damage the code, managers can sleep easier knowing that their code is protected.

Also, managers will be happy to learn that version control can help foster a culture of accountability. All major version control systems allow for post-commit and pre-commit hooks which can do a number of important things, including send emails when code is committed, write to databases or add information to ticket tracking systems, and even validate that the code being committed contains no syntax errors or coding standard violations.

Finally, version control can help control costs and save money. Because code can be centrally stored and managed, developers are more likely to reuse code. Additionally they have the ability to see what has already been done before, which reduces the time they spend writing new code. Because developers are unlikely to overwrite each other, there is less time and money spent redoing work that was overwritten.

Which Version Control Should You Use?

The choice of VCS really depends on the user and their preferences. People are much attached to their VCS and for good reason: mastering one can take some time, with developers needing to learn how they work and solve problems with them. That being said, there are two major VCS that PHP developers tend to make use of and that we will focus on: Subversion and Git.

Subversion

The Subversion version control system is a "next generation" iteration of the CVS system of years past. It provides significant enhancements and features over CVS, especially in the area of branching and merging.

Subversion is almost ubiquitous in the version control world right now. Most developers have used it, and are very familiar with it. It is currently being actively developed, and there are a number of new features planned for the future. It is robust, with its own TCP protocol, and can be integrated with Apache or accessed via SSH tunnel.

Subversion's core feature is that the repository is centrally hosted in a custom file system on a single server. Developers check out a copy of the current working copy (or a subset of the current working copy) and have the ability to make modifications and then check in files. Subversion tracks the differences between the files (the diff) and applies them like patches.

The benefit of this kind of system is that all work is committed centrally, which means there is little chance of data loss (as long as developers commit regularly and the server is backed up). The obvious drawback here is that in the event that the server crashes or the repository is corrupted, the repository can be lost, because an entire copy of it does not exist anywhere except on the centralized server.

Subversion also has a significant advantage in the number of GUI tools, with a wide range of tools available for Windows, Mac OS X and Linux. Some of these tools are free and open source, while others cost money (some are very expensive).

Git

Unlike Subversion, Git is designed to be a Distributed Version Control System (DVCS). This eliminates the absolute hard and fast need for a central repository, because when a user creates a copy of the repository (clones the repository) they get a complete copy of all of the history, files, and components. At the same time, Git is designed to allow users to push and pull changes to/from other users, which allows users to collaborate more effectively.

DVCS can be complicated to understand. In a typical version control system, each computer pulls a working copy but the versions are stored on the central server. This is how Subversion and CVS work. However, Git works differently: even though there may be a central server where all files are stored (for dissemination between developers), when a repository is cloned with Git, an entire copy of the repository is provided to the user - all revisions, commits, history, everything.

This provides some distinct advantages to Git: developers are able to commit without being on the network (since they can commit locally), and the repository can be pushed to another user or to a master repository at a later date. Additionally, Git offers branch and tagging support built right into the core. Subversion allows for branching, but treats the branch as a copy of the entire file system, while Git treats the branch as having a common ancestor to the master line.

Much about how Git operates is based on how it is designed. Subversion tracks change sets, which are essentially diffs over time. Unchanged files are essentially considered to be "at" the revision they at which they started. Git, on the other hand, takes essentially a "snapshot" of what the file system looks like on each commit, and for files that are unchanged since the last commit, it creates a link to the previous iteration of that file.

Git has some drawbacks, however. Because changes are committed on the local file system and to a local repository, a disk failure between pushes to another server can result in a complete and total loss of that particular repository and, along with it, all of the work committed but not pushed. As Git is designed to be distributed, it uses SHA1 commit markers to indicate revision numbers, rather than a sequential set of integers (like CVS and Subversion). Finally, Git is not well developed for use in the Windows environment, and there are limited GUI tools for working with it (for example, NetBeans offers a plug-in, but that plug-in is significantly lacking when compared to the Subversion tools for Windows).

Essential Subversion Syntax

Subversion has a number of crucial commands that aid developers in using it properly. These few commands are essential for every developer using Subversion.

It is important to note that with Subversion, all commands are prefixed with the `svn` executable name. Additionally, developers can get help by typing `svn help` and they can get help on a specific command by typing `svn help <command>`.

Getting a Working Copy

In order to get a working copy, developers make use of the `checkout` command. Subversion works by creating a local copy of the current revision (or the specified revision). To get a copy of the latest revision, use the most basic version of the `checkout` command:

```
svn checkout svn://path/to/repository
```

Developers can specify the path for the checkout by adding a second, optional argument that specifies the path. If this second argument is absent, Subversion will create the working copy in the current working directory. Additionally, unless a trailing slash is added to the end of the repository path, the name of the directory in which all the files are created will be named whatever the last directory in the repository was. To check out a repository to a particular path, use the following command:

```
svn checkout svn://path/to/repository /local/path/to/working/copy
```

Finally, developers can check out their repository from a particular revision by adding an `@REV` option on the end:

```
svn checkout svn://path/to/repository@123
```

The above command would check out revision 123 to the local working directory.

Keeping the Working Copy Updated

From time to time, other developers may make revisions to the repository that are not reflected in your working copy. In order to maintain an up-to-date working copy, the `update` command should be used:

```
svn update
```

The `svn help update` command will display a number of options including updating to a particular revision, or updating only a particular part of the working copy.

Managing the Working Copy File Structure

Subversion tracks changes to individual files: therefore, when moving or changing files and directories, it is important to tell Subversion what has been done, so that Subversion can associate the history with that move. Subversion mimics the file system commands on Linux and UNIX, by offering a `copy`, `mv`, `rm` and `mkdir` command. Each works similarly to the UNIX equivalent, and will make changes to the working copy's change history, which will be committed on your next commit.

Committing Changes to the Repository

You have made some changes to your working copy; great! They are ready to go live. How do we ensure that the changes you have made get committed? We can do so by making use of the `status`, `add`, and `commit` commands.

When we have untracked files in our working copy, they show up with question marks next to them:

```
$ svn status
?       example.txt
```

In order to commit this file to the repository, we need to instruct Subversion to add this file:

```
$ svn add example.txt
A         example.txt
```

Note that there is now an "A" next to the `example.txt` file. This indicates that the file is staged for committing. Next, we can commit this file:

```
$ svn commit  m "Initial commit."
Adding         example.txt
Transmitting file data .
Committed revision 1.
```

Note that I have committed the first revision to this repository. Also note the `-m` flag that is executed along with the commit: this designates to Subversion that we are adding a commit message. Developers should always commit with a commit message. For most repositories, if one is not provided they will be prompted in the editor of their choice to set one.

Now imagine that we make a bunch of changes to the `example.txt` file, and we run `svn status` again:

```
$ svn status
M       example.txt
```

Now the file has an "M" next to it, indicating that it has been changed since our last commit. We no longer need to run the `svn add` command. This file, because it is tracked by Subversion, will automatically be committed during our next commit.

Imagine for the moment that the changes made to `example.txt` were inappropriate or wrong. We are not forced to commit those changes, but instead we can roll them back using `svn revert`:

```
$ svn revert example.txt
Reverted 'example.txt'
```

Now a `svn status` command shows a clean working copy:

```
$ svn status
$
```

It is possible to revert a file to a particular revision. Consult the `svn help revert` command for more documentation on how this operation functions.

Reviewing History

From time to time developers want to see what commits were made against the repository. Developers can see a history of commits with the `svn log` command:

```
$ svn log

r70 | brandon | 2009 11 30 20:32:40  0500 (Mon, 30 Nov 2009) | 1 line

Moving everything into trunk.

r69 | brandon | 2009 11 30 20:29:12  0500 (Mon, 30 Nov 2009) | 1 line

Removing lib; it's an external and should not have been moved.

r68 | brandon | 2009 11 30 20:27:26  0500 (Mon, 30 Nov 2009) | 1 line

Updating the repo.

r67 | brandon | 2009 11 30 20:25:28  0500 (Mon, 30 Nov 2009) | 1 line

Adding trunk directory.
```

The `svn log` command is useful for seeing the commit messages, and has many useful features, like limiting the revision number and seeing lots of extra data:

```
$ svn log  l 1  v

r70 | Brandon | 2009 11 30 20:32:40  0500 (Mon, 30 Nov 2009) | 1 line
Changed paths:
   M /financials
   D /financials/files
   D /financials/install
   D /financials/logs
   M /financials/trunk
   A /financials/trunk/webapp (from /financials/webapp:69)
   D /financials/webapp

Moving everything into trunk.
```

For example, in the above command we limited output to one entry (which is going to be the last revision) and asked it to be verbose, which listed the files that were changed in that particular commit.

Subversion also makes it easy to see *who* changed the file, and in fact, the *lines* that were changed. This command has several aliases: `blame`, `praise`, and `annotate`. The purpose is to show each line of a file, and who was responsible for editing it.

```
$ svn annotate example.txt
   2    brandon Lots of new changes.
```

This shows that in revision 2, user "brandon" (that would be me) changed the first line to read "Lots of new changes." This can be useful for determining when, where, and by whom a bug was introduced in the code, or for seeing what changes were made by a user line-by-line.

Essential Git Syntax

Just like with Subversion, Git also has a number of commands that will aid developers in using it properly.

Setting Up a New Git Repository

There are two ways to get a new Git repository: either clone an existing Git repository or initialize one in a current directory.

To clone a repository, we use the `git clone` command:

```
$ git clone git@github.com:brandonsavage/developersplaybook.git
Initialized empty Git repository in /Users/brandon/developersplaybook/.git/
remote: Counting objects: 56, done.
remote: Compressing objects: 100% (53/53), done.
remote: Total 56 (delta 21), reused 0 (delta 0)
Receiving objects: 100% (56/56), 653.13 KiB | 799 KiB/s, done.
Resolving deltas: 100% (21/21), done.
```

Git initializes an empty repository in a directory of the same name as the repository we are cloning, and then downloads the objects into the directory. In the event that

we wanted to have the cloned repository in a different location, we could add an optional file path as a fourth argument.

Developers may also want to create an empty repository, or create a repository in a directory that already has files. This is an easy task to complete: developers need only to initialize the repository:

```
$ git init sample
Initialized empty Git repository in /Users/brandon/sample/.git/
```

This creates the directory and an empty Git repository. Alternatively, if you want to create a Git repository in the current working directory, simply call `git init` and the repository will be created on the present working directory.

Of course, Git assumes with the initialize command that the developer intends to actually place files inside the directory; thus it creates a hidden directory, called `.git`, and places the repository files inside of that. However, from time to time the developers may just wish to initialize a bare repository - one that will contain no files. This is useful in situations where the repository will simply be pushed to and pulled from. To do this, we use a flag:

```
$ git init   bare sample.git
Initialized empty Git repository in /Users/brandon/sample.git/
```

Note that the hidden `.git` directory was not created. Examining the file system will show us a layout which differs from our initialized repository.

Using a Remote Repository

Git is not as useful unless the information stored in it can be shared. To that end, developers have devised a system whereby commits can be "pushed" and "pulled" from remote sources, such as other developers, a central server setup similar to Subversion, or even a web service called Github. Whatever the case may be, working with remote repositories is a critical skill.

To work with a remote repository, it must first be added as a remote location. To do this, we use the `git remote` command:

```
$ git remote add origin /path/to/remote
```

The syntax here is pretty simple: we tell the `remote` command to `add` an alias called `origin` with the file path of `/path/to/remote` which is a remote repository.

Once we have made changes (more on how to make changes later on), we can "push" these changes to our remote repository. We use the `git push` command:

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 219 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /Users/brandon/remote
 * [new branch]      master  > master
```

The syntax here is also fairly simple: we direct Git to `push` to the `origin` alias everything in branch "master" and the output we get is a log of what Git is doing. It finishes by telling us that it created a new branch called "master" in the remote repository.

As developers work on their own repositories and push their changes, Git will eventually force us to pull those changes before we can push additional changes. We use a `git pull` request to do this:

```
$ git pull origin master
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /Users/brandon/remote
 * branch            master      > FETCH_HEAD
Updating bb2e6b4..78a9bd1
Fast forward
 example.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions( )
```

The syntax is exactly the same as the `push` request.

When creating the aliases like "origin" it is important to note that, while by convention, "origin" is the upstream repository alias name, you can name aliases anything. This is particularly useful if you have two or more repositories that you want

to push and pull from (say you are collaborating on a very small team without a central repository).

One of the major differences between Git and Subversion is that Git does not actually require you to pull in order to commit. It will allow you to diverge as far as you want. The only time you are forced to pull from a remote source is when you wish to push to a source that is behind yours or that you have diverged from.

Managing Files in the Local Repository

We have a blank repository with nothing in it. Now that we have made some changes to the file structure, we need to add those files to the repository in order to save the history or push those files to our master repository.

Git has four different statuses for files: untracked, unstaged, staged, and committed. Untracked files are files that are not part of the repository and will not be committed on the next commit. Unstaged are files that have changed since the last revision, but will not be committed unless we explicitly add them for the next commit. Git allows us to control our commits, adding certain files but not others. Staged files are files that will be committed during the next commit, and committed files are unchanged files that are stored in the repository.

Staging a file is a fairly easy process making use of the `git add` command:

```
$ git add example.txt
```

Committing works similarly to most other version control systems as well, using the `git commit` command:

```
$ git commit  am "Initial commit."
[master (root commit) bb2e6b4] Initial commit.
 0 files changed, 0 insertions(+), 0 deletions( )
 create mode 100644 example.txt
```

Note that we used the `-a` and `-m` flags. The `-a` flag would have committed any unstaged files automatically. The `-m` flag followed by a message stores a commit message in the system. Commits must have a message associated with them. If one is not specified, Git will use the default editor and direct you to enter a message.

From time to time we may want to see what is tracked, untracked and staged. We can do this using the `git status` command:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  modified:   example.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout    <file>..." to discard changes in working directory)
#
#  modified:   unstaged.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  untracked.txt''
```

Here we see that `example.txt` is staged for committing. `unstaged.txt` is not staged, but is tracked; it is listed as well. Finally, we have an untracked file which is listed under "Untracked files".

Git is also very helpful in its output here: it tells us exactly how to fix things if we do not want to commit or have a file changed. We can unstage files using `git reset HEAD <file>` and we can revert files by using `git checkout - <file>`.

Branching the Repository

Developers often work on more than one bug at a time, but do not necessarily want those two components interacting with one another until a later date. To better facilitate this, Git has a very powerful branching model that can be used to improve workflow. Branching is the process of directing the version control system to essentially duplicate the current file structure so that features can be added or removed without affecting the main line of development or other developers. Branches can be created on the fly, and do not necessarily get pushed up to the master repository; branches can be anonymous, or they can be included in commits.

To create a branch, users make use of the `git branch` command:

```
$ git branch testbranch
$ git branch
* master
  testbranch
```

Note that the syntax is to simply state the name of the branch we are creating. Git will form a branch off the branch we are currently using. Alternatively, we can list the branch we want to branch from as a second argument.

Without any arguments, the `git branch` command lists all branches available to us. The asterisk indicates the branch we are currently using.

Switching between branches is just as easy, using the `git checkout` command:

```
$ git checkout testbranch
Switched to branch 'testbranch'
```

We are now on the `testbranch` branch. We can make changes and commits against this branch.

Once we have made our changes and are satisfied, we will want to merge those changes back with our master branch (or with other branches). We can do this using the `git merge` command:

```
$ git merge testbranch
Updating dbb00cf..fa4022d
Fast forward
 example.txt |    2 +
 1 files changed, 1 insertions(+), 1 deletions( )
```

The syntax here is simple enough: we tell it which branch we want to merge from, and without an optional second argument telling which branch we are merging into, Git automatically merges the changes into the branch we are currently on.

Once we are finished with our branch, we can use the `git branch` command to remove it as well:

```
$ git branch  d testbranch
Deleted branch testbranch (was fa4022d).
```

The branch is removed by using the `-d` flag with the branch name.

Stashing Changes for Later

From time to time, we may not be ready to commit our changes to the repository quite yet, but we may need to pull in additional changes from other users, either from Subversion or from another repository. Or, we may be working on master and need to revert it back to a known state, while still preserving our changes for the future.

Git offers a function to do this called `stash` which allows us to stash our files in a temporary storage, and retrieve them later.

```
$ git stash
Saved working directory and index state WIP on master: 0adce2e Adding new
    content.
HEAD is now at 0adce2e Adding new content.
```

Git has now stashed the changes to the working copy and has reverted the directory back to the `0adce2e` commit. We can now freely make changes, and when we are ready we can see the stashes that already exist:

```
$ git stash list
stash@{0}: WIP on master: 0adce2e Adding new content.
```

This shows all the stashes we have available to us. There are a number of ways to access a stash: we can pop the stash or apply it. If we "pop" the stash we remove it from the list and apply the changes; if we simply "apply" the stash we apply the changes while leaving the stash in the stash list. Alternatively, we can also direct it to apply a particular stash (if we have more than one).

```
$ git stash pop
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout    <file>..." to discard changes in working directory)
#
#  modified:   example.txt
#
no changes added to commit (use "git add" and/or "git commit  a")
Dropped refs/stash@{0} (a79efc499ea8fd6cec59adb5acb4ae2caa768ef3)
```

This command applied the changes, then showed us a current status of the working directory and told us that the stash has been dropped (because we used `git stash pop`).

Looking at History

Developers often want to take a look at the commit logs or see who changed what in which revision. Git offers a powerful feature for seeing revision history in the `git log` command:

```
commit 0adce2e0c892950bdac39c91eddc6ef3f245cd00
Author: Brandon Savage <brandon@brandonsavage.net>
Date:   Sat May 1 08:14:23 2010  0400

   Adding new content.

commit fa4022d6c421cbcddba73fb982b0a2be8f237976
Author: Brandon Savage <brandon@brandonsavage.net>
Date:   Sat May 1 08:10:58 2010  0400

   Making some changes.

commit dbb00cf3947228349d6e1ea5f649dc2e003be0b2
Author: Brandon Savage <brandon@brandonsavage.net>
Date:   Sat May 1 08:06:21 2010  0400

   Removing unstaged.

commit d14d9078db6d55176a4dcf608cd8d093398ab68f
Author: Brandon Savage <brandon@brandonsavage.net>
Date:   Sat May 1 08:02:17 2010  0400

   Adding unstaged.txt for demonstration.
```

Here we have a few commits. Note some things that are different from Subversion: first and foremost, Git does not use sequential revision numbers. This is because Git is designed to be used collaboratively. Therefore, sequential revision numbers would increase the chances for revision number collisions. Git instead uses SHA1 hashes, which have nearly a zero chance of colliding.

This command takes a number of arguments that can be used to limit the search for commit history. The vast number of combinations makes it difficult to list everything here, but using the `git help log` command will show all the combinations for `git log`.

Git also contains a way to see who changed what in which revision on a line-by-line basis in the form of the `git annotate` command. This command shows line-by-line of a particular file and is useful for seeing when, or by whom, a bug was introduced.

```
$ git annotate example.txt
0adce2e0 (Brandon Savage 2010 05 01 08:14:23  0400 1)Adding some content
fa4022d6 (Brandon Savage 2010 05 01 08:10:58  0400 2)
0adce2e0 (Brandon Savage 2010 05 01 08:14:23  0400 3)And some additional content
```

Some important things of note here: note that Git is using only the first eight characters of the revision ID. This is because it is almost a given certainty that the first eight characters will be unique to any repository (in fact, in small repositories it is almost a certainty that the first four or five are). This is a useful shortcut for referring to commits, since developers can refer to the commit by the shortcut in each of Git's commands.

Rules of Version Control

Version control can be a double-edged sword, with benefits on one side but danger and difficulty on the other. Understanding the hard and fast rules of version control etiquette is essential to effectively using it as a member of a team, and enjoying the teamwork and collaboration that comes with version control.

These rules are general etiquette tips to help developers be successful at using version control with others.

Always Include a Useful Commit Message

A commit message of "added stuff" does not help anyone understand the work that was done. In fact, it is antithetical to the goals of version control: communicating changes over time to a group of individuals who did not make those changes.

Commit messages should be as descriptive as possible for understanding what was accomplished in a particular revision. A commit message that states "revised

`execute()` method to incorporate changes listed in Ticket #3271" is much more descriptive and provides a developer who is investigating a changeset a clear and concise understanding of the change that was made, and the reasons for it (which are probably in Ticket #3271).

Some companies enforce the rule of descriptive commit messages by forcing a linkage between a ticket and a commit. This is a good option for those who are concerned with whether or not commit messages will adequately describe the changes made.

Some tools are focused around the subject/body structure of writing commit messages (Git has many tools like this). Whatever the convention, enforced or expected, it is a great idea to be complete and comprehensive in a commit message and to follow the expected convention, in order to aid others.

Do Not Break the Build

A continual frustration for developers is to update their working copy, only to discover that the last developer to make a commit included a syntax or otherwise fatal error in the code, and then left for the day. This shuts down production until the issue can be resolved, and if the issue is related to something both developers were working on, the problem only gets worse from there.

Do not be the guy who "breaks the build" by including syntax errors, or not communicating clearly about the changes made that will affect everyone. Many companies institute pre-commit hooks that test for syntax and refuse to commit code that contains syntax errors, but it is still possible to break the build with database changes or syntactically-correct but incorrect code implementation.

Test your code before you commit it, and communicate well with your development team. Also, consider continuous integration, which when used properly can help alleviate some of these concerns (because it "builds" the code, and runs the unit tests against it right away).

Update Your Code Base Often from the Repository

If you are using Subversion, run `svn up` frequently. If you are using Git, use `git pull origin master` with some regularity to keep your code base up to date. This is critical, for two reasons:

First, it is important for you to have a good idea what other developers are doing. Changes to the code might just change how you develop particular features or design an aspect of the product. Second, failing to update regularly only heightens the likelihood that a conflict will occur later on.

Small conflicts are easy to deal with, but large conflicts are extremely hard to manage because they often involve large sections of the code and contain nuanced changes that require some time to understand. The worst conflict I ever dealt with was committing at the end of the project. The **entire file** was a conflict because I had not updated my code base consistently.

Never Rebase After Pushing

This tip is for Git users, but is so important it bears inclusion in the rules of version control section in its own right. Git contains a feature, called rebase, which can be exceptionally powerful and used to literally rewrite the commit history of the repository. This power allows developers to squash commits together, change the commit message, or even change the files in the commits. It also allows developers to combine changesets without merging. This can be useful for bringing a particular branch back to a certain state.

However, rebasing does not come without risk: in particular, when `git rebase` is run over a group of commits, *it rewrites those commits and the SHA1 commit ID*.

This might not seem like a big deal, and in many cases it is not, unless the commits have already been pushed to another repository. Then it becomes a big deal because Git will be confused by the new commit IDs, and it will create serious problems for merging changes further down the road.

Rebasing commits that have already been pushed is a serious mistake that will cause headaches to all the developers involved. Avoid it.

Do Not Simply Override Someone Else's Changes During a Conflict

From time to time, every developer experiences commits that conflict with their own work. This can be and sometimes is an extremely frustrating situation to fix, especially if the work done mimics or closely relates to the work being committed by the developer. However, while it is easy to take advantage of Subversion's option to simply override the conflict with your working copy, do not do this.

This might seem like an obvious point, but it is not. It is especially not obvious when a developer does a quick look to see what the conflict is and concludes that the conflict is a minor one that does not actually need resolution. However, developers are wise to skip resolving the conflict, and then resolve it using a text editor and reviewing the lines between the conflict markers.

This is a bigger issue with Subversion than with Git, which does not offer the ability to just override the incoming copy with your own. Instead, it forces developers to merge the two copies together, and then add the file for committing to resolve the conflict.

Diff Each File Before Committing It and Commit Frequently

Developers often forget that there can be situations where changes get made unintentionally - debugging code is added, verbosity is increased, or settings are changed. Before committing code, every developer should diff the files to be committed to ensure that the changes being sent are exactly what that developer wants to send.

This is of course more difficult the more files there are to be committed. This leads into the second part of the rule: commit frequently. Small, individual commits are almost always preferable to large commits. If developers are concerned with breaking functionality or going down the wrong path with their code changes, they should branch. That is what branching is for.

Some developers believe that code commits should be feature-complete. I disagree. Code commits should not break the application, but by no means need to be feature complete if being "feature complete" will take days or weeks. While developers should follow the rules against breaking the build they should commit small units of code that do not break the build regularly (even if the commit does not fully work).

Merge Frequently

Anyone who uses Subversion with any regularity has experienced one of the greatest nightmares in history: a large merge from one branch to the trunk. And while Subversion has gotten better at this over time, merging frequently still helps developers prevent the issue of a large, complicated merge.

Git does not suffer so much from this problem; its operational model is to make use of small, short-lived branches that are worked on and then merged quickly into master, to be destroyed. But Subversion, by virtue of the fact that the entire file system is copied over to a new directory, does not foster this regular merging. So it is important that developers do it purposely.

If a major feature or overhaul is being worked on, it is more than alright to merge trunk into the branch to add features currently being worked on in trunk to the branch. Once the branch is stable, this makes merging the final product back into trunk that much easier. Also, it is suggested that developers take very good notes about at which revisions they branch and merge. This will help alleviate some of the problems with Subversion's merging command.

Because Git is better at merging, for developers who intend to branch frequently, Git is a much better choice. However, Subversion does offer some branch support, so long as this rule is followed.

Guerrilla Version Control

When it comes to version control, everyone has an opinion, and they do not always line up. For cases when developers cannot use version control, or the version control system they desire, it might look hopeless. But it does not have to be hopeless: it is still possible to use version control.

Some Words for Developers Who Do Not Have Version Control

Sometimes, a manager cannot be convinced to make use of version control, or a team cannot be persuaded to take it seriously. In these cases, you as a developer do not need to abandon version control altogether; there are still distinct benefits that you can take advantage of, even if your team or management will not.

Version control by definition is designed to improve the performance of individuals, and even though it is often team focused, it is individuals who benefit the most from version control. This means that developers can use version control by themselves and without the involvement of their teams.

Probably the best suited VCS for this is Git. Because Git allows for creation of a local repository without the need for it to be hosted elsewhere, Git offers the lone developer the ability to make use of version control without anyone ever knowing they

are doing it. And because Git places a single hidden file in the root directory where the repository is initialized (versus Subversion's hidden `.svn` directories in every directory) it is easier to upload whole directories without worrying about accidentally adding a `.svn` directory.

Git also has a feature that makes it better suited to merging changes with others: the "stash" command. Git's ability to stash your current changes and then apply them later on gives developers working in a non-version-controlled environment a unique opportunity: the ability to mimic version control merges.

The way this works is that a developer stashes the changes and then downloads new copies of changed files worked on by others. Then, when they reapply the stashed files, Git automatically "merges" them and highlights conflicts. This helps keep changes from being overwritten.

The most important thing to remember about version control is that it gives a developer a history of changes, of past revisions, of the thinking process of those who used it. A single developer can make almost as effective use of version control as a team can.

Some Words for Developers Stuck on Subversion Wanting to Use Git

For several months, I used Subversion, but also used Git. How is this possible? Git has an optional add-on called `git-svn` which allows Git users to bridge the gap between Git and Subversion nearly seamlessly. I say "nearly seamlessly" because there are some scenarios where Git users can run into trouble. However, for those deeply committed to Git but forced to use Subversion, git-svn is a great option.

There are some precautions to be taken by git-svn users, though. The first is that git-svn users should avoid branch merges - these are merges where Git merges two branches and then makes a commit indicating that they have been merged. Additionally, git-svn users should avoid using a separate Git repository hosted elsewhere, especially since the process for incorporating Subversion changes into the Git repository is to rebase the Git commits - something that changes the SHA1 hashes, the commit messages, and breaks the rules of good Git collaboration.

The third danger in git-svn is the chance that there are problems with branching. As a general rule of thumb, once a branch is merged with the master, that branch should be deleted and a new branch started if a developer wishes to continue work

on a branch. This is to ensure that Subversion and Git stay current with each other. Obviously, this does take away some of the power of Git but it is a necessary evil for using git-svn. Finally, svn:externals do not work with git-svn; the solution here is to check out the external as a Subversion working copy, and then ignore that entire tree by editing the exclude file.

That said, git-svn does offer Subversion-bound developers the opportunity to make use of Git, if they prefer it.

Chapter 7

Refactoring Strategies

Code released into production is not always entirely finished, despite what many developers would like to believe. The maintenance component of the development cycle is often the longest and, many times, most important aspect of software development. A significant component of the maintenance cycle ought to be improving on the existing code, through the process of refactoring it into better, more easily maintainable code.

Introduction to Refactoring

Refactoring is one of those aspects of development that is often misunderstood. Many people believe that to successfully refactor an application, they must somehow change the way the application works. This is not necessarily true.

Refactoring is the process of making improvements to the code, whether they are performance-related, structure-related or just general improvements, that *do not alter the functionality of the product.* This is an important concept to understand: refactoring takes place solely in terms of the code, and does not involve the functionality or behavior of the application. Refactoring is transparent to the end user.

Why Refactor?

There are a number of reasons why refactoring is an important aspect of development. Many developers feel that refactoring is difficult, time consuming and pointless, yet it is a critical step, and every developer should practice it.

Refactoring is important because it costs less to improve the code than to rewrite it. Many developers look at old code that they have written and are surprised at how much they have advanced, even in a span of time as short as a few months. They may look at the last product and decide that they should stop and rewrite the product given their newer skills. But this is a fool's errand.

Instead of rewriting applications, refactoring gives us the opportunity to improve existing applications slowly, in iterations, over time. Because refactoring is focused on improving the code itself, rather than the functional behavior of the application, refactoring is a perfect opportunity to apply those new skills to old code.

Refactoring the existing code takes advantage of bug fixes, preserves the institutional memory embedded in that code, and can take advantage of the test suite for that application, which helps prevent bugs from being introduced by the refactoring. These protections often do not exist when applications are rewritten: rewritten code does not benefit from institutional memory, does not contain the bug fixes already highlighted in the code, and since the test suite is unproven, it is possible that there will be many bugs.

Refactoring is also important because it allows developers to make improvements as their understanding and knowledge improves. The reality is that for most developers, they grow and change. Their understanding of core principles and the practice of their art improves over time. They are liable to look at their code in the future and see problems with it, as well as areas for improvement.

Choosing to refactor gives developers an opportunity to bring these improvements to older products and code. It allows them not only to see areas where code can be improved, but to actually improve it. In turn, older applications continue to get better, which provides value to clients and users.

Along the same lines, developers that refactor often bring enhancements learned from other applications and projects, which in turn improves the overall quality of an application. For example, a particular algorithm or process for doing something with the database is often portable from one application to another. A developer's

ability to take those improvements and discoveries, and then to add them to other applications, improves everyone's experience overall.

Finally, refactoring often improves the structure and design of an application. Since the point of refactoring is to enhance the structure, functionality, performance and design of an application, as developers improve in these areas, the applications they refactor benefit from an improved understanding of these concepts and practices.

Things Developers Must Do Before Refactoring

Refactoring is an important tool in a PHP developer's toolkit, but it is not without its problems and potential hazards.

In order to refactor effectively, developers must pay attention to some "gotchas" about refactoring and do what they can to mitigate these areas.

The first and most obvious is that developers must understand what it is that the code is trying to accomplish before trying to change the way it functions. Otherwise, developers run the risk of breaking the implementation out of a misunderstanding of how the code works.

Developers also often overlook the need to have a comprehensive test suite built up for code they intend to refactor. This is essential, because without a comprehensive test suite there is no way for developers to know whether or not the refactoring they completed has broken the application. Many small bugs may only manifest themselves in test suites or edge cases within an application. Having a test suite that can highlight these edge cases is of great value to developers.

How to Refactor

Refactoring is not an exact science. Instead, it is an abstract art, one that developers consistently improve upon as they grow as developers. Thus, this chapter contains few hard and fast rules, and more of a discussion of strategies that will lead to improved code when refactoring.

This discussion requires a sample of code to be refactored, which will reappear throughout this chapter as refactoring takes place:

```
// class_Twitter.php   NPC TWITTER AUTO FEED
```

```
error_reporting(E_ALL);

// SEND AN AUTOMATED TWEET
// USAGE EXAMPLE
//   require_once('class_Twitter.php');
//   $status = 'Hello World';
//   $t      = new Twitter;
//   $t >tweet($status);
//   unset($t);

// DO NOT RUN THIS SCRIPT STANDALONE (HAS PASSWORD)
if (count(get_included_files()) < 2) {
   header("HTTP/1.1 301 Moved Permanently"); header("Location: /"); exit;
}

class Twitter {
   private $ch;
   private $you;
   private $user;
   private $pass;
   private $test;
   private $host;
   private $done; // ALREADY TWEETED?
   public  $res;

   public function __construct() {
      $this >user = "??? TWITTER USERNAME";
      $this >pass = "??? TWITTER PASSWORD";
      $this >you  = "??? YOUR EMAIL ADDRESS";

      $this >host = "http://twitter.com/";
      $this >done = FALSE; // DEFAULT   NOT ALREADY TWEETED
      $this >test = FALSE; // DEFAULT   THIS IS LIVE, NOT A TEST

      $this >ch   = curl_init();
      curl_setopt($this >ch, CURLOPT_VERBOSE, 1);
      curl_setopt($this >ch, CURLOPT_RETURNTRANSFER, 1);
      curl_setopt($this >ch, CURLOPT_USERPWD, "$this >user:$this >pass");
      curl_setopt($this >ch, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_1);
      curl_setopt($this >ch, CURLOPT_POST, 1);
   }

   public function __destruct() {
      curl_close($this >ch);
   }
```

```
// SET AN INDICATOR THAT THIS IS NOT A LIVE TWEET
public function test() {
   $this >test = TRUE;
}

// DETERMINE IF THE MESSAGE HAS ALREADY BEEN SEND
private function already_tweeted($message) {
   $text     = mysql_real_escape_string(trim($message));
   $date     = date('Y m d');
   $code     = md5($text . $date);
   $sql      = "SELECT id FROM twitterLog WHERE thash = \"$code\" ORDER BY id
        DESC LIMIT 1";
   if (!$res = mysql_query($sql)) { die(mysql_error()); }
   $num      = mysql_num_rows($res);
   if ($num) { return TRUE; }
   $sql      = "INSERT INTO twitterLog (tdate, thash, tweet) VALUES ( \"$date
       \", \"$code\", \"$text \" )";
   if (!$res = mysql_query($sql)) { die(mysql_error()); }
   return FALSE;
}

// POST A MESSAGE TO TWITTER
public function tweet($message) {
   if(strlen($message) < 1) { return FALSE; }
   if ($this >already_tweeted($message)) { $this >done = TRUE; }

   // IF ONLY A TEST, JUST EMAIL THE INFORMATION   DO NOT TWEET
   if ($this >test) {
      $msg = '';
      if ($this >done) { $msg .= "ALREADY DONE "; }
      $msg .= "TWEET: $message";
      mail($this >you, 'What We Would Have Tweeted', $msg);
      return TRUE;
   }

   // DO NOT REPEAT YOURSELF
   if ($this >done) { return TRUE; }

   // STATUS UPDATE ON TWITTER
   $this >host .= "statuses/update.xml?status=".urlencode( stripslashes(
       urldecode($message )));
   curl_setopt($this >ch, CURLOPT_URL, $this >host);
   $xxx       = curl_exec($this >ch);
   $this >res = curl_getinfo($this >ch);
   if ($this >res['http_code'] == 0) { return TRUE; }
   if ($this >res['http_code'] == 200) { return TRUE; }
```

```
            return FALSE;
        }
    }
```

This code sample code sample contains many things that we can improve upon. It is a great example of code that might have made its way into production, but should be refactored to more closely adhere to standards, best practices, and well understood principles.

Truth be told, this code also contains many problems: it lacks a cohesive coding standard, is poorly abstracted, is untestable, combines testing and production in the same code base, does not separate concerns, and contains some suspect logic. To refactor this, a number of areas must be improved and changed.

Developing a Coding Standard

The first thing that becomes immediately obvious in our code sample is that it is difficult to read. This is for a combination of two reasons: the first is that the code is complex, in fact needlessly complex (something we will address in a future section), and also because the code follows no consistent standard.

Coding standards are something that people argue about all the time. An anecdote from a friend was that whenever a developer on their team did not want to do any work for a week, they would nitpick about something in the coding standard, resulting in a week-long argument that would stall the production of code entirely. But coding standards need not be so acrimonious. In fact, let's redefine what we mean by "coding standard" here.

By "coding standard" in this section, we mean that the code follows a consistent standard - *regardless of what that standard actually is*. This is an important point to make: it does not matter whether a developer follows the PEAR convention or the Zend Framework style.

For the purposes of this section, it is important to implement three elements: consistency in the code, adding comments to improve our understanding, and preventing the introduction of bugs as we move forward.

Consistency is important for obvious reasons: having consistent code makes it easier to understand because the brain finds it easier to pick up consistent patterns than inconsistent ones. We will add some comments (Docblocks) in this code, for

two reasons: first because it improves our understanding of the API to at least document the arguments and the methods, and secondly because it forces us to go through the code and understand how it works.

This is an important aspect of refactoring. You must know what it is that you are working on before you can refactor it. Finally, we will go through and make sure that our coding conventions don not inadvertently add bugs, by ensuring things like bracketed conditionals and loops, appropriately-named functions, and that variables are initialized properly.

Starting from the top of the sample, we see that the developer is using something similar to the PEAR style, with a space after the word "if" and opening bracket on the same line. They are using four spaces, which we will use for indentation.

```
// DO NOT RUN THIS SCRIPT STANDALONE (HAS PASSWORD)
if (count(get_included_files()) < 2) {
   header("HTTP/1.1 301 Moved Permanently"); header("Location: /"); exit;
}
```

However, we note on Line 64 that the developer has included a single-line conditional. This violates the coding standard we seem to be using, and should be fixed. Furthermore, the fact that the conditional does not make use of brackets makes it easy to introduce a bug. For example:

```
if($var == true) {
   echo "PARTY!";
}
```

In the example, we will only echo the word "PARTY!" if the `$var` variable is equal to true. However, imagine that we are trying to debug the call, and insert something immediately preceding the echo command:

```
if($var == true) {
   echo "I made it here!";
   echo "PARTY!";
}
```

Whoops! Now we are going to have a party every time, regardless of whether or not the `$var` variable is true! This is because PHP allows only a single line to follow a conditional that is not wrapped in brackets. The same rule applies for loops. This means all conditionals and loops should always be surrounded by brackets:

```
if ($var == true) {
   echo "PARTY!";
}
```

In the case of our code sample, we need to do the same thing throughout. This problem appears on lines 64, 66, 68, 75, 76, 89, 96 and 97. Additionally, we have inconsistency with the conditional beginning on line 79, as the bracket is on a new line. Since we have decided to use a PEAR-like standard, we will need to fix that as well:

```
// class_Twitter.php   NPC TWITTER AUTO FEED
error_reporting(E_ALL);

// SEND AN AUTOMATED TWEET
// USAGE EXAMPLE
//   require_once('class_Twitter.php');
//   $status = 'Hello World';
//   $t      = new Twitter;
//   $t >tweet($status);
//   unset($t);

// DO NOT RUN THIS SCRIPT STANDALONE (HAS PASSWORD)
if (count(get_included_files()) < 2) {
   header("HTTP/1.1 301 Moved Permanently"); header("Location: /"); exit;
}

class Twitter {
   private $ch;
   private $you;
   private $user;
   private $pass;
   private $test;
   private $host;
   private $done; // ALREADY TWEETED?
   public  $res;

   public function __construct() {
      $this >user = "??? TWITTER USERNAME";
```

```
   $this >pass = "??? TWITTER PASSWORD";
   $this >you  = "??? YOUR EMAIL ADDRESS";

   $this >host = "http://twitter.com/";
   $this >done = FALSE; // DEFAULT   NOT ALREADY TWEETED
   $this >test = FALSE; // DEFAULT   THIS IS LIVE, NOT A TEST

   $this >ch   = curl_init();
   curl_setopt($this >ch, CURLOPT_VERBOSE, 1);
   curl_setopt($this >ch, CURLOPT_RETURNTRANSFER, 1);
   curl_setopt($this >ch, CURLOPT_USERPWD, "$this >user:$this >pass");
   curl_setopt($this >ch, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_1);
   curl_setopt($this >ch, CURLOPT_POST, 1);
}

public function __destruct() {
   curl_close($this >ch);
}

// SET AN INDICATOR THAT THIS IS NOT A LIVE TWEET
public function test() {
   $this >test = TRUE;
}

// DETERMINE IF THE MESSAGE HAS ALREADY BEEN SEND
private function already_tweeted($message) {
   $text     = mysql_real_escape_string(trim($message));
   $date     = date('Y m d');
   $code     = md5($text . $date);
   $sql      = "SELECT id FROM twitterLog WHERE thash = \"$code\" ORDER BY id
        DESC LIMIT 1";
   if (!$res = mysql_query($sql)) {
      die(mysql_error());
   }
   $num      = mysql_num_rows($res);
   if ($num) {
      return TRUE;
   }
   $sql      = "INSERT INTO twitterLog (tdate, thash, tweet) VALUES ( \"$date
       \", \"$code\", \"$text \" )";
   if (!$res = mysql_query($sql)) {
      die(mysql_error());
   }
   return FALSE;
}
```

```
    // POST A MESSAGE TO TWITTER
    public function tweet($message) {
       if(strlen($message) < 1) {
          return FALSE;
       }
       if ($this >already_tweeted($message)) {
          $this >done = TRUE;
       }

       // IF ONLY A TEST, JUST EMAIL THE INFORMATION   DO NOT TWEET
       if ($this >test) {
          $msg = '';
          if ($this >done) $msg .= "ALREADY DONE ";
          $msg .= "TWEET: $message";
          mail($this >you, 'What We Would Have Tweeted', $msg);
          return TRUE;
       }

       // DO NOT REPEAT YOURSELF
       if ($this >done) {
          return TRUE;
       }
       // STATUS UPDATE ON TWITTER
       $this >host .= "statuses/update.xml?status=".urlencode( stripslashes(
           urldecode($message)));
       curl_setopt($this >ch, CURLOPT_URL, $this >host);
       $xxx       = curl_exec($this >ch);
       $this >res = curl_getinfo($this >ch);
       if ($this >res['http_code'] == 0) {
          return TRUE;
       }
       if ($this >res['http_code'] == 200) {
          return TRUE;
       }
       return FALSE;
    }
}
```

Great. Now we have added some consistency to our application, and removed any conditionals or loops that do not automatically have brackets. The next step is adding comments to our code so that we better understand its behavior and function.

There is a great amount of debate as to whether or not comments make a whole lot of sense. I think they help tremendously. If comments have not been written,

writing them forces a developer to understand the code they are working with, which is a crucial component to understanding and properly refactoring a code segment. I tend to use Docblocks (which I am using here) which differ from in-line comments, and document each method rather than documenting the code as it runs. Here's the same code, now documented with Docblocks:

```
// class_Twitter.php   NPC TWITTER AUTO FEED
error_reporting(E_ALL);

// DO NOT RUN THIS SCRIPT STANDALONE (HAS PASSWORD)
if (count(get_included_files()) < 2) {
   header("HTTP/1.1 301 Moved Permanently"); header("Location: /"); exit;
}

/**
 * Twitter Class
 * @author Anonymous
 *
 */
class Twitter {
   /**
    * @var resource Curl connection
    */
   private $ch;

   /**
    * @var string Email address for administrator/tester
    */
   private $you;

   /**
    * @var string The username for the Twitter account.
    */
   private $user;

   /**
    * @var string The password for the Twitter account.
    */
   private $pass;

   /**
    * @var bool The flag for test mode
    */
   private $test;
```

```
/**
 * @var string The URL for the API call
 */
private $host;

/**
 * @var bool Flag for whether or not this tweet has been sent today
 */
private $done;

/**
 * Object constructor.
 */
public function __construct() {
   $this >user = "??? TWITTER USERNAME";
   $this >pass = "??? TWITTER PASSWORD";
   $this >you  = "??? YOUR EMAIL ADDRESS";

   $this >host = "http://twitter.com/";
   $this >done = FALSE; // DEFAULT   NOT ALREADY TWEETED
   $this >test = FALSE; // DEFAULT   THIS IS LIVE, NOT A TEST

   $this >ch   = curl_init();
   curl_setopt($this >ch, CURLOPT_VERBOSE, 1);
   curl_setopt($this >ch, CURLOPT_RETURNTRANSFER, 1);
   curl_setopt($this >ch, CURLOPT_USERPWD, "$this >user:$this >pass");
   curl_setopt($this >ch, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_1);
   curl_setopt($this >ch, CURLOPT_POST, 1);
}

/**
 * A destructor.
 * @return void
 */
public function __destruct() {
   curl_close($this >ch);
}

/**
 * Method sets a flag as to whether the object is in test mode
 * @return void
 */
public function test() {
   $this >test = TRUE;
}
```

```
/**
 * Check to see if this tweet has already been posted and add it to the DB
 * if it has.
 * @param string $message The tweet
 * @return bool
 */
private function already_tweeted($message) {
   $text     = mysql_real_escape_string(trim($message));
   $date     = date('Y m d');
   $code     = md5($text . $date);
   $sql      = "SELECT id FROM twitterLog WHERE thash = \"$code\" ORDER BY id
        DESC LIMIT 1";
   if (!$res = mysql_query($sql)) {
      die(mysql_error());
   }
   $num      = mysql_num_rows($res);
   if ($num)  {
      return TRUE;
   }
   $sql      = "INSERT INTO twitterLog (tdate, thash, tweet) VALUES ( \"$date
       \", \"$code\", \"$text \" )";
   if (!$res = mysql_query($sql)) {
      die(mysql_error());
   }
   return FALSE;
}

/**
 * A public method to post a tweet. If in test mode, this method will email
 * the tweet instead of posting it.
 * @param string $message The tweet to be posted.
 * @return bool
 */
public function tweet($message) {
   if(strlen($message) < 1)  {
      return FALSE;
   }
   if ($this >already_tweeted($message)) {
      $this >done = TRUE;
   }

   // IF ONLY A TEST, JUST EMAIL THE INFORMATION   DO NOT TWEET
   if ($this >test) {
      $msg = '';
      if ($this >done) {
```

```
            $msg .= "ALREADY DONE ";
         }
         $msg .= "TWEET: $message";
         mail($this >you, 'What We Would Have Tweeted', $msg);
         return TRUE;
      }

      // DO NOT REPEAT YOURSELF
      if ($this >done) {
         return TRUE;
      }

      // STATUS UPDATE ON TWITTER
      $this >host .= "statuses/update.xml?status=".urlencode(stripslashes(
          urldecode($message)));
      curl_setopt($this >ch, CURLOPT_URL, $this >host);
      $xxx       = curl_exec($this >ch);
      $this >res = curl_getinfo($this >ch);
      if (($this >res['http_code'] == 0) || ($this >res['http_code'] == 200)) {
         return TRUE;
      }
      return FALSE;
   }
}
```

Note that we also indented our in-line comments to follow the indenting style of the rest of the page. This is important because it helps to preserve the consistency in the patterns our brain recognizes.

We have not done much to improve the code yet, only the way it looks. This important step should not take too long (and can often be done automatically with new IDEs), but is a crucial step to improving the readability and maintainability of the code. Next, we will move on to actually changing the code to make it better.

Refactoring for Testability

Earlier in the chapter we discussed that refactoring should only take place with a robust test suite already built and designed to test the application being refactored. However, it is clear from the our example that there are no obvious unit tests. This is for a few reasons, but the biggest one is that this code is largely untestable.

Untestable code is code that is written in such a way as to be difficult if not impossible to effectively develop unit tests against it. But since code that cannot be tested

is asking for bugs to crop up, it is imperative that we refactor the code just enough to allow for some unit testing to take place.

The first thing that pops up as a confrontation to testability is the first seven lines of the application:

```
// class_Twitter.php   NPC TWITTER AUTO FEED
error_reporting(E_ALL);

// DO NOT RUN THIS SCRIPT STANDALONE (HAS PASSWORD)
if (count(get_included_files()) < 2) {
   header("HTTP/1.1 301 Moved Permanently"); header("Location: /"); exit;
}
```

These lines are intended, ostensibly, to prevent access to this class from the web browser. However, classes should generally be placed outside of the web root, meaning that these settings are meaningless. Furthermore, the `error reporting(E ALL)` should be handled in an INI setting, rather than at the script level. So, we can safely remove these lines of code from our example.

Next, we note that the example code has a "test mode" - this can theoretically be used to conduct a "dry run" against the sample. That being said, testing should be done at the unit test level, not in our example. If we want to have a "dry run" mode, we should mock an object rather than rely on settings to the object to conduct the dry run.

It is worth pointing out that the author had good intentions when they wrote this component into their application. The developer wanted to avoid a scenario where they would inadvertently post to a Twitter account during functional testing, certainly a noble goal. But as we will examine in the next section, there are far better ways to mock up an HTTP connection and avoid this type of inadvertent posting. It is safe to remove this behavior from the class.

The example also has a problem in terms of how the object is created, which will hinder testing. In particular, this class is designed to have hard-coded credentials for Twitter and the user's email address. This makes our constructor very untestable because we cannot easily alter the credentials that we are passing without overriding the entire constructor, and then we are testing a mock object rather than a real one.

Rather than using hard-coded credentials, we should pass our constructor some parameters. This will improve the flexibility of this object, and allow us to test it more effectively.

```
/**
 * Constructor.
 * @param string $username The username of the Twitter user
 * @param string $password The password of the Twitter user
 * @param string $email Optional email address for testing
 */
public function __construct($username, $password, $email = null) {
   $this >user = $username;
   $this >pass = $password;
   $this >you  = $email;

   $this >host = "http://twitter.com/";
   $this >done = FALSE; // DEFAULT   NOT ALREADY TWEETED
   $this >test = FALSE; // DEFAULT   THIS IS LIVE, NOT A TEST

   $this >ch   = curl_init();
   curl_setopt($this >ch, CURLOPT_VERBOSE, 1);
   curl_setopt($this >ch, CURLOPT_RETURNTRANSFER, 1);
   curl_setopt($this >ch, CURLOPT_USERPWD, "$this >user:$this >pass");
   curl_setopt($this >ch, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_1);
   curl_setopt($this >ch, CURLOPT_POST, 1);
}
```

The last thing of note is that the author of this class has decided to make all the properties and the `already tweeted()` method private. Private methods and properties cannot be accessed directly by subclasses, making the process of creating mock objects much more difficult. In addition, in situations where extension is necessary (say, to access certain Twitter API functionality that is needed in one application but not in another), private methods and properties make this difficult.

In order to improve the testability of the object, we will instead change the properties and methods to "protected". Because protected methods can be accessed by subclasses (though not from outside the object), this improves the overall ability of our object to be tested and extended later on.

We have done a lot to improve the testability of this object. While we will not show the unit tests that could be written (we will assume that they were), you can see the progress we have made thus far:

```
/**
 * Twitter Class
 * @author Anonymous
 *
 */
class Twitter {
   /**
    * @var resource Curl connection
    */
   protected $ch;

   /**
    * @var string Email address for administrator/tester
    */
   protected $you;

   /**
    * @var string The username for the Twitter account.
    */
   protected $user;

   /**
    * @var string The password for the Twitter account.
    */
   protected $pass;

   /**
    * @var string The URL for the API call
    */
   protected $host;

   /**
    * @var bool Flag for whether or not this tweet has been sent today
    */
   protected $done;

   /**
    * Constructor.
    * @param string $username The username of the Twitter user
    * @param string $password The password of the Twitter user
    * @param string $email Optional email address for testing
    */
   public function __construct($username, $password, $email = null) {
      $this >user = $username;
      $this >pass = $password;
      $this >you  = $email;
```

```
        $this >host = "http://twitter.com/";
        $this >done = FALSE; // DEFAULT   NOT ALREADY TWEETED

        $this >ch   = curl_init();
        curl_setopt($this >ch, CURLOPT_VERBOSE, 1);
        curl_setopt($this >ch, CURLOPT_RETURNTRANSFER, 1);
        curl_setopt($this >ch, CURLOPT_USERPWD, "$this >user:$this >pass");
        curl_setopt($this >ch, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_1);
        curl_setopt($this >ch, CURLOPT_POST, 1);
    }

    /**
     * A destructor.
     * @return void
     */
    public function __destruct() {
        curl_close($this >ch);
    }

    /**
     * Check to see if this tweet has already been posted and add it to the DB
     * if it has.
     * @param string $message The tweet
     * @return bool
     */
    protected function already_tweeted($message) {
        $text     = mysql_real_escape_string(trim($message));
        $date     = date('Y m d');
        $code     = md5($text . $date);
        $sql      = "SELECT id FROM twitterLog WHERE thash = \"$code\" ORDER BY id
             DESC LIMIT 1";
        if (!$res = mysql_query($sql)) {
            die(mysql_error());
        }
        $num      = mysql_num_rows($res);
        if ($num)  {
            return TRUE;
        }
        $sql      = "INSERT INTO twitterLog (tdate, thash, tweet) VALUES ( \"$date
             \", \"$code\", \"$text \" )";
        if (!$res = mysql_query($sql)) {
            die(mysql_error());
        }
        return FALSE;
    }
```

```
    /**
     * A public method to post a tweet.
     * @param string $message The tweet to be posted.
     * @return bool
     */
    public function tweet($message) {
       if(strlen($message) < 1)  {
          return FALSE;
       }
       if ($this >already_tweeted($message)) {
          $this >done = TRUE;
       }

       // DO NOT REPEAT YOURSELF
       if ($this >done) {
          return TRUE;
       }

       // STATUS UPDATE ON TWITTER
       $this >host .= "statuses/update.xml?status=".urlencode(stripslashes(
            urldecode($message)));
       curl_setopt($this >ch, CURLOPT_URL, $this >host);
       $xxx       = curl_exec($this >ch);
       $this >res = curl_getinfo($this >ch);
       if (($this >res['http_code'] == 0) || ($this >res['http_code'] == 200)) {
          return TRUE;
       }
       return FALSE;
    }
}
```

Now, let's take a look at abstracting the code to improve reusability.

Refactoring for Abstraction

We have done a lot of work already, just in restructuring the coding standards and making the application more testable (and theoretically writing unit tests). Now that we have accomplished these steps, we can get into the heart of the issue: refactoring.

A common mistake that many developers make in their applications is asking a particular object or function to do too much. Objects should be designed to accomplish a particular task, but no object should have more than a single task to complete.

Why is this true? When objects have limited responsibilities, it is easier to reuse, extend, modify and change them to fit the needs of a particular implementation.

Looking back at the code we are working on, it is easy to see that this application has several situations where it is doing far too much. This object is responsible for a core behavior - posting a message to Twitter - but it is also responsible for a few other things. It is responsible for opening, maintaining and handling a cURL connection to the Twitter website, and it is also responsible for opening, maintaining and executing database tasks.

To make matters worse, this object is bound to using MySQL as its data layer, and cURL as its HTTP client. This is because these items are hard-coded in. Rather than using some kind of abstraction, we have assumed and decided that in fact the server executing this application *must* use MySQL and cURL in order to use this object. This is severely limiting, and would require that the code be rewritten in order to use Postgres or any other database, or another HTTP client.

Right away, we have opportunities to refactor the application to improve its abstraction. A properly abstracted object will pass off the implementation of the HTTP connection and database to other objects, but this w will not be an easy task. It will require refactoring a large part of the application. Let's get started with the constructor.

Our primary task here is to modify the way information is given to this object when the object is created. In order to properly handle the HTTP and database connections, we will want to pass these objects into our object, through a process known as *dependency injection*. Dependency injection is the process of passing dependencies (other objects that an object relies on) into the object at creation, or before, using the object for its intended purpose. These dependencies allow for us to create mock objects, which in turn improves testability and allows us to more adequately test objects free of true dependencies.

We are going to pass two objects into this object: an HTTP connection object (we'll assume the object is named HTTPCon) and a database connection object (we'll use the PDO object).

```
/**
 * Constructor.
 * @param HTTPCon $http The HTTPCon object
 * @param DBCon $db The database connection object.
```

```
     * @param string $username The username of the Twitter user
     * @param string $password The password of the Twitter user
     * @param string $email Optional email address for testing
     */
    public function __construct(HTTPCon $http, PDO $db, $username, $password, $email
          = null) {
       $this >user = $username;
       $this >pass = $password;
       $this >you  = $email;

       $this >initializeHTTP($http);
       $this >db = $db;

       $this >host = "http://twitter.com/";
       $this >done = FALSE; // DEFAULT   NOT ALREADY TWEETED
    }

    /**
     * Method to initialize and set up the HTTP object properly.
     *
     * @param HTTPCon $http The object to be initialized.
     */
    public function initializeHTTP(HTTPCon $http) {
       $http >setUsername($this >user);
       $http >setPassword($this >pass);
       $http >setHttpVersion(HTTPCon::HTTP_11);
       $http >setHttpRequestType(HTTPCon::REQUEST_POST);
       $this >http = $http;
    }
```

We have made several changes here, all of which are important. First and foremost, we introduced PDO instead of the MySQL API, which means that we have the ability to use any kind of database we like (so long as the SQL is compatible between them). Next, we have introduced an `HTTPCon` object, which is to be used to make our connection to Twitter and return data. More importantly, we have implemented a new method called `initializeHTTP()`.

This initialize method is designed to reduce the workload on the constructor and also allow for us to pass a new HTTP object later on, if we wish to replace the existing one. This may be important if we wish to reuse the object, or want to pass a mock object for testing purposes later on.

Most importantly, we have abstracted out two large components to other objects that are better suited to handle them, and can be reused throughout the application

should we need them elsewhere. This is an important concept, because it reduces the overall code we use while encouraging reuse.

Also of note is that, since we removed the cURL connection, we also removed the `destruct()` method from the class. This method is often unnecessary; as PHP will automatically garbage collect all objects, resources and variables at the conclusion of a request.

Obviously by changing the database connection, we will also need to refactor the database connection method.

```
/**
 * Check to see if this tweet has already been posted and add it to the DB
 * if it has.
 * @param string $message The tweet
 * @return bool
 */
protected function already_tweeted($message) {
   $pdo = $this >db;
   $statement = $pdo >prepare('SELECT id FROM twitterLog WHERE thash = ? ORDER
       BY id LIMIT 1');
   $hash = md5($message . date('Y m d'));
   try {
      $statement >execute(array($hash));
   } catch (PDOException $e) {
      throw $e;
   }

   if ($statement >rowCount())  {
      return TRUE;
   }

   $insert = $pdo >prepare('INSERT INTO twitterLog (tdate, thash, tweet)
                            VALUES (:date, :code, :text)');
   $params = array(':date' => date('Y m d'), ':code' => $hash, ':text' =>
       $message);
   try {
      $insert >execute($params);
   } catch (PDOException $e) {
      throw $e;
   }

   return FALSE;
}
```

There are a lot of important changes in this section of the code. The API has been ported from the MySQL API over to the PDO API, making the code database agnostic. More importantly, we have further protected ourselves against SQL injection because we are using the PDO prepare options, which improves our security. Finally, we have removed the `die()` construct, opting instead to use PDO's built-in exceptions for control. In this case we have simply caught and rethrown the exception, which we will resolve in a later component.

This is a good point at which to discuss one of the hard and fast rules of refactoring: remove `die()` statements from production code (`exit()` calls too). Calls that terminate the execution of your script, and worse, output error messages, have no place in a production-quality application. Halting execution is a useful tool, but one that places you and your users at a tremendous disadvantage when things go wrong.

Exceptions are far better for use in object-oriented programming. Graceful degradation in procedural applications (like error handlers) is a much more appropriate way to handle problems than a sudden stop with a cryptic error message. Errors should never be displayed to the end user.

The last component in need of refactoring is the part that makes the HTTP request to Twitter, because we modified the API we are using. This is a fairly straightforward refactoring:

```
/**
 * Twitter Class
 * @author Anonymous
 *
 */
class Twitter {
   /**
    * @var HTTPCon HTTP object
    */
   protected $http;

   /**
    * @var string Email address for administrator/tester
    */
   protected $you;

   /**
    * @var string The username for the Twitter account.
    */
```

```
protected $user;

/**
 * @var string The password for the Twitter account.
 */
protected $pass;

/**
 * @var string The URL for the API call
 */
protected $host;

/**
 * @var bool Flag for whether or not this tweet has been sent today
 */
protected $done;

/**
 * @var PDO The database connection object.
 */
 protected $db;

/**
 * Constructor.
 * @param HTTPCon $http The HTTPCon object
 * @param DBCon $db The database connection object.
 * @param string $username The username of the Twitter user
 * @param string $password The password of the Twitter user
 * @param string $email Optional email address for testing
 */
public function __construct(HTTPCon $http, PDO $db, $username, $password,
    $email = null) {
   $this >user = $username;
   $this >pass = $password;
   $this >you  = $email;

   $this >initializeHTTP($http);
   $this >db = $db;

   $this >host = "http://twitter.com/";
   $this >done = FALSE; // DEFAULT   NOT ALREADY TWEETED
}

/**
 * Method to initialize and set up the HTTP object properly.
 *
```

```
 * @param HTTPCon $http The object to be initialized.
 */
public function initializeHTTP(HTTPCon $http) {
   $http >setUsername($this >user);
   $http >setPassword($this >pass);
   $http >setHttpVersion(HTTPCon::HTTP_11);
   $http >setHttpRequestType(HTTPCon::REQUEST_POST);
   $this >http = $http;
}

/**
 * Check to see if this tweet has already been posted and add it to the DB
 * if it has.
 * @param string $message The tweet
 * @return bool
 */
protected function already_tweeted($message) {
   $pdo = $this >db;
   $statement = $pdo >prepare('SELECT id FROM twitterLog WHERE thash = ?
       ORDER BY id LIMIT 1');
   $hash = md5($message . date('Y m d'));
   try {
   $statement >execute(array($hash));
   } catch (PDOException $e) {
      throw $e;
   }

   if ($statement >rowCount())  {
      return TRUE;
   }

   $insert = $pdo >prepare('INSERT INTO twitterLog (tdate, thash, tweet)
       VALUES (:date, :code, :text)');
   $params = array(':date' => date('Y m d'), ':code' => $hash, ':text' =>
       $message);
   try {
      $insert >execute($params);
   } catch (PDOException $e) {
      throw $e;
   }

   return FALSE;
}

/**
 * A public method to post a tweet.
```

```
     * @param string $message The tweet to be posted.
     * @return bool
     */
    public function tweet($message) {
       if(strlen($message) < 1)  {
          return FALSE;
       }
       if ($this >already_tweeted($message)) {
          $this >done = TRUE;
       }

       // DO NOT REPEAT YOURSELF
       if ($this >done) {
          return TRUE;
       }

       // STATUS UPDATE ON TWITTER
       $this >host .= "statuses/update.xml?status=".urlencode(stripslashes(
           urldecode($message)));
       $this >http >setRequestUrl($this >host);
       $xxx       = $this >http >executeRequest();
       $this >res = $this >http >getResponse();
       if (($this >res['http_code'] == 0) || ($this >res['http_code'] == 200)) {
          return TRUE;
       }
       return FALSE;
    }
}
```

Refactoring for Logic

Our class is more properly abstracted, but how robust is the logic used within the methods? Once an object has been properly abstracted, it is easy to see and refactor the logic in the methods, to improve their robustness and reduce the chance of errors, while improving error handling when errors do crop up.

There are a number of logical problems with this application. For example, the assumption is made that the object cannot be reused once a tweet has been sent. In fact, if you were to try, the code designed to append the URL would break.

Also, we insert the tweet into the database but we do that prior to the tweet being successfully sent. There is no rollback, and we are not using transactions here.

The bottom line being that if there is a Twitter connection problem, or the service is down, we cannot send the same message today.

The database code actually has a bit of a problem, too. What happens if the date changes from today to tomorrow in the process of sending the tweet? Surely this is an edge case, but we do not take that into account; it is possible to repeat ourselves under limited circumstances.

Finally, we are passing the user's email address to the constructor, but we do not actually use it anywhere. Its presence is a vestige of the time and place where we had a dry run scheme in place (we would email the tweet instead of post it to Twitter).

As we refactor for logic, we can remove the `$you` property since we no longer need it. Additionally, we can remove the `$done` property, since we want to be able to reuse this object once it has completed its task. This means refactoring the `tweet()` method:

```
/**
 * A public method to post a tweet.
 * @param string $message The tweet to be posted.
 * @return bool
 */
public function tweet($message) {
   if (strlen($message) < 1)  {
      return FALSE;
   }
   if ($this >already_tweeted($message)) {
      return TRUE;
   }

   // STATUS UPDATE ON TWITTER
   $this >host .= "statuses/update.xml?status=".urlencode(stripslashes(urldecode
       ($message)));
   $this >http >setRequestUrl($this >host);
   $xxx       = $this >http >executeRequest();
   $this >res = $this >http >getResponse();
   if (($this >res['http_code'] == 0) || ($this >res['http_code'] == 200)) {
      return TRUE;
   }
   return FALSE;
}
```

Note that we have removed the `$done` property, and refactored the logic so that if the tweet has been sent, we do not resend it.

Let's also consider the purpose of storing data in the database: it is designed to prevent us from sending the same message we just sent by mistake. This is a valid goal, and certainly one worth writing some protections against. However, this can be done without a database connection. Instead, we will add a property called `$lastTweet` that will store the last tweet's MD5 hash; this will ensure that the next tweet is not the same as the last one.

We are also going to want to store the last tweet in the database, to record what messages are sent using our application. This can be for audit purposes or simply historical context. However, we will want to do this after we have successfully sent the tweet out. The nature of the HTTP protocol prevents us from using transactions effectively in this case. We are going to have to hope that the database is available once the tweet has been posted, but the risk here is lower than storing it beforehand.

We also want to refactor the logic we are using to construct the URL. By doing so, we ensure that the item is reusable. Since the Twitter URL will never change, we will add a class constant called `Twitter::TLD`, which is equivalent to the Twitter URL we want to use. This means refactoring out the `$host` property and its assignment in the constructor.

There is also a logic problem with testing the response which is that the HTTP status code it tests for can either be 200 or 0. This is incorrect, as a properly-formed HTTP response should contain some kind of status code. Therefore, we will test for that. Also, we will remove the logic that made the response a property, and instead make it a local variable within the `tweet()` method.

```
/**
 * A public method to post a tweet.
 * @param string $message The tweet to be posted.
 * @return bool
 */
public function tweet($message) {
   if (strlen($message) < 1)  {
      return FALSE;
   }

   // STATUS UPDATE ON TWITTER
   $url = self::TLD;
```

```
    $url .= 'statuses/update.xml?status=' . urlencode($message);
    $this >http >setRequestUrl($url);
    $this >http >executeREquest();
    $httpResponse = $this >http >getResponse();
    if ($this >res['http_code'] == 200) {
       $this >lastTweet = md5($message);
       $this >recordTweet($message);
       return TRUE;
    }
    return FALSE;
}
```

The last logical issue we need to resolve concerns the responses issued by the `tweet()` method. There are three possible response types: success (the tweet was posted), failure (the tweet was not posted), or improper parameters. This is solved by adding an exception to the `strlen()` check. Additionally, we will do `strlen()` once, but test to make sure the string is between 2 and 143 characters, because Twitter has a maximum number of characters.

```
$length = strlen($message);
if ($length < 1 || $length > 140)  {
   throw new Exception('The length of the message must be between 1 and 140; ' .
        $length . ' given.');
}
```

Take a moment to compare the previous example of our Twitter code with the following:

```
/**
 * Twitter Class
 * @author Anonymous
 *
 */
class Twitter {
   const TLD = 'https://www.twitter.com/';

   /**
    * @var HTTPCon HTTP object
    */
   protected $http;
```

```
/**
 * @var string The username for the Twitter account.
 */
protected $user;

/**
 * @var string The password for the Twitter account.
 */
protected $pass;

/**
 * @var PDO The database connection object.
 */
protected $db;

/**
 * @var string The last tweet that was sent.
 */
protected $lastTweet

/**
 * Constructor.
 * @param HTTPCon $http The HTTPCon object
 * @param DBCon $db The database connection object.
 * @param string $username The username of the Twitter user
 * @param string $password The password of the Twitter user
 * @param string $email Optional email address for testing
 */
public function __construct(HTTPCon $http, PDO $db, $username, $password) {
   $this >user = $username;
   $this >pass = $password;

   $this >initializeHTTP($http);
   $this >db = $db;
}

/**
 * Method to initialize and set up the HTTP object properly.
 *
 * @param HTTPCon $http The object to be initialized.
 */
public function initializeHTTP(HTTPCon $http) {
   $http >setUsername($this >user);
   $http >setPassword($this >pass);
   $http >setHttpVersion(HTTPCon::HTTP_11);
   $http >setHttpRequestType(HTTPCon::REQUEST_POST);
```

```
   $this >http = $http;
}

/**
 * Record a tweet in the database for retrieval or audit at a later date.
 * @param string $message The tweet
 * @return bool
 */
protected function recordTweet($message) {
   $pdo = $this >db;

   $insert = $pdo >prepare('INSERT INTO twitterLog (tdate, thash, tweet)
                            VALUES (:date, :code, :text)');
   $params = array(':date' => date('Y m d'), ':code' => $this >lastTweet, ':
       text' => $message);
      try {
         $insert >execute($params);
      } catch (PDOException $e) {
         throw $e;
      }

   return FALSE;
}

/**
 * A public method to post a tweet.
 * @param string $message The tweet to be posted.
 * @throws Exception
 * @return bool
 */
public function tweet($message) {
   $length = strlen($message);
   if($length < 1 || $length > 140)  {
      throw new Exception('The length of the message must be between 1 and
          140; ' . $length . ' given.');
   }

   // STATUS UPDATE ON TWITTER
   $url = self::TLD;
   $url .= 'statuses/update.xml?status=' . urlencode($message);
   $this >http >setRequestUrl($url);
   $this >http >executeRequest();
   $httpResponse = $this >http >getResponse();
   if ($this >res['http_code'] == 200) {
      $this >lastTweet = md5($message);
      $this >recordTweet($message);
```

```
            return TRUE;
        }
        return FALSE;
    }
}
```

They do not look similar to each other at all. This is the art of refactoring. By refactoring the code, we have improved it overall, while still keeping the functionality basically the same. We have added new features, removed unnecessary logic, improved the overall logic, created tests, made the code testable, and ensured that the code will be usable for a long time, and reusable in other projects.

Refactoring's power comes from the fact that it changes the underlying code without changing the functionality, feature set, or usability of the code to any significant extent. While some new features may be added (like the ability to configure the HTTP object, for example), for the most part the outcome remains the same. But this object will be better off for the work done, and developers who employ refactoring as a regular tool will experience great benefits.

Chapter 8

Worst Practices

First, let's get these out of the way:

- Spaghetti code
- References
- Too much OOP
- 80/20 rule micro optimization

Developers tend to focus on "best practices" as a way to help improve development and get everyone on the same page. These best practices are a great way to highlight development practices that should be employed. But what practices are explicitly bad for development?

This chapter focuses on these practices and highlights the ways in which they are both bad and can be improved upon. We focus on the reasons why security should be a part of your application from the ground up, talk about how many developers spend too much time coding and not enough time designing, expose not-invented-here syndrome and talk about style guides for applications.

Thinking Security Is for When an Application is Finished

"I'll just finish writing the application, then I'll make it secure." Sound familiar? This is a common thread in development for many. Security is secondary in their minds to

getting a working prototype assembled and demonstrated. This can be for a variety of reasons: they do not care to focus on security, or the deadline for the prototype is extremely tight, or they simply do not understand the value that security adds to the picture.

Developing security last, though, is a dangerous practice because it undermines the entire concept of security. It ensures that when you go through your application to "add" security later, inevitably you are going to miss something. You might miss adding security at the database layer or at the view layer or somewhere else in your application. These are poor security practices that can and should be avoided.

Security should be included from the architecture phase all the way up through the demonstration phase. Good developers might even include security as one of the items demoed to a client or to an internal team before an application can be declared to be "ready" for production.

When writing individual portions of the application, a good deal of time should be spent determining what security must be included and how to implement that security.

During the architecture phase, an application developer should determine what security measures will be necessary, and how they will work together to provide overall security for the application. It is at this stage that one can determine when and where to place the security checks, which security checks to enforce, and how best to protect the application from malicious users.

When designing the database layer, protections against SQL injection should be undertaken by the development team. A consideration of where data is coming from and how it is being used should also play into the decisions made at this stage, whether the data is coming from internal (generally safer) sources, or is user-supplied (and wholly unsafe). These decisions will largely rest on the architectural design supplied in the first phases of development.

Moving on to the user interface development, there are two unique protections that should be undertaken here: what Chris Shiflett calls "Filter Input, Escape Output". First, any input submitted by the user must be filtered for a specific set of criteria that is unacceptable - HTML, injection possibilities (usually handled at the database layer), invalid data (alpha characters where numeric characters are expected, non-emails where email addresses are required), and other checks.

These checks should always be done on the server side (despite the popularity of JavaScript-based validation, which can be used to provide immediate feedback if so desired). Next, it is critical to escape output - anything that was not screened for or filtered out should be rendered harmless at this point. This means converting any missed HTML to harmless HTML, ensuring that XSS attacks are not possible, and neutralizing other types of attacks.

There are also areas that are often overlooked that should be addressed. One of those areas is in session protection. It is critical to design sessions to be protected against session hijacking, by verifying sessions, changing session IDs after permissions elevation, considering checking User-Agent headers, and other standard protections.

Additionally, standard practices like requiring a password to change sensitive settings or the password itself, and using HTTPS connections for security-related data are essential components of your security model.

It should be obvious by this point that planning to incorporate security at the conclusion of development presents extreme challenges. Many of these security changes would require significant code revision; some would require the additional work of designers and testers, ensuring a second round of development. While it is likely that a second round will be necessary to make adjustments requested by clients or product managers, this still requires significant refactoring.

Developing with a security mindset instead ensures that security will be included in each step, helping to eliminate the chances that something will be missed. While it is impossible to guarantee that every security hole is closed; developing with security in mind, rather than as an afterthought, helps make sure that many are in fact closed, and improves the overall security of your application.

Spending Too Much Time Coding, Not Enough Time Designing

Developers enjoy writing code and presenting a solution that is expressed as a finished product. They would rather write code than write specs. This component of developer behavior makes the design process hard on them, because they have a hard time putting their thoughts in writing or coming up with ideas that are not expressed in some sort of code.

In many cases, this will result in a developer simply diving into a project and writing code without having developed a plan, architecting the product, or thinking through the implications of design choices they are now making on the fly. This is dangerous, because it inhibits the inclusion of security measures, ensures that massive amounts of refactoring will be necessary when changes are needed, and ultimately will end up delaying the product past any deadline the developer might have insisted he could meet.

The problem here is that any software development process expert would tell that developer that design is not an inconvenient waste of time; but actually requires a greater portion of the time than is required by the actual programming. This might seem strange; however, according to the author of "The Mythical Man Month", the following is the breakdown of a proper application development cycle:

- 1/3rd planning
- 1/6th coding
- 1/4th early testing
- 1/4th complete application testing, once all components delivered

In other words, only 17% of the application development is spent writing code. A full third of the time is spent planning the application, designing it, architecting it, getting feedback from stakeholders and planning out exactly what will be written, by whom, and by when.

An odd reality that I have become familiar with is that software development is going to take as long as it is going to take. There is nothing you can do to speed the process. If the process was meant to take six weeks and you try to compress it into four weeks, you will spend those extra two weeks playing catch-up no matter what. You cannot ignore the 1/3 planning time, or the 1/2 testing time, because the application will have bugs and will delay the deadline to the point where you will have ultimately spent that amount of time fixing it. The proportions here have an eerie way of being dead-on right.

Developers who realize this have an easier time than those who do not. They know that they have to take the time to properly develop their applications, and as such they spend time in design. They step back, architect how the application will work,

talk amongst themselves about new ideas that might improve the process, and generally examine the application from an objective point of view.

The best team leads recognize that seeing developers sitting in a room together, talking, is not a sign that nothing is getting finished. They realize that planning and design are critical components of the application development process, and they leave their teams to think, plan, dream, and ultimately walk out with a comprehensive design in hand that will make development that much easier.

There is an old adage that "those who fail to plan, plan to fail." This is true when it relates to software development. Failure to plan is the greatest single cause of project failure. The next greatest cause would be failure to plan realistically, not realizing the time requirements, and thus not building those requirements into the overall system. At the end of the day, spending too much time coding and not enough time designing is a dangerous move, and one that will surely reap undesirable dividends.

Catching NIH Syndrome

When I first began to develop software, I had a great idea for a blogging platform. It would be bigger, better, more powerful and easier to use than WordPress. It would have lots of new features, and would take the world by storm. And most importantly, I would write the entire thing from the ground up: By myself.

Sound familiar? Chances are that every developer in the world has experienced this from time to time. We think that we are smarter, better, more able to implement, insert-your-own-description-here than the next guy.

This is what is referred to as the Not-Invented-Here (NIH) Syndrome. It is a belief that products or ideas that come from other places are somehow inferior to our own products or ideas. It leads to the regular reinvention of standardized libraries, the reimplementation of existing products, and the creation of code that has already been developed elsewhere.

There are certainly some cases in which redeveloping something is a wise choice: when you have a bona fide performance improvement to contribute, or when you have special needs that are underserved or not served by the existing ideas. Certainly lots of companies make livings by reinventing old products and progress is predicated on the idea that someone can build a better mousetrap.

The NIH syndrome becomes a problem, though, when developers automatically assume that everything that has been written is not good enough for them and that they can do a better job. Rejection of existing ideas often leads to reinvention of something that was usually good enough to begin with.

NIH syndrome comes with a number of disadvantages. First, it requires the developer to rewrite everything that has already been written. It leads to things like development of a new framework where other frameworks were adequate, or worse, the implementation of native PHP functions in PHP code - severely inhibiting the performance of those functions.

It also creates problems in maintainability. If everything a developer does is proprietary, finding new developers to maintain those products or libraries means they must first learn those libraries - often with a considerable amount of lead time. This reduces productivity, and simultaneously reduces the pool of talent that might be acquired in the future.

Reinventing everything also limits how much a library, product or service can learn from the rest of the world. Development practices and standards change; improvements to standard libraries are almost immediately available to all who use them, while a custom-written library has to be rewritten to incorporate any changes - if those changes are known in the first place.

Rather than going through the hassle of developing tools and products that have already been invented, developers would be wise to learn to examine existing products, tools, libraries and services for their usefulness, applicability, and acceptability. This leaves more time for the development of the application at hand - likely a new application that has never been developed previously - which is really the fun part of development.

Trying to Micro Optimize

A regular debate throughout the PHP community focuses on the idea of "micro optimizations" - small changes to code that effect small but meaningful performance gains. Most of these optimizations focus on replacing things like `print()` with `echo()`, or replacing double quotation marks with single quotation marks and concatenating. A great deal of time and energy is spent talking about these optimizations, and lots of people regularly blog about them.

Micro optimization is a complete waste of time and energy. It is a massive refactoring effort for minimum - if any - performance gains. There are a number of reasons why this is the case.

First and foremost, most people who micro optimize do not first macro optimize. They have not read Chapter 3 of this book, which discusses how to optimize code, and the things that must be done. Chances are good that most applications that need optimization, and thus are micro optimized, have bottlenecks that are not related to the micro optimizations that are purported to improve performance.

Often those who micro optimize also forget that PHP is already a pretty smart compiler. Those who insist that PHP must run through double-quoted strings to identify and handle variables buried within that string are only partially right - my own performance tests indicate that PHP only does this (with any sort of performance hit) when there are actually items to be parsed. In other words, PHP is smart enough to know whether or not to give the string a second look. Concatenating those items out has no meaningful performance impact.

Another common issue amongst those who micro optimize is that they often turn their code into a mess. One blog post (published in 2009) suggested that those using objects adjust their application design. They articulated that "calling a static method is faster than calling an object method." This may well be true but presents any number of problems, not the least of which is that this would require a substantial refactoring and breaks pretty much every paradigm of object-oriented programming.

Those who insist on micro optimizing often ignore the cost associated with doing it. Having a developer run through an application and replace double quotes with single quotes for "performance reasons" will cost a considerable amount of money in terms of time and salary paid. This amount could even exceed the cost of purchasing additional hardware which would make such a micro optimization a moot point.

Finally, it is worth noting that some micro optimization proponents actually stumble onto good design practices without realizing it when suggesting micro optimizations. For example, one person encourages use of "`isset()` over `array key exists()`" and states that "a switch is sometimes faster than `if...elseif...else`, which are both potentially good design concepts regardless of performance questions."

Rather than micro optimizing, which has dubious performance consequences, it is better to actually optimize. To get started with optimizations that will make a difference, read Chapter 3 and learn what optimizing actually means.

Not Developing with Strictness Operators

For those not intimately familiar with the ins and outs of the INI configuration file, the `E_ALL | E_STRICT` notation means that PHP will notify you of all errors (including notices) AND all `E_STRICT` errors. Most developers never bother to develop this way, yet this is an important element of development and one that should be taken seriously.

Most developers figure that notices are unimportant since they do not typically break the application and they do not interrupt anything. And with regards to `E_STRICT`, these coding standards warnings have no real bearing on the way a PHP application actually performs - they just warn of coding standard violations. It is often hard to justify spending time fixing these errors when there are lots of other things to get done.

Developing with these flags also ensures that you are able to catch errors and mistakes early on. Some problems manifest themselves as minor errors, but can morph into large bugs later on in your application. For example, an undefined variable being used will generate a notice. If you then try and use the variable as an object, this generates a fatal error that can (and usually will) manifest itself in the most inopportune or inappropriate time.

Similarly, following the coding standards ensures that the code does not violate certain voluntary rules, which can later lead to the violation of certain involuntary rules relating to how PHP code is supposed to be written.

Developers who develop with these flags on typically produce a higher quality code that complies with all the standards of PHP, ensuring that their code will ultimately be more bug-free and less prone to mistakes and omissions later on. While it is possible to develop entire applications that have notices and coding standard violations, it is not advisable and for any professional developing an application, these errors should be caught, fixed and avoided.

Not Developing with a Style Guide

Every developer ultimately develops their own sense of coding styles, whether they employ a particular defined style or invent one of their own. This is usually acceptable for small projects that the developer undertakes by themselves, but for larger

projects, this can create a hodgepodge of varying code that confuses other developers and makes code difficult to maintain.

Many developers are still resistant to implementing a coding standard, even given this potential for conflict. Their reasons are many: good code does not result from a coding standard, or the fact that a coding standard means they have to spend precious time formatting their code instead of making it work properly.

However, coding standards have two important applications that make them necessary in large teams: first, they make code maintenance considerably easier by codifying and unifying the way code is written and developed. Second, they can help prevent inadvertent bugs by their implementation.

Most developers will agree that it is easier to write code than it is to read it. Given the difficulty in reading code, developing without a coding standard makes this even more difficult, because in addition to reading code that is unfamiliar the developer must also adapt to a coding style that is unfamiliar.

A standardized coding style, regardless of whether it is each developer's personal preference, establishes a familiar baseline that ensures that developers will be able to more easily read and understand source code not written by them. This further ensures that maintainability is made easier.

Style guides, when well written, can also help prevent the introduction of bugs into your code. For example, requiring brackets around if-else statements, even if they are a single line, helps prevent the introduction of bugs when a developer goes to add a second behavior in that if-else loop. Without such a coding standard, it is possible that the developer might forget and introduce something that causes the original item to be executed regardless; this condition would obviously introduce a significant bug.

Joel Spolsky once wrote an article called "Making Wrong Code Look Wrong" in which he argues that developers can designate between unsafe and safe (unclean and clean) variables in code. This helps prevent security problems, and might also help prevent bugs in code.

Another strategy suggested by some is the reversal of the order in which items are compared in if-else statements, generating a fatal error in the event that a developer uses the wrong symbol. For example:

```
if($var = count($array)) { ... }
```

This is clearly wrong - in this if statement we are assigning the value, which will always return true, but that is obviously not the point of the `if-else` statement. Instead, we meant to use a double-equals (==) to evaluate whether the variable `$var` was equivalent to the `count($array)` result.

If we reverse these items, our mistake would generate a fatal error:

```
if(count($array) = $var)) { ... }
```

The fatal error would help us to realize that we had omitted the double-equals required to evaluate, and instead we would realize we that were attempting to assign, resulting in the error.

One of the easiest ways to enforce coding standards is to make use of something like PHP_CodeSniffer. This PEAR package can be integrated into continuous integration servers like PHPUnderControl, which will produce regular reports about compliance with your coding standards. PHP_CodeSniffer examines your code against a specified set of rules, and flags errors in that code for review. It can be as lenient or as strict as you like. There are several standards already developed for use with it, and developers can develop their own set of standards as well.

There are many resources available to help developers create a coding standard that fits their needs. Some existing ones will automatically fit the bill, at other times developers will want to adopt a coding standard to fit themselves or their needs. Whatever approach is taken, developers should simply ensure that the coding standard of their choice is followed and applied to all new code written, and all old code maintained, to ensure future compliance with such a standard for ease of reading and prevention of bugs.

Chapter 9

Becoming a Happy Developer

Development is a great trade to be in. It is growing, and people have a great amount of respect for those that can "do things" with computers. There are new openings every day for skilled developers, and becoming a highly skilled PHP developer will bode well for future career opportunities.

Having a job in development, though, is not always the same thing as being happy in development. There are a number of factors that contribute strongly to whether or not a developer is happy at what they are doing. Some of these items, including version control and testing have been discussed in this book to this point. Others like effective issue tracking, spec development and quiet working conditions have not been discussed in very much detail to this point.

This chapter focuses on some of these "soft skills" that help contribute to happy, healthy developers that love their work environments and help produce exceptional code day in and day out.

The Importance of Quiet

In what has turned into a pervasive trend, developers at the entry, mid and even senior levels are finding themselves thrust into open bullpens or cubicle farms, devoid of sound dampening barriers like walls and office doors. Startups with fancy, vowel-devoid names corral their developers to "improve communication" or "boost

synergy." But the reality is that developers need quiet conditions to work effectively and productively.

In the groundbreaking book "Peopleware", authors Tom DeMarco and Timothy Lister argue that having quiet working conditions is not just a benefit, but an essential component of software development. They say that "management, at its best, should make sure there is enough space, enough quiet, and enough ways to ensure privacy so that people can create their own sensible workspace." They point out statistical benefits of silence.

Today's "productivity environment" is based on the bullpen, the cubicle farm, or the "collaborative room" concept where there are a bunch of desks collected together, all for the purpose of "improving communication." The reality though is that creative workers, like software developers, need quiet to work. And without quiet, they do not work quite as effectively.

Some Specs on Spec Development

Another common feature of the workplace in software development is the "agile" workplace. In the agile workplace, small teams output quick iterations of a changing specification in short bursts of time known as "scrums." Agile's proponents advocate this approach as a rapid-fire way to put together shipping software **right now**, as opposed to the traditional "waterfall" method which results in software taking months, years or decades to ship.

However, there is an underlying problem here which agile's backers fail to realize or mention: software developers have a particular skill set, and innovating on the fly is not usually a part of that. In my opinion, many software developers like to have all the puzzle pieces in hand before they begin working; they are not the types who collect the puzzle pieces. Without all the components, they are lost, and unproductive.

The need to give a developer a clear, concise overview of what they are working on does not necessarily mean a trip back to the tome-like specifications of years past, where large requirements documents were written and never read by anyone but their authors. No, specification development need not be that complicated or boring.

To understand modern specifications, one needs to understand what the goal of the specification is. The goal of a spec is to highlight for the developer *what* is being

built, and *how* it will function. Therefore, an annotated wireframe is technically a complete specification, provided it highlights the demands, desires and needs of the individuals compiling it.

I have worked on many teams before: the "we don't have a plan; let's go by the seat of our pants and do 'agile"', as well as the "here's a long tome that compiles all of our needs and took six months to write." In my experience, the easiest teams to work on were the ones who said "here's a wireframe, build this." It makes it easier to innovate (because the developer decides how the button works and what features to add) while also making it easy to deliver a useful product to the rest of the team.

Development requires some degree of planning. Failing to plan is planning to fail. Developers should insist on a basic level of spec development, even if it is a complete wireframe.

Effective Issue Tracking

Think back quickly to your first project: how much do you remember about the way you implemented it or the bugs you found but never fixed? Chances are good there is a limited amount of memory devoted to that particular project. In fact, it is possible you may not even remember (or care to remember) it at all.

The reality is that human beings can only remember so much. Studies show we can only do a few things at a time, and when we start doing more than one our attention to all of them takes a hit. Given our imperfect ability to remember things, it is of vital importance that we learn to write things down to aid in our memories.

Nowhere is this more important than in the tracking of issues, feature requests and bugs associated with our products. Consider: it is possible that for a medium-sized project there may be 100 outstanding bugs, tasks and feature requests. No one person, no matter their memory skills, can remember them all, resulting in issues going unresolved into future releases.

Effective bug tracking is a key component of being a happy developer. Once all issues are dumped into a tracker of some kind the mind is freed to start working on other things, one at a time. Furthermore, tracking issues is like creating shared memory: it is possible to share the issues with other developers, who can address them, improving productivity and reducing the workload.

A good rule of thumb is that each discrete action in a project requires its own ticket. For example, if a form needs to be created, and a backend handler for that form needs to be implemented, that is two tickets: they are discrete actions that will be done individually. Despite the fact that they are linked together by their shared purpose (the form submission and storage), the actions are discrete components.

Another good rule of thumb is that whenever an issue is resolved, but additional issues are raised as a result (e.g. with our form example, if a database change is required to make the form save properly), a new issue should be created and mentioned in the old issue. That way, it is obvious to other developers (and you, two weeks from now) that this issue was resolved, but created another issue to resolve as well.

Effective Project Management

Many developers think that project management is handled entirely by the project manager. They could not be more wrong. Good project managers rely on developers and team leads to help manage the project, by providing feedback, estimates, and issue management.

The most effective way to help manage a project is to communicate clearly, completely, and often. Whether through filing issues or bringing up concerns during daily standup meetings, developers must take initiative at being involved in the overall management of their projects.

It is easy for developers to think that they do not have anything to contribute to a project manager or to the management of the project, but they really do: they are the way the work gets done, and they are the reason the project is finished at all. So being an effective, involved member of the team is everyone's responsibility.

Developers who are engaged in the project, have a mental stake in its success, and help aid the proper management of the project will be much happier than developers who just sit by and watch a project move towards completion (or failure). And project managers want the input and feedback of developers. Successfully managing a project to completion is the responsibility of the entire team, not just the project manager; the manager is the coordinator, but their team members are the eyes and ears of what is really going on.

Picking the Right Company to Work For

Selecting a position is not always an easy decision for developers to make. Picking the right company is crucial to a developer's long-term happiness and success, yet it is not always apparent how well a developer will do at a particular firm. Unhappy developers are a leading cause of turnover in development firms. Thus, it is important that developers and companies select well.

While it is not always easy to figure out whether or not a company will be a good fit, there are questions a developer can ask before starting work that can highlight possible future problems. In particular, developers should ask questions that relate directly to other experiences they have had. They should ask about process, the expectations of their supervisors, the availability of tools like bug tracking, version control and project management software. They should investigate whether or not there is a specification and what kind of work they will be doing (direct client work, product work, etc).

Developers also have an obligation to determine what their own needs and desires happen to be. This introspection may take some time, and should be done before receiving an offer, and optimally before even sending out resumes to other companies. The easiest way to ensure that a developer will be unhappy at a new place is to not have an idea of what their own happiness should look like. Developers can and should avoid this if possible.