PHP BEYOND THE WEB

shell scripts, desktop software, system daemons and more

A complete guide to taking your skills beyond the web with PHP

Rob Aley

PHP Beyond the web

Shell scripts, desktop software, system daemons and more

Rob Aley

This book is for sale at http://leanpub.com/php

This version was published on 2013-11-25

Tweet This Book!

Please help Rob Aley by spreading the word about this book on Twitter!

The suggested hashtag for this book is #phpbeyondtheweb.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#phpbeyondtheweb

Contents

Welcome

About the author

I've been programming in PHP since late 2000. Initially it wasn't by choice as my preferred languages at the time were Perl and Delphi (also known as Object Pascal). Things began to change after graduating from the University of Leeds with a degree in Computer Science in 1999 and starting out in a career as a freelance web developer. After only a couple of months I was offered the opportunity to take over a (relatively speaking) substantial government website contract from a friend who was exiting the freelance world for the safer and saner world of full-time employment. The only catch was that several thousand lines of code had already been written, and they were written in a relatively new language called "PHP". Oh, and the only other catch was that I had about a week to learn it before taking over the site. So, as was the way at the time, I popped down to the local Waterstones bookshop (for the younger among you that's where we used to get books. Books made of paper. And we had to go out and get them. Or order online and wait for days for them to be delivered. Did I mention they were made of paper?). With my paper copy of "The Generic Beginners Complete Guide to PHP and MySQL for Dummies Compendium" book in hand (I may not have recalled the title completely correctly) I settled down with a pint of ale (I'm in Yorkshire at this point, remember) and set about reading it. A few days later I was coding like a pro (well, stuff was working) and 12 years later I haven't looked back. After a varied career as a freelancer and starting up a couple of, er, start-ups (IT related and not) with varying (usually dismal) success, I'm now a full-time programmer at The University of Oxford. My day job involves medium-scale data acquisition and management, statistical analysis and providing user interfaces for researchers and the public. The majority of my development work is done in PHP, either developing new projects or gluing together other peoples software, systems and databases.

Through-out my career I've always used PHP for web development, but for Desktop GUI work I initially used Delphi (and then Free-Pascal/Lazarus), complimented with bash shell scripting for other tasks. This was mainly due to learning them while at University. However, as PHP has matured I've increasingly used it beyond the web, and now I rarely use anything else for any programming or scripting task I encounter. Having been immersed in other languages like C++, Javascript, Fortran, Lisp (and probably others that my brain has chosen deliberately not to remember) by necessity during University and in some of my freelance jobs, I can honestly say that PHP is now my language of choice, rather than of necessity.

When I'm not tied to a computer, I would like to say I have lots of varied and interesting hobbies. I used to have. I could write a whole book about where I've been and what I've done and I'd like to think it's made me a well rounded person. But these days I don't. In large part this is due to the demands of my three gorgeous young daughters Ellie, Izzy and Indy, my gorgeous wife Parv and my even more gorgeous cat Mia. And I wouldn't have it any other way. That's what I tell myself, anyway…

Acknowledgements

Isaac Newton said “If I have seen further it is by standing on the shoulders of giants”. This book builds on, and hopefully adds to, the work of many others. The most notable of whom I would like to acknowledge below.

The authors of, and contributors to, the official PHP manual
An invaluable reference for PHP functions and syntax, to which I referred frequently during writing this book, both for fact checking and as an aide-memoir. Thanks!

The collective PHP wisdom of the Internet
For over 12 years I’ve used you for learning, research, play and profit. Too many sites & too many people to list here, if you’ve written about PHP on web then you may well be one of them. Thanks!

My Family
For allowing me a modicum of time to write this book, and supporting me unconditionally in everything I do. Usually. If I ask first. And there’s not something more important going on.

1 Introduction

If your e-reader has bought you directly to this page, then it has (perhaps wisely) skipped the books "front matter", containing the front cover, table of contents, about-the-author section, acknowledgements and a few more bits and bobs. If that interests you, then scroll backwards. Otherwise, read on to start your journey through the wider world of PHP software.

1.1 "Use PHP? We're not building a website, you know!".

Both it's current formal recursive moniker ("PHP : PHP HyperText Preprocessor"), as well as the name originally bestowed upon it by it's creator Rasmus Lerdorf ("PHP : Personal Home Page") reinforce the widely held view that PHP is a scripting language for the web. And that was true, back in 1995 when PHP was first created, and for a number of years afterwards. In the web arena PHP excels : it's easy to use, quick to develop in, widely deployed, tightly integrated into web stacks (it's usually the P in LAMP,WAMP,MAMP etc.), and of course because it is free and open source.

But many people don't realise (or haven't noticed, or choose not to notice) that PHP has evolved and grown up. It now closely resembles a modern, general purpose programming language. This lack of recognition is partly PHP's own fault as it took a long time to get some of the fundamentals in place, such as OOP language constructs and even the ability to run without a web server being involved. However the programming community hasn't helped; many programmers had a hard time seeing the potential for PHP to bring its rapid dynamic development model out of the web and into the wider computing environment, and many simply stuck with the "web scripting for beginners" dogma that was only really true of the early years.

Recent releases in particular have brought mainstream language features (e.g. closures, traits, better language support, namespaces & late static binding among many others) to the table. Performance has jumped up and up and up, memory usage (a bugbear of older versions) has dropped considerably, and PHP is now one of the more lean popular scripting languages. For even higher performance needs PHP now has a built in Op-code cache which is useful for oft-used scripts, and Facebook (the biggest user of PHP around) are one of several groups who are releasing alternative interpreters/VM's with JIT compilers, leading to performance increases of up to 6 times that achievable with the standard Zend engine. And of course PHP core language development continues apace, version 5.5 has recently been released introducing more modern features like Generators, indicative that reports of PHP's demise are quite premature!

If you're a PHP web developer looking to work beyond the web, there has never been a better time to leverage your existing skill set and try your hand at desktop and system coding in PHP. If you're a new programmer pondering which language to learn you can pick up skills to use in a wider range of scenarios than many other similar languages.

But why, why, oh why do it in PHP? Why not whip up a shell script? Why not learn C++ or another language typically used for software projects? The truth is these are valid options, and life may well work out just fine for you. But why turn down the opportunity to use your existing skills? Why not use PHPs integrated database access, reuse existing code and data from your web projects, take advantage of PHPs easy to use network libraries and functions, wallow in flexible text and data handling, and mix shell commands and other languages into PHP as needed to get best of both worlds? In short, why the heck not?

The aim of this book is to give you, the PHP coder, an insight into using PHP beyond the web. We'll look at building desktop apps, command line tools, system daemons, network servers and other native applications. Importantly I'll show you how you can do all of this without leaving the comfortable world of PHP and its friends.

Hopefully I've sold you on the story that PHP is a cross-platform, rapid-development focused, versatile language and is ideal for many different types of software. But if you still doubt the power of PHP to deliver real, non-web, software, and don't want to read the whole book to find out what is achievable, then skip to Appendix D at the rear of the book to find examples of projects such as web servers, database engines, machine learning tools and many others all written in PHP. If they don't inspire you to create your software in PHP, then maybe nothing will!

Further Reading

"PHP is much better than you think" - An article by Fabien Pontencier outlining the positives of PHP development and talking about the changes in the PHP ecosystem
http://fabien.potencier.org/article/64/php-is-much-better-than-you-think

1.2 Are you new to PHP?

This book is aimed at showing PHP developers how they can use their existing skills to write software instead of web pages. However I appreciate that some readers may be new to PHP and are reading this book to get a feel for what PHP is capable of. If you're already a programmer, the comprehensive official PHP manual may be the best place to begin to get a feel for the differences between PHP and the languages you are used to.

If you're not already a programmer, there are numerous "beginning PHP" books available from your favourite e-book retailer. In either case, the wider web also provides its usual breadth of in-depth knowledge and tutorials, just a judicious google away.

And finally, newcomer or experienced programmer, if you're thinking of getting serious about PHP, it may be worth having a look at some PHP "best practices" websites before diving headlong into coding. It may save you a lot of trouble in the long run.

Further Reading

The Official PHP Manual
http://www.php.net/manual

A free online PHP course for beginners
http://ureddit.com/class/55471/programming-in-php

"PHP The Right Way" - A quick reference for PHP best practices, accepted coding standards, and links to authoritative tutorials
http://www.phptherightway.com/

"PHP Best Practices" - A short, practical guide for common and confusing PHP tasks
http://phpbestpractices.org/

"PHP Study Guide" - A PHP Study Guide aimed at those wanting to pass the ZCE (Zend Certified Engineer) exam
http://php-guide.evercodelab.com/

"PSR-What?" - A guide to the "PSR" PHP Standards Recommendations
http://www.lornajane.net/posts/2013/psr-what

1.3 Reader prerequisites. Or, what this book isn't

To make the most of this book, you should have basic experience of programming in PHP (most likely for the web), a general programming or IT background, and a willingness to learn and be taken outside of your comfort zone.

This book isn't an introduction to PHP or programming in general. Although you don't need a computing degree or knowledge of advanced programming concepts, the book is hopefully pitched to the level of an average PHP programmer (one who has explored more than the very basics of PHP) and tries to explain any necessary concepts as we go along. It is also useful for advanced programmers who may choose to use it as a quick reference for exploring PHP beyond the web.

The book gives code samples to illustrate concepts and usage of functions, but generally doesn't provide complete scripts or programs. It is written as a reference and introduction, to give you information and inspiration to help you develop your own PHP solutions. Likewise, when we discuss external tools and third party libraries, I will generally give an overview and basic introduction or example code, and then link to the official documentation and online tutorials. There is little point in a book like this, which covers a general area, going into great depth and giving extensive coverage to a particular piece of software, particularly those which already have comprehensive documentation available. We will of course cover any relevant areas where the official or online documentation is lacking, and point you directly at the relevant sections where necessary.

1.4 An important note for Windows and Mac users

Most of the examples given in the book are run and tested under Linux (the latest Ubuntu Desktop/Server flavour in most cases). PHP runs on many operating systems, including MS

Windows and Apple OS X, and code often runs in an identical manner. However there are, of course, differences due to the file system, OS, available libraries etc., but to cover these all in this book would not be practical. In addition, some features such as those reliant on POSIX standards aren't (easily) available on some systems like Windows. As OS X was derived from a POSIX compliant operating system, you will likely find more of the code compatible than with Windows, but your mileage will likely still vary. A good source of information for OS specific issues is the official PHP online manual, and in particular the user comments at the bottom of each page. Where possible areas specific to these other OSes have also been covered, for instance we will look at how to access the Windows Registry.

1.5 About the sample code

As you'll see throughout the book I mainly use "traditional" imperative/procedural PHP in my coding examples to keep things as simple as possible for coders of all abilities. This book isn't designed to be a lesson in coding best practices or style, a guide to OOP programming or to push any kind of programming model or dogma. I also avoid the use of any code frameworks. Many frameworks are based around the web model and don't always perform as intended in the kind of applications we'll be looking at, although some do now have "console" or "cli" modules. MVC style frameworks can be useful when building GUI applications (indeed the MVC paradigm pre-dates the web considerably!), but because of the many different implementation details and styles we'll stick to framework-less code here.

It should be clear from the plain, straight forward PHP code presented how it can be used or adapted to suit your own programming style, framework or model. We will be mainly looking at task specific implementation details, leaving the hot topic of programming itself to the many other books available.

The sample code also generally doesn't try to implement any particularly useful programs. It is designed to be as minimal as possible, in order to be as clear as possible and outline the scaffolding needed for a particular type of task. For example, the sample code for creating daemons focuses on the mechanics of creating a background process and running an event loop. This is to keep the code clear, as a PHP programmer you will likely already know how to check your Twitter account for the latest Justin Beiber updates, and just want to know how to daemonise the process. The code aims to be something of a reference for general-purpose PHP programming, rather than a tutorial on programming itself.

All of the sample PHP source code in this book is available for you to use and do with as you please without limit or license. Use or abuse it as you see fit!

If you have trouble running or understanding the sample code, see the feedback section in the last chapter of this book for details on how to contact the author for help.

1.6 External resources

Through out the book we will point you in the direction of external tools, resources and information, in one of two ways :

A "toolbox" like this

Toolboxes like this contain details of useful tools and software

Main website : http://www.a-useful.tool/

Main documentation and installation info : http://wiki.a-useful.tool/

Further Reading

Useful articles, tutorials and reference information will be presented in Further Reading sections like this
https://www.very-useful-info.book/

1.7 Book formats/versions available, and access to updates

The book is currently available in pdf, epub and mobi format, all included in the same price! The epub and mobi formats are best for reading on dedicated e-readers. Even if you are reading it on an e-reader, you may wish to download the PDF version as well for use on your computer when coding as there are many links to useful websites through-out this book, and it is also easier to copy and paste code directly from the book. Whichever versions you use, you will always get free access to all future updates to the e-book.

1.8 English. The Real English.

Just a quick note to say that, no, those aren't spelling mistakes, it's British English (you know, the one that kicked it all off) as your humble author hails from the other side of the pond. Deal with it, we spell things differently. Any genuine spelling mistakes you spot will be dismissed as part of an evolving language, and then quietly fixed while you're not looking.

2 Getting away from the Web - the basics

This chapter has an overview of the basic steps involved in breaking free from the web with PHP. We'll look at the technical steps involved, and also at the differences in programming practises and focus.

2.1 PHP without a web server

Most PHP programmers have used PHP strictly in a web-server environment. In such an environment PHP is a CGI/Fast GGI or server module, called and controlled by the HTTP server (usually Apache, IIS, Nginx or similar). The HTTP server receives a request for a PHP based web page and calls the PHP process to execute it, which usually returns output back to the HTTP server to be sent on to the end user.

There have been a number of attempts to create local applications (desktop apps, scripting systems) with PHP using a locally installed web server, with some success. However the downsides of using this web-type model for local fapps are numerous:

- For the value they provide in these scenarios, web servers such as Apache are often very over specified and resource hungry
- Unless properly locked down, a web server running locally introduces a point of access to your machine for malicious visitors from the outside world
- HTTP is a verbose protocol and (arguably) ideally suited for the web, however it's often overkill for local inter-process communication and adds another resource and complexity overhead to your application
- Your application interface typically runs in a web browser, and therefore looks anything but local (and comes with additional support/upgrade headaches unless you install a browser specifically for your app)

PHP, as of version 5.4, includes a built-in web-server which removes some of the problems described above. However it was designed for local testing of PHP scripts designed to be deployed on a fully-fledged HTTP server such as Apache in a live environment. It is a route that can be explored for some local apps, particularly where you are wanting to run a private instance of a PHP app that already exists. However its stability, reliability and suitability for production-ready local apps is yet to be proven, and it still comes with the baggage of HTTP and web browsers.

Since version 4.3, PHP has had a golden nugget hidden up it's sleeve which solves all of these problems. The PHP CLI SAPI (Command Line Interface Server Application Programming Interface), to give it it's formal name, is essential a long way of saying "stand-alone PHP". It cuts out the need for a web server and provides a stand alone PHP binary for you to interact with. For instance, typing the following at a shell prompt

```
~$ php /home/myfiles/myprogram.php
```

will simply execute your `myprogram.php` file (which you write mostly like any other PHP script) and return any output back to the terminal (unless you tell it otherwise) instead of being sent to the web server (which doesn't exist!).

In general, PHP scripts called directly using PHP CLI will behave in the same way as when called through a web server, although there are some differences. For instance, you won't find any `$_GET` or `$_POST` arrays, and PHP won't send any HTTP headers by default, as these concepts don't mean much beyond the web. Default settings like `max_execution_time` are set to sensible values for local use (in this case to `0` so your scripts won't time out), output buffering is turned off, and error messages are displayed in plain text rather than HTML.

The PHP CLI SAPI is what we will be using for most of the examples in this book as it designed with exactly the same motivations as this book, taking PHP beyond the web. Full details of how to install and use the CLI SAPI are given in the next chapter.

2.2 PHP versions - what's yours?

PHP has supported the CLI SAPI since version 4.3.0, so many of the examples in this book will run on any PHP version since then. At the time of writing, the current version of PHP is 5.5.1 and the code in this book has been tested against this version. If you find that a particular function doesn't appear to exist or work as expected, check the online PHP manual for that function. This shows which versions support which functions, and describe any breaking changes created by newer versions.

If you are using an older version, there are some very good reasons to upgrade.

- Performance has increased markedly (and, correspondingly, resources used have decreased) in recent versions
- Although security is not always as critical with non-web applications (see the discussion on security later in this chapter for caveats), the security enhancements and security related bug fixes in recent versions are essential if you're handling data from external sources, and if you're using the same version for web work as well
- Modern language features are available, which can help your coding productivity, as well as helping others take your code more seriously

As with the web versions of PHP, you can compile your own version of the CLI SAPI if you find the need. If you want to include extensions not pre-packaged by your OS software repositories, remove non essential code for performance reasons, or use any of the other compile-time options that PHP supports then learning to "roll your own" version may be worthwhile. A starter guide to compiling and installing PHP and related extensions are covered in Appendix A.

2.3 A few good reasons NOT to do it in PHP

This book is focused on doing everything in PHP, and for 90% of tasks PHP does all that is asked of it. But before you commit to rewrite your world in PHP, there are a few very good reasons that you may want to consider carefully before jumping in.

- **High performance requirements**

If you need very high performance, particularly on limited hardware, PHP may not be for you. It has come on leaps and bounds in recent versions and regularly trounces languages like Python and frameworks like Ruby-on-Rails in benchmarks. However at its heart PHP is an abstracted scripting language which is never going to get the same performance for some tasks as lower level languages like C, which are closer to the bare metal. Don't write it off for all high-performance tasks though, if you're looking at performance with an eye to costs, you may find that the cost of savings on developer time through speed of development in PHP outweighs the cost of the extra hardware you throw at it to get the performance. Take a look at Chapter 9 to learn more about increasing performance, and find out how you can interact with languages like C from PHP so that you can still get some of the benefits of RAD (Rapid Application Development) in PHP.

- **Don't (necessarily) re-invent the wheel**

You can write a web server in PHP (see Appendix D for examples), but won't existing servers like Apache do what you need? For many or most infrastructure "itches", software exists (written in many different languages) that will give you the "scratch" you need. Unless you really need other features, the time spent writing your own is usually going to be greater than the time learning and implementing an existing piece of software. Of course, for some people the converse is true. "Because I can" can be a valid reason, and writing software is always a valuable learning experience.

- **Keeping the source closed**

PHP is a scripting language, and this means that if you are writing software to sell or deploy elsewhere, you will be revealing your source code to the recipients. Many (including your author) will argue that being "open-source", even commercially, is a good thing. But if it's not your bag you will need to go to greater lengths to protect your code. Source code obfuscater are available online which use various tricks to make your code hard (but not impossible) to read, and a number of PHP compilers are now available to produce binary programs (albeit with limitations in terms of syntax and extension coverage). But the main PHP project hasn't expressed any intention to support code hiding or compiling, and so the viability of such methods long term is not certain. The business case for closed-source software is also not a done-deal in the medium to long term.

2.4 Thinking about security

Every programmer worth their salt is at least aware of the security implications of building web sites and apps. You deliberately expose your code to the public, to the world, to anyone and everyone who will come (good people and bad). One of the early failings of PHP was to put ease of use over security of code (the horror stories from relics like `register_globals` are only a quick google away). With newer, more secure defaults and depreciated code like `register_globals` PHP is safer than ever online. And although most security problems are caused by the programmer rather than the language, even the newest web coders seem to have an appreciation of security issues right off the bat these days.

Step into the world of off-line software, however, and things are markedly different. Typically it's software for trusted users only, deployed locally on trusted machines and under the full control of the benevolent user. The user isn't going to deliberately attack the software or machine, they're working with their own data, and functionality absolutely can't be compromised. Security is rarely considered at all when developing desktop apps and the like, let alone being features specified at design time, because it's not "necessary".

Except the world doesn't work like that. Take a look at any software vulnerability mailing list like BugTraq and you'll find an abundance of desktop apps like Adobe Reader, Microsoft Word and even open-source stalwarts like GIMP. The fact is, security is important even in local apps, for two main reasons. The first is the perennial problem of "typical user" behaviour. This is not a problem for an intelligent tech-literate user who never makes mistakes (like yourself of course), but for any software used by the rest of us disaster is only an accidental-click-on-a-dodgy-email away. The second reason security is important that for most machines, most non-web applications aren't really "off-line". Even when a desktop app or a system daemon doesn't interact with the web, local network or other external services itself, the machine it is connected to will invariably have an Ethernet cable plugged into it or a WiFi/3G connection active. Your software will not run in its own cosy little realm, insulated from the world outside. Perfectly sand-boxed virtual machines not withstanding.

Software security is the topic of a whole other book, and many of the same principles apply to systems software as to web software so you will be able to use your existing knowledge of web based PHP security practices to guide you. The following list of typical vulnerability types and attack vectors in both user-facing and systems software should be considered when planning your application security measures and monitoring.

- **Compromised data files**

Compromised files from external sources (loaded deliberately or accidentally by users). These are usually data files, and particularly at risk is software registered as a default viewer for a particular file type, as accidental and malicious file activation is much easier.

- **Exploitation by malware**

Malware looking for innocent software to exploit to gain privilege escalation. Scripted software like PHP code can be easier for malware to re-write or alter, and the availability of the source code in an un-compiled form can be of help to the malware authors.

- **User misbehaviour**

Legitimate users misbehaving. John Smith looking for a way to view the files or surfing history of his boss Jane Doe on their shared business system, for instance.

- **Privilege escalation**

Similar to legitimate users misbehaving, this is legitimate software misbehaving, either accidentally or deliberately trying to gain & use access permissions it does not have.

- **Dependency vulnerability**

PHP vulnerabilities, and vulnerabilities in other dependencies and related software. Your software will be completely free of security issues, of course, but it inevitably depends on other libraries, software and PHP extensions, and of course lets not forget the PHP interpreter itself. Any of these can contain security bugs and attack vectors.

The above sections list common sources of security issues in all types of software, not just in PHP. Minimise your risks by planning for these in the design stage, and testing for them before deployment. Then cross your fingers…

Further Reading

A free online course in Penetration Testing. Focuses on web based penetration, but relevant also to off-line software.
https://www.pentesterlab.com/

Free online book covering PHP security
http://phpsecurity.readthedocs.org/en/latest/index.html

3 Understanding the CLI SAPI, and why you need to

As we mentioned in Chapter 2, most systems programming you do will involve using the PHP CLI SAPI. It's therefore important to have a good grasp of how to use it, the options for configuring and running it, and how it differs from the web based SAPIs you are used to. Luckily the differences are minimal, and many are intuitive, although it's still worth having a thorough read of this chapter as the CLI SAPI forms the basis upon which most of your code will run.

3.1 What's different about the CLI SAPI?

The following is a list of the main differences in the CLI SAPI implementation :

- No HTTP headers are written to the output by default. This makes sense as they hold no meaning in the command line and so would be just extraneous text printed above your genuine output. If your output will later be funnelled out to a web browser, you will need to manually add any necessary headers (for instance by using the `header()` PHP function).
- PHP does not change the working directory to that of the PHP script being executed. To do this manually, use `getcwd()` and `chdir()` to get and set the current directory. Otherwise, the current working directory will be that from which you invoked the script. For instance, if you are currently in `/home/rob` and you type `php /home/peter/some_script.php`, the working directory used in PHP will be `/home/rob`, not `/home/peter`.
- Any error or warning messages are output in plain text, rather than HTML formatted text. If you want HTMLified errors, for instance if you are producing static HTML files, you can override this by setting the `html_errors` runtime configuration directive to true in your script using `ini_set('html_errors', 1);`
- PHP implicitly 'flushes' all output immediately and doesn't buffer by default. Online performance can often be harmed by sending output straight to a browser, so instead output is buffered and sent in optimal sized chunks when the chunk is full. Off-line this is not likely to be an issue, and so HTML blocks and output from constructs like print and echo are sent to the shell straight away. There is no need to use `flush()` to clear a buffer when you are waiting for further output. You can still use PHP's output buffering functions to capture and control output if you wish, see the "Output Control Functions" section in the PHP manual for more information.
- There is no execution time limit set. Your script will run continuously until it exits of its own volition, PHP will not terminate it even if it hangs. If you want to set a time limit to reign in misbehaving scripts you can do so from within the script using the `set_time_limit()` function.
- The variables `$argc` and `$argv` which describe any command line arguments passed to your script are automatically set. These are discussed fully later in this chapter.

- PHP defines the constants `STDIN`, `STDOUT` and `STDERR` relating to the standard streams of the same name, and automatically opens I/O streams for them. These give your application instant access to "standard input" (STDIN), "standard output" (STDOUT) and "standard error" (STDERR) streams.

Further Reading

Output Control Functions in the PHP manual http://www.php.net/manual/en/ref.outcontrol.php

Standard Streams (STDIN, STDOUT, STDERR) on Wikipedia http://en.wikipedia.org/wiki/Standard_streams

3.2 CLI SAPI installation

To use the PHP CLI SAPI, you may need to install it first. Appendix A gives details on installing (and compiling, where necessary) PHP. However you may find that it is already installed if you have PHP installed (often in a folder called sapi/cli in the PHP program folders), and if not it is usually available in modern OS software repository (e.g. in Ubuntu a package called `php5-cli` exists and can be installed from any package manager or via the command line with `sudo apt-get install php5-cli`). If it is installed in the command line search path, typing `php -v` on the command line will print the version details, confirming it is indeed installed.

3.3 PHP command line options

The PHP binary will accept a number of command line options/switches/arguments which affect its operation. A full list can be seen by typing `php -h`. Although some apply only to the CGI SAPI, the following are some of the more interesting and common ones used when interacting with the CLI SAPI:

- `-f` *or* `--file`

This allows you to specify the filename of the script to be run, and is optional. The `-f` option exists to allow compatibility with software and scripts such as automation software which can programmatically call command line programs but require filename arguments to be formed in this way. It also allows default filetype handlers to be easily set on Windows for PHP scripts. The only real difference in usage between the two versions of the command above come when interpreting command line arguments passed to the script, which we look at in the "Command line arguments for your script" section below. In most cases, the two following lines are mostly equivalent :

```
~$ php -f myscript.php
~$ php myscript.php
```

- `-a` *or* `--interactive`

Runs PHP interactively. This allows you to type in PHP code, line by line, rather than executing a saved PHP script. This mode of operation is often called a "REPL" (Read-Eval-Print-Loop). As well as providing an interactive interface for testing and developing code, it can also act as an enhanced PHP enabled shell or command-line, and in the next chapter on development tools we'll look at this and third party PHP REPLs more closely.

- `-c` *or* `--php-ini`

Specifies the PHP ini file that PHP will use for this application. This is particularly useful if you are also running web services using PHP on the same machine, as if it is not specified PHP will look in various default locations for `php.ini` and may end up using the same one as your web service. By providing one specifically for your CLI applications you can "open up" various restrictions that make more sense for offline applications. Note that by using the CLI SAPI, PHP will automatically override several `php.ini` settings regardless of whether you specify a custom ini file. These overridden settings are those that affect the behaviour outlined in the "Whats different about the CLI SAPI?" above, and while the `php.ini` file is ignored in these cases you can revert or change these settings directly in your code using `ini_set()` function or similar. You can also use the -d or –define option to set options (e.g. `php -d max_execution_time=2000 myscript.php`). If you are deploying software onto machines that you do not control (e.g. if you are selling software for users to install on their own machines), it makes sense to use one of these mechanisms to ensure that PHP will be running with the settings you expect, not the settings the user may happen to have. See `-n` below as well.

- `-n` *or* `--no-php-ini`

This tells PHP not to load a `php.ini` file at all. This can be useful if you do not wish to ship one with your application and instead set all of the settings directly within your code using `ini_set()` or similar. PHP will use it's default settings if no ini file is provided, so remember that these default settings may change from version to version of PHP (and indeed have done so in the past), and you shouldn't rely on the current defaults being suitable for your application. `php --ini` shows the default path that PHP will look for ini files if the `-n` option isn't used and `-c` isn't used to specify a file.

- `-e` *or* `--profile-info`

This puts PHP into "Extended Information Mode". EIM generates extra information for use by profilers and debuggers. If you're not using a profiler or debugger that requires this mode, you should not enable it as it can degrade performance. More information on profilers and debuggers can be found in the next chapter.

- `-i` *or* `--info`

Calls the `phpinfo()` function and prints the output. This outputs a *large* range of information about the PHP installation, in plain text format rather than the usual HTML (it detects we are calling it from the CLI SAPI). This can be useful in tracking down issues with the installation itself, version information, extensions installed, file paths etc. As with any other shell command, the output can be piped to other commands, e.g. grep. So if you wanted to check if IPv6 was enabled in your PHP binary for instance, you could try :

```
~$ php -i | grep -i "ipv6
```

- `-l` *or* `--syntax-check`

Parses the file, checking for syntax errors. This is a basic "lint" type checker, more advanced static code analysis tools are discussed in the next chapter. Be aware that this option only checks for basic syntax errors, the sort that cause the PHP engine to fail. More subtle bugs, problems in your program logic, and errors that are created at run time will not be detected. Your code is not executed, so it can help pick up basic errors before running code that may alter data and cause problems if it fails. Even when you run such code in a testing environment, resetting data and setting up for another test can take time, so a quick check for basic syntax errors first can be a time saver.

- `-m` *or* `--modules`

Lists all of the loaded PHP & Zend modules/extensions. These are modules that PHP has been compiled with, and may include things like "core", "mysql", "PDO", "json" and more. This is useful for checking the PHP installation has the functionality that your application requires. You can also check from within your scripts using the `extension_loaded()` function or by calling the `phpinfo()` function. `-m` provides a subset of the information given with the `-i` flag described above, and `-i` (or the `phpinfo()` function) will return more information about the configuration, version etc. of the modules.

- `-r` *or* `--run`

Runs a line of PHP code supplied as the argument, rather than executing it from a file. The line of code should be enclosed by single quotes, as shells like bash will try and interpolate PHP variables as if they were shell variables if you use double quotes. This performs a similar role to the `-a` interactive mode, except that PHP's "state" is cleared after each line is executed. This means that the line of code supplied is treated as the whole script to be executed, and execution is terminated once it has been run. An Example would be:

```
~$ php -r 'echo (2+2)."\n";'
```

which will print out 4 followed by a new line. Note that the line must be well-formed syntactically correct PHP, so don't miss out the semi-colon at the end! We will return to `-r` later in this chapter in the section "The many ways to call PHP scripts".

- `-B` *or* `--process-begin`
  `-R` *or* `--process-code`
  `-F` *or* `--process-file`
  `-E` *or* `--process-end`

These four arguments allow you to specify a PHP code to be executed before, during and after input from `STDIN` is processed by PHP. `-B` specifies a line of code to execute before the input is processed, `-R` specifies a line of code to execute for every line of input whilst `-F` specifies a PHP file to execute for each line. Finally `-E` executes a line of code at the end of the input process. In `-R` and `-F`, two special variables are available; `$argn` - the text of the line being processed, and `$argi` - the number of the line being processed. This is mainly useful when using PHP directly in shell scripts. For instance, to print out a text file with line numbers, you can do something like:

```
~$  cat my_text_file.txt | php -B 'echo "Lets add line numbers...\n";'
        -R 'echo "$argi: $argn\n";' -E 'echo "That''s the end folks\n";'`
```

which will output something like

```
Lets add line numbers...
1: Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
2: eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
3: minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip
4: ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
That's the end folks
```

- `-S` *or* `--server`

This starts the built-in development web server. This is like a stripped down version of Apache or Nginx, and can be used for testing or running single instances of websites locally. We'll look at this in depth in Chapter 5, when we look at how it can be used to create local HTML-based apps.

- `-s` *or* `--syntax-highlight`

This outputs an HTML version of the PHP script, with coloured syntax highlighting. The PHP script is not executed or validated, it's simply made "pretty". The pretty HTML is printed to STDOUT, and can be useful when pouring over code looking for errors, issues and optimisations. This only works with PHP in files, not with code provided by the `-r` option. Most modern IDE's and code editors provide syntax highlighting by default, however this can be useful if your only access to a machine is on the command line and the editor you are using doesn't do syntax highlighting. In this case, use `-s` to create a coloured version of your script, and either download it or view it through your web browser if the machine has a web server installed.

- `-v` *or* `--version`

Outputs the PHP version information. This can also be found in amongst the output of the `-i` option described above. Be careful when assuming a particular format, some package repositories (Ubuntu for instance) include their name and their own build numbers in the version string, so don't just filter it for any numerics.

- `-w` *or* `--strip`

This outputs the contents of the source code with any unnecessary white space and any comments removed. This can only be used with code files (not with lines of code supplied by `-r`) and does not work with the syntax highlighting option above. This is used to "minify" a file, i.e. reduce the file size. Contrary to popular opinion, this will not speed up most scripts, the overhead of parsing comments and white space is extremely negligible. You should also be wary of support & debugging issues even if a copy of the "full" code is kept, as line numbers in error reports will no longer match between the versions. It also does not minify identifies like variable names, and so cannot be used to obfuscate your code. There are very few reason to use this option these days. To make a file smaller for distribution, using proper compression (e.g. add it to a zip file) is usually a better way.

- `-z` *or* `--zend-extension`

This specifies the filename/path for a Zend Extension to be loaded before running. This can allow dynamic loading of these extensions, which can also be specified in the `php.ini` file for extensions that are always to be loaded.

- `--rf` *or* `--rfunction`
  `--rc` *or* `--rclass`
  `--re` *or* `--rextension`
  `--rz` *or* `--rzendextension`
  `--ri` *or* `--rextinfo`

The first three options above print reflection information about a named function, class or extension. The last two print basic information about a Zend extension or a standard extension, as returned by the `phpinfo()` function. Reflection is the process by which PHP can perform runtime "introspection", which is the means to allow you to look into elements and structures of your code at run time. The reflection information, which is much more detailed, is only printed if PHP is compiled with reflection support. These options can be used as a quick but precise reference guide to the entities listed above, and are particularly useful in interrogating unknown code written by others.

Further Reading

Reflection information in the PHP Manual
http://www.php.net/manual/en/book.reflection.php

"Introspection and Reflection in PHP" by Octavia Anghel
http://www.sitepoint.com/introspection-and-reflection-in-php/

3.4 Command line arguments for your script

As we've seen above, passing arguments to PHP itself is straightforward and done in the normal way. However passing arguments for use by your PHP script is a little more complicated, as PHP needs to know where it's own arguments stop and where your script's start. The best way to examine how PHP deals with this is through some examples. Consider the following PHP script:

```
<?

echo "Number of arguments given :".$argc."\n";

echo "List of arguments given :\n";

print_r($argv);
```

There are two special variables in the script above :

- `$argc` : This records the number of command line arguments passed to the script
- `$argv` : This is an array of the actual arguments passed.

Lets save the script as `arguments.php`. Now lets call it as follows :

```
~$ php -e arguments.php -i -b=big -l red white "and blue"
```

You will get the following output :

```
Number of arguments given :7
List of arguments given :
Array
(
    [0] => arguments.php
    [1] => -i
    [2] => -b=big
    [3] => -l
    [4] => red
    [5] => white
    [6] => and blue
)
```

As you can see, all of the arguments given from the filename onwards in the command are passed to the script. The first, `-e`, which is used by PHP itself is not passed through. So, as a general rule, everything after the filename is treated as an argument to the script, anything before the filename is treated as an argument for PHP itself, and the filename is shared between the two.

Of course there is an exception. As we learned above, as well as specifying the filename of your script on it's own, we can also pass it as part of the -f flag. So if we execute the following command:

```
~$ php -e -f arguments.php -i -b=big -l red white "and blue"
```

we get the following unexpected output :

```
phpinfo()
PHP Version => 5.4.6-1ubuntu1.3

System => Linux dev-system 3.5.0-37-generic #58-Ubuntu SMP Mon Jul 8 22:10:28 U
TC 2013 i686
Build Date => Jul 15 2013 18:23:34
Server API => Command Line Interface
Virtual Directory Support => disabled
Configuration File (php.ini) Path => /etc/php5/cli
<rest of output removed for brevity>
```

You may recognise this as the output of calling `php -i`. Rather than treating arguments after the filename as belonging to the script, PHP has treated the -f argument as one of it's own, and continued to "own" all of the following arguments. As `-i` is a valid PHP argument, it decides that it was what you wanted and invokes its "information" mode. If you need to pass the filename as part of the -f flag rather than as an argument on it's own, you will need to separate your scripts arguments using `--`. So, for the command above to work as expected, we need to alter it to read:

```
~$ php -e -f arguments.php -- -i -b=big -l red white "and blue"
```

Everything after the `--`, plus the script filename, is passed as arguments to the script, and we get the output we were expecting.

This can make your scripts a little messy, particularly if you are passing lots of arguments, so you may want to look at the sub-section on self-executing scripts in the next section, which will show you how to embed PHPs arguments in the script itself, allowing the script to claim any and all arguments passed as its own.

3.5 The many ways to call PHP scripts

As you can probably tell from the command line options in the previous section, there are several different ways to execute PHP code when using the CLI SAPI. Although we've covered a couple of these already, they are discussed here again for completeness.

3.5.1 From a file

You can tell PHP to execute a particular PHP source code file. For instance:

```
php myscript.php
php -f myscript.php
```

Note that -f is optional, the above two lines are functionally equivalent. The PHP command line options detailed above, where appropriate, work in this method. E.g.

```
~$ php -e myscript.php
```

will execute the file `myscript.php` in Extended Information Mode.

As with the web version of PHP, source files can be interpolated (mixed) with HTML (or, more usefully on the command line, plain text). So you will still need your opening `<?` or `<?php` tags, otherwise your source code will just be printed straight out without being executed.

3.5.2 From a string

You can execute a single line of code with the `-r` flag :

```
~$ php -r 'echo("Hello World!\n");'
```

Many of the other command line options are not available with the `-r` method, such as syntax highlighting. Watch out for shell variable substitution (use single quotes rather than double quotes around your code) and other mangling of your code by the shell. Unless it really is a quick one-off, it is likely safer and easier to pop the relevant line into a file and execute that instead. One common use of the `-r` option is for executing PHP generated by other (possibly non-PHP) shell commands where the whole shell script needs to execute in memory without touching disk (for instance where permissions prohibit disk access).

3.5.3 From STDIN :

If you do not specify a file or use the `-r` option, PHP will treat the contents of `STDIN` as the PHP code to be executed, as shown here:

```
~$ echo '<? echo "hello\n";?>' | php
```

You can also use this method with `-B`,`-R`,`-F` and `-E` to makes PHP a first class citizen in shell scripting, giving you the ability to pipe data in and out of PHP. For instance, to reverse every line of a file (or any data source that you pipe into it), use

```
~$ cat file.txt | php -R 'echo strrev($argn)."\n";' | grep olleh
```

In this line of code, we pipe the contents of a text file into PHP. The `-R` option tells PHP to execute the following PHP code on each line of input, where the line is stored in the special variable `$argn`. In this case, we reverse `$argn` using the string reversing function `strrev()` and then echo the reversed string out again. Any `echo`'d output goes to `STDOUT`, which is either printed to the shell or, as in this case, can be piped to another shell command. In our case we then use `grep` to only display the lines containing the string 'olleh', which is 'hello' backwards. More details on `-R` and its siblings are given in the previous section.

If you want to use options like `-R`, but have too much PHP code to fit comfortably on the command line, you can put the code in a normal PHP source code file and `include()` it, e.g.

```
~$ cat something.txt > php -R 'include("complicated.php");'
```

If it is a non-trivial PHP script, it may be more efficient to package it up into functions and include it once with -B (-B means it's executed before the main code), and then execute the function each time with -R. The following example loads the content of `my_functions.php` once at the start, and then the function `do_something_complicated()` from that file is called on each line (each `$argn`) from the data file (`data.txt`).

```
php -B 'include("my_functions.php");' -R 'do_something_complicated($argn);'
        -f 'data.txt'
```

Although these commands look relatively simple, there is of course no arbitrary limit to the PHP code you can put behind them. You can use classes and objects, multiple files, and most of the code and techniques explored in this book, exposing only functions or methods at the shell level as an interface for the user. You can also open the standard streams as PHP streams within PHP and access their file pointers to read data from, negating the need to use -R, as discussed in the next chapter.

3.5.4 As a self executing script : Unix/Linux

On Unix/Linux type systems you can make a PHP script file into a directly executable shell command. Simply make the very first line of the script file a `#!` line (usually pronounce "shebang line" or "hashbang line") with path to the PHP binary, e.g.

```
#!/usr/bin/php
<?

echo('Hello World!');
```

and set the executable bit using `chmod` or similar, e.g.

```
~$ chmod a+x myscript.php
```

Then simply typing `myscript.php` at the command line to execute it. You can also rename the file to remove the `.php` extension (assuming you had one in the first place), so you would just type

```
~$ myscript
```

at the shell prompt to run it. Note that when running a script in this manor, any command line options are passed directly to the script and not to PHP. In fact, you cannot pass extra command line parameters to PHP at runtime using this method, you must include them in the "shebang" line when constructing your script. For instance, in the example above, if you wanted to use "Extended Information Mode", you would alter the first line of the script to read:

```
#!/usr/bin/php -e
```

If you were to instead call the script as follows :

```
~$ myscript -e
```

then the `-e` flag would be passed as an argument to the script, not to PHP directly, and so PHP would not enter EIM. This is useful for scripts that have lots of user supplied arguments, but also makes options like `-B` and `-R` discussed in the method above cumbersome to use for processing `STDIN` data, as you have to include all of the PHP on the shebang line where it is harder to change. However you can simply `include()` the necessary files and use standard file streams to process the `STDIN` stream (created and opened by the CLI SAPI automatically for you) line by line instead.

Further Reading

Standard IO Streams information in the PHP Manual
http://php.net/manual/en/features.commandline.io-streams.php

If your script may be used on other systems, please bear in mind that the PHP binary will quite often be located in a different directory to your system. In this scenario, you will need to change the shebang line for each system if you hard-code the location in it. Fortunately, if installed correctly, PHP sets an environment variable with its location, available via the `/usr/bin/env` shell command. So if you change the shebang line as follows, your script should be executable wherever PHP is located :

```
#!/usr/bin/env php
```

On Windows, the "shebang" line can be left in, as PHP will recognise it and ignore it. However it will not execute the file as it does on *nix.

3.5.5 As a self executing script : Windows

In a similar manner, scripts can be executed by calling them directly under Windows. However, the process for setting Windows up to do this is slightly more involved, requiring changes to the Registry and file type associations. I therefore recommend reading the official PHP documentation which covers this in detail.

Further Reading

Setting up the PHP Command Line environment on Windows
http://php.net/manual/en/install.windows.commandline.php

3.5.6 Windows php-win.exe

PHP for Windows also ships with php-win.exe, which is similar to the CLI build of PHP, except that it does not open a command line window. This is useful for, running apps with a graphical interface (which will be covered later in this book) or daemon type software.

Futher Reading

An instructive comment on spawning php-win.exe processes on Windows
http://www.php.net/manual/en/features.commandline.php#62162

3.6 “Click to Run” your PHP

In the previous section we looked at the different ways the PHP commands can be formatted, and showed them in the context of the command line, i.e. typing them in. But lets make things easier on ourselves and give them clickable icons so that we can run them direct from the desktop. How we achieve this depends on the operating system.

3.6.1 Clickable icons : Linux

On most Linux systems, specifically those that support the freedesktop.org “Desktop Entry Specification” standard, you can create a clickable “launcher” icon by simply creating a text file. Most main-stream Linux distributions and their window managers follow at least the basics of this standard.

To create a launcher for your app, create a text file in the folder where you want it to appear (e.g. in `/home/rob/Desktop`) called, e.g. `myscript.desktop`. It is important it has the `.desktop` extension. In the file, add the following lines :

```
[Desktop Entry]
Type=Application
Name=My Funky App
Terminal=true
Exec=/usr/bin/php /home/rob/scripts/myscript.php
Icon=/usr/share/icons/gnome/32x32/actions/up.png
```

The first two lines tell the system what we are creating here. The `Name=` line gives your launcher a name which will be shown below the icon. The `Terminal=` line determines whether a terminal window is opened for this program. If you are creating a shell script that takes input from, or sends output to, the terminal you will want to set this to `true`. If your script interacts with the user via GUI elements, you will likely want to set it to `false`. `Exec=` specifies the command to execute which the user clicks the icon. This is the command to call your PHP script, as we discussed in the previous sections. Finally, the `Icon=` line points to the location of a pretty icon for your app. You can supply your own icon file, or use one of the system provided ones. Here we have use the Gnome provided "up" arrow icon, to indicate that our script will cause profits at your company to rise and rise. "Down" arrows are also available. You will usually need to also give the `.desktop` file executable permissions, using `chmod u+x myscript.desktop` or similar.

There are many other options you can set in this file, see the following standards site for more info. Support for some options varies from distribution to distribution.

Further Reading

Freedesktop.org Desktop Entry Specification standard
http://standards.freedesktop.org/desktop-entry-spec/latest/

3.6.2 Clickable icons : Windows

As mentioned previously there are two executables for Windows, the standard `php.exe` and the newer `php-win.exe` which allows windowless running of scripts. There are likewise two ways to make "clickable" scripts.

First is the simple Windows Batch File. This is a simple text file with a list of commands to execute. Batch files have the `.bat` extension in their name, and are executable by default. So if we have a PHP script called `display_stats.php` that displays a list of numbers and then exits, and we want to run it in Extended Information Mode, we can make it a clickable script by creating a file called, say, `our_stats.bat` with the contents :

```
"C:\Program Files (x86)\PHP\php.exe" -e "C:\users\Rob\PHP
        Scripts\display_stats.php"
pause
```

If you click on `our_stats.bat` it will open a command prompt window, execute our PHP script and display our statistics, and finally wait for the user to press a key before closing the command

prompt window. The last step, achieved here with the `pause` command, is important, as the command prompt window will close again as soon as the batch file finishes executing, and if you have output for the user to view you will want to keep it open. This can of course be achieved directly in your PHP script by pausing execution or waiting for user input, but the script above shows how you can mix PHP and other commands in the same batch file. Don't forget to change the paths to `php.exe` and your script as appropriate. You can use relative filenames or rely on default search paths, but it is often best to use the full path name to ensure that the correct instance of `php.exe` is used and avoid problems if moving your PHP script relative to the batch file.

The second way of making a clickable PHP script is by calling it from within a Windows VBScript. The example above using a batch file will always open a command prompt window, even if you use `php-win.exe` rather than `php.exe`, as it does it by default to execute the commands in the file. A VBScript on the other hand needs no command prompt or other visual form by default. So if we want to use `php-win.exe` to run a script that either provides it's own GUI interface or runs hidden in the background, we should create a script with a `.vbs` extension. Lets say we have PHP-GTK script (more on that in chapter 5) called `text_editor.php` that provides a visual text editor like Notepad, we can make it clickable by creating a file called, say, `our_text_ed.vbs` with the following contents:

```
1  Set WinScriptHost = CreateObject("WScript.Shell")
2
3  WinScriptHost.Run Chr(34) & "c:\Program Files (x86)\PHP\php-win.exe" &
4  Chr(34) & " -e c:\Users\me\text_editor.php", 0
5
6  Set WinScriptHost = Nothing
```

Click on the .vbs file and your PHP script should spring into life, sans command line prompt. The `WinScriptHost.Run` line does the heavy lifting of running a command that we provide. We format that command in the same way we would if we were typing it in by hand at the command prompt, and pass it as a string. Note that in VBScript `&` is the string concatenation operator (the same as `.` in PHP), and that `Chr(34)` is the `"` double quote character (needed as the directory path to `php-win.exe` has a space in). Don't forget to change the paths to the `php-win.exe` and your script as appropriate.

The final piece of the puzzle, whichever of the two methods you use, is to give your clickable file a better icon. The files will have the default icon for Batch Files or VBScripts, and it is not possible to change this directly without changing it for all files of the same type. There are however two workarounds. Windows allows you to change the icon of a short-cut (via `Right-Click -> Properties`), so the first workaround is to hide your script file somewhere out of sight, and create a short-cut icon where you want to be able to click it. Then change the icon of the short-cut. The other workaround is to change the file extension to one that Windows does not recognise, e.g. `.phpsc` and create a new file extension with the same actions as `.vbs` or `.bat` but with a different icon.

3.6.3 Clickable icons : Ubuntu Unity

To add an icon to the Unity Launcher in Ubuntu, we create a .desktop file in a similar manner to the Linux launcher we created before, but with a few more options. So lets create a new `myscript.desktop` :

```
1  [Desktop Entry]
2  Name=My Super Script
3  Exec=/usr/bin/php /home/rob/scripts/myscript.php
4  Icon=/usr/share/icons/gnome/32x32/actions/up.png
5  Terminal=true
6  Type=Application
7  StartupNotify=true
8  Actions=Window;
9
10 [Desktop Action Window]
11 Name=Open me a new window please
12 Exec=/usr/bin/php /home/me/scripts/myscript.php -n
13 OnlyShowIn=Unity;
```

Don't forget to make it executable with chmod. This should work in its current location as with the previous Linux example. However, if you drag it to the Unity Launcher (or "pin" it while the script is running), it will remain there and add a new feature, the "action" menu. Notice the `Actions=Window` line and the `[Desktop Action Window]` section in the code above. These define a new action, (in this case opening a new window, if our script had that capability via the `-n` flag) which is accessible by right-clicking on the icon. Through this you can add any actions you want, by for example calling your script with different parameters, or even calling a completely different script in the `Exec` line. If you want to automatically add Unity entries, or make them available to all users, I suggest the following articles.

Further Reading

"Unity: adding items to the dock" by Daan van Berkel
http://themagicofscience.blogspot.co.uk/2011/05/unity-adding-items-to-dock.html

Unity Launcher Documentation and Guide at Ubuntu.com
https://help.ubuntu.com/community/UnityLaunchersAndDesktopFiles

All of the methods described above can be executed in any number of scenarios. You can use them directly from a command line, as part of a shell script, as a cron job, as a system call from other PHP (and non PHP) scripts, as default file type handlers and so on. With appropriate precautions and permissions, you can even use these methods to call scripts from web based PHP pages (there is usually a better and safer way to invoke them than via a web page, so we'll leave the details of how to implement and secure it as an exercise for the reader if you really must do so).

3.7 Quitting your script

We've looked at starting your scripts, but what happens when it comes time to finish running them?

Like web based PHP scripts, CLI scripts will terminate happily when we hit the end of the script file, and will tidy up all the resources used in the same way. Likewise, if we want to end early, we can call the `exit` language construct (or the equivalent `die` construct).

However in the world of CLI scripts this isn't considered very polite. Because they are designed to work together, often in chains of commands, most shell programs and scripts will provide an "exit code" when they terminate to let the other programs around them know *why* they finished. Were they done? Did they encounter an error? Were they called incorrectly? Enquiring minds want to know.

It is particularly important to supply an exit code when your script may be the last item in a shell script, as the exit code of the script as a whole is taken to be the last exit code returned within it. We can make our PHP scripts provide an exit code, simply by including it as a parameter to `exit` or `die`. An exit code is an integer, and there are a number of common exit codes :

- **0** : Success - we've exited normally.
- **1** : General Error - usually used for application/language specific errors and syntax errors
- **2** : Incorrect Usage
- **126** : Command is not executable - usually permissions related
- **127** : Command Not Found
- **128+N** (up to **165**): Command terminated by POSIX Signal number N - e.g. In the case of `kill -9 myscript.php` it should return code 137 (128+9)
- **130** : Command terminated by Ctrl-C (Ctrl-C is POSIX code 2)

Further Reading

POSIX Signals on Wikipedia
http://en.wikipedia.org/wiki/Unix_signal#POSIX_signals

Any other positive integer is generally construed as exiting due to an unspecified error. So, for instance, if you decide the command line arguments provided by your user are not in the correct format, you should terminate your script using `exit(2)`. If instead all goes well and your script continues to the end of its script file, you can actually let it exit by itself (or by calling `exit` without a parameter), as it returns status code `0` by default.

As with web scripts, you can register functions to be executed when your PHP script exits using the `register_shutdown_function()` function. One use for this may be to check that all is well and evaluate which exit code should be returned. The exit code used as the parameter to `exit` or `die` within a registered shutdown functions overrides the code used in the initial exit call that initiated shutdown. This means that you can happily `exit(0)` everywhere, then `exit(76)` from your shutdown function if you detect that the foo conflaganation isn't aligned with the bar initispations in your metaspacialatific object. Or similar.

4 Development tools

So far we've outline some of the basics of using PHP without a web server, but before we really get into the nitty-gritty details of developing useful non-web software we're going to take a slight detour into the world of PHP development tools. This is for two reasons. Firstly, before deciding which techniques and methods you are going to use (Interactive CLI script or GUI interface) it is important to understand what tools are available to help support your choice of programming methodology. Secondly, you will often need to adopt a different workflow when developing a new type of software that you may not be familiar with, so you may need to find new tools to fit the new way of working.

In general, you can use the same development tools for general purpose programming as you do for web development, after all PHP syntax is the same wherever its executed. The main differentiator is your workflow, and you may find yourself being more productive with different tools.

Of course one of PHP's dirty secrets is that many PHP programmers don't use debuggers, unit testing, build systems or code profilers, and in many cases don't even know what they are. Usually it's because many PHP scripts are developed as small, uncomplicated, low-risk projects by single developers, and the overhead of learning and deploying extra tools is an unnecessary burden (particularly for those without a formal computer science or programming education). Don't worry if you fall into this camp, we've all been there (and like to re-visit it from time to time!).

However when you start venturing into general purpose programming, you'll find that the complexity of projects often increases. Rather than a selection of short lived scripts with limited shared state executing in less than a second, you'll get into larger code bases with longer running processes and more complex dependencies. In such scenarios, the effects of bugs can be magnified and tracking them down becomes increasingly hard. Code is revamped less often and can be relied upon by an organisation and remain deployed for much longer, increasing the burden of maintaining code in the future. If you're not a regular user of the type of tools described in this chapter, you might find this the ideal opportunity to delve into the wider world of PHP development tools, and give yourself a good foundation on which to start building real software.

We'll look in turn at different classes of tool, and give some examples of, and links to, popular tools of each type.

4.1 PHP REPLs

When you want to test out a few lines of PHP, your default instinct may be to create a new PHP file, save it, and then execute it with PHP. There is a better, faster and more interactive, way however. The PHP "Interactive Shell", also known as the PHP REPL (Read-Eval-Print-Loop), is a quick and easy way to type in code and have it execute immediately. Unlike executing single lines of code using `php -r`, the REPL (started by calling `php -a`) keeps the scripts state (e.g.

contents of variables and objects) in-between each line that you type, until you exit. You can use all of PHP's functions, although no libraries are loaded by default, and you can `include()` or `require()` existing files of PHP code. This latter capability is useful for debugging the final output of a problematic script; simply `include()` your script which will execute the script and, as long the script doesn't terminate prematurely, then you can `echo()` or `print_r()` or otherwise explore the state of the variables and other resources at the end of the run. The following example is a capture of an actual interactive REPL session using the standard PHP REPL. Other brands of REPL are available, and are listed later in this section.

```
~$ php -a
Interactive shell

php > # As we can type any valid PHP, I have added comments
php > # directly to the REPL, rather than afterwards in editing!
php >
php > # Lets start with some simple assignments :
php >
php > $a = 5;
php > $b = 6;
php >
php > # The REPL will throw Notices, Warnings and Errors as appropriate,
php > # in real-time :
php >
php > $c = nothingdefined;
PHP Notice:  Use of undefined constant nothingdefined - assumed
'nothingdefined' in php shell code on line 1
php >
php > # Just as with normal PHP source files, we can split commands across
php > # lines. The interpreter only kicks in when it hits the terminating
php > # semicolon :
php >
php > $d
php > =
php > 7
php > ;
php >
php > # The following shows that the state in the variables above has been
php > # kept :
php >
php > echo $a + $b + $c + $d ."\n";
18
php >
php > # Next, a more interesting example. Use the REPL instead of the
php > # shell to get a line from a file :
php >
php > echo file ('/proc/version')[0];
```

```
Linux version 3.5.0-21-generic (buildd@roseapple) (gcc version 4.7.2
(Ubuntu/Linaro 4.7.2-2ubuntu1) ) #32-Ubuntu SMP Tue Dec 11 18:52:46 UTC
2012
php >
php > # Of course all of the usual protocol wrappers are available, so we
php > # can see what is happening in the world...
php >
php > $page = file ('http://news.bbc.co.uk');
php >
php > echo $page[0];
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN" "http://www.w3.org/
MarkUp/DTD/xhtml-rdfa-1.dtd">
php >
php > # and maybe get a hash of that...
php >
php > echo md5 ( implode ( $page, "\n" ) ) . "\n";
0319bf4e62db39fb2c89210e48783d70
php >
php > # when we are done ...
php >
php > exit;
php >
php > # doesn't work, as its just evaluated as PHP (and the REPL ignores
php > # exit/die calls. To exit the REPL, enter the word 'exit' on its own
php > # on a new line
php >
php > exit
~$
```

Sometimes you'll want to execute your commands within the "environment" of other scripts. For instance, you may have a script that declares constants, sets up database connections and does other routine tasks that you normally `include()` at the start of your main PHP scripts. As noted above you can `include()` these files in the REPL too, but you may forget to do so and then wonder why things didn't work as they should. One facility PHP offers us, which applies not only to the REPL but to all forms of PHP execution, is the `auto_prepend_file` configuration directive. This tells PHP to execute a given file each time PHP is run before it starts to do anything else (like executing the script you have asked it to). This can either be set in php.ini, or via the `-d` flag on the command line. The following is an example of presetting some constants/variables. First, we create a script called `initialise.php` with the following content.

```
<?

const FOUR = 4; # Declare a constant value

$five = 5; # Instantiate a variable with another value
```

and then, at the command line, start and run a REPL session as follows, using `-d` to execute the `initialise.php` script first.

```
$> php -d auto_prepend_file=initialise.php -a
Interactive shell

php > echo (FOUR + $five)."\n";
9
php > exit
$>
```

As you can see, the constant and variable we had set up in the initialise.php file were available for use from the REPL without having to manually declare them. The `-d` flag is used here, but the option could be set in php.ini as well if you want to always use the same file. If you regularly use a few different initialisation files like this, you can create shell aliases to commands using the `-d` flag. For instance, you could add lines similar to the following to your `~/.bash_profile`:

```
alias php-cl="php -d auto_prepend_file=clientSetup.php -a"
alias php-in="php -d auto_prepend_file=ourSiteSetup.php -a"
```

As well as the built in PHP REPL explored above, there are a number of third-party REPLs available, some of which include features such as a history of commands typed, tab-completion of commands, protection from fatal errors and even abbreviated function documentation.

phpsh

Developed at Facebook, phpsh is an interactive shell for PHP that features readline history, tab completion, quick access to documentation

Main website & documentation : http://phpsh.org/

Installation info :
https://github.com/facebook/phpsh/blob/master/README.md

php shell

A REPL with tab-completion, fatal error handling, inline help and default autoload() inclusion.

Main website, installation info & documentation :
http://jan.kneschke.de/projects/php-shell/

phpa

A simple replacement for `php -a`, written in PHP.

Main website, installation info & documentation : http://david.acz.org/phpa/

PHP Interactive

An web based REPL that allows better support of displaying HTML output. Project is an Alpha release.

Main website : http://www.hping.org/phpinteractive/

Installation info : Readme file in source download
http://www.hping.org/phpinteractive/phpinteractive-0.2.tar.gz

Main documentation : None

Boris

A small but robust REPL for PHP

Main website : https://github.com/d11wtq/boris

Main documentation & installation info :
https://github.com/d11wtq/boris#usag

Extension for Symfony and Drupal :
http://vvv.tobiassjosten.net/php/php-repl-for-symfony-and-drupal/

Sublime-worksheet

An inline REPL for the Sublime Text editor.

Main website : https://github.com/jcartledge/sublime-worksheet

4.2 Build systems

Build systems are used for building and deploying software to development and/or live systems. They automate many of the repetitive tasks that occur when deploying a new revision of a software system. Build scripts and build systems originated with languages like C which required compilation before the software was able to run, and covered steps such as compiling and linking various software modules as well as managing source code dependency issues. Although PHP is not a compiled language, a build system can still be a great time saver performing tasks like :

- gathering and managing assets (images, data files etc.)
- packaging or moving scripts and assets
- archiving older versions
- running automated tests and other quality control processes
- running code minification or obfuscation processes
- starting and stopping related services
- refreshing and resetting your test environment

- cleaning or initialising databases
- documentation generation
- team notification

or indeed anything else needed to ensure the latest version of your script is deployed successfully. As well as saving time, it also helps to ensure consistency and prevent mistakes. As you'll no doubt know, trying to solve an obscure bug that occurred because you forgot to copy over the latest copy of a minor data file isn't fun.

You can of course construct your own build system using shell scripting (or even better, PHP CLI scripts!), which may suffice for small or simple projects. For larger or more complex systems an existing build system may be a time saver. Many existing systems can cope with PHP, but one of the best for PHP work is Phing. Phing was built for PHP, and can be easily extended with PHP classes and has a wide range of PHP related tasks already built in. Telling Phing what to do is a matter of creating simple XML files, which you can even create programmatically using the PHP XMLwriter extension if you need to.

Phing

The leading PHP build system, modelled on Apache Ant, a Java build system.

Main website : http://www.phing.info/

Installation info : http://www.phing.info/trac/wiki/Users/Installation

Main documentation : http://www.phing.info/trac/wiki/Users/Documentation

Tutorials

"Deploy and Release your Applications with Phing" by Vito Tardia
http://phpmaster.com/deploy-and-release-your-applications-with-phing/

"Automating software development and deployment" by Grzegorz Godlewski
http://blog.twelvecode.com/2012/06/20/automating-software-development-and-deployment/

4.3 Continuous Integration

One step beyond build systems are Continuous Integration (CI) systems. Particularly useful in large, team-developed projects, CI systems allow multiple developers to continuously (or,

regularly) integrate their individual development work into the final complete software system. The CI system typically takes individual user commits from a version control system and builds ("integrates") them into the final "product" along with other developers work, typically testing and deploying the build as it goes. With CI systems, developers are usually encouraged to make at least one commit each day, or more often if possible. In projects without CI, teams often developed on their own, and integrated their work with each others at the end of the process. In a large project with many developers the integration stage is often lengthy and is a frequent cause of time (and money) over-runs, as incompatibilities and problems are found and fixed. CI systems effectively remove this integration step as integration is done from day one. Problems are revealed early on and fixed on an on-going basis, with rewrites of incompatible code kept to a minimum. When the project is finished, the work of different developers or teams is more or less guaranteed to be interoperable and fully integrated into the project.

Further Reading

An in-depth primer article on CI from software engineer Martin Fowler
http://martinfowler.com/articles/continuousIntegration.html

CI on Wikipedia, with background, principles of CI and links to CI software
http://en.wikipedia.org/wiki/Continuous_integration

There are a growing number of CI systems, most of which can cope with PHP projects, and indeed for smaller projects CI systems can be "hand-rolled" with judicious use of existing build systems. Some of the more popular off-the-shelf ones for use with PHP are listed below.

Jenkins

A popular, versatile and extensible open source CI server, with PHP support

Main website : http://jenkins-ci.org

Installation info :
https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins

Main documentation : https://wiki.jenkins-ci.org

Tutorials

PHPMaster tutorial on using PHP and Jenkins
http://phpmaster.com/series/continuous-integration-with-jenkins/

“Zero to Jenkins - PHP Continuous Integration” by David Adams - Video
https://www.youtube.com/watch?v=PklYO2vYIfc

Books

Integrating PHP Projects with Jenkins, by PHPUnit creator Sebastian Bergmann
http://shop.oreilly.com/product/0636920021353.do

Other related resources

Jenkins jobs template for PHP projects
http://jenkins-php.org/

Plugin to allow you to capture code coverage reports from PHPUnit in Jenkins
https://wiki.jenkins-ci.org/display/JENKINS/Clover+PHP+Plugin

Plugin to allow you to use the Phing build system to build PHP projects in Jenkins
https://wiki.jenkins-ci.org/display/JENKINS/Phing+Plugin

phpci

A CI system designed specifically for PHP, written in PHP, which integrates easily with tools like Composer, PHP Unit and so on.

Main website : http://www.phptesting.org/

Main documentation and installation info :
https://github.com/Block8/PHPCI#phpci

Travis CI

The CI system used at Github. Supports PHP. Available as a download or commercial hosted service.

Main website : http://travis-ci.org/

Commerical hosted version : http://travis-ci.com/

Main general documentation : http://about.travis-ci.org/docs/

Main PHP documentation : http://about.travis-ci.org/docs/user/languages/php/

Sismo

A Continous Testing Server (a sub-set of CI) written in PHP from Fabien Potencier, the creator of the Symfony PHP framework

Main website, documentation and installation info :
http://sismo.sensiolabs.org/

Criterion

Criterion is a Continuous Integration app built in PHP.

Main website, documentation and installation info :
http://romhut.github.io/criterion/

4.4 Debuggers

A debugger provides an easy way for you to inspect the internal state of your application, often while it runs, at fixed points in the code, or after a crash. This can give you valuable insight into bugs, particularly those in long running code or those relating to how external data affects variables, as well as direct flaws in your code. Debuggers can give you a "stack trace", which shows the nested set of functions (or "stack") you are currently in. They can also provide the current contents of variables, objects and their members, the state of resource identifiers and so on. Some additionally provide profiling capabilities (see the section on profilers later in this chapter).

There are a number of debuggers available for PHP, the most prolific (and arguably the most versatile) is Xdebug.

Xdebug

A versatile, comprehensive debugger and profiler.

Main website : http://xdebug.org/

Installation info : http://xdebug.org/docs/install

Main documentation : http://xdebug.org/docs/

VIM Interface for Xdebug : https://github.com/joonty/vdebug

MacGDBp

A Mac-only debugger, sits on top of Xdebug.

Main website : http://www.bluestatic.org/software/macgdbp/

Main documentation and installation info :
http://www.bluestatic.org/software/macgdbp/help.php

VLD (Vulcan Logic Dumper)

An advanced tool by the author of Xdebug. It allows you to dump the opcodes for a script to aid in debugging.

Main website, documentation & installation info :
http://derickrethans.nl/projects.html#vld

Article

“Print vs echo, which one is faster?” - the age-old, irrelevant, debate solved by VLD
http://fabien.potencier.org/article/8/print-vs-echo-which-one-is-faster

Zend Studio Debugger

The commercial Zend Studio IDE contains an integrated debugger

Main website, documentation & installation info :
http://www.zend.com/en/products/studio/

How-to

"Debugging a PHP CLI script" with Zend Studio by Kevin Schroeder
http://www.eschrade.com/page/debugging-a-php-cli-script/

PHP DebugBar

Displays basic debugging information as part of the PHP script output

Main website, documentation & installation info : http://phpdebugbar.com/

APD (Advanced PHP Debugger)

An official PHP Debugger. It has not been updated for a few years, but appears workable.

Main website : http://pecl.php.net/package/apd

Installation info : http://www.php.net/manual/en/apd.installation.php

Main documentation : http://php.net/manual/en/book.apd.php

4.5 Testing and Unit Testing

There are many strategies for testing all types of code, including PHP, and many full books have been written on the subject, so we won't cover testing here except to make a couple of salient points and introduce a couple of PHP related tools.

As your development projects increase in size (and in general purpose programming, it's likely you will be creating much longer and more complex scripts than when programming for the web), testing becomes more and more important even for developers working on their own. Your ability to keep all of the details of the code you've written in your head diminishes. Occurrences of changes that, when made to one section of code, cause problems for other parts of the software, increase.

There are many types of software testing methodologies, but "unit testing" in particular can help keep these things straight. For the uninitiated, unit testing involves creating tests for individual sections of code (such as functions, classes, methods etc.) which test the outputs of that portion of code against the specification/expected behaviour for that code. For instance, for a given function `Foo()` you may write a test to check that it will always output values between `1` and `10` for any given input except the value `bar` which should return the value `-1`. You can then use that function elsewhere and have confidence that it will always return value within the specified range no matter what you throw at it. If you later make a change to how `Foo()` is implemented, your unit test will pick up if the range of output values `Foo()` gives has changed (for instance, if it now also outputs `0` and `-2` in certain conditions), and it will fail the test. This will alert you to the fact that either you have a bug in the new code you have put into `Foo()`, or your specification for `Foo()` has changed and you will need to check carefully the places you have called `Foo()` from to make sure that the new range of output values won't cause issues elsewhere. Without such a test, it is easy to fail to notice the possible knock-on effects of code changes, in particular on how they may affect other potentially distant sections of code.

Before you start programming though, it's important to consider what type of testing you intend to use and plan your design accordingly. Some types of testing require a firm specification for the software from the start, others (like unit testing) are made easier by structuring your code in certain ways to make tests easier to construct and maintain.

Further Reading

Test Driven Development on Wikipedia
http://en.wikipedia.org/wiki/Test-driven_development

A reference for structuring PHP programs to make Unit Testing easier using Test Driven Development, "The Grumpy Programmer's Guide To Building Testable PHP Applications" by Chris Hartjes…
http://leanpub.com/grumpy-testing
… and a 5 Minute video introduction to "Test Driven Development" also by Chris
http://www.littlehart.net/atthekeyboard/2012/08/16/5-minute-tdd/

An assortment of popular PHP testing tools are listed below.

PHPUnit

PHPUnit is the de-facto standard for unit testing in PHP projects

Main website : http://phpunit.de/

Installation info :
https://github.com/sebastianbergmann/phpunit/blob/master/README.md

Main documentation : http://phpunit.de/manual/current/en/index.html

Tutorials

“Let’s TDD a Simple App in PHP” by Patkos Csaba
http://net.tutsplus.com/tutorials/php/lets-tdd-a-simple-app-in-php/

“Bulletproofing Database Interactions with PHPUnit Database Extension” by Jeune Asuncion
http://phpmaster.com/bulletproofing-database-interactions/

“Debugging PHPUnit Tests in NetBeans with XDebug” by Rafael Dohms
http://blog.rafaeldohms.com.br/2011/05/13/debugging-phpunit-tests-in-netbeans-with-xdebug/

Video : “Leveraging 12 Years of PHPUnit” by Sebastian Bergmann (PHPUnit’s author)
http://thephp.cc/viewpoints/blog/2013/09/drupalcon-europ

Codeception

Describes itself as “PHPUnit on steroids”. Covers Unit Testing, Acceptance Testing and Functional Testing, with a focus on ease of use.

Main website : http://codeception.com/

Installation info : http://codeception.com/quickstart

Main documentation : http://codeception.com/docs/01-Introduction

Enhance-php

Enhance-PHP is a lightweight Open Source PHP unit testing framework written in PHP

Main website : http://www.enhance-php.com

Installation info :
https://github.com/Enhance-PHP/Enhance-PHP/wiki/Quick-Start-Guid

Main documentation : https://github.com/Enhance-PHP/Enhance-PHP/wiki

Atoum

A simplified unit testing framework, aiming for simplicity and rapid implementation.

Main website : https://github.com/atoum/atoum

Main documentation & Installation info :
https://github.com/atoum/atoum/blob/master/README.md

SimpleTest

A simple unit testing and web testing framework.

Main website : http://www.simpletest.org/

Main documentation and installation info :
http://www.simpletest.org/en/overview.html

4.6 Static code analysis

Static code analysis is the process of testing or examining code without executing it. The above examples of testing and debugging all involve running the code to look for bugs and issues. However there are times when checking the code for potential errors before executing it can be beneficial. Many simple errors such as bad syntax (misspelling function names, forgetting closing brackets, using non-existing operators and so on) can be picked up before running code. Code that takes a long time to run, or where the test environment and data take a while to set-up for each run, can benefit from being statically analysed to catch glaring errors before a run starts. It can be particularly galling to happen across a fatal error in a script `include()`'d at the end of an hour long data analysis run which could have been caught before you even started! Of course static analysis tools can't catch every kind of bug, they have no knowledge of the data our scripts will encounter or the environment in which they will run, or indeed our expectations of what the script will achieve.

A programming "lint" is a simple type of static code analysis tool. "Lint" was the name given to one of the first analysis tools for C many years ago, but like "Hoover" it has become a generic term for static analysis tools. PHP has a number of lint tools, including one built right into PHP itself. To use the built in lint, simply call `php -l yourfile.php` and any syntax errors found will be printed to the terminal.

Other 3rd party lints and similar tools go beyond syntax checking and can also check your code against particular coding standards that you may choose to use. Sticking to a given coding standard won't prevent bugs by itself, but can help code readability and maintainability particularly if multiple developers are involved. It doesn't really matter which coding standard you use from the many available, but if you do use one it is important that you use it consistently and across all of your code, and tools like these can help with doing that. Some tools, such as RIPS can look for code patterns which are indicative of particular faults (such as security vulnerabilities) but which aren't specific violations of a coding standard or a syntax error. The output of tools like these requires a bit of further consideration, as they are only indicators of potential problems. Whether a particular item of code, in your particular case, constitutes an actual security vulnerability will depend on the use case and code surrounding it. That said, even if a code scanner like RIPS detects something that isn't currently a security problem as your script is structured now, it may be worth changing that section (or adding strict unit tests to it) to ensure that any changes to related code in the future won't allow it to become exploitable then.

Further Reading

How (PHP founder Rasmus Lerdorf's employer) Etsy.com implements static analysis for it's PHP stack
http://codeascraft.etsy.com/2012/08/10/static-analysis-for-php/

"PHP Static Analysis in Sublime Text" by Phil Sturgeon
http://philsturgeon.co.uk/blog/2013/08/php-static-analysis-in-sublime-text

PHPLint

A comprehensive PHP lint tool, available to download or use online

Main website, documentation & installation info :
http://www.icosaedro.it/phplint/

Online tool : http://www.icosaedro.it/phplint/phplint-on-line.html

PHPHint

A PHP lint tool, available to download or use online, which can also attempt to clean code to PSR standards. A front end to other projects, and an Alpha release.

Main website and online tool : http://phphint.org

Main documentation & installation info :
https://github.com/klaussilveira/PHPHint/

PHP_Codesniffer

Pear package for detecting violations of a defined set of coding standards

Main website : http://www.squizlabs.com/php-codesniffer

Installation info : https:
//github.com/squizlabs/PHP_CodeSniffer/blob/master/README.markdown

Main documentation :
http://pear.php.net/manual/en/package.php.php-codesniffer.php

PHP Depend

A static analysis and metric generation tool. Recommended for use with PHP Mess Detector below.

Main website : http://pdepend.org/

Installation info : http://phpmd.org/download/index.html

Main documentation : http://pdepend.org/documentation/getting-started.html

PHPMD (PHP Mess Detector)

An spin-off tool that uses PHP Depend to look for bugs, suboptimal code and more.

Main website : http://phpmd.org/

Installation info :
http://pdepend.org/documentation/handbook/installation.html

Main documentation : http://phpmd.org/documentation/index.html

PHPLOC

A simple tool that outputs a range of statistics about your code.

Main website : https://github.com/sebastianbergmann/phploc

Main documentation & installation info :
https://github.com/sebastianbergmann/phploc/blob/master/README.md

PHP Analyzer

An advanced static analysis tool. Designed for use as part of a CI workflow but can be used as a stand-alone tool as well.

Main website : https://github.com/scrutinizer-ci/php-analyzer

Installation info : https://github.com/scrutinizer-ci/php-analyzer#installation

Main documentation : https://scrutinizer-ci.com/docs/tools/php/php-analyzer/

RIPS

An analysis tool primarily aimed at security vulnerability detection. Aimed at websites, but can analyse CLI code too.

Main website : http://rips-scanner.sourceforge.net/

Installation info : http://rips-scanner.sourceforge.net/#download

Main documentation : None

PHP_CodeCoverage

An advanced tool for testing code coverage (the code actually executed in a run).

Main website : https://github.com/sebastianbergmann/php-code-coverag

Main documentation & installation info :
https://github.com/sebastianbergmann/php-code-coverage/blob/master/README.markdown

PHP_sat

A general static analysis tool for PHP. Unstable release only.

Main website : http://www.program-transformation.org/PHP/PhpSat

Installation info : http://www.program-transformation.org/PHP/PhpSatDocumentation#Installation

Main documentation : http://www.program-transformation.org/PHP/PhpSatGettingStarted

Going beyond basic Lint type capabilities and single purpose scanning, there are a number of advanced open source and commercial analysis platforms available for PHP that provide a much wider range of analysis and reporting capabilities, including various code quality metrics, code duplication analysis, security pattern analysis and more, as well as integration with other tools such as unit testing software and IDEs. These tools often have a non-negligible learning curve and deployment/use "cost" (in terms of time and resources), and are more suited to larger projects, mission critical projects and projects with multiple developers. However even when starting small with a single developer, if you honestly believe that your project will scale up it may be worth considering using such a tool from the start to help minimise "technical debt" building up to be "called in" later in the project. Such tools include :

Sonarqube

An extensive open platform for managing code quality through code analysis.

Main website : http://www.sonarqube.org/

Installation info : http://docs.codehaus.org/display/SONAR/Installation+and+Upgrad

Main documentation : http://docs.codehaus.org/display/SONAR/User+Guid

PHP specific plugin : http://docs.codehaus.org/display/SONAR/PHP+Plugin

Understand

A professional, commercial, source code analysis & metrics tool

Main website : http://www.scitools.com/

Main documentation & Installation info :
http://www.scitools.com/support/manuals.php

4.7 Virtual development & testing environments

When you develop for the web, you will usually control or at least specify the environment in which your PHP scripts are deployed (i.e your web server). When writing general purpose software, this often isn't the case, particularly where you sell or otherwise distribute your software to others. You can specify Operating Systems and other software/hardware pre-requisites for your software to match your own development environment, but it is often more desirable to make your software work on as wide a range of platforms as is possible. However this can raise issues when developing and testing as you will require access to all of the platforms that your users will use. It used to be the case that you would require many different physical machines to run different Operating Systems and configurations. However a modern solution for many cases is virtualisation, running different platforms on the same hardware in "virtual machines". There are a number of different available virtualisation solutions, but one of the easiest (and cheapest) to use is VirtualBox from Oracle. Simply install VirtualBox, create and run a new virtual machine, and install an operating system as you would if you were setting up a physical machine from scratch. You can install most common operating systems including Linux and Windows variants, assign various levels of memory, processor and disk space, and run many different virtual machines at the same time. You can also take "snapshots" of a virtual machine at a given time, and roll-back to a given snapshot when you need to. This is useful if your testing environment needs setting to an initial state (data, machine state, software state etc.) before each run. VirtualBox doesn't have all the bells and whistles of some virtualisation environments and sometimes has lower performance characteristics than those that run at a lower level, however for many cases it is perfectly performant and versatile.

VirtualBox

An easy to use virtualisation system.

Main website : https://www.virtualbox.org/

Installation info :
https://www.virtualbox.org/manual/ch01.html#intro-installing

Main documentation : https://www.virtualbox.org/wiki/Documentation

A note of caution : When using virtualisation with commercial operating systems such as Microsoft Windows, you always need to be aware of licensing issues. Many commercial OSes require separate or additional licenses for virtual machines, even when you are using a fully licensed "host" operating system. In particular the licensing requirements & costs for server variants can vary depending on not just the number of virtual machines installed but on the number of processor cores used by each virtual machine (a particular headache if you regularly vary the number of virtual cores provided to a given VM for testing purposes).

Virtualisation can also be useful for development environments. It can be a pain to set up all your preferred tools on a new machine, or configure a machine to match a development environment specified for a team or company. Having your development environment as a portable virtual machine, or deploying a new virtual machine to a predefined recipe using a tool like Vagrant can make life a lot easier.

Vagrant

Create and configure lightweight, reproducible, and portable development environments.

Main website : http://www.vagrantup.com/

Installation info : http://docs.vagrantup.com/v2/installation/index.html

Main documentation : http://docs.vagrantup.com/v2/

Further Reading

"Make $ vagrant up yours" by Juan Treminio
An article on Vagrant, Puppet and PuPHPet.
https://jtreminio.com/2013/06/make_vagrant_up_yours/

4.8 Source/version control systems & code repositories

Version control systems and code repositories help you to manage different versions and branches of code, share and deploy code, track errors and changes, and can be useful in most projects. Virtually all common version control systems and code repositories work with PHP, including SVN, Git, CVS and others, and there are no PHP specific reasons to recommend one over another. There are no particular PHP specific tools to note here, other than to mention that most build systems (see the section above) can be configured to work with most of these systems. There are several PHP code bindings of note in this area, a binding for the Git system, an API wrapper for the popular Git based host Github.com and a pecl package for interacting with SVN repositories. These libraries can be used in PHP based build systems, for example, although they do not require knowledge of the Git or SVN systems.

PHP Git Bindings

PHP bindings for libgit2, the Git implementation used by GitHub, Microsoft and others.

Main website : https://github.com/libgit2/php-git

Installation info : https://github.com/libgit2/php-git#installing-and-running

Main documentation : https://github.com/libgit2/php-git#api

Gittern

A library for reading/writing Git repositories, that doesn't depend on the Git binary.

Main website : https://github.com/e-butik/Gittern

Installation info : http://gittern.readthedocs.org/en/latest/#installation

Main documentation : http://gittern.readthedocs.org/en/latest/

PHP GitHub API

A simple Object Oriented wrapper for the GitHub API

Main website : https://github.com/KnpLabs/php-github-api

Installation info : https://github.com/KnpLabs/php-github-api#requirements

Main documentation : https://github.com/KnpLabs/php-github-api#basic-usage-of-php-github-api-client

Tutorial

"Talking to GitHub with PHP" by W.J. Gilmore
http://www.phpbuilder.com/columns/github/github-api-php_11-29-2011.php3

svn

PHP Bindings for the Subversion Revision control system

Main website : http://pecl.php.net/package/svn

Installation info : http://www.php.net/manual/en/svn.setup.php

Main documentation : http://www.php.net/manual/en/book.svn.php

Further Reading

Free Apress "Pro Git" book by Scott Chacon
http://git-scm.com/book

A practical tutorial on using GitHub with a PHP project by Lorna Jane Mitchell
http://www.lornajane.net/posts/2012/do-open-source-with-git-and-github

4.9 IDEs and editors

IDEs (Integrated Development Environments) and code editors for PHP are plentiful, and you will likely have your own favourite you use for web based PHP development. There are currently no IDEs or Editors specifically aimed at PHP CLI-based programming, but as such PHP code is syntactically compatible and very similar to web based PHP programming, you should have few problems using your existing PHP editing environment. Some additional features may not work correctly, such as web based debuggers, or may require additional tinkering to use. A list of popular PHP IDEs is given in Appendix E at the end of the book.

4.10 Documentation generators

Documentation Generators (also known as documentors) automatically generate documentation from your source code. They do this by using the comments you put in the code and by picking up the syntactical structures in your code. They are useful for creating base documentation for programming APIs and technical documentation for other developers working on your code. The documentation provided usually requires further finenessing or additional information adding to be truly useful and is usually of little use to end users of your software. However it provides a good overview of the code structure along with a useful reference for larger code-bases, where finding the details of particular functions from the code itself can be time consuming. Some IDEs and other programming tools can use the unmodified output from documentors to provide additional functionality, such as project based context specific help and auto-completions.

phpDocumentor

Popular and versatile PHP based documentor

Main website : http://www.phpdoc.org/

Installation info :
http://www.phpdoc.org/docs/latest/for-users/installation.html

Main documentation : http://www.phpdoc.org/docs/latest/index.html

phpDox

A fast modern documentor which can pull in information from other static analysis tools

Main website : http://phpdox.de/

Main documentation and installation info :
http://phpdox.de/getting-started.html

phpSimpleDoc

A simplified documentor which outputs HTML docs

Main website : http://phpsimpledoc.tig12.net/

Installation info : http://phpsimpledoc.tig12.net/user-guide/quick-start#toc_1

Main documentation : http://phpsimpledoc.tig12.net/user-guid

Doxygen

A comprehensive and extensive documentor, which supports PHP but is not PHP specific.

Main website : http://www.stack.nl/~dimitri/doxygen/

Installation info : http://www.stack.nl/~dimitri/doxygen/manual/install.html

Main documentation : http://www.stack.nl/~dimitri/doxygen/manual

Sami

The documentor created and used by the Symphony framework project for their API, although it is not Symphony specific.

Main website : https://github.com/fabpot/Sami

Installation info : https://github.com/fabpot/Sami#installation

Main documentation :
https://github.com/fabpot/Sami/blob/master/README.md

4.11 Profilers

Profilers let you measure and view the execution time and path of your code, usually with a view to increasing the performance of your scripts and removing code bottle-necks. We will look at these in detail in Chapter 9 when we look at further at script performance.

4.12 Other tools

This section lists some further tools that are useful for serious development, but don't fit neatly into the categories above.

Composer Dependency Manager

The easy way to keep libraries consistent and up-to-date on a per-project basis. See Appendix A for more information.

Main website : http://getcomposer.org/

Package Repository : https://packagist.org/

Installation info : http://getcomposer.org/doc/00-intro.md#installation-ni

Main documentation : http://getcomposer.org/doc/

Tutorial : http://hassankhan.me/post/58193034824

PHP Beautifier

Takes your PHP source code and formats it for readability by, for example indenting code, adding new lines where needed and formatting data structures.

Main website : http://pear.php.net/package/PHP_Beautifier

Installation : `pear install PHP_Beautifier`

Main documentation : http://beautifyphp.sourceforge.net/docs/

PHPLighter

Takes your PHP source code and creates a syntax-highlighted HTML version. Uses advanced tokenisation to highlight more features of code. Useful for reviewing and displaying code.

Main website : https://github.com/brandonwamboldt/PHPLighter

Main documentation & installation info :
https://github.com/brandonwamboldt/PHPLighter#usag

PHP Coding Standards Fixer

Attempts to fix your code to meet PSR standards.

Main website, documentation & installation info : http://cs.sensiolabs.org/

PHP Refactoring Browser

A command line refactoring tool for PHP

Main website, documentation & installation info :
http://qafoolabs.github.io/php-refactoring-browser/

phptidy

A tool for formatting PHP code for readability.

Main website, documentation & installation info : http://phptidy.berlios.de/

Phabricator

An open software engineering platform created at Facebook that inclues many tools designed to help create better software. Written in PHP.

Main website, documentation & installation info : http://phabricator.org/

5 User facing software

After slugging through the preliminary information necessary to understand developing PHP in a non-web context, we're now getting to the nitty-gritty of how to start communicating with our users without the rendering engine of a web browser.

Some software sits and happily runs without any interaction from humans, we'll call this "system software" and will examine it in Chapter 6. However a large proportion of software requires interactivity with the user, and there are a number of different ways of performing the interaction in PHP. From interactive command lines through to fully fledged GUI (Graphical User Interface) based software, it can all be done with PHP.

When choosing how your software interacts with humans, you need focus on the needs of the user and put your own preferences to one side if at all possible. Command line interactions are (usually) very straight forward to code, text is, well, just text and it's what PHP excels at. However many users, particularly non-technical users, shy away from text based CLI software. Unless you are meticulous about your software structure and interface flow, text based input can be :

- prone to error
- hard to navigate
- require mental agility on the part of the user
- have low discoverability
- and be initially slower to use than a GUI

If you choose a text based interface, make sure your target users will be happy with it (you may be, but often your users aren't much like you!). Text interfaces, because of the simplicity of coding, can often be a good choice for proof-of-concept and prototype software (particularly where the main benefits of the application are in what it does, not in how the user interacts with it), but be aware that temporary interfaces have an uncanny way of "sticking around". Additionally, some non-technical commissioning users (the words "pointy-haired-boss" spring to mind here) can't see past an interface to the application beneath and may not be happy with your proposed project, even after you explain that the interface is just temporary. This is understandable to some extent as, for most users, the application IS the interface. It is all they see and use day-in day-out. The changes to their files and other functions performed just happen by "magic" of course. Good user interface design (whether it be for text based interfaces or fully-fledged GUIs) is a field of endeavour in and of itself, so get help from the professionals where you can, read some of the multitude of books available on the subject, or take a course. If you ever looked at a piece of software and wondered how it got so popular while your preferred, much more functionally superior software, languished in obscurity, the answer is often at least partly down to good user interface design.

5.1 Command line interface basics

As described above, there are still uses for text-based interfaces, particularly in environments with technically-adept users. When creating a text based program to run on the command line, there are three primary considerations over-and-above the PHP you are already accustomed to. These are

1. Getting keyboard input
2. Outputting text (and graphics) to the screen
3. Program flow control

Rather than describe each one in isolation, we will instead look through a simple program that contains elements of each. Read through the code and comments below. The program is a screen-saver like routine which fills the shell with colour via a wiggling snake-like cursor.

```
1  <?
2
3  # First we will define some named constants.
4  # These are shell escape codes, used for formatting
5  # Defining them as names constants helps to make our code more readable.
6
7  define("ESC", "\033");
8  define("CLEAR", ESC."[2J");
9  define("HOME", ESC."[0;0f");
10
11 # We will output some instructions to the user. Note that we use
12 # fwrite rather than echo. The aim is to write our output back to the
13 # shell where the user will see it. fwrite(STDOUT... writes to the
14 # php://stdout stream. Echo (and print) write to the php://output
15 # stream. Usually these are both the same thing, but they don't have to
16 # be. Additionally php://output is subject to the Output control &
17 # buffering functions (http://www.php.net/manual/en/book.outcontrol.php)
18 # which may or may not be desirable.
19
20 fwrite(STDOUT, "Press Enter To Begin, And Enter Again To End");
21
22 # Now we wait for the user to press enter. By default, STDIN is
23 # a blocking stream, which means that when we try to read from it,
24 # our script will stop and wait some input. Keyboard input to the shell
25 # is passed to our script (via fread) when the user presses Enter.
26
27 fread(STDIN,1);
28
29 # We want the program to run until the user presses enter again. This
30 # means that we want to periodically check for input with fread, but not
```

```
31  # to pause the program if there isn't any input. So we set STDIN to be
32  # non-blocking.
33
34  stream_set_blocking(STDIN, 0);
35
36  # In preparation for our output, we want to clear the terminal and draw a
37  # pretty frame around it. To do this we need to know how big the terminal
38  # window currently is. There is no in-built way to do this, so we call an
39  # external shell command called tput, which gives information about the
40  # current terminal.
41
42  $rows = intval(`tput lines`);
43  $cols = intval(`tput cols`);
44
45  # We now write two special escape codes to the terminal, the first
46  # of which (\033[2J) clears the screen, the second of which (\033[0;0f)
47  # puts the cursor at the top left of the screen. We've already defined
48  # these as the constants CLEAR and HOME at the start of the script
49
50  fwrite(STDOUT, CLEAR.HOME);
51
52  # Now we want to draw a frame around our window. The simplest way to draw
53  # "graphics" (or "semigraphics") in the terminal is to use box drawing
54  # characters that are included with most fixed-width fonts used in
55  # terminals.
56
57  # Draw the vertical frames by moving the cursor step-by-step down each
58  # side. The cursor is moved with the escape code generated by
59  # ESC."[$rowcount;1f"
60
61  for ($rowcount = 2; $rowcount < $rows; $rowcount++) {
62          fwrite(STDOUT, ESC."[$rowcount;1f"."║"); # e.g. \033[7;1f║ for line 7
63          fwrite(STDOUT, ESC."[$rowcount;${cols}f"."║");
64  }
65
66  # Now do the same for the horizontal frames.
67
68  for ($colcount = 2; $colcount < $cols; $colcount++) {
69          fwrite(STDOUT, ESC."[1;${colcount}f"."═");
70          fwrite(STDOUT, ESC."[$rows;${colcount}f"."═");
71  }
72
73  # And finally fill in the corners.
74
75  fwrite(STDOUT, ESC."[1;1f"."╔");
76  fwrite(STDOUT, ESC."[1;${cols}f"."╗");
```

```
77  fwrite(STDOUT, ESC."[$rows;1f"."╚");
78  fwrite(STDOUT, ESC."[$rows;${cols}f"."╝");
79
80  # You can see the range of box drawing characters available at
81  # http://en.wikipedia.org/wiki/Box-drawing_character
82  # They are just "text" like any other character, so you can easily copy
83  # and paste them into most editors.
84
85  # $p is an array [x,y] that holds the position of our cursor. We will
86  # initialise it to be the centre of the screen.
87
88  $p = ["x"=>intval($cols/2), "y"=>intval($rows/2)];
89
90  # Now for our first element of flow control. We need to keep the program
91  # running until the user provides input. The simplest way to do this is to
92  # use a never-ending loop using while(1). "1" always evaluates to true, so
93  # the while loop will never end. When we (or the user) are ready to end
94  # the program, we can use the "break" construct to step out of the loop
95  # and continue the remaining script after the end of the loop.
96
97  while (1) {
98
99          # Each time we go through the loop, we want to check if the user has
100         # pressed enter while we were in the last loop. Remember that STDIN is
101         # no longer blocking, so if there is no input the program continues
102         # immediately. If there is input we use break to leave the while loop.
103
104         if (fread(STDIN,1)) { break; };
105
106         # We will step the position of the cursor, stored in $p, by a random
107         # amount in both the x and y axis. This makes our snake crawl!
108
109         $p['x'] = $p['x'] + rand(-1,1);
110         $p['y'] = $p['y'] + rand(-1,1);
111
112         # We check that our snake won't step onto or over the frame, to keep
113         # it in its box!
114
115         if ($p['x'] > ($cols-1)) { $p['x'] = ($cols-1);};
116         if ($p['y'] > ($rows-1)) { $p['y'] = ($rows-1);};
117         if ($p['x'] < 2) { $p['x'] = 2;};
118         if ($p['y'] < 2) { $p['y'] = 2;};
119
120         # We want a pretty trail, so we need to pick random colours for the
121         # foreground and background colour of our snake, that change at
122         # each step. Colours in the terminal are set with yet more escape
```

```
123         # codes, from a limited palette, specified by integers.
124
125         $fg_color = rand(30,37);
126         $bg_color = rand(40,47);
127
128         # Once chosen, we set the colours by outputting the escape codes. This
129         # doesn't immediately print anything, it just sets the colour of
130         # whatever else follows.
131
132         fwrite(STDOUT, ESC."[${fg_color}m"); # \033[$32m sets green foreground
133         fwrite(STDOUT, ESC."[${bg_color}m"); # \033[$42m sets green background
134
135         # Finally we output a segment of snake (another box drawing character)
136         # at the new location. It will appear with the colours we just set, at
137         # the location stored in $p
138
139         fwrite(STDOUT, ESC."[${p['y']};${p['x']}f"."╬");
140
141         # Before we let the while loop start again, we need to do one more
142         # very important thing. We need to give your processor a rest.
143         # If we just continued our loop straight away, you would find your
144         # processor being hammered, just for our relatively simple program.
145         # Our snake would also consume the screen at super-speed!
146         # usleep pauses execution of the program, so others can use the
147         # processor or the processor can "rest". Every little bit helps the
148         # responsiveness of your machine, so even if you need your program
149         # to loop as fast as possible, consider even a small usleep if you can
150
151         usleep(1000);
152 };
153
154 # If this line of code has been reached, it means that we have 'break'd
155 # from the while loop.
156
157 # To be a good citizen of the terminal, we need to clean up the screen
158 # before we exit. Otherwise, the cursor will remain on which-ever line
159 # our snake left it, and the background/foreground colours will be
160 # the last ones chosen for our snake segment.
161
162 # The following escape code tells the terminal to use its default colours.
163
164 fwrite(STDOUT, ESC."[0m");
165
166 # We then clear the screen and put the cursor at the top-left, as we
167 # did earlier.
168
```

```
169 fwrite(STDOUT, CLEAR.HOME);
```

This program should demonstrate the three basics we listed earlier.

1. Getting keyboard input. You can read from `STDIN` in the same way you would any stream.
2. Outputting text (and graphics) to the screen. You can output to `STDOUT` (or use `echo`/`print`), control the appearance and cursor with escape characters, and use block drawing characters to make "semigraphics".
3. Program flow control. A `while(1)` loop is useful for keeping a program running, with `break` to continue flow outside the loop. It's important to use `usleep` or `sleep` to stop your process hogging a processor.

5.2 Advanced command line input

In the section above we used `fread()` to read keyboard input. This is suitable for simple programs, but if you are looking to create a more complex interface to allow users to issue commands then you may wish to look at the readline extension, which you can use to implement a shell-like editable command line program.

The following example script shows how to implement a simple bespoke command line type interface with the readline library.

```
1  <?
2
3  # Create arrays to hold our command history and list of valid commands.
4
5  $history = array();
6  $validCommands = array();
7
8  # Define some valid commands.
9
10 $validCommands[] = 'kill';
11 $validCommands[] = 'destroy';
12 $validCommands[] = 'obliterate';
13 $validCommands[] = 'history';
14 $validCommands[] = 'byebye';
15
16 # We want to enable tab-completion of commands, which allows the user to
17 # start typing a command and then press tab to have it completed, as
18 # happens in Bash shells and the like. We need to provide a function (via
19 # readline_completion_function) that will provide an array of possible
20 # functions names. This can be based on the $partial characters the user
21 # has typed or the point in the program we are at, or any other
22 # factors we want. In our case, we'll simply provide an array of ALL of
23 # the valid commands we have.
```

```
24
25  function tab_complete ($partial) {
26          global $validCommands;
27          return $validCommands;
28  };
29
30  readline_completion_function('tab_complete');
31
32  # We now enter our main program loop. Note that we don't include a usleep,
33  # as readline pauses our program execution while it waits for input from
34  # the user.
35
36  while (1) {
37
38          # We call readline with a string that forms the command prompt. In our
39          # case we'll put the date & time in there to show that we can change
40          # it each time its called. Whatever the user enters is returned. This
41          # one simple line implements most of the readline magic. At this stage
42          # the user can take advantage of tab-completion, history (use up/down
43          # cursor keys) and so on.
44
45          $line = readline(date('H:i:s')." Enter command > ");
46
47          # We need to manually add commands to the history. This is used for
48          # the command history that the user accesses with the up/down cursor
49          # keys. We could choose to ignore commands (mis-typed ones or
50          # intermediate input, for example) if we want.
51
52          readline_add_history($line);
53
54          # If we want to programmatically retrieve the history, we can use a
55          # function called readline_list_history(). However, this is only
56          # available if PHP has been compiled using libreadline. In most cases,
57          # modern distributions compile it using the compatible libedit library
58          # for licensing and other reasons. So we will keep a parallel copy of
59          # the history in an array for programatic access.
60
61          $history[] = $line;
62
63          # Now we decide what to do with the users input. In real life, we
64          # may want to trim(), strtolower() and otherwise filter the input.
65
66          switch ($line) {
67
68              case "kill":
69                  echo "You don't want to do that.\n";
```

```
70                  break;
71
72              case "destroy":
73                  echo "That really isn't a good idea.\n";
74                  break;
75
76              case "obliterate":
77                  echo "Well, if we really must.\n";
78                  break;
79
80                  case "history":
81
82                          # We will use the parallel copy of the command history that we
83                          # created earlier to display the command history.
84
85                  $counter = 0;
86
87                          foreach($history as $command)
88                          {
89                                  $counter++;
90                                  echo("$counter: $command\n");
91                                  };
92
93                  break;
94
95              case "byebye":
96
97                      # If it's time to leave, we want to break from both the switch
98                      # statement and the while loop, so we break with a level of 2.
99
100                 break 2;
101
102             default :
103
104                     # Always remember to give feedback in the case of user error.
105
106                     echo("Sorry, command ".$line." was not recognised.\n");
107         }
108
109 };
110
111 # If we reached here, the user typed byebye.
112
113 echo("Bye bye, come again soon!\n");
```

You may have noticed that I chose to use `byebye` as the command to quit the program. This was

not just a whimsical choice on my part, but to illustrate the need to think about discoverability. If you were presented with this program, without seeing the source code above, and asked to close it, it's likely you would try `quit`, `exit`, `end` and so on, before resorting to a good old `Ctrl-C`. In a GUI interface, you would have no such problems when faced with a button that said "Bye Bye!". With text based input, it is best to stick to common and memorable formats for commands, provide visual guidance and clues where possible, and aid in discoverability with good documentation, a `help` command, and user training.

Further Reading

Readline extension in the PHP Manual
http://www.php.net/manual/en/intro.readline.php

5.3 Using STDIN, STOUT & STDERR

The PHP CLI SAPI automatically opens the standard streams for you when it starts, so there is no need to issue commands like `fopen('php://stdin', 'r')`. You can treat them just like any other PHP stream, and start using them straight away. We've included some examples above, but here are a few more to illustrate the options available.

```
1  <?
2
3  # Get one line of input from STDIN
4
5  echo ('Please Type Something In : >');
6
7  $line1 = fgets(STDIN);
8
9  echo ('**** Line 1 : '.$line1." ****\n\n");
10
11 # Get one line of input, without the newline character
12
13 echo ('Please Type Something Else In : >');
14
15 $line2 = trim(fgets(STDIN));
16
17 echo ('**** Line 2 : '.$line2." ****\n\n");
18
19 # Write an array out to STDOUT in CSV format.
20 # First, create an array of arrays...
21
22 $records[] = array('User', 'Full Name', 'Gender');
23 $records[] = array('Rob', 'Robert Aley', 'M');
24 $records[] = array('Ada', 'Augusta Ada King, Countess of Lovelace', 'F');
```

```
25 $records[] = array('Grete', 'Grete Hermann', 'F');
26
27 echo ("The following is your Data in CSV format :\n\n");
28
29 # ...then convert each array to CSV on the fly as we write it out
30
31 foreach ($records as $record) {
32         fputcsv(STDOUT, $record);
33 };
34
35 echo ("\n\nEnd of your CSV data\n");
36
37 # Pause until the user enters something starting with a number
38
39 echo ('Please type one or more numbers : >');
40
41 while (! fscanf(STDIN, "%d\n", $your_number) ) {
42
43         echo ("No numbers found :>");
44
45 };
46
47 echo ("Your number was $your_number\n\n");
48
49 # Send the text of a web page to STDOUT
50
51 echo ("Press enter for some interwebs :\n\n");
52
53 fread(STDIN, 1); # fread blocks until enter pressed
54
55 fwrite(STDOUT, strip_tags( file_get_contents('http://www.cam.ac.uk') ) );
56
57 # Send an error message to STDERR. You can just fwrite(STERR,...
58 # if you want, or you can use the error_log function, which uses the
59 # defined error handling routine. By default for the CLI SAPI this is
60 # printing to STDERR.
61
62 error_log('System ran out of beer. ABORT. ABORT.',4);
```

The error logged on the last line will usually appear in your shell along with the other output, as that is where most shells put `STDERR` by default. If you want to check that it did come via `STDERR` rather than `STDOUT`, the following bash command will highlight any `STDERR` output (denoted by the `2>`) in red. It uses escape codes to colour the error (`31` sets the colour to red, `07` reverses it, then `0` clears it) :

```
php script.php 2> >(while read errors; do echo -e "\e[07;31m$errors\e[0m"
        >&2;  done)
```

In short, we can use the standard streams in any number of ways, often treating them as standard file pointers or streams.

5.4 Partial GUI elements - dialogs

Later in this chapter we will look at systems for producing complete graphical interfaces, but there is an intermediate stage for simpler programs that only need to occasionally interact with or notify the user, such as when popping up a warning or a graphical request for input. With PHP there are several methods of calling or creating visual display elements, as discussed below.

5.5 Dialogs invoked from the shell

There are various shell commands that will invoke graphical elements on screen. You can call shell commands in various ways from PHP, including using `shell_exec` or backticks as we do below. See the "Starting external processes from PHP" section in Chapter 7 for more details and possibilities. These commands include :

- **notify-send**

On Debian based systems, `notify-send` will pop up a notification "bubble" on a users screen. Exactly how it appears will depend on the distribution, but it will usually be a system standard notification similar to those displayed when you get new e-mail or a system event occurs. It uses lib-notify from the Gnome project, which may need to be installed (`sudo apt-get install libnotify-bin`). Call it from PHP as follows :

```
<?

shell_exec('notify-send -i error "Flange Error" '.
        '"An error occurred with the Flange Grommet. Flange 2.0 not found."');

# or

$command = 'notify-send -i info "Flange Completed" '.
        '"Flange has been Grommeted. See  manual for de-grommeting info"';

`$command`;
```

The `-i` flag allows you to specify an icon. You can provide either the full path to an image, or reference a standard icon (usually found in a location such as `/usr/share/icons/gnome/32x32/`) as we have done above.

Use `man notify-send` to find out the various options available to customise the notification.

Ubuntu Unity users note : The flag `-t` allows you to specify a timeout after which the notification disappears. However this flag is ignored in Ubuntu Unity, instead the duration of the notification depends on the length of its text. Setting `-t=0` in Unity will turn the notification "bubble" into an alert box which requires the user to explicitly close it instead.

- **zenity**

`zenity` allows you to display a number of common dialogs, from calendars to colour choosers. Based on GTK+, it allows various formatting and content options, and returns any input back for use. `Zenity` is installed in most Gnome based systems.

```
1  <?
2
3  # Execute zenity using backticks. Zenity returns the users input as text
4  # which we collect into a string variable. Lets use it to pop up a
5  # calendar, then tell the user what day of the week it is.
6
7  $day = `zenity --calendar --text="Choose a day" --date-format="%d %b %Y"`;
8
9  if ($day) {
10         echo('The date chosen is a '.date('l', strtotime($day)).".\n");
11 };
12
13 # Now we'll show a file selector and then show an "info" dialog to tell
14 # the user the size of the file selected.
15
16 $filename = chop (`zenity --file-selection`);
17
18 if (file_exists($filename)) {
19
20         $command = 'zenity --info --text "The size of the file chosen is  '.
21                 filesize($filename).' bytes."';
22
23         `$command`;
24
25 };
```

The full range of dialogs available include a colour selector, various notification dialogs, text inputs, a progress meter and more.

Further Reading

The full list of dialogs and options available in `zenity`
http://library.gnome.org/users/zenity/3.4/

- **kdialog**

`kdialog` is the equivalent of `zenity` above for KDE desktops.

Further Reading

More information and tutorial on kdialog
http://techbase.kde.org/Development/Tutorials/Shell_Scripting_with_KDE_Dialogs

5.6 Windows dialogs

A useful, Pecl package allows you to access the Win32 API from PHP. The API provides access to create common Win32 dialogs. We will look further at this library in Chapter 8 when we look at how it can be used to interact with the Windows Registry.

win32std

Set of standard Windows API functions.

Main website : http://pecl.php.net/package/win32std

Main documentation & Installation info : See README file within the download.

Alternative documentation :
http://wildphp.free.fr/wiki/doku.php?id=win32std:index

Compiled Version download : http://downloads.php.net/pierre/

5.7 Static HTML output

PHP is of course excellent for creating HTML output, that is its original reason d'etre after all. HTML can be useful for displaying information, charts, data etc. at the end of a CLI run, and you can use PHP's built in HTML tools to do this. Usually this occurs via a web server like Apache, so it may not be immediately obvious how to create and display HTML from CLI scripts. The end goal is to create one or more HTML files to display your output and save them to disk, to be displayed by a local web browser. You can probably imagine creating this HTML in strings and sequentially writing out to disk in a similar manner to creating other text/data files. However

PHP provides us with a neat trick called Output Buffering that allows us to pretend we are actually sending out HTML in the normal web server environment. This allows us to use facilities like intermingling blocks of HTML with code, echo'ing and print_r'ing data and so on, but instead of sending it out to a web server, PHP captures it for us and allows us to write it down to disk en-mass. This can be particularly useful if you have existing reporting or templating code from a web based project that you want to easily re-use from the command line.

The following example shows how to tell PHP to start collecting output, create some output ourselves, save it to disk and finally open it in a local web browser.

```
<?

echo("This text will go to STDOUT (your terminal) as normal\n");

# Start buffering our output rather than sending it STDOUT

ob_start();

# Echo and print write to the php://output stream. By default,
# php://output writes to STDOUT, which is why echo normally prints stuff
# back to the terminal. ob_start() captures the php://output stream,
# however we can still write to STDOUT directly (rather than via echo) if
# we want to tell the user what we're up to.

fwrite(STDOUT, "Starting HTML Generation...\n"); # displayed in terminal

# So we create our HTML as we would if we were "on the web"

echo('<html>');

print("<head><title>My HTML Page</title></head>\n");

?>
<body>
<h1> Intermingle Some HTML</h1>
<p>In the traditional PHP Way</p>
<p>
<a href="http://www.php.net">An Important Link</A>|||
<?

echo('</body></html>');

fwrite(STDOUT, "Finished HTML Generation\n"); # displayed in terminal

# ob_get_contents() creates a string with everything buffered so far.

$ourHtml = ob_get_contents();
```

```
38
39 # We can continue with buffering if we need to, or in this case we
40 # end "ob_end_clean"ly. If we ob_end_flush() instead, then the contents of
41 # the buffer would be pushed to php://output, which after ending the
42 # buffering is STDOUT, which we don't want in this case.
43
44 ob_end_clean();
45
46 echo ("This Text will go to STDOUT\n"); # now that we've ended buffering
47
48 # Finally, we want to save the buffered HTML to a file. Lets create
49 # a unique temporary file name ....
50
51 $filename = tempnam(sys_get_temp_dir(), 'my_report_').'.html';
52
53 # and write the HTML string to it.
54
55 file_put_contents($filename, $ourHtml);
56
57 # Finally, we want to open a web browser to view the HTML "report" we
58 # just created. Here we use the command "see" (available on most Debian
59 # based distros) to open the default viewer for the filetype (HTML).
60 # The command "open" achieves the same thing on other platforms. You
61 # can also specify a particular browser if you want,
62 # e.g. `firefox $filename`
63
64 `see $filename`;
```

Further Reading

Output Buffering Control in the PHP Manual
http://www.php.net/manual/en/book.outcontrol.php

5.8 Complete graphical interfaces (GUIs)

Many users expect full graphical user interfaces (GUIs) these days, with very good reason. GUIs, when done right, are good for discoverability and ease of interaction. They can provide intuitive human-friendly ways of presenting data and information and are ideal for event based programming.

Most GUI programs are written to use “Widget Toolkits” (also known as GUI Toolkits). These are toolkits or frameworks that provide code for the building blocks of GUIs; windows, forms, buttons, lists, mouse interactions, menus, events and more. There are a large number of widget toolkits, some are low-level and platform specific such as the Windows API for Windows or

Carbon for Mac OS X, some are higher level and cross platform like GTK+ and QT, and some are language specific, such as Swing for Java and LCL for Object Pascal. There is a fairly comprehensive list on Wikipedia.

Further Reading

Widget Toolkits on Wikipedia
http://en.wikipedia.org/wiki/List_of_widget_toolkits

Two approaches exist in PHP for using these toolkits to make graphical interfaces. The first is via direct bindings - your PHP script invokes the toolkit directly via a library or PHP extension. The second is via a helper application - your PHP script tells the helper application what it wants to display using an intermediate language like HTML or XUL, and the helper application deals with calling the necessary elements of the toolkit (or its own code) and displays the interface for you. The former is the traditional way of writing graphical applications, the latter is more akin to the web approach (where the browser is the helper application to which your scripts send HTML). Both approaches have pros and cons, below we will look at some of the current implementations of both methods.

5.9 Understanding GUI and event-based programming

It's worth spending a few moments at this point talking about one of the main differences between GUI apps and, for instance, command line programming. Most GUI programming is event-based programming, which can sometimes take a while to get your head around if you've not programmed using this style before.

With a GUI program, you typically start programming "sequentially" as normal by opening a window or form and populating it with buttons, text, images, data or whatever.

At this stage your program then goes into a waiting loop, usually waiting for the user (or sometimes the system) to do something. That may be waiting for them to click a button, enter some text, select an option or wait for a new item of data to arrive. Each of these things is called an event, and you typically can't predict in which order they will happen. So your code will bind these events to functions or methods, and call these as the relevant events occur.

This means that your code is often executed out of sequence, so keeping abreast of state becomes much more nuanced than in command line scripts and even web scripts, which run sequentially from top to bottom. The most common way of dealing with event based programs is to use Object-Oriented (OO) programming techniques, and indeed the rise of GUI software drove the rise in OO programming (or followed on from it, depending on who you talk to). Its still perfectly possible to use traditional "imperative" style programming of course, particularly if you are careful around issues such as state and scope, but if you intend to do a lot of GUI programming and aren't familiar with OO, you may be doing yourself a favour if you pick up an OO primer and have a quick flick through.

Further Reading

Object-oriented Programming on Wikipedia
http://en.wikipedia.org/wiki/Object-oriented_programming

"Introduction to PHP OOP" by Lorna Mitchell
http://www.lornajane.net/posts/2012/introduction-to-php-oop

"Object-Oriented PHP for Beginners" by Jason Lengstorf
http://net.tutsplus.com/tutorials/php/object-oriented-php-for-beginners/

Web programming is of course event-based in a sense, you typically present an HTML page to the user and then wait for them to click a button or link, and handle that "event" with a different HTML page (or another run through of the same script with different parameters). Each of your HTML pages (or the PHP scripts that generate them) is in some-way equivalent to the individual functions or methods called in our GUI scripts, and your web server (Apache etc.) is equivalent to the waiting loop in our GUI program that holds things together and reacts to our events. The difference is that in web based programming, with PHP in particular, we operate on a shared-nothing (or almost nothing) basis - each script execution or HTML page fetch is deliberately separate from another unless we otherwise act to share some state. In GUI programming, we are typically operating from within the same script set under one process, and thus it is a share-(almost)-everything architecture. Once you've got your head around this concept, you will find it much easier to plan and map out your software.

Now lets look at some of the GUI toolkits available.

5.10 PHP-GTK

PHP-GTK is an official PHP extension which provides direct bindings to the GTK+ widget toolkit from PHP. The first version was released in 2001 and the project has enjoyed some success. However activity on the project has waned in recent years, the most recent posting on the official website (gtk.php.net) at the time of writing was three years ago in 2010. That said, the code is mature and reasonably complete/stable and it is relatively easy to begin with.

Pros :

- Provides direct binding to the GTK+ toolkit
- Fast and complete control over the toolkit elements
- Officially supported PHP project
- Availability of tools such as Glade for help with interface design and layout Cross platform
- PHP compilers such as PriadoBlender, Roadsend PHP and bcompiler offer some support for php-gtk
- Various additional GTK components are available through the PEAR repository

Cons :

- Lack of activity on the project raises questions on its future direction (if any)

- GTK+ doesn't look native on all platforms, the look and feel of GTK+ apps is most at home on Linux type OSes.
- GTK+ itself is less popular than it once was, although as a project it is still very active and progressing

php-gtk

Official PHP GUI toolkit

Main website : http://gtk.php.net/

Main documentation & Installation info : http://gtk.php.net/docs.php

Github Repository : https://github.com/php/php-gtk-src

Mailing list archives : http://marc.info/?l=php-gtk-general

Active Community Sites

Worldwide Community
http://php-gtk.eu/

Brazilian Portal
http://www.php-gtk.com.br/

Glade

GTK+ visual interface design tool, can be used with php-gtk.

Main website : http://glade.gnome.org/

Main documentation & Installation info : https://wiki.gnome.org/Glade/

Tutorial :

Using Glade with php-gtk
http://gtk.php.net/manual/en/tutorials.helloglade.php

5.11 wxPHP

Started in 2005, wxPHP has recently gained steam and is currently the most active of the widget library bindings listed here. wxPHP provides bindings for wxWidgets, a cross-platform library of native widgets that supports Linux, Windows and Mac OS X. The latest builds are available directly from the wxPHP Github repository, followed later by releases on the main site. wxWidgets is primarily a C++ based library, but has bindings for a wide range of other languages, and is well supported, comprehensive and extensive, and under active development. Some observers have criticized many wxWidget applications as having a bland, corporate look (think “lots of grey”), but it is perfectly possible to create quite attractive interfaces with it too.

A quick word about documentation; The documentation (class references etc.) provided by wxPHP are fairly perfunctory, they are automatically generated each time a new version of the bindings are built and basically just list which widgets and properties are supported. A better source of useful information is the official wxWidgets documentation, as well as the wxWidgets wiki. Between them they provide full details about what each widget does along with guides and tutorials. These are not PHP specific, but generally useful anyway regardless of the language you are using. You should probably consider the wxPHP documentation as a simple reference as to whether a particular widget or method has been implemented in PHP.

Further Reading

Official wxWidgets documentation
http://wxwidgets.org/docs/

Official wxWidgets wiki
http://wiki.wxwidgets.org/

Once installed, creating applications is fairly straight forward. Windows (often called forms or

frames) can be built up programmatically by adding buttons, boxes, inputs and such directly from your PHP script. For more complex layouts this can be a tedious way to design the interface. Luckily, WxFormBuilder (available from the software repositories on most Debian based Linux distros) is a graphical layout tool for wxWidgets that lets you design an interface by dragging and dropping elements onto your window, and now supports outputting the necessary PHP code to use your design from within your program. You can also use it to manage events, specifying which functions to execute when, for instance, a button is clicked. With the addition of wxFormBuilder, wxPHP is branding itself as a leading PHP RAD (Rapid Application Development) environment, with good reason.

wxPHP

RAD type toolkit for wxWidgets

Main website : http://www.wxphp.org/

Main documentation & Installation info : http://www.wxphp.org/docs, also see https://github.com/wxphp/wxphp#table-of-contents

Interface design tool : http://sourceforge.net/projects/wxformbuilder/

Documentation for design tool : http://sourceforge.net/apps/mediawiki/wxformbuilder/index.php?title=HomePage

5.12 Local web server & browser

One general technique that has been employed with varying degrees of success when creating local PHP apps is to deploy the whole web stack locally on each PC. This means installing a copy of PHP and Apache (or Nginx, or similar) and accessing the PHP webpages over a local port (e.g. `http://127.0.0.1:80`) in a web browser.

Pros:

- Re-use existing web code and web skills, mostly as-is
- Cross platform

Cons:

- Heavy resource overhead for the web-server
- Non-native experience

- Must be secured to stop external access
- Web servers can be temperamental if not properly tuned to the system it's on
- Large maintenance overhead
- Larger vector for security exploits
- Performance hit of using the verbose HTTP protocol for UI->backend interaction
- Cross-browser compatibility issues occur in the same way as the web, unless a particular browser is deployed or required.

In general this is definitely one to avoid if at all possible, certainly for public distribution of software. It may be worth considering where you have existing web applications you need to deploy in short-order on local machines that you have responsibility for and control over.

If you are going to go down this route, it is worth considering deploying an existing stack "solution" which have already ironed out some of the issues involved in local deployment of web stacks. A handy list of "LAMP" (LinuxApache/Mysql/(PHP|Perl|Python)) type stacks can be found on Wikipedia, and likewise a comparison of WAMP (Windows/AMP) stacks can be found there too. Remember that while the solution is generally cross-platform, the set-up and fine-tuning of the web stack can vary a lot across different OSes, depending on exactly what you are trying to achieve and the complexity of your application.

Further Reading

List of LAMP type stacks on Wikipedia
http://en.wikipedia.org/wiki/List_of_Apache%E2%80%93MySQL%E2%80%93PHP_packages

Comparison of WAMP stacks on Wikipedia
http://en.wikipedia.org/wiki/Comparison_of_WAMPs

5.13 PHP's Built-in testing server

Since v5.4, PHP comes equipped with a built-in web server, primarily designed for testing purposes. It is intended to be used locally by single users rather than on the public web, so lacks the performance, security and resource management controls a fully-fledged web server like Apache has, and is not extensible by means of modules and the like. Nevertheless, a lightweight server like this makes more sense for local apps than the Apache based solution described above, where these extra facilities aren't generally required. That said, it still has a longer than desirable list in the "Cons" section.

Pros:

- Re-use existing web code and web skills, mostly as-is
- Cross platform
- No extra server to deploy beyond PHP itself

Cons:

- Non-native experience
- Must be secured to stop external access
- The PHP developers have explicitly stated that it was designed specifically for the testing/development environment.
- Performance hit of using the verbose HTTP protocol for UI->backend interaction
- Cross-browser compatibility issues occur in the same way as the web, unless a particular browser is deployed or required.
- Not all features of servers like Apache are included, so some re-writing of the web app may be required if these are relied upon.

PHP Built In Web server

PHP's own built-in test web server

Main website : http://php.net/manual/en/features.commandline.webserver.php

Installation info : Installed as part of standard PHP install

Main documentation :
http://php.net/manual/en/features.commandline.webserver.php

Tutorial

"Taking Advantage of PHP's Built-in Server" by Vito Tardia
http://www.sitepoint.com/taking-advantage-of-phps-built-in-server/

5.14 Web sockets & browser

A variation on the web server/browser scenarios above is using Web Sockets, part of the evolving HTML5 specification designed to allow 2-way communication and "push" type data flows in the browser. LAMP type stacks and the PHP Built in Web Server don't, by default, handle Web Socket connections. The specification is very new and browser support varies, and you'll need to deploy a dedicated web socket server or a PHP web socket server library (usually based on spawning PHP CLI processes to handle communications). Beyond that, the pros and cons are roughly the same as those discussed for traditional web server communications. Due to the "push" ability of sockets, communications may be quicker and less frequent.

Ratchet

Popular PHP WebSocket library. Uses ReactPHP.

Main website : http://socketo.me/

Installation info : http://socketo.me/docs/install

Main documentation : http://socketo.me/docs

whippy.php

A pure PHP WebSocket server

Main website : https://github.com/rthrfrd/whippy.php

Web Socket Service

A PHP package to handle Web socket accesses using child processes

Main website : http://www.phpclasses.org/package/7259-PHP-Handle-Web-socket-accesses-using-child-processes.html

Web Socks

A Web Socket server written in PHP. It implements the version 76 handshake.

Main website : http://code.google.com/p/web-socks/

5.15 SiteFusion

SiteFusion is a GUI solution based on the Mozilla XULrunner. XULrunner is the Mozilla technical platform on which applications like Firefox and Thunderbird are built, and it is a general purpose XUL/HTML/Javascript runtime. Rather than providing a totally local solution, SiteFusion focuses on the server-client scenario. A local client helper "XULrunner" application in installed on the client PCs, but the PHP application code lives on the server running under custom SiteFusion server software. It is possible to install the client and server on the same machine, however many of the "cons" listed in "Local web server & browser" apply to this configuration. SiteFusion exposes XUL as the primary method of building the interface, which although similar to HTML still has a learning curve. HTML can of course be used as XUL has the browser element, however SiteFusion doesn't provide any abstraction of the HTML/DOM to assist working with the contents of the browser element. Some HTML tags can be used directly within XUL code (outside of browser element), however Mozilla advise against this for various reasons. SiteFusion's main market target and the area in which it is definitely worth considering is with client-server apps, where central control over the application is desired and it is deployed on a local network within one organisation.

Pros:

- Native experience
- Cross-platform
- Designed for client-server applications
- Stable, mature project with ongoing development

Cons:

- Responsiveness can be subject to network delays
- Not ideal for local-only installs
- XUL is less common than HTML, so less documentation, support, toolkits etc., although HTML components can be run in the browser element and some Javascript tools/components will operate in/on the XUL interface.

Comprehensive tutorials and documentation can be found on the SiteFusion.org website.

SiteFusion

XUL based client-server PHP GUI system

Main website : http://sitefusion.org/

Installation info : http://sitefusion.org/10

Main documentation : http://sitefusion.org/tutorials

5.16 Winbinder

Winbinder provides PHP direct bindings for the Windows API. Thus, it only works on MS Windows based platforms (or under Windows compatibility layers like Wine on Linux). The current status of the project is unknown, the website is still functioning and the files are available for download. However the community forum has been closed down, there are no news posts and the system only appears to support older Win32 style interfaces.

Pros :

- Provides direct bindings to the Windows API
- PECL package available

Cons :

- Questionable project status/activity/future
- Not cross-platform
- Out of date interfaces

WinBinder

Windows API based GUI toolkit

Main website : http://winbinder.org/

Main documentation & Installation info : http://winbinder.org/manual.php

5.17 Adobe AIR

An alternative to a web browser for LAMP stack or PHP test server deployments, Adobe AIR lets you create local HTML/Javascript applications that run using the Adobe Integrated Runtime (or AIR). While Adobe envision you creating your whole application in HTML and Javascript, and allow privileged access to the file system, off-line storage and more, you can easily hook up your "web" application to PHP scripts in the same way a browser would. In essence, the AIR app is your browser. This provides a better visual environment than a browser, but still has most of the same draw-backs listed above. Adobe have also recently withdrawn support for Linux, although they do support mobile and e-reader platforms. In the latter case you may not be able to run the PHP portion of your app locally.

Adobe AIR

Runtime supporting HTML & Javascript apps that can replace a browser

Main website : http://www.adobe.com/products/air.html

Main documentation & Installation info :
http://www.adobe.com/devnet/air/documentation.html

Commercial video course :
http://my.safaribooksonline.com/video/programming/air/01220110006si

5.18 Titanium

Titanium is similar in concept to Adobe Air, and essentially the pro's, con's and usage of both is the same. However this could easily have been so different, as the early versions of Titanium had integrated support for PHP as a first class language. That means that you could use PHP *within* your HTML, in the same way you used Javascript. No server. This meant that you could run PHP code on your page, directly access the page DOM and do everything that PHP does. However as of v2 Appcelerator withdrew that functionality to focus on Javascript only. Although older versions of the opensource core product are still available with the PHP functionality, this entry is more of a historical note on what could have been.

Appcelerator Titanium

Runtime supporting HTML & Javascript apps that can replace a browser

Main website : http://www.appcelerator.com/

Main documentation & Installation info : http://docs.appcelerator.com/

Now removed PHP functionality : http://matthewturland.com/slides/titanium/

Appcelerator Licensing Controversy : http://arstechnica.com/information-technology/2012/09/when-free-isnt-developer-accuses-tool-vendor-of-extorting-customer/

5.19 PHP-Qt

PHP-Qt is an extension to provide PHP bindings for the popular cross-platform QT widget toolkit. This project appears however to have last released code in 2007 and been abandoned in 2009. It is mentioned here only for completeness, the code is still available for download and may be of interest for someone attempting to develop a similar solution.

PHP-Qt

Defunct project to provide bindings for the QT toolkit.

Main website : http://developer.berlios.de/projects/php-qt/

5.20 PHP/TK

PHP is a project last updated in 2004 proving bindings for the TCL/TK X-windows interface toolkit. As with PHP-Qt above, it is mentioned here only for completeness.

wxPHP

Bindings for the TCL/TK toolkit.

Main website : http://php-tk.sourceforge.net/

Using PHP/TK with Delphi/Lazarus :
http://web.fastermac.net/~MacPgmr/PhpTk/PhpTkStatus.html

6 System software

In the previous chapter we looked at user facing software, that is, software which the human user interacts with directly. In this chapter we are going to look at what I will term "system software", software that does things (generally) without a human driving it.

To write system software we typically need to create a "daemon", which is a continuously running process. Once we've created the Daemon, we need to decide its primary function. Some daemons work continuously (measuring, monitoring, communicating and so on), others wait and work in response to events (network connections, system changes, user actions). We will look at how to create a daemon in PHP and set it working continuously. We will then look at how we can take that basic daemon and instead of working continuously, make it to just spring into life to react to certain events that occur in our system.

In the user facing software we looked at in the previous chapter, the software is typically used by one person or process at a time. System software, on the other hand, often serves many different "clients" at the same time. A "client" in this context may be a network client (e.g. a web browser for a web server), a user process (a GUI client accessing your API server), a file (a log file for a logging server) or similar. While doing so it needs to remain "responsive", that is one client shouldn't have to wait while another client's request finishes. Think how backed up the web would get if Apache could only serve one web page at at time! To manage concurrent tasks in PHP and maintain a responsive daemon, we can use task dispatch and management systems, which we will look at in the final part of this chapter.

6.1 Daemons in PHP

A daemon is a program that runs, usually continuously, as a background process. It often doesn't interact with users directly, but performs background tasks or responds to system events or calls from other software, network requests or other machine-to-machine events. Examples of programs that run as daemons include Cron (which waits in the background and executes tasks based on the current time), and Apache (which sits and waits for calls from remote machines for web resources). Daemons usually :

- run permanently (or for a long time, or until a predetermined event),
- start up at boot time
- perform useful tasks
- are owned by root or a (non-human) system user.

However these criteria don't universally apply to all daemons, in fact the only concrete thing daemons have in common is that they don't have a controlling terminal (tty), and thus are deemed to be running "in the background". Without a tty the software cannot get user input from the keyboard or display output back to the user, via the terminal (though there are other

ways to directly and indirectly interact with a user). Although consuming minimal resources was traditionally a key trait of background processes, that is not now commonly the case. Software "servers" such as Database Management Systems and Web Servers run as background daemons but often consume large (or even all) of the system resources and often have machines dedicated just to running them. So it may be best to think of daemons as any permanently running software which doesn't usually directly interact with the user.

6.2 Creating a daemon

To create a daemon in PHP we use the PHP process control extension, or PCNTL, which is only available on Unix/Linux type systems. Most pre-compiled versions of PHP include the PCNTL extension, but if not you will need to re-compile using the `--enable-pcntl` option, or install the extension using your package manager. See Appendix A for details.

The outline process for creating a daemon is as follows :

- We run a process (PHP script). We will call this the parent process.
- From the parent, we fork (copy) a child process.
- The parent process then exits. The child process is now parent-less.
- `init` adopts the parent-less child process. `init` is the original process started by the kernel when it boots and is the ancestor of all processes.
- We then dissociate (detach the child) from the terminal we started the parent in. This is so that
  - any of our output doesn't appear in the terminal
  - killing the terminal won't kill our child process
  - we are truly running in the background
- To dissociate (detach), we need to :
  - move the child process into it's own POSIX process session
  - fork it once again (into the grandchild process) and kill the child process
  - close any file descriptors such as STDIN that may tie it to the terminal

Once all this is done, you will be returned to the command prompt in the terminal (assuming thats where you started your original parent process from) and your (grandchild) daemon will be running on its own. You will only be able to interact indirectly with your daemon from now on. Finally, assuming that the daemon is to run continuously (or for a set period of time) rather than just completing a task and exiting, the process will need to enter a loop where it will continuously cycle and, for instance, await events or perform continuous tasks. Its probably also wise to give it the ability to exit upon demand.

This may sound quite a long and involved process, however it is fairly straight forward in PHP. The following script follows this process and outlines the basics.

```
<?

# We start by forking this script, which creates the child process.

$pid = pcntl_fork();

# The child process will start running from here and will be a copy of the
# parent process, which includes all opened resources, variable values and
# so on, with the sole exception of the $pid variable above which is not
# set for the child. The parent process will keep running from here as
# well, with the process ID of the child process assigned to $pid

# If for some reason a child process could not be forked, for example
# the system is low on memory, $pid will be set to -1

if ($pid == -1) { exit("Could not fork the child process"); };

# If $pid is set to a process ID, then we must be the parent, and
# should exit.

if ($pid) { exit(0); };

# As the parent has exited above, the following code is now executed
# solely by the child process.

# We detach from the TTY (terminal) by becoming the "session leader"
# (instead of the TTY being leader. This starts a new POSIX session) ...

if ( posix_setsid()  === -1 ) {
        exit("Could not become the session leader");
};

# ... and then by forking again, to create a grandchild process

$pid = pcntl_fork();

if ($pid == -1) { exit("Could not fork child process into grandchild"); };

# Exit the child process, leaving only the grandchild beyond this point.

if ($pid) { exit(0); };

# Now to finally dissociate from the TTY and run in the background, we
# need to close our input and output streams to it. These are
# automatically defined and opened in the CLI SAPI, so will always need to
# be closed.
```

```
47
48  if (!fclose(STDIN)) { exit('Could not close STDIN'); };
49  if (!fclose(STDERR)) { exit('Could not close STDERR'); };
50  if (!fclose(STDOUT)) { exit('Could not close STDOUT'); };
51
52  # STDOUT is now closed, if you echo or print anything or interpolate HTML
53  # after this point, your script would crash. Likewise, any error messages
54  # would have nowhere to go, and any inadvertent attempts to read input
55  # would end badly. So instead we will recreate these three streams, but
56  # with sensible destinations. When we fclose the standard streams, and
57  # because you cannot redefine a constant, PHP simply assigns the standard
58  # streams to the next three new file descriptors opened (whatever they
59  # are called and whereever they point to). Thus the following also acts
60  # to protect any other streams you may open from "pollution" caused by
61  # accidental writes to those standard streams
62
63  $STDIN = fopen('/dev/null', 'r');
64  $STDOUT = fopen('/dev/null', 'w');
65  $STDERR = fopen('/var/log/our_error.log', 'wb');
66
67  # We are now a free-floating daemon, fully detached from our TTY!
68
69  # Now we go into our main loop, and do something useful.
70
71  # We set a variable which will allow us to escape from the loop if we
72  # want to shut down our daemon.
73
74  $stayInLoop = true;
75
76  while ($stayInLoop) {
77
78      # do useful stuff here, like listening for connections,
79      # monitoring things, or whatever else your daemon does.
80
81      # When it's time to exit, set $stayInLoop = false at some point in the
82      # loop and this loop will be the last. If you need to exit before the
83      # loop finishes you will need to call any clean-up code and exit()
84      # directly. Here, we will end if it's a Tuesday, looping only once.
85
86      if (date('l') == 'Tuesday') { $stayInLoop = false; };
87
88      # For this example, we're going to execute a cli program called
89      # notify-send to periodically pop up a notification to say hello.
90
91      `notify-send 'Hello, The daemon is alive!'`;
92
```

```
93      # The following line adds a "sleep" to each cycle of the loop. If we
94      # didn't do this, our daemon would (try to) consume 100% of the CPU
95      # time as it constantly cycles and evaluates the conditions for
96      # looping. You can adjust the time it sleeps for depending on the
97      # "responsiveness" required of your daemon. Giving it a break for even
98      # a few 100 or 1000 milliseconds (using usleep) helps to maintain
99      # overall system responsiveness.
100
101     sleep(15); # Loop every 15 seconds
102 };
103
104 # We've exited our loop, so do any clean-up required here
105
106 `notify-send 'The Daemon is now finished. Bye Bye.!'`;
107
108 # And then exit
109
110 exit(0);
```

The code above, while it doesn't do anything particularly useful, is an outline of the basic steps involved in creating a PHP daemon and shows how simple it can be. You can combine it with the other techniques outlined in this book to create some useful systems and networking tools that run permanently in the background.

There are a number of important things to bear in mind when writing daemons in PHP, to ensure they keep running as expected.

Firstly, when you fork a process as we have done to create a child process, the child is an almost exact copy of the parent, including the state of variables and resources. This means that if you do anything in the parent script before forking, it will be replicated in the state of the child. A common gotcha, particularly if forking multiple daemons from the same script, is setting up database connections in the parent before forking. The children will then share the same resource identifier (i.e. exactly the same connection) as the parent and the other children, and as soon as one of them closes the connection (e.g. by exiting) the database will become unavailable to all of the others. On the other hand, this doesn't apply to, say, variables. If you set the variable `$foo = 6;` in your parent, when each child starts `$foo == 6` will be true, but if you alter it in one child, it will not alter in the others, as they are separate variables (just set to the same value at the start). Technically the resource identifiers in the database example are a also separate identifiers in each child, they just happen to point to exactly the same external database connection.

Secondly, resource usage and garbage collection are important to handle correctly for very long running scripts, such as daemons. PHP has traditionally had a "designed to die" model, for instance it wasn't until recently that garbage collection was turned on by default. In web scripts you usually don't have to worry too much about resource usage, short running scripts typically don't use much and PHP will clean up after you when your script exits. In long running scripts, PHPs garbage collection can only do so much (and may do it at inconvenient times), so you will need to manage your resources carefully. This is particularly pertinent if you repeatedly create new variables, objects, resource handlers and so on, or rely on high

responsiveness from your script. See Chapter 9 for a detailed look at performance, garbage collection, measuring/minimising resource usage and generally keeping scripts up and running.

Finally some platforms have standard practises for daemons, which help everything to get along smoothly. You may wish to consider those applicable to your platform. These can include things such as setting your working directory to `/` (e.g. `chdir /`), and using `init` scripts to start and stop your daemon. These vary from platform to platform and are not PHP specific, so are beyond the scope of this book.

Further Reading

"Thorough look at PHP's pcntl_fork()" by Frans-Jan van Steenbeek
https://sites.google.com/a/van-steenbeek.net/archive/php_pcntl_fork?q=php_pcntl_fork

6.3 Network daemons using libevent

In the previous section we used a fairly basic `while(1) { }` event loop to keep our daemon running and responding to events or doing useful work. The advantage of that approach is that is very simple for basic needs and is implemented natively in PHP with no external dependencies. The downside however is that it leaves you to implement all of the details, and the complexity increases as your project grows.

One popular alternative to consider is libevent, a library which provides a framework for dealing with event based programming. This library can be accessed in PHP though two different PECL modules :

- pecl-libevent : This is an older module, and is fairly simple and straight forward to use. However it doesn't support libevent2 (only 1.x versions, 1.4.0 or above.), and thus has fewer features
- pecl-event : This is a complete rewrite of previous Pecl module of the same name abandoned in 2004. It is currently actively developed, and supports libevent2. This has more options including specific classes tailored for HTTP, DNS, SSL and other types of event connections. For these reasons this is the module we will use in the examples below.

Libevent describes itself as "… a library that provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a time-out has been reached". In layman's terms, this means that libevent will execute a function of your choice either at pre-determined time intervals, or when a particular "file descriptor" event occurs. "File descriptors" in PHP cover not just events occurring on actual files, but anything that can be treated as a file or stream. This includes network sockets and system streams like `STDIN`. In fact, due to a problem with epoll (a Linux kernel event notification system, used by libevent) compatibility, libevent typically cannot be used for file events (detecting file accesses and modifications etc.) on many platforms. Because of this, we will additionally look at inotify in the next section for use with file events. Indeed, you should only really consider libevent for network/stream type events, which is where it really shines.

Libevent also offers event buffering, so in demanding environments it will queue events for you to process at your leisure, and you won't risk missing something because your script was off doing something else. This is particularly important in a non-multitasking environment like PHP. Note that libevent just deals with responding to events, not creating a daemon in the first place, so you will still need to use code from the previous section to turn your script into a daemon before using Libevent to do the work.

The following example shows how we can use the pecl-event module to call libevent to act as a very simple http server. For brevity the following example runs as a standard CLI process, you can daemonise it using the techniques discussed in the previous section if you need to.

```
<?

# Before we start doing any actual work, we first define a series of
# functions that will provide responses to our "events"; in this case
# http requests. The real magic of libevent occurs at the end of this
# script.

function techInfo($req) {

        # This is our first response function. The object $req passed in is
        # the current http request/connection.

    # First we will set a "Content-Type" output header to tell the web
    # browser to expect plain text rather than HTML. If we were outputting
    # HTML we would still need to send a header, but the event library
    # does this for us if we don't add one ourselves.

          $req->addHeader ( 'Content-Type' , "text/plain; charset=ISO-8859-1" ,
                                          EventHttpRequest::OUTPUT_HEADER );

    # Next we'll gather some information about the request, and format it
    # into a string to send back to the web browser.

    $replyText .= 'Command : ' . $req->getCommand() . "\n";
    $replyText .= 'Host : ' . $req->getHost() . "\n";
    $replyText .= 'Input Headers : ' .
                                          var_export($req->getInputHeaders(),true) . "\n";
    $replyText .= 'Output Headers : ' .
                                          var_export($req->getOutputHeaders(),true) . "\n";
    $replyText .= 'URI : ' . $req->getUri() . "\n";

    # To send a reply back, we create an "EventBuffer" containing the
    # reply contents, in our case the $replyText above

    $reply = new EventBuffer;

```

```
37      $reply->add($replyText);
38
39      # Finally we send our EventBuffer to the browser, with an HTTP status
40      # 200-OK to confirm everything happened correctly.
41
42      $req->sendReply(200, "OK", $reply);
43
44  };
45
46  function closeServer($req) {
47
48          # Our next function allows the visitor to shut down the server by
49          # simply visiting a URL. We'll send them a message before we shut down
50          # to let them know.
51
52      $reply = new EventBuffer;
53
54      $reply->add("Ok 1337 haxor, you've killed the server...");
55
56      $req->sendReply(200, 'OK', $reply);
57
58      # We then call the exit method of the event base, to exit the event
59      # loop, which we'll look at towards the end of the program.
60
61      global $base;
62      $base->exit();
63  };
64
65  function notFound($req) {
66
67          # This function handles the case where we can't find a resource
68
69      $req->sendError(404, 'Does not appear to be here. Sorry.');
70
71  };
72
73  function cat($req) {
74
75          # This function is one of the most important on the internet. It
76          # returns a picture of a cat. You will need a cat picture named
77          # cat.jpg in the same directory for this to work, but that shouldn't
78          # be too difficult to arrange...
79
80          # As we're returning a binary image file, we need to set the
81          # appropriate mime-type output header.
82
```

```
83      $req->addHeader ( 'Content-Type' , "image/jpeg" ,
84                                              EventHttpRequest::OUTPUT_HEADER );
85
86      # Get the contents of the image file ....
87
88      $cat = file_get_contents('cat.jpg');
89
90      # and add them to a new EventBuffer ...
91
92      $reply = new EventBuffer;
93
94      $reply->add($cat);
95
96      # finally delivery the cat to an appreciative audience ....
97
98      $req->sendReply(200, "OK", $reply);
99
100 };
101
102 function genericHandler($req) {
103
104         # This function will handle any requests that the previous functions
105         # haven't. We'll use the opportunity to serve up an HTML page with a
106         # title and a picture of a cat. The <img> tag will cause the browser
107         # to make a second call, which will be routed to the cat() function
108         # above to deliver the image file.
109
110     $replyText = '<html><head><title>'.$req->getUri().'</title>';
111     $replyText .= '<h1>Picture of cat</h1><br>';
112     $replyText .= '<img src="/images/cat.jpg">';
113
114     $reply = new EventBuffer;
115
116     $reply->add($replyText);
117
118     $req->sendReply(200, "OK", $reply);
119 };
120
121 # Now we've defined all of our functions for delivering content, we need
122 # to actually set up our server.
123
124 # First we create an "EventBase", which is libevent's vehicle for holding
125 # and polling a set of events.
126
127 $base = new EventBase();
128
```

```
129 # Then we add an EventHttp object to the base, which is the Event
130 # extension's helper for HTTP connections/events.
131
132 $http = new EventHttp($base);
133
134 # We'll choose to respond to just GET and POST HTTP requests
135
136 $http->setAllowedMethods(
137                 EventHttpRequest::CMD_GET | EventHttpRequest::CMD_POST);
138
139 # Next we'll tie the functions we created above to specific URIs using
140 # function callbacks.
141
142 $http->setCallback("/info", "techInfo");
143 $http->setCallback("/close", "closeServer");
144 $http->setCallback("/notfound", "notFound");
145 $http->setCallback("/images/cat.jpg", "cat");
146
147 # Finally we'll add a default function callback to handle all other URIs.
148 # You could, in fact, just specify this default handler and not those
149 # above, and then handle URIs as you wish from inside this function using
150 # it as a router function.
151
152 $http->setDefaultCallback("genericHandler");
153
154 # We'll bind our script to a address and port to enable it to listen for
155 # connections
156
157 $http->bind("0.0.0.0", 12345);
158
159 # Then we start our event loop using the loop() function of our base. Our
160 # script will remain in this loop indefinitely, servicing http requests
161 # with the functions above, until we exit it by killing the script or,
162 # more ideally, calling $base->exit() as we do in the closeServer()
163 # function above.
164
165 $base->loop();
```

To try out this script, place one of your many cat pictures in the same directory, calling it cat.jpg, then run the script. Open a browser and navigate to http://localhost:12345 whereupon you should be greeted by a cat. To get some information about your connection, go to http://localhost:12345/info and to shut down the server go to http://localhost:12345/close.

This is a fairly simple example, but it demonstrates the basic structure of an event driven program. The Event library (via libevent) can respond to a wide variety of event types and helpers for a number of network style responders are included.

A listing of other daemon/event tools, frameworks and information is included below.

PHP Simple Daemon

A stable, production-ready PHP Daemon library

Main website : https://github.com/shaneharter/PHP-Daemon

Main documentation & Installation info :
https://github.com/shaneharter/PHP-Daemon#php-simple-daemon

Kellner framework

Asynchronous, scalable I/O framework for PHP based on libevent

Main website : https://github.com/fhoenig/Kellner

Main documentation & installation info :
https://github.com/fhoenig/Kellner#readme

Nanoserv

Network oriented server daemon framework for PHP

Main website : http://nanoserv.si.kz/

Main documentation & installation info : http://nanoserv.si.kz/doc/

phpDaemon

Asynchronous server-side PHP framework for Web and network applications using libevent2

Main website : http://daemon.io/

Main documentation & installation info : none, see example code at http://daemon.io/#outofbox

Further Reading

Socket programming tutorials, useful for adding socket-based network capabilities to your daemons

http://christophh.net/2012/07/24/php-socket-programming/
http://www.binarytides.com/php-socket-programming-tutorial-for-beginners/
http://www.adayinthelifeof.nl/2010/07/30/creating-a-traceroute-program-in-php/

Reddit thread linking to many different network utilities written in PHP http://www.reddit.com/r/PHP/comments/s9t3k/im_trying_to_find_really_unique_mindboggling_php/

6.4 File monitoring daemons using inotify

Daemons that respond to events in the filesystem are often useful. Example uses include :

- File conversion daemons : the daemon monitors a particular folder, when a file of a specified type is added to that folder the daemon converts it into another format automatically
- File sync daemons : the daemon watches one or more folders, and automatically syncs any changes with an external storage service
- Change notification daemons : want to know when someone else updates your important files? Get a daemon to watch them and report back to you!
- File search services : a daemon watches the file system and indexes files as they are created or modified, for rapid searching later

And of course there are many more uses for daemons that can watch for file and directory events. As noted above, libevent can be used to monitor individual files for changes, though this doesn't work on all platforms and monitoring directories is more complicated.

Step forward inotify. Inotify provides a simple way to monitor a file or directory (or entire filesystem if you want). Available in Linux type systems, it talks directly to the kernel to get file system events as they happen, and buffers them for your script to deal with.

Inotify can be accessed in two different ways from your PHP script. The first is by using the inotify PECL extension to directly interact with it. The second way is to call one of the `inotify-tools` command line programs from your script. By default, the inotify extension allows you to monitor either *one* file or *one* directory under one *watch*. So, if you want to recursively watch all of the sub-directories of a given directory, you need to traverse the directory tree and set up watches for each directory (as well as watching for the creation of new directories and setting up a watch for each of those as well). If you are only interested in monitoring a couple of files or a couple of directories, then using the PECL extension is a straight-forward way to do this whilst staying within PHP. On the other hand, if you want to recursively watch one or more directories and their sub-directories (or an entire filesystem), then the command line tool `inotifywait` is usually easier to use, as it will automatically recursively set up the watches for you.

6.4.1 Using the inotify PECL extension

The following script is an example of using the PECL extension, as a daemon, to monitor the current directory for any file `access` or `modify` events, and pop up a notification bubble.

```
1  <?
2
3  # Set us up a daemon (see the first section in this chapter for details)
4
5  $pid = pcntl_fork();
6
7  if ($pid == -1) { exit("Could not fork the child process"); };
8
9  if ($pid) { exit(0); };
10
11 if ( posix_setsid()  === -1 ) {
12         exit("Could not become the session leader");
13 };
14
15 $pid = pcntl_fork();
16
17 if ($pid == -1) { exit("Could not fork child process into grandchild"); };
18
19 if ($pid) { exit(0); };
20
21 if (!fclose(STDIN)) { exit('Could not close STDIN'); };
22 if (!fclose(STDERR)) { exit('Could not close STDERR'); };
23 if (!fclose(STDOUT)) { exit('Could not close STDOUT'); };
24
```

```
25 $STDIN = fopen('/dev/null', 'r');
26 $STDOUT = fopen('/dev/null', 'w');
27 $STDERR = fopen('/var/log/our_error.log', 'wb');
28
29 $stayInLoop = true;
30
31 # Let the user know we are now up and running
32
33 `notify-send -i face-glasses 'File monitoring daemon started'`;
34
35 # We create an inotify instance to use
36
37 $inotify = inotify_init();
38
39 # We then add one or more "watches" to the instance. Each watch gives
40 # inotify a file or directory to monitor, and tells it which events to
41 # watch for. In this case, we want to watch the current directory
42 # (specified by the magic constant __DIR__ ), and we want to look for
43 # file accesses (IN_ACCESS) or (|) file modifications (IN_MODIFY). This
44 # is a "bit mask" of events, which will discuss later.
45
46 $watch = inotify_add_watch($inotify, __DIR__, IN_ACCESS | IN_MODIFY);
47
48 # Once our watch is set up, it will keep running, so we can enter our
49 # usual program loop and wait for inotify to tell us when an event occurs
50
51 while ($stayInLoop) {
52
53         # We now call inotify_read to get an array of file events that our
54         # watch has spotted. inotify_read blocks execution until an
55         # event occurs, so if nothing is happening our script will sit and
56         # wait at this point until it does.
57
58         $events = inotify_read($inotify);
59
60         # inotify_read returns an array of events. Often this may just be an
61         # array with one event on a non-busy file/directory. However, multiple
62         # events may occur at the same time, and inotify queues events while
63         # your program is doing other things (like processing previous events)
64         # so you should always be prepared to handle multiple events. So we
65         # loop through the $events array...
66
67         foreach ($events as $event) {
68
69                 # The "mask" value tells us which type of event (or events)
70                 # have occurred
```

```
71
72                  $type = $event["mask"];
73
74                  # "name" gives us the file or directory name of the event
75
76                  $filename = $event["name"];
77
78                  # We'll compose a human readable string to present to the user
79
80              switch ($type) {
81
82                      case IN_ACCESS :
83                          $what =  "accessed";
84                          break;
85                  case IN_MODIFY :
86                          $what = "modified";
87                          break;
88                  case (INACCESS+IN_MODIFY) :
89                          $what = "accessed and modfied";
90
91          };
92
93          # We'll now pop up a notification bubble with the event details
94
95          `notify-send -i document '$filename was $what'`;
96
97          # Finally, if a file called bye.txt was accessed or modified,
98          # we'll exit our loop and exit the daemon
99
100                 if ($filename == 'bye.txt') { $stayInLoop = false; };
101         };
102
103 };
104
105 `notify-send -i face-raspberry 'File monitoring daemon stopped'`;
106
107 exit(0);
```

In the above script, we tell inotify what types of events we are interested in by creating a "bit mask". If you are not familiar with bit masks, you can think of them in this case in the following (simplified) way :

- Each possible event type is represented by a constant (e.g. `IN_ACCESS` and `IN_MODIFY`)
- Each constant is simply an integer (e.g. `echo IN_ACCESS;` prints 1, `echo IN_MODIFY` prints 2)

- Each integer is chosen so that adding any combination of these integers together always gives a unique number. (technically the numbers are powers of 2).
- `echo IN_ACCESS+IN_MODIFY;` will print 3. No other addition of any of the constants will give 3. So if in your event the "mask" value is 3, you know that both IN_ACCESS and IN_MODIFY occurred in that event.

Further Reading

Full explanation of bit masks on Wikipedia
http://en.wikipedia.org/wiki/Mask_%28computing%29

In the above example, we use `inotify_read()` to get details of occurring file events, which is a blocking function causing our script to pause. If you want to occasionally check for events, but not cause your script to block if none have happened, you can treat the inotify instance as a stream (the `inotify_init()` function actually returns a standard file descriptor). In this way you can, for instance, use `stream_set_blocking()` to make it a non-blocking stream.

Further Reading

Example of accessing inotify as a stream, in the PHP manual (see example section)
http://www.php.net/manual/en/function.inotify-init.php

6.4.2 Using the `inotifywait` command

The second way to call inotify is by "shelling out" to the `inotifywait` shell command. The following example sets up a watch to look for any file modifications or deletions anywhere in the filesystem under the `/home` directory. For brevity I've shown this as a simple CLI script, you can use the previously shown technique for turning it into a daemon as necessary.

```
1  <?
2
3  # Create the command line string to execute inotifywait with the options
4  # we want. In this case, we use the following options :
5  #         --csv : Returns the output in easy-to-parse csv format
6  #        -q : Suppresses any messages and only shows the event output
7  #        -r : Run in recursive mode, so all sub-directories are included
8  #        -m : Run in "monitor" mode, which simply means it runs continuously
9  #        -e : Specifies which events to listen for (modify and delete)
10 # Finally we give it the directory to watch (/home). This could be just
11 # '/' if you want to watch the whole filesystem.
12
13 $command = 'inotifywait --csv -q -r -m -e modify -e delete /home';
14
15 # Because we want to start displaying the events as they happen, rather
```

```
# than after the command finishes (as it won't finish), we can't use
# methods like backticks or shell_exec() to run the script. Instead we use
# popen() to treat it like a file stream.

$handle = popen($command, 'r');

# read each line of output as it occurs

while ($line = fgets($handle)) {

        # The data is in CSV format (due to using --csv), so parse it into
        # a PHP array

        $event = str_getcsv($line);

        # Output details of the event to the shell

        echo 'A '.$event[1].' event occured in Directory '.$event[0];
    echo ' on file '.$event[2]."\n";

};
```

`inotifywait` automatically sets up watches on all sub-directories when you use the `-r` recursive option. However on a large filesystem this can take several seconds or more when it starts for the first time. See the `man` page for `inotifywait` for all the possible options. Read the next chapter for details on `popen()` and other ways to interact with shell commands from PHP.

6.4.3 Inotify limits

While inotify is a very versatile tool, be aware of a couple of limits. Firstly as noted above setting up recursive watches can take some time, during which events will not be listened for. Secondly, the default limit for the number of "watches" (directories or sub-directories) is usually set to 8192. You will need to increase this in `/proc/sys/fs/inotify/max_user_watches` if your filesystem has more directories. Finally, although inotify buffers notifications for you while your script is busy, that buffer has a finite size. Thus, if the task that your script performs in response to events takes a long time (e.g. file type conversion), you may wish to farm out that processing to external scripts to allow yours to get back to responding to events quicker. See the following section on task dispatch and management systems for ways to do this.

6.5 Task dispatch & management systems

We've looked in this chapter at how you can fork new processes, and we'll look in the next chapter at how to execute or call, and talk to, other external commands. These examples should give you some good ideas of how you can create worker tasks to carry out processing in parallel with your main PHP scripts. There are many good cases for rolling your own task/worker dispatch

and management scripts, but there are also many times when it may be more prudent to use something already written by an expert. Luckily in these cases we can take advantage of any one of a number of excellent task dispatch and management systems that work with PHP. A few of the most common and useful systems are listed below, but first we'll look at one particular system - Gearman - which has fantastic PHP bindings and good community support, and is an ideal task system to break your teeth in with.

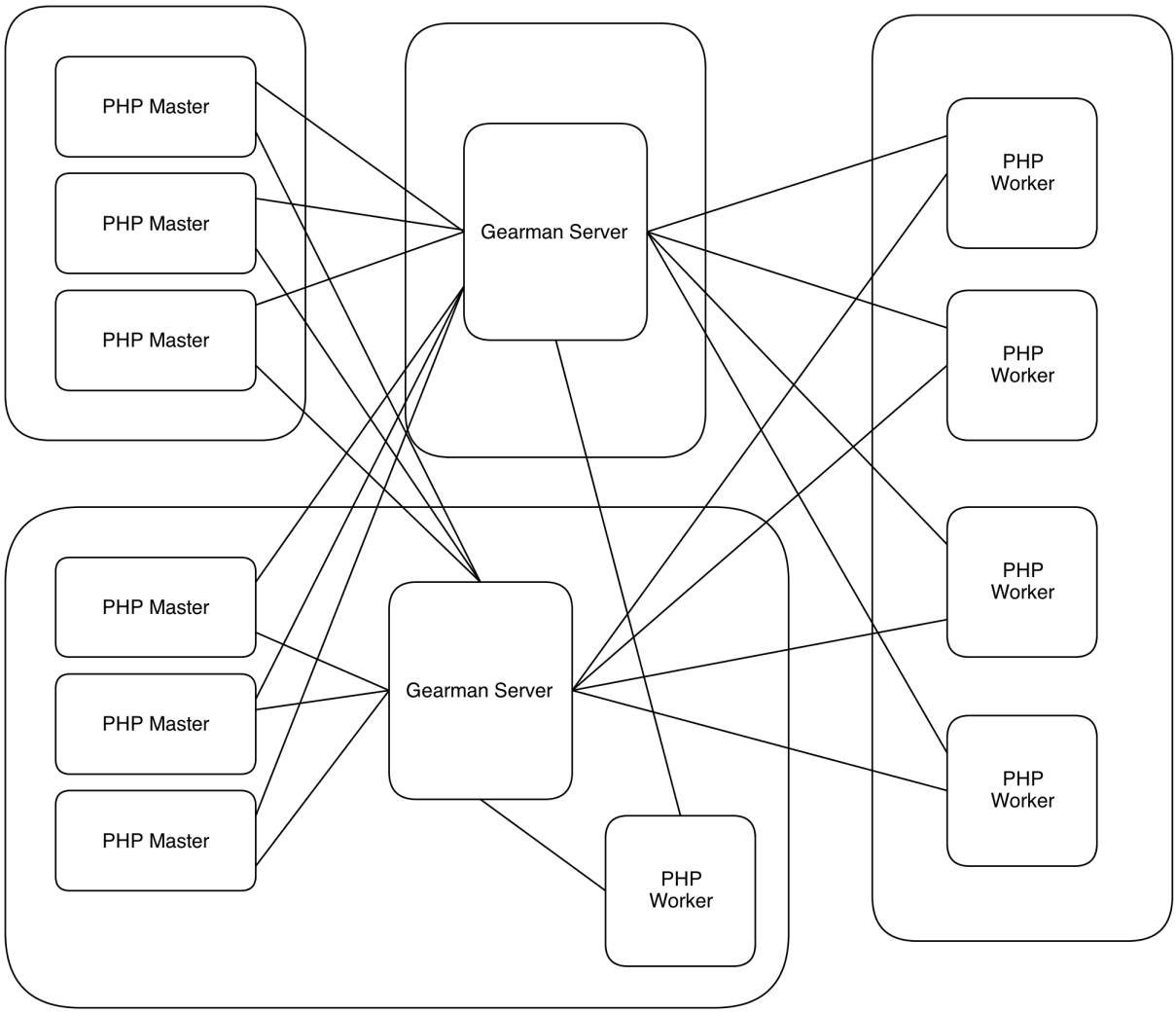
6.6 Gearman and PHP

From the PHP manual :

- *"Gearman is a generic application framework for farming out work to multiple machines or processes. It allows applications to complete tasks in parallel, to load balance processing, and to call functions between languages. The framework can be used in a variety of applications, from high-availability web sites to the transport of database replication events."* http://www.php.net/manual/en/intro.gearman.php

In essence, Gearman is a "middle-man" between your main scripts and your worker scripts. Your main scripts can fire off tasks to Gearman, and Gearman will allocated those tasks to workers when they become available. It will then monitor the workers and report their progress back to the main script along with the results of the task. You can have multiple Gearman servers for redundancy, and you can operate everything in a distributed manner across many different machines. You simply tell your main scripts where your Gearman server(s) are, and start firing off tasks. You don't even have to wait for the results of the tasks if you don't want to, and you can register call-back functions to handle results from multiple tasks as they come in. To configure the worker scripts, you tell them where the Gearman server(s) are and specify which tasks that particular script can handle, and add a little code to feed back progress during the task if you wish. You then fire up your worker scripts and they sit and wait for tasks to come in.

To help you visualise the possibilities available with Gearman, the diagram below shows a multi-machine setup with redundant Gearman servers.



Example with multiple machines/masters/clients/servers

In this particular network, we have six PHP "master" scripts running on two different machines. Each of these masters are connected to two Gearman server instances, one running on its own machine and one on a shared machine. In turn, there are five PHP "worker" scripts, four running on their own machine and one on the shared machine. All of the worker scripts are also connected to both of the two Gearman server instances. In this set up, you can see that if one of the Gearman server instances are unavailable, the system would continue to run as all components are connected to both server instances at the same time, and so can use either. Likewise if one or more master or worker scripts are unavailable, work still happens as the Gearman server instances dish out the jobs to the remaining scripts. And the best thing of all is that Gearman takes care of this "fallback" system for you. You simply point your masters and workers at the Gearman servers and Gearman deals with routing the jobs appropriately.

You can, of course, run it all on the same machine, and this is a common configuration. This makes it suitable for using Gearman to support parallel computations in PHP where there is no need for extra hardware but where PHP's single threaded operation hampers full utilisation of the machine.

Further Reading

"Using Gearman from PHP" by Lorna Mitchell
http://www.lornajane.net/posts/2011/using-gearman-from-php

"Queues are your friend" section in "5 Things You Should Check Now to Improve PHP Web Performance" by Gonzalo Ayuso gives example PHP/Gearman code
http://php.dzone.com/articles/5-things-you-should-check-now

6.7 Other task dispatch systems

A couple of other common task dispatch systems with PHP bindings are listed below. If none of them quite meet your needs, you can use the inter-process communication techniques described in the next chapter as the basis for creating your own. We will also look at simulating multi-threading in PHP in Chapter 9.

Beanstalkd

A simple, fast task dispatch system

Main website : http://kr.github.com/beanstalkd/

Main documentation & Installation info :
https://github.com/kr/beanstalkd/wiki

PHP Libraries: https://github.com/kr/beanstalkd/wiki/client-libraries

php-resque

A PHP task queue based on Redis. A port of the Github Ruby based Resque.

Main website : https://github.com/chrisboulton/php-resque

Main documentation & Installation info :
https://github.com/chrisboulton/php-resque#background

Further Reading

“Offline Processing in PHP with Advanced Queuing” by Christopher Jones
https://blogs.oracle.com/opal/entry/offline_processing_in_php_with

7 Interacting with other software

Few pieces of software run in total isolation, most will talk to various other processes and programs on your system. Particularly on open source systems (where licensing is less of an issue) there is often little point re-inventing the wheel when, with minimal effort, you can simply call upon other existing software and libraries to perform tasks that are already a "solved problem".

Although you may have come across some of these methods when programming for the web, there is generally less call for interactions when your script will execute and be gone in the blink of an eye. There are a number of different methods for invoking and interacting with other software, which we will cover below. The primary use for these methods is interacting with non-PHP software, however by virtue of the fact that these methods are language agnostic they can of course be used for communication between two or more scripts written in PHP.

7.1 Starting external processes from PHP, or "shelling out"

Before we can start talking to another process, that other process has to start running. You can of course manually start another piece software at the same time as your PHP script, but often it is useful for the PHP script itself to start the other software when it needs to. This is often referred to as "shelling out".

There are a number functions availble in PHP to do this, each doing it in a slightly different way:

- `exec()` : Executes a program and sends the text output to the user
- `passthru()` : Executes a program and sends the binary output to the user
- `system()` : Executes a program and gathers the output for use by PHP
- `shell_exec()` : Executes a command via a shell and gathers it's output for use by PHP
- Backtick operator (e.g. ‘command‘) : identical to `shell_exec()` above
- `pcntl_exe()` : Executes a program in the current process space, that is to say it stops the current PHP script and replaces it with the specified program
- `popen()` : Executes a program and opens a file pointer (identical to the pointers returned by `fopen()` for example) to read or write to the process via STDOUT or STDIN. Can only read or write, not both.
- `proc_open()` : Like `popen()`, but with more control. Allows both reading and writing at the same time. Not as simple to use as `popen()`.

Which method you choose depends on what you intend to do with the newly opened process and how you want to talk to it. If you're not sure from the descriptions above which function is appropriate for your use, the Further Reading list below gives some pointers, information and examples of implementations which should give you some direction.

Further Reading

"Cookbook" recipes for using the functions listed above
http://pleac.sourceforge.net/pleac_php/processmanagementetc.html

"Proc_Open: Communicate with the Outside World" tutorial by Timothy Boronczyk
http://www.sitepoint.com/proc-open-communicate-with-the-outside-world/

"Shelling Out Sucks" by Stefan Karpinski, an article on the downsides of calling external programs via an intermediate shell
http://julialang.org/blog/2012/03/shelling-out-sucks/

When calling external scripts, remember that using untrusted user input in command names or options is a recipe for a bad security day! The `escapeshellarg()` function can protect you from some inadvertent mistakes, but won't stop you executing "bad" functions or files.

Further Reading

Sanatising shell arguments using `escapeshellarg()` in the PHP manual
http://www.php.net/escapeshellarg

7.2 Talking to other processes

Once we've got other processes up and running, PHP supports a number of ways to talk to them, which we'll look at below. These methods mainly apply to processes that have been started up by other means, or which your script has started and disassociated itself from. If you've used `popen()` or `proc_open()` to run them, you can simply communicate via PHP streams as normal. You can still use the methods below as well though in addition, if you need additional channels of communication.

7.3 Semaphores

A basic form of interaction between programs is "negotiation over shared resources". If you have a data source, peripheral, data set, or any other kind of resource for which you only want one program to have access to at a time, you need some form of communication between the programs to decide who is using it and work out when they have finished with it. Since the advent of System V, Unix has had the concept of "semaphores" to manage this process, and PHP embodies their use in the `sysvsem` extension, which is usually already compiled into most PHP distributions. Semaphores are like a flag that gets passed around, and only the program holding the flag can access the resource. The following code shows an example of passing a semaphore. Open two shell windows, and run the script in each one. You will see the scripts getting and releasing the semaphore, such that at any one time only one script will have the semaphore.

```
<?

$process = getmypid();

$semaphore = sem_get('123456', 1, 0666, 1);

if (!$semaphore) { echo("Couldn't get semaphore\n"); exit;};

while (1) {

        sem_acquire($semaphore);

        echo ($process." has the semaphore\n");

        sleep(rand(0,5));

        if (!sem_release($semaphore)) {

                echo("Couldn't release semaphore\n");

                exit;

        };

        echo ($process." has released the semaphore\n");

};
```

First we get the pid (Process ID) of our script to be sure we are running different processes. Next, we create a semaphore using `sem_get()` and return an ID for that semaphore (you can run multiple semaphores for different reasons at the same time, so the ID is important). If another script has already created it, this will just return the ID. Note that all of your scripts/programs that want to share a particular semaphore must use the same parameters for the `sem_get()` function (or its equivalent in another language) otherwise a new semaphore will be created. The first parameter for `sem_get()` is a unique integer key with which to identify the semaphore (this is different from the ID returned by the `sem_get()` function, which is PHPs internal resource ID for the semaphore and will vary from script to script for the same semaphore). The second parameter is the number of unique users permitted to hold the semaphore at any one time. This allows you to, for instance, allow 2 (or more) people to "hold the flag" at the same time, which can be used to place an upper limit on the number of consecutive users of a resource. The third parameter specifies the standard Unix permissions (as you would use with `chmod` for instance), and the fourth parameter specifies whether to automatically release the semaphore if the request goes away.

Once we have our semaphore set up with `sem_get()`, we then enter a loop. Each time through the loop we try and get the flag with `sem_aquire()`. This is a blocking function, which means that our program will sit and wait until the semaphore is available and we can grab it. Once

we have it, the program will continue. We sleep for a random time (in "real life" we would do something useful with the resource we are protecting with the semaphore), and then release it back again. At this point the next waiting process will pounce in and "grab the flag", as we go back to the start of the loop and try and acquire it again.

It's important to note that semaphores work on the honour system. Every program is expected to implement and respect the semaphore. The system itself doesn't know about or enforce semaphores, so a "rogue" program can simply nip in and use the resource regardless of who is "holding the flag". Thus semaphores shouldn't be used for any systems which you don't absolutely control (or don't mind if un-entitled software grabs your resource), and only where all programs *want* to share the resource. There is also little protection against deadlocks, where one program gets the semaphore and fails to release it due to an error with that program. The final parameter of the `sem_get()` function does allow for the automatic releasing of the semaphore if the request disappears, e.g. if the process crashes or the semaphore is closed in the current script with `sem_remove()`, but this may not protect against the case where the process simply hangs.

The `sem_get()` function requires a unique integer key to identify the semaphore, and this key (like a telephone number) must be known to all of the programs in advance. If you are having trouble deciding on a key or sharing it, take a look at the `ftok()` function which generates a key based on the pathname to a known file and a project identifier. However read the user comments on the PHP manual page for some potential gotchas.

Further Reading

ftok() in the PHP Manual
http://php.net/manual/en/function.ftok.php

When you have your semaphores running, you can check the status of them from the shell with the command `ipcs -s` which will show details of all of the semaphores on the system.

While semaphores don't directly allow you to exchange data with other programs, they can be used to facilitate communication via other means, such as shared files or shared memory segments which we look at below.

Further Reading

Semaphore function in the PHP Manual
http://www.php.net/manual/en/ref.sem.php

"What Is Wrong With PHP's Semaphore Extension" by Jonathon Hill
http://jonathonhill.net/2012-12-08/what-is-wrong-with-phps-semaphore-extension/

7.4 Shared Memory

"Shared Memory" is a simple and widely supported method of passing data between two or more processes, and it really is as simple as it sounds. A process can create a segment of memory, with

a key as a unique ID, assign standard Unix type permissions to it, and then it and other processes can read, write and delete data from that segment as necessary (and as permissions allow).

Shared memory is probably the fastest way of sharing information between two processes. There is no disk IO or database access for instance to slow you down, and no intermediate message broker to spend time processing and distributing the information. The downside is that you have to manage the whole process yourself, although this isn't as hard as it sounds.

Shared memory is supported in two extensions, the SystemV `sysvshm` extension (part of the Semaphore extension like `sysvsem` described in the previous section) and the more recent `shmop` extension. We will look at the latter as it is based on the C `shm` api which makes it easier to share data with non-PHP programs, unlike the former which used its own proprietary data format making it hard to share with anything other than PHP. `Shmop` is also usually faster than the `sysvshm` as it stores its data in a raw form.

Like files, shared memory can become corrupt if multiple processes try to write to it at the same time. For our example below, we don't need to worry about locking the shared memory segment as only one process will be writing to it so we won't include any locking code for brevity and clarity. However where locking is an issue, you can use the semaphore method outlined in the previous section to ensure only one process writes to the memory segment at a time. Using multiple memory segments at the same time by the same or different processes doesn't require locking, as long as only one process is writing to a specific segment at a given time.

It's time to look at an example of shared memory in action. Below are two scripts. The first script below, `generator.php`, generates an array of three random numbers every second, encodes them as JSON and puts them into a shared memory segment. The second script, `display.php`, retrieves that data and outputs it to the terminal. To try these, open two terminal windows and run one in each, at the same time.

- This first script is **generator.php**

```
1  <?
2
3  $segment = shmop_open('1234456', 'c', 0755, 1024);
4
5  for ($counter=0; $counter < 20; $counter++) {
6
7          $jsonArray = json_encode(
8                                                  array(rand(0,50000),
9                                                          rand(0,2000),
10                                                         rand(5000,100000))
11                                 );
12
13         $jsonArray = str_pad($jsonArray, 1024-strlen($jsonArray), ' ');
14
15         $dataSize = strlen($jsonArray);
16
```

```
17          $bytesWritten = shmop_write($segment, $jsonArray, 0);
18
19          if (!$bytesWritten) { echo("Error - couldn't write to memory\n"); };
20
21          if ($dataSize != $bytesWritten) {
22                  echo("Error - couldn't write all data to memory\n");
23          };
24
25          sleep(1);
26
27  };
28
29  shmop_delete($segment);
30
31  shmop_close($segment);
```

- This second script is **display.php**

```
1   <?
2
3   $segment = shmop_open('1234456', 'c', 0755, 1024);
4
5   while (1) {
6
7           sleep(1);
8
9           $size = shmop_size($segment);
10
11          $jsonArray = shmop_read($segment, 0, $size);
12
13          echo("Fetched $size bytes of data at ".date("H:i:s").
14                  ". Our random numbers are :\n");
15
16          print_r(json_decode(chop($jsonArray)));
17
18  };
```

Lets look through `generator.php`. First we create a shared memory segment using `shmop_open()`. This takes four parameters, the first is a unique integer key to identify this segment, and as with the section on semaphores above must be unique and shared with the other processes that are going to access our shared memory segment. See the note about using `ftok()` to create a unique id in the semaphore section for one way of generating and sharing a unique id. The second parameter is the "mode" in which we will open the segment. The options are :

- `a` - open for read-only access on existing segment

- `c` - create a new segment, or open for read and write if it already exists
- `w` - open for read and write access on existing segment
- `n` - create a new segment, or fail if it already exists

Note that the "read-only" access above only applies to the process we are in, another process can open the segment for writing (if it has write permissions, see below).

The third parameter to `shmop_open()` specifies the permissions for the memory segment, in octal. These are the same as unix file permissions (that you assign using `chmod` for example). These only apply to segments that have not already been created, and are ignored for existing segments. The last parameter is the size, in bytes, of the shared memory segment. You should ensure that the segment will be big enough for the data you want to put into it (otherwise your data will be truncated when writing), but not larger than necessary (otherwise you will waste memory that could be being used by other processes). Once created, `shmop_open()` returns an identifier that we can use to refer to this segment, which we place in `$segment`. Note that this is an internal PHP resource identifier, and is different to the unique key that we used as the first parameter for `shmop_open()`.

Once we have created our memory segment, we perform a loop 20 times and then mark the segment for deletion and close it. In the loop, we create an array of three random numbers and encode it into a JSON string called `$jsonArray`. We then pad this with spaces up to 1024 using `str_pad()`. This padding is important, as a memory segment isn't like a variable in PHP, it has a fixed size. If you store a 5 character string in it, for example, and then store a 3 character string instead, you will find that when you try and read it you will have the last 2 characters of your initial string on the end of your 3 character one. Thus, if you are going to be repeatedly using the same segment, you need to find a way of clearing any extra space in the segment that you aren't using at that moment, or find a way of passing the length of data that you have written to the other process so that it only reads the correct amount of data. One common way to do this is to write the length of your string to the memory first (always in, say, the first 4 bytes of memory) and then write the actual data after this. The receiving process then reads the length data first and then reads only the required number of bytes from the remaining data in the segment. The other way, as we are doing here, is to always fill the segment completely. We do this by `str_pad()`ing our data with spaces up to 1024 bytes, and then `chop()`ing off the spaces on the receiving end.

Once we've padded `$jsonArray`, we get its size (in this case it should always be 1024) and write it to the memory segment using `shmop_write()`, which returns the number of bytes it has successfully written into `$bytesWritten` (hopefully 1024 if all went well). `shmop_write()` takes three parameters. The first is the `$segment` PHP resource identifier, the second is the string to write to memory, and the third is the offset. The offset specifies where in the memory segment we want to start writing the data. In our example we want to start at the beginning so we use 0, if you were using the first 4 bytes, say, to write the data size then your second `shmop_write()` writing the actual data would start with an offset of 5.

After we've done the writing, we check to see that we managed to write (that `$bytesWritten` isn't -1) and that we managed to write *all* of the data (we didn't try to write more than 1024 bytes in our case).

Once we've been through our loop 20 times (sleeping for a second each time to keep things readable in the output!) we mark the memory segment for deletion and close it. We'll come back

to those last two lines in a moment.

Now lets look at our other script, `display.php`. We open the shared memory segment with `shmop_open()` in exactly the same way as our first script. Because we use the "c" mode which creates the memory segment, it doesn't actually matter which of our scripts we open first. We could have used the "a" read-only mode if we knew that `generator.php` was running first (or were willing to wait for it).

Once the segment is open, we continually loop (sleeping for a second each time as we know the `generator.php` script will only update the segment once a second) getting and displaying the contents of the memory segment.

To get the contents, we use `shmop_read()`. This takes three parameters, the first is the `$segment` PHP resource identifier, the second is the offset where we will start reading, and the third is the amount of data we want to read. In our case we want the entire segment, so we start at the beginning (offset = 0) and read to the end (1024 bytes along). If you want to read to the end of the segment but don't know it's size, you can use `shmop_size()` to find it as we do here, although in our case it's superfluous as we already know it will be 1024.

Once we have our data, which should be the `$jsonArray` string padded with spaces, we `chop()` off the spaces and `json_decode()` the string back into an array which we `print_r()` out.

Now we will return to the last two lines of `generator.php`, calling `shmop_delete()` and `shmop_close()`. `Shmop_delete()` doesn't actually delete the memory segment, instead it "marks it for deletion". This means that no new processes can open the memory segment, but existing processes which have already opened it can continue to access it. It will remain accessible until all of those existing processes close it (or exit), at which point it will be deleted.

You can see this deletion process in action by running these two scripts. `generator.php` will exit after 20 seconds, and `display.php` will keep running forever (due to the `while(1)` loop). Once `generator.php` exits, you will notice that `display.php` keeps outputting the same 3 random numbers each time. This is because although we've marked the memory segment for deletion in `generator.php`, `display.php` still has it open and so it can still read it (albeit getting the same value each time as nothing is updating it). If you kill `display.php` and start it again (without starting `generator.php`), you will find that it now reads nothing, as the memory segment was deleted when the previous `display.php` process (the last one using it) exited.

The other peculiarity is that we call `shmop_delete()` *before* we call `shmop_close()`. The reason is that if we close it first, PHP will not have the internal reference any-more and so won't know what memory segment to mark for deletion. Once you have called `shmop_delete()`, you can't do anything else within your script to the memory segment other than closing it. You must have appropriate permissions to be able to mark a segment for deletion. You should ensure that you mark your memory segments for deletion at some point before all of your processes are finished with it. Otherwise you will end up with a "memory leak", i.e. memory is "allocated" and thus can't be used by other programs, lowering the amount of available memory to the system as a whole.

Of course the `shmop` functions only deal with transferring the data between processes. Understanding or parsing the data is left as an exercise for the programmer. If the application you are communicating with doesn't dictate the format (for instance if you are developing the system yourself rather than trying to hook into an existing system), then you will need to decide upon

a suitable format. Although we have used JSON to encode our array into a string in the example above, you could also use PHP's `serialize()` and `unserialize()` functions if you knew you would only be sharing the data with other PHP scripts. However if you think your software may be interacting with programs written in another language in the future, sticking to language independent encodings like JSON or XML is a wiser choice.

Although it should be obvious, remember that any data in memory is volatile. It won't be there if your system crashes, if the memory segment is deleted (usually when one or more programs exit), or after a reboot. Write anything critical to disk as well.

Further Reading

SHMOP Shared Memory in the PHP Manual
http://php.net/manual/en/book.shmop.php

SimpleSHM

An abstraction layer for `shmop` to simplify getting and setting shared memory

Main website : https://github.com/klaussilveira/SimpleSHM

7.5 PHP message queues

Message queues provide an easy way for multiple processes to interact. At its most simple, one process adds messages to the queue, and another takes them off. The queue is maintained usually by a intermediate process or the OS itself. Queues make a programmers life easier than, say, messaging using shared memory as the programmer doesn't need to worry (too much) about timing and synchronisation, and there are usually no need for locks. The sending process can fire off a message and forget about it (if its wants to), the receiving process can simply sit around and wait for a message to arrive (or do other things and check back for the messages later). There are a number of message queue APIs and extensions available to PHP, and we will look at most of those later in this chapter. In this section, we will look at the `sysvmsg` message queue which is part of the Semaphore extension that we looked at in the previous sections. This is a basic but useful message queue system, and has the advantage that it is usually compiled into most PHP distributions and it doesn't require any third party APIs, daemons or libraries.

Lets jump right in with an example. We're going to modify our random number generator scripts from the previous section to use a message queue. Our first script, `generator2.php` is going to fire off 20 messages, each with 3 random numbers, in quick succession and then exit. Our second

script, `display2.php`, will pull those messages out at its own pace (1 per second) and display them.

- This script is **generator2.php**

```
<?

$queue = msg_get_queue('1234456', 0666);

for ($counter=0; $counter < 20; $counter++) {

        $phpArray = array(rand(0,50000), rand(0,2000), rand(5000,100000));

        if (!msg_send($queue, 3, $phpArray, true, true, $errorCode)) {

                echo("Error - couldn't send message - code $errorCode\n");

        };

};
```

- This script is **display2.php**

```
<?

$queue = msg_get_queue('1234456', 0666);

while (1) {

        sleep(1);

        if (!msg_receive($queue, -4, $realType, 1024, $phpArray, true,
                0, $errorCode)) {
                        echo("Error - Couldn't receive message - code $errorCode\n");
        };

        echo("Fetched message at ".date("H:i:s").". Random numbers are :\n");

        print_r($phpArray);

};
```

If you run generator2.php first, you will see it complete and exit almost straight away. You can then wait as long as you like before running display2.php, hours or days even, as the messages will happily sit there in the queue (unless you turn your computer off). When you do run it, you will see the random numbers appearing, one set every second. If you use ctrl-c to kill the script before it has read all 20, you can simply start it going again and it will pick up where it left off, until all 20 have been displayed. At that point, it will sit and wait for any new messages to appear on the queue. If you run generator2.php again in another terminal while display2.php is sitting and waiting, you will see display2.php spring into life again almost immediately and start displaying the new set of numbers we just generated.

So lets look at how these scripts work. They should seem similar to, but simpler than, the earlier shared memory examples. In each script, we first open the message queue with the msg_get_queue() function. This either creates, or if it already exists, opens the message queue. We supply it with a unique integer key as the first parameter to indicate which queue we want to use. See the notes on the ftok() function in the semaphore section above regarding generating unique keys. The second parameter is standard Unix permissions for the queue. The function returns a unique ID for the $queue which we use with the msg_send() and msg_receive() functions. This ID is an internal PHP resource ID and is different to the queue key we specified earlier.

In generator2.php we then generate a PHP array of random numbers as before, and we put this array into the queue with the msg_send() function. We do this 20 times in quick succession and then exit. This function takes a number of parameters:

- queue : This is the $queue resource identifier
- msgtype : This is an arbitrary integer which specifies the "type" of message. You can choose to assign your messages an integer corresponding to any kind of criteria you want, and then when receiving messages you can filter by this type field. For instance, you can choose to receive the next message of type "3", or the next message of any type. This can be useful, for instance to assign a priority to messages using the type parameter and use that to deal with them in order of priority at the receiving end.
- message : This is the actual message to send. This should be a string, UNLESS you have the next parameter (serialize) set to true, in which case you can pass a variable or object of any type which PHP can serialize into a string. In our example for instance we pass a PHP array directly to the msg_send() function without calling any kind of encoding function on it first (e.g. json_encode() or serialize()) because we have the next parameter set to true.
- serialize : If this is set to true, this will automatically call the session module serialization mechanism to serialize the message into a string. This is usually the serialize() function (unless you have chosen to use the WDDX serializer or similar). If you are planning to pass data to non-PHP programs using the message queue, you should not use this automatic serialization (which is unique to PHP) and instead use another encoder like json_encode() or format your data as XML for instance, depending on which formats the programs you are communicating with will accept.
- blocking : If the queue is full and you try and send a message, the msg_send() function will "block". This means it will sit and wait for space to become available in the queue and hold up your script until that happens. If you set this parameter to false, instead of blocking the function will return false, set the errorcode parameter to MSG_EAGAIN (to

indicate you should try again later) and let the script continue to run. In this case, it is your responsibility to remember the message and try to send the message again as the `msg_send()` function will forget all about it.
- `errorcode` : This parameter stores any error code generated in case of an error. If an error occurs, the function will return `false` as well as setting this code.

So in our example above, we use `msg_send()` in blocking mode to send the PHP array `$phpArray` as our message, assigning it as a type "3" message (which is purely arbitrary in our case), using automatic serialization of the array into a string and catching any error codes in `$errorCode`.

Now we look at our `display2.php` script. After calling `msg_get_queue()` to join the queue, we start a continuous loop (using `while(1)`) which will try and read a message from the queue each time we go through the loop. We've added a `sleep(1)` one second delay to space out our message requests and show that the timing of our message requests isn't important (you can change this to another value and it will work just as well, or indeed remove it completely and grab all the messages without delay). To get a message, we use `msg_receive()`, which takes the following parameters:

- `queue`: The `$queue` resource identifier
- `desiredmsgtype`: The type of message you want to receive. If this is 0, you will simply get the next message in the queue. If it is set to a positive integer, say 5, you will get the next message that was sent using 5 as the type parameter in `msg_send()`. If it is a negative integer, say -3, you will get the next message on the queue with the lowest type that is less than or equal to the absolute (non-negative) value of this parameter. e.g if we specify -3 we will get the first message with type 1, 2 or 3 (and if there are multiple messages with these types, then we take the first message that has type 1, then when all the 1's are gone the first with type 2 and so on.).
- `msgtype`: This is the actual type of the message that we have received. This is useful when we specify a negative value for `desiredmsgtype` above to get a message from a range of types, and want to see what type the message we were given actually is.
- `maxsize`: This specifies the maximum size (in bytes) of the message that you are willing to receive. If the message that you would receive is bigger than this size then the `msg_receive()` function will either return false and not return a message at all, or if `MSG_NOERROR` is set in the `flags` parameter below, it will instead truncate the message and return just the first `maxsize` bytes. Be aware that the `msg_receive()` function allocates a block of memory equal to `maxsize` before it tries to read the message. This means that even if the message is small (e.g. 10kb), if you specify too large a size here (e.g. 1Gb when you only have 512Mb memory free) your script will crash.
- `message`: This is the actual message you receive. If the `unserialize` parameter below is set to `true`, any PHP types like arrays or objects will be unserialized and returned as the correct type. If `false`, you will receive the message as a (potentially serialized) string.
- `unserialize`: If true, the contents of message will be automatically unserialized.
- `flags`: You can specify the following flags:
  - `MSG_IPC_NOWAIT` : If there are no suitable messages to receive, by default `msg_receive()` will "block" the execution of the script and sit and wait for a new message to arrive. If you set this flag, the the script will not block, will return the integer value for `MSG_ENOMSG`, and your script will continue.

  - MSG_EXCEPT : If you ask for a message of a particular type using a positive integer in desiredmsgtype above, and you set this flag, you will get a message of ANY type EXCEPT the one you have specified.
  - MSG_NOERROR : Stops errors when the message is too big, see msgtype above for details.
- errorcode: If an error occurs, the msg_receive() function returns false and this parameter is set to the relevant error code.

So in our example, we ask for a message of maximum 1024 bytes where the type is -4 (that is to say, its type is 1, 2, 3 or 4). Our generator2.php script creates messages of type 3 and all are less than 1024, so they will all match and be returned. $realType will be set to 3, and $phpArray will contain our data as an array because we have set unserialize to true. We haven't set the MSG_IPC_NOWAIT flag, so if there are no messages our script will "block" at the msg_receive() command waiting for new messages.

You can use msg_queue_exists() to check that a message queue exists before trying to use it, and msg_remove_queue() to destroy a queue when you are done with it. To get more information about a queue, you can call msg_stat_queue(), which returns an array with the following keys:

- msg_perm.uid : The uid (User ID) of the process that owns (created) the queue.
- msg_perm.gid : The gid (Group ID) of the process that owns (created) the queue‘.
- msg_perm.mode: The file access mode (permissions) of the queue.
- msg_stime : The time that the last message was added to the queue.
- msg_rtime : The time that the last message was removed from the queue.
- msg_ctime : The time that the queue was last changed (either a message was added or removed).
- msg_qnum : The number of messages currently in the queue.
- msg_qbytes : The maximum size in total of the message queue, in bytes.
- msg_lspid : The pid (Process ID) of the process that last added a message to the queue.
- msg_lrpid: The pid (Process ID) of the process that last removed a message from the queue.

The maximum size of the queue (all of the messages in the queue at any one time) can be read from this array as msg_qbytes. This is often set at 16kb. If you want to change this value, on Linux you can change the file /proc/sys/kernel/msgmnb to a higher value (in bytes). Remember that you will need to do that on any system that you deploy on and will need to have root privileges to do so.

The message queues we've used are "bi-directional". This means, assuming each process has the appropriate permissions, each one can both add AND remove messages from the queue. When doing this you need to be sure that you don't inadvertently remove messages that you have added yourself before the other process has had a chance to do so, for example by using the message "type" field to specify the intended recipient. Alternatively, you can open two message queues and use one for sending and one for receiving in your process (and vice-versa in the other process). It is particularly important to keep things "straight" in this regard when you start adding more than two processes to your queue, bearing in mind that the number of processes that can join your queue is only limited by your available system resources.

Futher Reading

Message Queues in the PHP Manual
http://www.php.net/manual/en/ref.sem.php

7.6 Third party message queues

As you can see, the `sysvmsg` message queue system is a simple and easy built-in way to pass asynchronous messages between processes. It provides only basic queue functionality and does have it limits, so if you have more advanced requirements you might want to check out one of the other messaging systems available with PHP bindings. A list of some of the more common ones are below, and between them cover just about any messaging need. Be aware that some of the systems are definitely not for beginners, and all of them require the deployment of additional software or libraries.

0MQ

An extremely fast and comprehensive messaging transport layer, with support for many languages and OSes. Quite advanced and not suited to beginners.

Main website : http://zero.mq

Installation info : http://www.zeromq.org/bindings:php

Main documentation : http://php.zero.mq

Example code from Rasmus Lerdorf : http://talks.php.net/show/phpuk2012/16

Tutorial : PHP to Java http://hashmade.fr/zeromq-communicate-from-php-to-java-with-multithreaded-request-reply/

SAM for IBM MQTT

An extension aimed at producing a simple extensible API for multiple messaging platforms. Currently only IBMs MQTT is supported.

Main website : http://www.php.net/manual/en/intro.sam.php

RabbitMQ

RabbitMQ is a messaging system that implements the widely supported AMQP messaging standard, a widely supported cross-platform messaging protocol.

Main website : http://www.rabbitmq.com/

Official extension : http://www.php.net/manual/en/intro.amqp.php

3rd party PHP libraries : http://www.rabbitmq.com/devtools.html

Tutorial :
http://www.slideshare.net/old_sound/integrating-php-withrabbitmqzendcon

Apache ActiveMQ Apollo

A part of the ActiveMQ Java based messaging system, supporting the STOMP, AMQP, MQTT, Openwire, SSL, and WebSockets messaging protocols. Designed for enterprise systems.

Main website : http://activemq.apache.org/apollo

PHP libraries for STOMP : http://stomp.github.io/implementations.html

PHP-Queue

A unified PHP front-end for different queuing backends

Main website : https://github.com/miccheng/php-queue

Installation info : https://github.com/miccheng/php-queue#installation

Main Documentation :
https://github.com/miccheng/php-queue#getting-started

Further Reading

"Publishing queue messages from PHP using different backends" by Artur Ejsmont
http://artur.ejsmont.org/blog/content/publishing-messages-from-php-to-different-message-queue-backends

7.7 APC cached variables

You may be aware from your web work with PHP of the APC Cache system. This can be used to cache variables in memory, which can then be accessed by different scripts. Unfortunately this only works on the web, as the cache operates on a per-process basis. When running through a server like Apache, PHP operates under the Apache process, which is a set of one (or more) long running processes. With the PHP CLI SAPI, the converse is true. Each run of each script creates its own process (the `php.exe` process), which terminates when the script terminates. APC can be enabled for CLI, but at the end of each script run the cache will be destroyed, and each process has its own cache. This can occasionally be useful for memory based caching in long running, single process programs (e.g. system daemons), but even then you may still decided that a dedicated memory caching system like memcached may be better as it can cope with unexpected restarts of the process. For these reasons, we won't cover the APC user variable cache in this book. If you are using an Apache based solution, more information can be found in the PHP manual.

Further Reading

APC in the PHP manual
http://www.php.net/manual/en/book.apc.php

7.8 Virtual files - tmpfs

Tmpfs is a filesystem which allows you to create and use files stored in virtual memory. This type of system is often called a "ram-disk". These virtual files act and operate like normal files on disk, and can thus allow your different processes to communicate by reading and writing data in files as you would with normal "on-disk" files. The advantages tmpfs brings are two fold; firstly it's fast, and secondly everything is temporary. Because the files are held in memory, there is no mechanical hard disk to wait for so I/O is very quick. And because they are held in memory, they are only temporary and will disappear upon a reboot if you haven't already deleted them. A further advantage is that, being normal files, they aren't a PHP specific technology, and so can be accessed from other software as needed.

We won't go into specific details of how to use them to communicate between processes, it is assumed that you will be familiar with accessing normal disk based files from PHP and creating and reading appropriate file formats. You can access files on a tmpfs file system in exactly the same way as normal files and streams. The fact they are in memory is transparent to your PHP script.

To create a tmpfs filesystem on Linux, you first create a directory on disk to use to "attach" the memory device to your file system. You then mount the memory device at that location and start using it.

```
1  mkdir /home/rob/myMemoryDrive
2
3  sudo mount -t tmpfs /mnt/tmpfs /home/rob/myMemoryDrive
4
5  php -r "file_put_contents('/home/rob/myMemoryDrive/test.txt','Hello');"
6
7  cat /home/rob/myMemoryDrive/test.txt
8
9  sudo umount -a /mnt/tmpfs
10
11 cat /home/rob/myMemoryDrive/test.txt
```

In the above script, we create a directory at `/home/rob/myMemoryDrive` to attach the memory device, and then `mount` it there. We execute a line of PHP to demonstrate creating a memory file as we would any other file, and then "cat" the file which should output Hello. Finally we `unmount` the device, and try to "cat" the file again, but as one would expect the file is gone, it is never saved to physical disk.

You can mount tmpfs devices using the `mount` command as shown above each time you boot your system or whenever you want to use them, or you can add an entry into your `fstab` file to have it automatically created each time your system boots. Which ever way you mount it, when you shut down or reboot always remember that it, and all of the files within it, will be destroyed.

As tmpfs operates in the same way as normal a normal file system, you need to make sure that you set the relevant file permissions to allow all of your applications to access it (or prevent access by those that shouldn't be able to meddle with it). Also bear in mind that memory swapping to

disk may occur if your system becomes short of memory, so your data may temporarily touch your hard disk in these cases, and under certain conditions may be recoverable from disk after that. Always consider the security implications of any messaging system you choose.

7.9 Standard streams

We looked at the standard streams (STDIN, STDOUT, STDERR) and how to use them in previous chapters. It should now be obvious that you can use them to communicate with other processes via shell piping and I/O redirection. If you start the processes you want to talk to from within PHP using `proc_open()` or `popen()`, you automatically use the standard streams when reading and writing to the file handles.

7.10 Linux signals

It is possible to crudely use Linux signals to communicate between processes, although this is generally used for parent-child control. See Chapter 8 for more information on Linux signals.

7.11 Task dispatch & management systems

One common use of messaging systems is to manage "worker" processes and dispatch tasks. If you want to use a pre-built solution, have a look at the task dispatch and management section in chapter 6.

8 Talking to the system

So far we've looked at software that communicates with your users, via text based or graphical interfaces, and system software that doesn't need to talk to users at all. One thing that both types of software have in common is the need to deal with the underlying system that it sits on top of. That system is a structure containing the file system, operating system, hardware interfaces and various system level services.

When programming for the web you typically don't interact with hardware, lower level aspects of the system and so on. Indeed in many cases you specifically take steps to prohibit your users from doing so! In contrast, dealing with printers, sound-cards and other hardware is a common requirement when constructing many types of software, and interfacing with system level services is often a necessity. You will work with portions of the file system from web pages, but with off-line software you usually have the freedom and resources to work with a wider range of file formats, larger file sizes and more privileged files.

In this chapter we'll look at some of the different ways to interact with these resources both from within PHP and with the aid of helper applications, and some of the issues to consider when doing so.

8.1 Filesystem interactions

There are many different types of data files that software needs to commonly interact with, from images to text files, formatted documents to videos, structured data to configuration files, and many more. PHP has built in functions for reading, writing, parsing and displaying many different types of data files, and between Pear, Pecl, Composer and third party libraries even more types are covered. On many systems, particularly unix variants, helper applications can also be called to further extend the range of file types covered. In fact, there are very few file types that you will struggle to deal with in PHP, and those tend to be proprietary formats with closely guarded specifications. If you stick to open formats, and in particular standards-based formats, you will invariably find the tools you need in the PHP eco-system.

8.2 Data files & formats

Appendix B contains a reference list of functions/libraries/helpers available for a wide range of common formats. Remember that where a particular version of a format doesn't appear, it is often usable using the functions for the generic format it is based on (e.g. many XML based formats are perfectly at home being manipulated by the XML tools listed).

Always bear in mind that data files are a large vector for security exploits, and even where software is operated locally by trusted users, those users may inadvertently try to open files from malicious sources. Always treat external data as potentially tainted, and treat un-vetted extensions/helpers as if they have potential security vulnerabilities.

8.3 Dealing with large files

When you don't have a limit on the time your script can run, you will find that you can work with bigger files than you may be used to when using PHP on the web. In general, you can deal with them in the same way as you would with smaller files. However one big problem you may run into is memory usage. It's important to understand how PHP uses memory when loading and processing files so that you can make appropriate choices in your code. Many of the libraries for handling particular file formats listed in Appendix B will deal with opening and processing large files efficiently on your behalf, so this section is most relevant when you do your own file processing, or use a library that requires you to pass it raw data (rather than a filename).

First lets look at simple methods for reading in a file in one go. PHP has two easy to use functions for doing this, `file()` and `file_get_contents()`. The former reads the file into an array, the latter into a single string.

```
<?

$filename = 'bigfile.csv';

echo("Size of file : ".filesize($filename)." bytes\n");

$memory1 = memory_get_usage();

$file_array = file($filename);

$memory2 = memory_get_usage();

$file_string = file_get_contents($filename);

$memory3  = memory_get_usage();

echo("Memory used by array : ".($memory2-$memory1)." bytes\n");

echo("Memory used by string : ".($memory3-$memory2)." bytes\n");
```

Running this on a sample large file I had lying around, gave the following output

```
Size of file : 186097433 bytes
Memory used by array : 296969824 bytes
Memory used by string : 186097588 bytes
```

So we can see that reading the file (about 177Mb) into a string adds an overhead of 155 bytes, which is not too bad at all. However reading this file into an array adds an additional 105Mb to the original size of the file! In PHP, arrays are a very versatile data structure, they *"can be treated as an array, list (vector), hash table (an implementation of a map), dictionary, collection, stack,*

queue, and probably more" (according to the PHP manual). However this versatility comes at a price, and that is the additional memory used for storing the structure information. So before loading the file, consider whether the processing you are going to do on the data can be done on a string, or whether the extra overhead of an array is worth it for the manipulation capabilities. If you need a traditional array or hash like structure without the versatility of the PHP array type, look into the PHP SPL (Standard PHP Library) which contains a range of "traditional" data structures that may be more optimised for your use case.

Further Reading

SPL book in the PHP Manual
http://www.php.net/manual/en/book.spl.php

Sometimes, no matter what type of data structure you read your file into, there isn't enough memory available on the system, you hit a memory limit imposed for your script, or you just need to keep memory usage low in general. Often processing of a datafile can be done on a line-by-line (or chunk-by-chunk) basis, and PHP allows us to read in a file piece by piece rather that in one go. Assuming you discard the data you've read before you read some more (unset it, overwrite it or write it out to a file, for example), then you will just use enough memory to store that one line or chunk.

```
1  <?
2
3  $filename = 'bigfile.csv';
4
5  $memory1 = memory_get_usage();
6
7  $file_string = file_get_contents($filename);
8
9  $memory2  = memory_get_usage();
10
11 unset($file_string);
12
13 $memoryBase = memory_get_usage();
14
15 $file_handle = fopen($filename, 'r');
16
17 while ($line = fgets($file_handle)) {
18
19 $memoryCurrent = memory_get_usage();
20
21 if ($memoryCurrent > $memoryBase) { $memoryHigh = $memoryCurrent;};
22
23 };
24
25 echo("Memory used by single string : ".($memory2-$memory1)." bytes\n");
```

```
26 echo("Max memory used when reading by line : ".
27                 ($memoryHigh-$memoryBase)." bytes\n");
```

On my sample file this gave the output :

```
1 Memory used by single string : 186097768 bytes
2 Max memory used when reading by line : 9000 bytes
```

which should illustrate the extreme differences in memory usage using the two different techniques.

If you're working with files that are, or may be, greater than 2Gb in size you should also be aware that some filesystem functions may not return the correct (or any) result for files bigger than that on many platforms. This is because these platforms use a 32-bit integer, PHP's integer type is signed, and 2Gb is the largest size that can be represented by a signed 32-bit integer. This affects functions like `filesize()`, `stat()` and `fseek()`. You can of course access external commands to replace some of these functions, for instance `wc -c` on Linux will return the number of bytes in a file for all files supported by the operating system. On 64 bit Linux, with a recent version of `glibc` installed, you can compile PHP with the `D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64` flag for better large file support, though be aware if you are writing scripts for distribution that this obviously won't be an available option for all.

In all of these examples, remember that your script has to work within any memory limit you or PHP has imposed. By default the PHP CLI SAPI turns off the memory limit, you can type `php -r "echo(ini_get('memory_limit'));"` at the command line to see the default memory limit (`-1` means no limit). From within your script, `ini_get('memory_limit')` will tell you the current maximum, and `memory_get_usage()` will tell you what you're currently using.

Handling memory usage when dealing with large amounts of data is something that often trips programmers up. As we've seen, understanding how the various functions we use operate can help us process that data more efficiently and help us to minimise the memory we use.

8.4 Understanding filesystem interactions

PHP has a number of filesystem related extensions, many of which are part of the PHP core or compiled into most PHP distributions and contain hundreds of useful functions. Most of them operate in a simple straight-forward manner, allowing you to get and set file and directory information and manipulate files and the filesystem as you would expect. We won't cover most of them here, as I'm sure many will be familiar to you from your web projects and they are covered well in the PHP manual.

Further Reading

Filesystem related functions in the PHP manual
http://www.php.net/manual/en/refs.fileprocess.file.php

Many of these functions ape command line programs that you may be used to, like `chmod`, `mkdir`, `touch` and so on, and operate broadly how you expect. However there is a difference that raises

its head in longer running script, and revolves around PHP caching filesystem information to increase performance, which we will look at next.

8.5 The PHP file status and realpath caches

On the web, speed is king. PHP operates two information caches to speed up access to the filesystem. The first is the file status cache, which caches information about a given file (like whether it exists, whether it is readable, it's size and type, and so on). The second is the realpath cache, which caches the actual, real path for a given file or directory (expanding out symlinks, relative paths, '.' and '..' paths, include_path's and so on). Information is added to the cache automatically by PHP each time it encounters a new file, and is then used by any number of functions the next time they attempt to look at that same file. With a web page that's gone in the blink of an eye where little may have happened on the filesystem, this is often a good trade-off for increased performance.

However the chances that the details of a file or path may change while your script runs obviously increase with the length of time that your script takes to execute. Therefore PHP gives us a couple of options for working with these two caches.

The following example shows the file status cache in action, and how to use `clearstatcache()` to clear it.

```
<?

# Create a file and add some text to it

$filename = 'test.txt';

$handle = fopen($filename, 'w+');

fwrite($handle, 'test');

# The following should print 4

echo stat($filename)["size"]."\n";

# Now write some data to the file, increasing the file size.

fwrite($handle, 'more test');

# Intuitively, the following command should print 13 as the file is now
# bigger than before. However it still prints 4, because the filesize
# value for this file is now cached.

echo stat($filename)["size"]."\n";

```

```
25 # If we clear the cache ....
26
27 clearstatcache();
28
29 # then the next line should print 13 as expected
30
31 echo stat($filename)["size"]."\n";
32
33 fclose($handle);
```

The realpath cache operates in a similar way, and can be cleared by calling `clearstatcache(true)`, i.e. by calling it with true as the first parameter. You can also clear the cache for just one particular file by calling `clearstatcache(true, 'myfile.txt')`, where the second parameter is the filename (and the first must be set to true, that is, you must also clear the realpath cache).

Of course clearing these caches may not be necessary in your application and doing so has performance implications. Consider each file access on a case-by-case basis.

8.6 Working with cross platform filesystems

While most filesystem related functions in PHP are platform "agnostic", you may find yourself writing your own or at the very least handling filenames and paths from assorted different platforms. The `pathogen` library is a useful library that takes away some of the pain of dealing with the different representations of filenames and paths across different platforms.

Pathogen

A general purpose library for working with file paths and schemas. Works with Unix and Windows path formats, URIs and more.

Main website : https://github.com/eloquent/pathogen

Main documentation & Installation info : https://github.com/eloquent/pathogen/#installation-and-documentation

8.7 Accessing the Windows Registry

The Windows Registry is a structured hierarchical database which Windows and other applications use to store configuration information. While not used universally by all applications, most

will store some information in the Registry, and the operating system itself uses it extensively. We can access the registry from PHP too, allowing us to check and set configuration information both for our own applications, and if we have the right permissions, other applications and the OS itself. To do this, we need to use the win32std extension with PHP.

win32std

Set of standard Windows API functions.

Main website : http://pecl.php.net/package/win32std

Main documentation & installation info : See README file within the download.

Alternative documentation : http://wildphp.free.fr/wiki/doku.php?id=win32std:index

Compiled Version download : http://downloads.php.net/pierre/

WARNING

Incorrectly changing or altering the Registry can cause a wide range of problems with individual applications, as well as system wide problems. In short, you can bork your machine if you do it wrong. Learn how to backup and restore the Registry BEFORE you start playing. Or play around in a disposable VM first before you use a production machine (and even then, backing up your production machine is still a good idea!).

Before you continue, you should understand the layout of the Registry. The Registry is divided into "keys" and "subkeys". Each subkey stores either a configuration value or futher subkeys, in a directory/tree like structure. Windows operates a permissions system based on users and keys/subkeys. Details of what information goes where and the specifics of the permission system are beyond the scope of this book.

Further Reading

"For Dummies" quick guide to the layout of the Registry
http://www.dummies.com/how-to/content/understand-how-the-windows-registry-works.html

A slightly more in-depth article on Wikipedia
http://en.wikipedia.org/wiki/Windows_Registry

Fairly comprehensive, official Registry documentation
http://msdn.microsoft.com/en-us/library/ms724871.aspx

Lets look at an example of reading a value from the Registry. We're going to get the e-mail address that is stored for use with the Firefox Crash Reporter software, which appears in the Registry at `HKEY_CURRENT_USER\Software\Mozilla\Firefox\Crash Reporter\Email`.

```
<?

# We first have to "open" a key before we can do anything with it.

$keyHandle = reg_open_key('HKEY_CURRENT_USER',
         'Software\Mozilla\Firefox\Crash Reporter');

if ($keyHandle) {

        $email = reg_get_value($keyHandle, 'Email');

        echo ("Crash Reporter Email  : $email\n");

        reg_close_key($keyHandle);

} else { die ("Couldn't open Registry key"); };
```

As well as accessing a subkey directly, you can loop through (or "enumerate") a particular subkey. For instance, lets check out what printers you have installed.

```
<?

$keyHandle = reg_open_key('HKEY_CURRENT_USER',
'Software\Microsoft\Windows NT\CurrentVersion\PrinterPorts');

if ($keyHandle) {

        $subkeys = reg_enum_key($keyHandle);

        foreach ($subkeys as $index => $subkey) {

                echo "Printer $index is $subkey \n";

        };

        reg_close_key($keyHandle);

} else { die ("Couldn't open Registry key"); };
```

Above we've used `reg_enum_key()` to get a list of the subkeys names. You could use `reg_enum_value()` in a similar way to get a list of the values for the subkeys instead.

To visually "browse" the Registry by hand, you can use the Windows application `regedit.exe`. This will give you a feel for where various subkeys are located. However be aware that many subkeys are either seemingly duplicated for the same app but in different areas (usually for per-machine vs. per-user settings), or hidden in completely unintuitive locations, like many of the operating system keys. The latter is usually due to either bad design or backwards-compatibility reasons. In any case, if you need specific information, consult the official developer documentation for the application or for Windows itself to determine exactly where that information is located (and what, exactly, it means).

Reading from the registry is a fairly benign activity, its hard to mess up your system by checking values (unless you act on bad or wrong information). Writing to the registry is much more dangerous, and the win32std extension makes it simple to do! We open the subkey with `reg_open_key()` as above, and them use `reg_set_value()` to write to it. If it doesn't already exist, it will be created for us. If it does already exist, the value will be updated.

```
1  <?
2
3  $keyHandle = reg_open_key('HKEY_CURRENT_USER',
4          'Software\My Php Software Co\My Software\login');
5
6  if ($keyHandle) {
7
8          reg_set_value($keyHandle, 'username', REG_SZ, 'rob');
9
10         reg_close_key($keyHandle);
11
12 } else { die ("Couldn't open Registry key"); };
```

The third parameter to `reg_set_value()` (in this case `REG_SZ`)is the data type of the value passed in the fourth variable. You can find all of the data types the Registry supports at http://en.wikipedia.org/wiki/Windows_Registry#Keys_and_values.

In summary, accessing the Registry in this way provides a powerful way to store software configurations and credentials, and allows wide-ranging access to Windows OS and application specific information. The downsides, aside from general criticisms of the Registry design and layout itself, are that it is platform dependent, and it is easy to damage your system if you aren't careful. Make those backups now, before you start playing.

8.8 Linux signals

On Unix and Linux systems, "signals" are a method that the Operating System (possibly at the behest of a user) can use to send, er, *signals* to a process. The OS interrupts the normal execution flow of the process to deliver the signal, allowing the process to act on the signal immediately (if it wishes to do so). There are a wide range of signals that can be passed, including requests to terminate, error condition signalling and polling notifications. Common signals have been

codified in the POSIX standards and a full list can be found on Wikipedia, and the list supported by PHP can be found in the PHP manual :

Further Reading

POSIX signals on Wikipedia
http://en.wikipedia.org/wiki/Unix_signal#POSIX_signals

PHP supported signals in the PHP manual
http://www.php.net/manual/en/pcntl.constants.php

We can listen and respond to these signals from our PHP scripts using the PCNTL extension. The following script demonstrates how to do this. It uses a PHP feature called "ticks", which allow a callback function be executed after every N statements. We don't need to do anything with ticks, we simply need to enable it (using a `declare` construct) so that it is available to the PCNTL functions to use as they deem fit.

```
1  <?
2
3  # Enable ticks every 1 `tick-able` statement
4
5  declare(ticks = 1);
6
7  # Declare a variable which will control whether our program keeps running.
8
9  $keepRunning = true;
10
11 # Output our PID (Process ID) so that we can use it in a kill command in
12 # another terminal to terminate the correct (this) script
13 # (e.g. ~$ kill 123456 )
14
15 echo("My PID is ".getmypid()." if you wanna kill me. I dare ya!\n");
16
17 # Now we create a function to handle signals when they come in.
18
19 function signalHandler($signal)
20 {
21
22         # $signal contains the signal that was received
23         # Always remember that this function can be called from any point in
24         # the script, so be careful if relying on the state of the program
25
26     switch ($signal) {
27
28         case SIGINT:
29
30                 # SIGINT is sent when a user wants to interrupt the process.
```

```
31                  # From the terminal this is usually by pressing Ctrl-C
32
33              echo ("No, you may NOT interrupt me. The cheek of it.\n");
34
35              break;
36
37          case SIGTERM:
38
39                  # Similar to SIGINT, SIGTERM is sent when a user requests
40                  # termination of the process, but is a slighter "stronger"
41                  # request.
42
43                  echo("Well, if you REALLY insist, lets go. Bye!\n");
44
45                  # set $keepRunning to false so that we exit at the start of
46                  # the next loop
47
48                  global $keepRunning;
49                  $keepRunning = false;
50
51                  break;
52      };
53  };
54
55  # Before we start the main body of our program, we need to tie the
56  # handler function we just created to the signals that we want it to
57  # handle.
58
59  pcntl_signal(SIGINT, 'signalHandler');
60  pcntl_signal(SIGTERM, 'signalHandler');
61
62  # Any signals received from now will be processed by the signalHandler
63  # function.
64
65  # Lets now enter a loop and do some work
66
67  while ($keepRunning) {
68
69          echo("Yawn, nothing happening...\n");
70          sleep(5);
71
72  };
73
74  # If we reached this point, then we must have exited the loop.
75  # That means that we must have received the SIGTERM signal as
76  # we haven't built any other means to exit the loop!
```

```
77
78 echo("That's it, you've stopped me. I hope you're happy.\n");
```

Run the script above and try pressing Ctrl-C. Normally, your PHP script would exit as PHP handles SIGINT by default by exiting the script. However we've handled it and refuse to close the script, so you will get a terse message and the script will continue! This is generally an impolite thing to do, as the user (or system) has requested to interrupt the process. If we want to be a good citizen, at this point we could choose to close or temporarily pause the process, or prompt the user (`We're still computing your Foobar value, are you sure you want to exit?`). If you still have some critical processing to do, it may be worth telling the user that you got their signal and will shut down in a moment, to save them trying other methods to kill the script.

Run the script again, take note of the PID shown (e.g. 123456), and in another terminal window type `kill 123456`. The `kill` command, by default, sends the SIGTERM signal to request the process terminates. In the first terminal window, you should see our PHP script "gracefully" exiting printing a couple of messages as it goes.

If you run the script once more, and this time use `kill -9 123456` to try and stop the script, you should see it stop dead in it's tracks without outputting any messages. This is because the `-9` flag tells `kill` to send the SIGKILL signal. We haven't handled this signal, because it is one of the few that we can't, and on POSIX systems it causes the process to terminate immediately. This is useful if a process isn't responding to any other signals and you need to end it. However it can be dangerous as the process doesn't have a chance to do any clean-up, such as closing files and releasing resources, and so can lead to errors like file corruption and resource leaks. If your system handles SIGTERM events, but may take a little while to clean up after itself, it is worth letting the user know that shut-down is in progress, otherwise they may resort to a SIGKILL thinking your process has just hung.

8.8.1 Sending Signals

It is possible to dispatch signals from your PHP scripts as well. This can be used to (crudely) communicate with other scripts, as well as to control processes if your script has sufficient permissions. To send a signal, simply use the misleading named `posix_kill()` function. This function actually lets you send *any* PHP supported signal, not just those used to kill a process. To use it, you need to know the PID (Process ID) of the process you want to talk to (e.g. 123456), and then you simply call `posix_kill(123456, SIGINT)` for example.

8.9 Linux timed-event signals

Sometimes we want our scripts to do something every-so-often, for example checking the status of a resource, do some clean-up, update a log file or similar. There are a couple of ways to achieve this. The simplest is by using the `sleep()` or `usleep()` functions to sit and wait for a number of seconds/microseconds before performing a task. This is not always of use, as when you call `sleep()` the script simply stops and waits for that amount of time rather than continuing to do other useful work. In the previous section we briefly looked at PHP "ticks", which allows us

to run a callback function every N (potentially useful) statements. This allows us to do useful work in-between calls to the callback function, however there is no guarantee on how long those statements will take to execute so we can't wait for a specific length of time. In fact, PHP has no internal way of keeping track of time in this way. However using POSIX signals, which we looked at in the previous section, we can ask the system to set an "alarm" for us a certain number of seconds into the future. When the "alarm" goes off, the system will interrupt PHP with a signal, which we can handle to run our callback function.

In the previous section we looked at how to listen for and process Linux signals, if you haven't read that section yet I recommend you do so now to fully understand the example that follows.

```
 <?

# We enable ticks for pcntl to use as before.

declare(ticks = 1);

$keepRunning = true;

# This is the function that we want to execute every 5 seconds.

function takeABreak($signal) {

   echo("\n\n ======== HAVING A BREAK, BE BACK SOON ========");

   sleep(3);

   echo("\n\n ======== BREAKS OVER, BACK TO WORK! ======== \n\n");

   # We need to request an alarm again each time.

   pcntl_alarm(5);

};

# This is a function to gracefully exit our program

function timeToGo($signal) {

        global $keepRunning;

        $keepRunning = false;

};

# In a moment we will ask the system to set an alarm, but before that
# we will register the callback function that will happen when the
```

```
37 # alarm goes off, i.e. when the system sends us a SIGALARM signal
38
39 pcntl_signal(SIGALRM, "takeABreak", true);
40
41 # Just to show that we can handle any and all signals with more than
42 # one callback function, we'll also register a different handler for
43 # the SIGINT/SIGTERM signals
44
45 pcntl_signal(SIGINT, 'timeToGo');
46 pcntl_signal(SIGTERM, 'timeToGo');
47
48 # Once we have got a callback function registered, we can go ahead and
49 # ask the system to set an alarm for us. The alarm only works once, so
50 # you'll notice this call to pcntl_alarm is repeated at the end of the
51 # callback function to set the alarm again
52
53 pcntl_alarm(5);
54
55 # Now we enter our main work loop
56
57 while ( $keepRunning ) {
58
59    echo('...doing work...');
60
61    usleep(50000);
62
63 };
64
65 # If we get here, then we've had the SIGINT or SIGTERM signals
66
67 echo("\nBye Bye, see you tomorrow.\n");
```

If you run the above script, you will see '...doing work...' printed to the screen every half a second. Every 5 seconds our alarm callback function will kick in and pause the work for 3 seconds, before the work resumes.

There are a couple of important concepts to understand when using alarm signals :

- The alarm callback function interrupts the normal script execution. That is, the normal script does not continue to run in the background while the alarm function is executing (it is not multi-threaded).
- The alarm function can kick in at any point in your script. If you modify the state of the script (e.g. setting global variables, using resources and connections), your script should be able to handle that new state at any point in it's execution
- You do not know how long your main script, or any part of it, will take to run in advance. This is affected by things like load on the system and resources, user interactions and so on.

Thus you should not set alarms that assume anything about the progress of the underlying script.
- Only one alarm can be set at a time. Any further calls to `pcntl_alarm()` will cancel the first alarm, although it will also return the number of seconds left on the previous alarm.
- You can cancel an alarm by calling `pcntl_alarm(0)`, which again returns the number of remaining seconds on that alarm.
- Any SIGALRM that is received will have the side-effect of exiting any system calls that happen to be in progress, such as calls to sleep(). Thus you should ensure that your script can tolerate such interruptions (or your alarm callback function should repair/reset any possible damage to the state of the script).
- If at any point you want your main script to wait until the next alarm occurs, you can call `pcntl_sigwaitinfo(array(SIGALRM))` which will pause execution until the alarm occurs.

8.10 Printing (to paper)

As much as we may wish to work in a paperless office, sooner or later we still have to artfully spill some toner onto a sheet of A4 or legal. PHP has a printing extension, but it is problematic as it is Windows only, it officially only supports versions up to Windows 2000 (although it generally works with versions up to Windows 7), and the required dll file no longer comes as standard with PHP. For those reasons we're not going to cover it in this book, however the PHP manual does cover it in detail if you're interested.

Further Reading

Printer extension in the PHP Manual
http://www.php.net/manual/en/book.printer.php

Instead, we're going to look at a different way to print from PHP that will generally work across all platforms with some tweaks. That way is to create and print PDF files from PHP.

We're going to use a free PDF library written in PHP call FPDF. No installation is necessary, you simply `require()` the `fpdf.php` file in your script and start using it. Once we've used it to create a PDF file, we then need to print it. The easiest way to do this is to open it with the systems default PDF viewer, which will act as a "print preview" allowing the user to then print it as they would a normal file. The alternative way is print it directly in the background without opening up a preview, which is done in a similar way.

Lets look at an example in Linux :

```
<?

# Load the fpdf.php library

require('fpdf.php');

# Create a new PDF object and start a new page

$pdf = new FPDF();

$pdf->AddPage();

# Add an image. Here we can use PHP's http wrapper to include a web image

$pdf->Image('http://static.php.net/www.php.net/images/php.gif',10);

# Now we'll set the font and add a header, plus some other text

$pdf->SetFont('Arial','B',24);

$pdf->Cell(0,30,'An Important Report','B',1,'C');

$pdf->SetFont('Arial',null,10);

$pdf->Cell(0,12,'Lots of really important text goes here. Etc.', null, 1);

# Generate a unique temporary file name

$filename = tempnam(sys_get_temp_dir(), "rep").'.pdf';

# Save the PDF to that temporary file

$pdf->Output($filename, 'F');

# Finally open the PDF for "print preview" in the default viewer

`xdg-open $filename`;
```

This will generate a PDF report and open it in your default PDF viewer for printing by the user. `open` and `gnome-open` also perform the same task as `xdg-open` on many Linux distros, which is to examine the file and open the appropriate registered viewer. If, instead, you would rather send the report directly to the printer yourself without displaying it to the user, simply replace the last line with

```
`lp $filename;`
```

The `lp` (or "line printer") command will print text, PDF or Postscript files. Unless you specify otherwise it sends it to the system's default printer. You can change which printer is used and other settings, `man lp` at the command line will give you all the options.

For Windows, the above example script will run with a change to the last line. To open the PDF with the default viewer, use

```
`start $filename;`
```

Printing in the background is dependent on the PDF viewer installed, as there is no equivalent of Linux's `lp` command. For Adobe Acrobat, you can use the following, with the path changed as appropriate :

```
`"C:\Program Files (x86)\Adobe\Reader 11.0\Reader\AcroRd32.exe" /t "$filename"`
```

This will open Acrobat, load the file, minimise the window and print it. The downside is that the minimised window is still visible and doesn't close after printing. There is also no control over which printer is used, it just uses the default printer.

For Foxit Reader, you can similarly do :

```
`"C:\Program Files (x86)\Foxit Software\Foxit Reader\Foxit Reader.exe" /t "$fi\
lename" "Oki C3800DN"`
```

This has a couple of advantages over the Adobe version. Foxit doesn't open a window, even minimised, and you can specify the printer to use in the second parameter.

Printing files other than PDF documents can usually be done in a similar manner, by calling an external print handler.

8.11 Audio

There are a number of useful libraries in PHP for dealing with Audio, from both making and recording sounds, to playback, effects and audio stream manipulation. Some of the more popular are listed below.

PHP-FFMpeg

PHP bindings for the FFmpeg Audio/Video recording, conversion and streaming library.

Main website : https://github.com/alchemy-fr/PHP-FFmpeg

Main documentation & Installation info :
https://github.com/alchemy-fr/PHP-FFmpeg#installation

OGG/Vorbis Extension

An extension to read and manipulate OGG/Voribis audio streams, including conversion to standard PCM audio.

Main website : http://pecl.php.net/package/oggvorbis

Main documentation & Installation info :
http://www.php.net/manual/en/book.oggvorbis.php

PHP MIDI

A class for reading, writing, analysing, modifying, creating, downloading and playing standard MIDI files.

Main website : http://valentin.dasdeck.com/midi/

Main documentation & Installation info :
http://valentin.dasdeck.com/midi/documentation.htm

Further Reading

On Linux, MPlayer can be controlled in "slave" mode by PHP writing commands to "named pipe file", to play and manipulate audio/video files.
http://stackoverflow.com/questions/4976276/is-it-possible-to-control-mplayer-from-another-program-easily

Introduction to HTML5 web audio. Useful for HTML5 based GUIs (see chapter 5)
http://www.html5rocks.com/en/tutorials/webaudio/intro/

8.12 Databases - no change here

We won't cover database access in this book, as by-and-large the processes for connecting to and querying databases are the same when using the PHP CLI SAPI as when using PHP on the web.

The only notes relating to databases refer to time-outs and disconnections, and are covered in the next chapter when we look at the stability of longer running PHP processes.

8.13 Other hardware and system interactions

We've covered a few of the common types of hardware and system interactions that you may come across, but there are of course many, many others out there. If you come across a piece of hardware that you need to control, or a system level process that you need to interact with, the first step is to check through the Pear, Pecl and Packagist repositories for relevant PHP libraries and extensions. If that fails, check your operating system's software repository for any helper applications or libraries that may be of use. Finally, try your favourite search engine. Although PHP is most common as a web tool, there are many thousands of developers out there that are already pushing the boundaries of what PHP can do (and generally blogging about it online!), so 9 times out of 10 you will find that someone else has already been there, done that and solved the problem for you. And if all else fails, try looking at how other developers have done it in other languages. PHP has many of the same functions as other languages including many system level calls, and so you may be able to work out a way to do it from translating their code into PHP.

9 Performance & stability - profiling and improving

Or, *"Why is PHP sooooo slow"* **(hint - it isn't, mostly)**

PHP script performance (a term which we use to encompass indicators such as speed of execution and usage of resources) is an issue for both PHP based web sites as well as other applications written in PHP. This chapter looks at the issues affecting PHP performance in general, specific performance considerations for non-web applications, and the tools and resources available for solving performance related problems. We also look at stability of long running scripts, which is often tied closely to performance issues.

9.1 The background on performance

PHP is generally considered a "scripting" or "interpreted" language. This means that rather than compiling or transforming source code into machine executable instructions and distributing those as a standalone program (as is usually the case with languages like C), PHP programs are distributed as PHP source code. The user of the software requires the PHP interpreter (also known as the PHP Virtual Machine) to run that source code, with the interpreter converting the PHP source code to machine instructions on the fly as the application executes. This style of execution has upsides and downsides. The main upsides are ease of code updating and deployment (no compilation steps) and fewer architecture issues (write-once-run-anywhere, no need to compile for specific platforms). The main downside is the performance hit of the interpretation stage as the application runs. Modern versions of PHP actually reduce this performance hit by first transforming the code into intermediate "opcodes" which are then executed. Those opcodes don't need to be re-interpreted every time the same code occurs (e.g. in a loop). For frequently used scripts, such as web scripts and some applications, these opcodes can even be cached between script runs for faster subsequent execution.

PHP is also a "high-level" language, which means that it uses abstractions to hide much of the lower-level detail of the code the computer needs from the user. Instead of dealing with implementation details like memory addressing and call stacks, PHP and other higher level languages present the computer to the programmer with abstract concepts such as variables, arrays and functions. Much of the appeal of a language like PHP is the wide range of in-built functions and operators to perform common tasks, which leads to higher developer productivity. Versatile data structures (like PHP's array type, which is actually a type of managed ordered map that can be used as a traditional array, a list, a hash table, a dictionary or several other common data structure types) hide details of memory management, simplify access, and provide a rich variety of related manipulation functions. However these abstractions comes at a performance cost. Behind every function, array operation, or disk access are lower level algorithms written in C. The implementation of these algorithms must cater to the "general" case and include all possible uses of the algorithm. Thus they will rarely be as optimal as C algorithms written

specifically for your particular need. This level of abstraction again adds an overhead to PHP when compared to lower-level languages like C, although the core PHP developers have invested a lot of time and effort in the last couple of years to optimise, trim, re-factor and otherwise increase the performance of the C back-end to great effect, wildly increasing speed and reducing memory consumption in many common cases. And for those common cases, these pre-built PHP algorithms are often better than those you may be able to write yourself!

The final reason for performance problems is perhaps one of the most prevalent issues but also one of the easiest to fix. And that reason is the PHP programmer. Many developers aren't aware of how to find and solve performance issues in their own code, often blaming their own failings on PHP itself even where PHP is just as performant as other languages. While PHP takes away a lot of the pain and slog from programming, hiding and dealing with numerous tasks for you, it is still a general purpose programming language and as such it is still perfectly possible to write poorly performing code. Even with an appreciation of the issues, many developers aren't aware of the tools and resources that are available to help them improve their programs performance.

For those that doubt that it is possible to write high performance systems in PHP, the set of slides in the Further Reading section below provides examples from one company that shows that it's not just possible, it happens in the real world. Appendix D also shows some of the things that people are doing with PHP, where performance clearly isn't holding them back.

Further Reading

"More Than Websites:PHP And The Firehose @ Datasift" by Stuart Herbert
http://blog.stuartherbert.com/php/2013/04/16/slides-from-brighton-php-more-than-websites-php-and-the-firehose/

9.2 Specific issues for general purpose programming

As noted above, both web and non-web PHP scripts can have performance problems. However there are some additional performance related issues to take into account when programming longer running scripts and those without memory restrictions. In the CLI SAPI, as with the traditional web PHP model of programming, PHP manages limits on memory consumption and executes garbage collection on your behalf. This works well on the web where the shorter lives & lower resource intensiveness of many scripts mean limits and management processes are rarely noticed. However when you program a general purpose application you will often want to remove the imposed memory limits to allow your application to consume all the memory it needs on different systems with different amounts of memory. Indeed, the default configuration for the CLI SAPI now sets the memory and execution time limits to 0, unless you specify otherwise. This transfers the burden of managing and limiting memory usage to your script. Likewise, in longer running, resource intensive and response-time-sensitive scripts garbage collection requires a different approach to avoid unwanted blocking or unnecessary conservancy. This involves the manual management of the garbage collection process within your scripts.

Inefficient programming and algorithm design on your part as the programmer can also be more noticeable than it is on the web. The time that an algorithm takes when generating a web

page can get lost in the other time overheads of transmitting and displaying a page, but when the exact same algorithm is executed on the command line a noticeable pause may be visible. Responsiveness to a user is critical on the web, but also very noticeable to local users as well.

The rest of this chapter looks at how to profile and manage the performance of your scripts and the resources they use, including strategies to target the problems described above.

9.3 Profile, profile, profile!

We've all come across slow running scripts (often our own!), and usually the first response is to start looking up ways to increase PHP's speed. Compiling, caching, re-factoring code, accelerators; these are topics that googling for PHP performance or speed issues will readily turn up, we may have already read about them, and then we dive right in to try them out. My advice, (derived from bitter personal experience), is to STOP RIGHT NOW. Throwing performance trick after performance trick against your code (often that you find online or in good books like these), even where they appear sensible and you can see the logic, can end up complicating your code or adding additional dependencies for no good reason. Why? because when we don't know what the root cause of the problem is, we don't know if a particular solution, no matter how good on paper, will actually addresses the issue we are having in this particular case. And even if it does appear to work, we don't know if it was the simplest way to fix it, and thus whether we're saddling ourselves with extra "technical debt" when we don't have to.

The step we often miss out is to ask our script directly : "why are you running so slowly?". If our script tells us we can then attempt to fix the issue directly without the use of external tools like compilers and caching systems. So how do we ask our script the "why" question? By "profiling" it.

A profiler, if you're not familiar with the term, watches a piece of software (usually from the "inside") as it runs and breaks down the time (and sometimes resources) that each part of the program uses. The profile information is often reported down to the level of an individual line of code or function call. This helps us spot exactly where our scripts are slowing down. Is it that complex database query? A badly written loop? A function that's called more times than expected? Disk or network access pausing execution? What ever the problem, the profiler will tell us. Once we know exactly what the cause of our slowdown is, the solution to the problem is usually apparent (or at the very least we can rule out potential solutions that won't actually fix it). It may just mean re-writing a few lines of code or caching some data instead of repeatedly generating it. It may point out problems external to PHP, such as a slow database server, or laggy network connection or resource. Of course in some cases it may end up being an intractable problem from a PHP programming point of view that does indeed require the help of an accelerator or external caching system. In any case, we will likely save time and prevent making unnecessary changes to our code or deployment environment by using a profiler to ask the "why" before we start trying the "what".

With PHP you have several choices when it comes to profiling. You can manually profile your code by adding profiling/measuring statements directly to your code base, or you can use one of a number of tools to automatically profile your code for you. The former is quick and easy to do with no changes to your development environment, if you know roughly where in your code the problem lies. The latter, whilst requiring the setup and configuration of the tools, and

learning how to use them the first time, provides more comprehensive profiling. It also doesn't rely on you knowing where your problems may be located, and usually requires minimal or no changes to your code base. We'll look at both options below.

9.4 Manual profiling

Manual profiling entails adding additional code to your source to measure time or resources directly from within the scripts. The following is an example of measuring execution time of different lines of code.

```
1  <?
2
3  # Lets fill some variables using loops
4
5  $something = $anotherthing = '';
6
7  # Lets create a "checkpoint" by recording the current time and memory
8  # usage
9
10 $time1 = microtime(true);
11 $memory1 = memory_get_usage();
12
13 # Now lets do a loop 10 times, having a quick usleep and adding just a
14 # little data to our variable each time
15
16 for ($counter = 0; $counter < 10; $counter++) {
17                 usleep(10);
18                 $something .= 'a';
19 };
20
21 # Now create a second checkpoint
22
23 $memory2 = memory_get_usage();
24 $time2 = microtime(true);
25
26 # Lets do this second loop 1000 times, having a longer sleep and adding
27 # lots of data to our variable each time
28
29 for ($counter = 0; $counter < 1000; $counter++) {
30                 usleep(100);
31                 $anotherthing .= str_repeat('abc',1000);
32 };
33
34 # and create a final checkpoint
35
```

```
36 $memory3 = memory_get_usage();
37 $time3 = microtime(true);
38
39 # Now lets output the time and memory each loop used.
40
41 echo ("1st Loop : ".($time2-$time1)." msecs, ".
42                 ($memory2-$memory1)." bytes\n");
43
44 echo ("2nd Loop : ".($time3-$time2)." msecs, ".
45                 ($memory3-$memory2)." bytes\n");
46
47 echo ("Peak memory usage : ". memory_get_peak_usage()." bytes\n");
```

Run the script above and you should see that the second loop took a lot longer and consumed a lot more memory. You now know that our "problem" is the second loop, and you can "fix" it by removing the usleep statement, and maybe removing the loop altogether and using `str_repeat('abc',1000000)` to fill your variable. You should also see that the peak memory usage reported on the last line is higher than that of the two loops. This demonstrates that without active management of variables (e.g. `unset()`ting them), memory usage accumulates throughout a script. This is obviously a simple, contrived example, but the principles used apply to real-world code as well.

As you can see manual profiling is quick and simple for a few lines of code. However profiling of larger code bases can quickly become cumbersome, massively increasing the size of the code base if you're not careful. When hunting down a particular problem, you can profile larger sections of code, and when the larger section at fault is found profile that into smaller chunks and so on until the problem code is found (effectively doing a binary search or similar). If you're spending a lot of time doing it this way, the time necessary to implement and learn a profiling tool like those detailed below would probably be time well spent instead.

One other thing to remember when manually profiling is that manual profiling code adds an additional performance penalty to your scripts, which although usually small can add up, especially if you are repeated logging profiling information to disk or such as you go. So it may be worth considering stripping out or disabling profiling code (perhaps as part of your build/deployment process) before the code hits production. There are cases, of course, where consciously adding profiling code to the production code base can be helpful, e.g. when it is necessary/useful to collect profiling information from your end users who aren't likely to have dedicated profiling software installed. Automatic profiling tools also usually add some overhead, although it is often smaller (they are typically written in lower level langauges and often integrate directly with the PHP interpreter), and are usually easier to switch on and off. They are often only used in the development environment and not on live production machines so any overhead is restricted to dev work.

9.5 Profiling tools

There are a several profiling tools available for PHP. While the xDebug debugger provides some profiling options (and is worth looking at if you already have it installed for debugging), the most

common and comprehensive tool is XHProf. Originally developed by Facebook, it is available as PECL extension and so can be simply and easily installed. The data collection side is written in C, and a graphical PHP interface is provided for viewing the collected profile data, including call graphs (visual graphs of which functions call which) if you have Graphviz installed . A related project, XHGui, which is a fork of XHProf, provides an expanded visual interface, stores multiple runs in a MySQL database and provides access for sorting and comparing multiple runs. XHGui is a little more work to get installed and configured, and is particularly geared towards web scripts, but provides a lot of flexibility if you are regularly profiling code, or profiling live code on production systems. For basic profiling to find obvious problems in development code though, XHProf is a good place to start.

Another up-and-coming profiler is php-profiler, a profiler written in pure PHP. Less comprehensive than XHProf, it is never-the-less useful and easy to deploy - simply `include()` profiler.php into your code. Note that as it is designed with the web in mind, the output from this profiler is attached to the web page it has profiled by default. If using it with non-web scripts, you will need to use output buffering or similar to redirect the output to (for example) a separate HTML file on disk for later viewing in a browser.

Further Reading

"The Need for Speed: Profiling PHP with XHProf and XHGui" by Matt Turland
http://phpmaster.com/the-need-for-speed-profiling-with-xhprof-and-xhgui/

"XHProf PHP Profiling" by Adam Culp
http://www.geekyboy.com/archives/718

"Profiling with XHGui" by Paul Reinheimer
http://webadvent.org/2010/profiling-with-xhgui-by-paul-reinheimer

"Profiling PHP Applications with XHGui" by Lorna Mitchell
http://techportal.inviqa.com/2013/10/01/profiling-php-applications-with-xhgui/

"Advanced CodeIgniter Profiling With XHProf" by James Constable
http://net.tutsplus.com/tutorials/php/advanced-codeigniter-profiling-with-xhprof/

XHProf in the PHP Manual
http://www.php.net/manual/en/book.xhprof.php

XHProf

Function level hierarchical PHP profiler

Main website : https://github.com/facebook/xhprof

Installation info : http://www.php.net/manual/en/xhprof.setup.php

Main documentation : http://www.php.net/xhprof

Tool for visual function graphs : http://graphviz.org/

XHGui

Expanded profiler based on XHProf

Main website : https://github.com/preinheimer/xhprof

Installation info & main documentation :
https://github.com/preinheimer/xhprof/blob/master/INSTALL

php-profiler

An embedded PHP profiler library

Main website : https://github.com/jimrubenstein/php-profiler

Installation info & main documentation :
https://github.com/jimrubenstein/php-profiler#php-profiler

PHP-benchmark

A profiling library like php-profiler, with an emphasis on profiling/comparing individual algorithms.

Main website : http://victorjonsson.github.io/PHP-Benchmark/

Installation info & main documentation :
https://github.com/victorjonsson/PHP-Benchmark#benchmarking

Kcachegrind

A tool for profile-data visualization. Use with Xdebug for visual profile information

Main website : http://kcachegrind.sourceforge.net/html/Home.html

Installation info : http://kcachegrind.sourceforge.net/html/Installation.html

Main documentation :
http://kcachegrind.sourceforge.net/html/Documentation.html

Webgrind

An alternative web based profiling front-end for Xdebug, implements a subset of Kcachegrind's features.

Main website : https://github.com/jokkedk/webgrind

Installation info & main documentation :
https://github.com/jokkedk/webgrind#webgrind

9.6 Low level profiling

When you really need to "go deep" into what's happening with your script, you sometimes need to look not at what your code is doing, but at what PHP itself is doing instead. To be clear, most of us will never need to do this to solve performance problems, though it can be quite interesting and instructive to look at how PHP translates your code into calls to the system on which it's running. PHP itself is a C program compiled into a binary executable, which means you can use general purpose tools like strace (shows system calls and signals), ltrace (shows library calls) and gdb (a debugger for C programs like PHP itself) to see what's going on under the hood. If this interests you, have a look at the tutorial below from Derick Rethans. As the author of Xdebug, he's somewhat of an expert on the mechanics of PHP.

Further Reading

"What is PHP doing?" by Derick Rethans
http://derickrethans.nl/what-is-php-doing.html

9.7 Profiling - the likely results

Profiling can reveal an inordinate number of different performance problems. However, the following are some of most common types of problems discovered when profiling PHP, along with some strategies for addressing them :

- Database access : Calls to databases (and indeed any other external service) can often be a source of slowness. What is one line in your code may actually be calling many thousands of lines of code in the responding software/service. Database performance is the topic of a whole book or two of its own, but areas to look at for improving database performance include : database indexes & normalisation, server load, database cache settings, and inappropriate SQL usage. You can side-step some database issues directly in PHP by thinking about whether you actually need to call the database at all. Can you, for instance, cache some oft-requested data? Can you batch multiple queries together in one request? Can you request more data in one query rather than over multiple queries? Do you really need the data in the first place?
- Disk access : Particularly for large files, and files from network drives, disk access can be time-expensive. Possible work arounds include caching data in PHP (or in Memcached or a similar extension), reading in a whole file at once (at the expense of memory) rather than repeated opening of the same file, using local disks (or caching to them before use) rather than network drives, using better hardware (e.g. SSDs), or using memory based drives (e.g. tmpfs). Beware that modern operating systems are usually quite good at optimising disk access and perform some memory caching, particularly for disk writes, and for reads of often-accessed files, and so some optimisations may not give the expected level of "speed-up". Always profile your proposed solutions!

- Loops : Repeated calculations and other operations inside loops can quickly eat up time in what appears to be a low number of lines. Where the same calculation (or partial calculation) is repeated in the body or control structure of a loop, move it outside of the loop and store the result in a variable to be used in the loop. Look at functions called from within the loop as well to see if anything in the function can be cached, removed or otherwise sped up. Often functions assume that they will only be called once or twice, and so using things like "static" variables to preserve state between calls can sometimes speed up a repeatedly called function.
- No clear cause and/or solution : Occasionally there is no clear single cause for the slowness, everything is a bit slow and it all adds up. Likewise, there may be a clear cause, but no clear solution as your script is doing something that is naturally time-expensive and it is the only way to achieve the outcome you need.

In this last case, it may be time to start looking for some "silver bullets" - generic solutions for increasing the speed of your script.

Further Reading

An article benchmarking "Fast line iteration in PHP" by Stephan Soller. Of particular interest is the comment section at the bottom, detailing why disk caching had a big effect on the authors numbers but not another commenter's.
http://arkanis.de/weblog/2013-09-27-fast-line-iteration-in-php

9.8 Silver bullets

Silver bullets are "solutions" that you can "throw" at your script, which will hopefully speed them up without you having to think too much about the cause of the slowdown. In the first version of this section that I wrote, the words "silver bullet" were always written with a question mark after them. I took it out as it cluttered the place up, but the point was that there are no guaranteed, universal silver bullets for increasing performance in PHP. Each of the oft-called "performance solutions" below have downsides, aren't suitable for everyone, and require a reasonable amount of thought to implement. Nevertheless they can be useful, particularly when you have improved your script as much as you can manually and you are stretching the capabilities of PHP or the available hardware.

9.9 Silver bullet #1 - Better hardware

If funding allows, sending out for a beefier server or desktop can often produce instant performance gratification. In some cases hardware is cheaper than developer time, so it's a no-brainer. However, there are some downsides, particularly for those who are cash-challenged:

- You still need to do some profiling to work out exactly *what* new hardware you need. Is your script processor-intensive? Does it guzzle RAM? Is it spending a lot of time hitting

the disks? Or do you need all of the above? Is it in-fact hitting problems with external access like network or database access? If that's the problem, new hardware likely won't help.
- Costs incurred include both purchase/lease costs, as well as potentially increased power consumption costs, maintenance costs and depreciation/replacement costs.
- With many algorithms, increased hardware capacity will eventually hit natural barriers where script performance doesn't scale linearly with increases in hardware performance.

9.10 Silver bullet #2 - Newer PHP versions

Each new version of PHP typically increases performance and reduces resource usage in many areas, so moving up to a new version of PHP can quite often give your scripts more headroom. For instance, benchmarks between 5.3 and 5.4 often show a performance increase of between 7% and 31% on common scripts, and memory usage improvement of between 17% and 50%. There are other benefits to, such as bug fixes and new features. However there are also some downsides:

- While generally backwards-compatible, most releases mean that a handful of functions are depreciated or fully removed, and the functionality of others is altered, sometimes subtly. Because of this, you need to thoroughly test your code when upgrading, and may need to make changes to your codebase.
- Some extensions (particularly older, less maintained or more obscure ones) that you use may not be compatible with the new version. This would require you to alter your codebase or that of the extension.

9.11 Silver bullet #3 - Opcode caching

Opcode caches work by caching the executable opcodes created by the PHP binary when a PHP script is run, and (assuming the PHP script remains unchanged) using the same opcodes the next time the script is called rather then re-generating them each time. This provides a performance increase mainly at the start-up of a script. Web scripts are short scripts called repeatedly, often many times a second on busy sites. In such cases opcode caches can provide a slightly faster experience for the individual user and a larger resource saving for the server as a whole. Opcode caches such as the PHP APC cache are usually easy to install and configure, and so are in most cases a no-brainer for web devs and server admins, providing the silver bullet of increased performance for not much work.

However they aren't always as effective for off-line applications. Such applications are typically longer lived and so the relative gain in startup performance is often much much less than with short lived web scripts. For this reason the main cache systems for PHP aren't written to deal with the CLI-SAPI, and only really run under the web server model where many requests are served by the same PHP process. You can enable the APC cache for use with the CLI-SAPI by setting `apc.enable_cli` in php.ini, however this is meant for testing purposes, and will usually not provide any benefit for your script as the cache is destroyed at the end of each process run. For this reason we won't cover opcode caches here, but further reading is given below if you are running on a web-style model.

Further Reading

List of Opcode caches and related software on Wikipedia
http://en.wikipedia.org/wiki/List_of_PHP_accelerators

"Using PHP 5.5s New OPcache" by Chris Jones
https://blogs.oracle.com/opal/entry/using_php_5_5_s

9.12 Silver bullet #4 - Compiling

PHP compilers have existed for a while, typically taking PHP source code and transforming it to an intermediate language (such as C++) before using a commonly available compiler for that language to compile it to a machine executable file. These compiled files typically execute faster, and with a lower resource overhead, than standard interpreted code. Previous incarnations have had limited success, often suffering from lacking coverage of all PHP syntax and functions, incompatibilities with common PHP extensions, and prohibitive licensing restrictions. The lack of decent compilation tools serves as a hat-tip to the notion that PHP is, in fact, perfectly performant for most use cases and so a strong market for them does not exist. One company with an understandable need for higher performance from PHP has started to change the game however. Facebook, the popular social networking company, has introduced and developed the PHP HipHop compiler project over the past couple of years.

Much of the Facebook interface was originally written in PHP, and although various subsystems have since been written in lower-level languages, much of Facebook's legacy (and new) interface code remains coded in PHP. Facing enormous scaling and performance challenges with a user base approaching 1 billion (a nice problem to have), but still recognising the benefits of developing in PHP and with a large existing codebase to support, Facebook has invested considerable resources into the HipHop project. The first fruits of the project, which have been released as open source, were the HPHPc compiler and HPHPi "developer mode interactive" version of HPHPc. Both are reliable, production ready tools with great features. Although, as with all programming, specific performance gains will depend on the ins and outs of what your PHP script is trying to do, various benchmarks have pegged HipHop compiled binaries with performance increases of up to 600% in real-world usage. HPHPc works in a similar way to other PHP compilers - it first converts your code to C++, and then uses G++ to compile that C++ down to an executable binary.

Whichever compiler you look at using, and several more are listed later, there are a number of downsides to consider:

- New tool chain & deployment process - None of the available compilers are drop-in replacements for the standard PHP interpreter, so you will need to invest time in learning to use the compiler and updating/changing your build process.
- Involved compiler deployment - PHP compilers are typically not included in software repositories like the standard PHP binaries, so deployment becomes more involved and often requires compiling the compilers themselves on each development/build machine.

- Compiling code takes time, particularly in large projects - Each build you make will be slowed down, although some of this is mitigated with the HPHPi tool in the the Facebook suite of programs.
- Lack of function coverage - all currently available compilers, including the HPHP suite, do not cover all of the core PHP functions, let alone many of the available extensions. Support for dynamically accessing other extensions varies by compiler and is often patchy or non-existent. Thus, you need to ensure that your code is supported by your chosen compiler, or you may need to re-write or remove sections of your code/functionality.
- Your users won't have access to the source code by default. For proprietary software this may be a plus. As a consequence if you use other open-source code within your project you may (depending on the license requirements) need to make arrangements to provide the source code to your users or bundle it with the executable. It also may dissuade knowledgeable users from helping you to develop your code and identify existing and potential bugs (and their fixes).

Because of (some of) these downsides, the Facebook project has (since early 2013) depreciated the HipHop compilers in favour of a JIT based Virtual Machine, discussed in the next section. Other available compilers are also now either defunct or haven't been updated to support recent PHP versions, however they may be of some use for certain projects so we have listed the main ones below.

Further Reading

HPHPc benchmarks for Drupal
http://php.webtutor.pl/en/2011/05/17/drupal-hiphop-for-php-vs-apc-benchmark/

An instructive rant by the developer of the phc compiler about PHP compilers
http://blog.paulbiggar.com/archive/a-rant-about-php-compilers-in-general-and-hiphop-in-particular/

phc

An open source compiler for PHP with support for plugins. Supports PHPv5.2 and earlier.

Main website : http://phpcompiler.org/

Installation info & main documentation :
http://phpcompiler.org/documentation.html

bcompiler

Pecl extension to compile PHP to bytecode and optionally into executables.

Main website : http://pecl.php.net/package/bcompiler

Installation info & main documentation : http://www.php.net/bcompiler

Roadsend PHP

No longer actively developed - An open-source compiler, source code still available.

Main website : http://www.roadsend.com/

Installation info & main documentation : https://github.com/weyrick/roadsend-php

9.13 Silver bullet #5 - JIT compilers and alternative Virtual Machines

Before we discuss them in detail, I'll just define a couple of terms that we're about to use, in-case you're not familiar with them.

- Virtual Machine, or VM. This is general used to mean what, in the old days, we would call the interpreter. Specifically we are talking about what is more formally called a "Process Virtual Machine" here, rather than a "System Virtual Machine" which is typically a virtualised Operating System running on something like VirtualBox. When you write PHP code (or code in many other programming languages like Python or Java), rather than being compiled down into machine executable instructions (like C is, for instance), it's first compiled into an intermediate set of instructions which are then executed by a virtual machine or interpreter. The VM is typically written in C and compiled down into machine executable code itself. This means that code written in PHP is effectively platform independent, that is, it doesn't need to know anything about the instruction set usable on

a specific platform. The VM worries about that (which is why you will find individual PHP executables for Windows, Linux, Mac and so on to download, but most PHP scripts themselves don't require different scripts for different platforms).

Further Reading

Process Virtual Machines on Wikipedia
http://en.wikipedia.org/wiki/Virtual_machine#Process_virtual_machines

- JIT compiler : JIT stands for "Just In Time". Rather than code being compiled completely in advance (like C), or at the start of each script run (like traditional PHP), JIT compilers compile code function by function as the code runs, in "real time". The advantages of doing this are that a) only the code that is actually executed needs to be compiled, and b) the compiler can use run-time information such as current variable types (which are not defined until they are run in dynamically-typed languages like PHP) to produce more optimised code. In short, they're faster than traditional interpreters.

Further Reading

JIT Compilation on Wikipedia
http://en.wikipedia.org/wiki/Just-in-time_compilation

There are a number of alternative VMs now available for PHP, with the most recent making some waves in the PHP community. HHVM is the latest output from the Facebook HipHop project (discussed in the compiler section above), and is hitting the news because of it's built in JIT compiler. It's certainly one to watch, as it's currently more than 40% faster than the HPHPc compiler it replaces in some benchmarks, it fits more naturally in most production workflows, and has greater compatibility with PHP & its extensions.

There are other VMs and JITs available, and these are listed below. All of them, including HHVM, however, share a few drawbacks.

- Limited coverage of PHP and extensions - None of them yet have full coverage of PHP and common extensions. The HHVM team have recently stated that their goal over the next 12 months is to reach parity with the top 20 PHP frameworks and applications, but they're not there yet. Other VMs like PHP-QB explicitly state that they are focused on a particular subset of PHP, and so aren't a general purpose solution. Thus you will need to thoroughly test (and possibly alter) your code when considering using an alternative VM.
- Not all alternate VMs are faster for all tasks - The JIT in HHVM for instance requires multiple runs of CLI scripts before it learns enough to produce optimised code. As Sara Golemon wrote on the "PHP internals" mailing list :

> *"As the maintainer of the OSS version of HipHop (HHVM), I'll be the first to admit that the official PHP engine and runtime have a broader range of platform/architecture support, and stronger community, and a larger library of extensions and functionality behind it. Also, because of it's lifecycle design, PHP outperforms HHVM on single-run command-line scripts.*
>
> *On the other hand, HHVM does outperform Apache+PHP for web requests quite well. Facebook would need at least 5x as many web-servers (and we use a lot as it is) if we were using normal PHP. That's not to say PHP isn't fast, but interpreted bytecode using loose typing will never keep pace with native machine code and strict typing."*
> http://marc.info/?l=php-internals&m=135399266632008&w=2

So you'll need to benchmark each possible VM against your typical use case, to make sure you are really getting the speed-up you require.

HHVM

Facebooks HipHop project VM. Includes a JIT compiler, and has fairly broad language coverage

Main website : https://www.facebook.com/hphp

Installation info & main documentation :
https://github.com/facebook/hiphop-php

Official Blog : http://www.hhvm.com/blog/

Other Information

Presentation on building HHVM by a Facebook engineer
http://www.infoq.com/presentations/PHP-HHVM-Facebook

"First steps on HHVM" tutorial by Allan MacGregor
http://coderoncode.com/2013/07/27/first-steps-on-hhvm.html

Official blog post on the future of HHVM
http://www.hhvm.com/blog/?p=875

Q&A about HHVM performance in CLI scripts
http://stackoverflow.com/questions/17898783/hhvm-poor-performance

Quercus

100% Java implementation of PHP. Supports a growing range of extensions and common PHP applications, claims a 4x performance boost and an api to allow PHP to access Java libraries.

Main website : http://quercus.caucho.com/

Installation info & main documentation :
http://quercus.caucho.com/quercus-3.1/doc/quercus.xtp

PHP-QB

A VM aimed at using PHP for graphics programming. Achieves impressive performance improvements but implements static typing and has array limitations. Can compile to a executable with G++ or other external compiler.

Main website : http://www.php-qb.net/

Installation info & main documentation :
https://github.com/chung-leong/qb/wiki

Roadsend PHP: Raven (rphp)

A rewrite of the Roadsend compiler using the LLVM VM framework and a C++ runtime. Includes a JIT compiler, still currently under development.

Main website : https://github.com/weyrick/roadsend-php-raven

Installation info & main documentation :
https://github.com/weyrick/roadsend-php-raven#readme

Parrot

A polyglot VM designed for optimal & efficient execution of dynamic languages like PHP, Perl, Python etc. The PHP specific project is PIPP.

Main website : http://www.parrot.org/

PHP main site : https://github.com/bschmalhofer/pipp/wiki

Installation info & main documentation :
http://docs.parrot.org/parrot/latest/html/

Further Reading

"HappyJIT - a tracing JIT compiler for PHP" - An academic paper exploring a tracing JIT engine for PHP
http://dl.acm.org/citation.cfm?id=2047854&dl=ACM&coll=DL

"HippyVM" - a Facebook sponsored project looking at how to use the Python PyPy toolchain to produce PHP VM.
http://morepypy.blogspot.co.uk/2012/07/hello-everyone.html

A very basic, proof of concept, PHP VM written in Javascript
http://phpjs.hertzen.com/

Presentation on "Building a JIT compiler for PHP in 2 days"
http://llvm.org/devmtg/2008-08/Lopes_PHP-JIT-InTwoDays.pdf

9.14 The SPL - Standard PHP Library

At the very beginning of this chapter, we discussed the fact that some of the apparent performance problems PHP has are down to the overhead necessary to provide users with easy to use and versatile data structures and functions. If you find these overheads start limiting your scripts, one port of call is the SPL, a core PHP extension that contains common & esoteric data structures and functions. These are designed to solve common programming problems, albeit with a little more thought needed to use than PHPs more common tools.

So for instance, if you find that large arrays of data are causing your script to hit memory limits, you might like to look at the `SplFixedArray` class. It has some restrictions (you can only use integers as indexes, the length of the array is fixed) but provides a faster implementation that uses less memory. The PHP documentation is fairly comprehensive. If you're not familiar with some of data structures (heaps, linked lists etc.) then most basic introductions to Computer Science (or programming in more traditional languages) should help you out.

Further Reading

PHP SPL documentation
http://www.php.net/manual/en/book.spl.php

Presentation "Intro to the SPL". A quick overview of the SPL.
http://www.slideshare.net/ctankersley/intro-to-the-php-spl

9.15 Garbage collection

In every programming language, creating variables, arrays and other resources uses up some of the memory on your PC. While that memory is in use, it isn't available for your software or other software to use. When you have no further use for that memory, it needs to be freed up so that it can be used again. Where memory isn't freed up (usually accidentally), we get what is called a Memory Leak, where the available memory is gradually (or in some cases, quickly!) filled up by unused data, or "garbage". In lower level languages, like C, programmers are responsible for memory management themselves, for instance assigning memory for a variable and freeing it again when finished with. In higher level languages like PHP, memory is managed for you and unused memory is returned to the available pile automatically.

In PHP, there are several mechanisms for this. The most basic is that when a program ends, all of the memory allocated is freed. Secondly, memory for resources that are created within a particular scope is automatically freed when we exit that scope as the resources are unset. For instance, variables declared within a function (rather than the global scope) will disappear when the function ends (unless we use "static" or similar) and the memory will be freed.

Finally, PHP keeps track of values by "Reference Counting". When you create a variable or an array, for instance, PHP will create a container in memory to hold the value of that variable, and start a counter to keep a count of how many things are using that value. When you refer to that variable, for instance by assigning it to another variable, PHP will increment the counter so that it knows who is using it. Once no one is using it, and the counter drops to 0, PHP knows it can safely remove that container and free up the memory.

However reference counting isn't perfect, in particular it has problems where circular references are created (a child with a reference to its parent, and vice versa for example) where reference may still exist even when the resource is no longer in use. In such cases, they aren't automatically freed and so can create a memory leak. To cope with this, PHP has Automatic Garbage Collection, also called Cycle Collection. Once in a while, PHP will go though everything it knows about looking for unused memory like this that hasn't been freed, and will then free it. The current algorithm for doing this, which is much more effective than previous ones, is available from PHP v5.3 onwards. A fuller description of reference counting and cycle collection can be found in the PHP manual :

Further Reading

Garbage Collection in the PHP manual
http://www.php.net/manual/en/features.gc.php

In theory this is great, it prevents memory leaks and prevents the programmer having to do too much work managing memory. However in some applications it has a downside. PHP keeps a "buffer" of the currently created values, and kicks in Garbage Collection each time the size of this buffer reaches 10,000. If your program is doing something particularly time sensitive at that point then tough, it will have to wait until garbage collection is done. This wasn't usually a problem with short lived web scripts, but definitely can be with longer running CLI based scripts which may frequently and repeatedly hit the 10,000 limit and kick off a collection cycle. That said, for many applications the slowdown may not be noticeable or a problem, and the advantages of the automatic memory management may outweigh the disadvantages.

PHP will let us manage garbage collection ourselves if we need to, turning it on and off as we require. As with any other performance issues, the benefit to your particular script is very hard to estimate in advance, so profiling/benchmarking your software (or the higher performance parts of it) with and without garbage collection disabled is advised.

To turn it off, simply call the `gc_disable()` function. When you are ready to turn it back on, use the `gc_enable()` function. If you want to manually run a garbage collection cycle at a time to suit you, use the `gc_collect_cycles()` function which will initiate a one-off collection of garbage.

Caution : To enable proper collection when you call `gc_collect_cycles()` or when you re-enable automatic collection with `gc_enable()`, PHP continues to keep and update its buffer of up-to 10,000 values in the background. This buffer is only of a fixed size, and so if you leave garbage collection off for too long, older values will get pushed out of the buffer. Some of these values may be the problematic variables that garbage collection is designed to free up, and so you may unwittingly introduce a memory leak even if you re-enable garbage collection later. Thus use these functions with caution.

9.16 Multi-threading and concurrent programming in PHP

Multi-threading is the use of concurrent programming "threads" within a process, allowing a given program to do multiple things at once. Multi-threading is commonly used to increase performance in processes with tasks that are suitable for concurrent processing (i.e. where one step of the process isn't reliant on the completion of the other). PHP was not designed with multi-threading capabilities, or facilities to otherwise natively do concurrent processing or take advantage of multiple processors/cores. In the past this wasn't a particular issue, web development rarely calls for the performance gains associated with multi-threading and a web server can spin up multiple PHP instances to prevent interface blocking in AJAX type applications. Typical CLI scripts also didn't require it, often being smaller scale or batch type applications. However as larger and larger application are developed in PHP, and web server admins are looking to get greater resource utilisation from their existing hardware, and the lack of multi-threading or other methods of parallel processing is beginning to become a more pertinent issue within the PHP community. Indeed, it is already a great "problem" that PHP's detractors often try and latch onto. As Moore's law begins to falter (depending on who you talk to), performance gains are typically coming instead from multi-core and multi-processor architectures where multi-threading and parallel programming will be more and more important.

The topic of multi-threading is a contentious one within the PHP community, with some people believing that it is needed for PHP to be taken "seriously" as a general purpose programming language, whilst others believe that it is of little use in PHPs core area of web interfaces and that it would add additional complexity to the language which, as a rule, has tried to stay as simple and straightforward as possible. It is true that even if threading was added to PHP with the simple, easy to grasp, interfaces that PHP is well like for, that is only half the story. Programming with threads, regardless of syntax, is a somewhat advanced discipline that delves deeper into the intricacies of code interdependence and structure than PHP programmers usually have to deal with. That the best way to take advantage of multiple processors and cores is through multi-threading is not a settled argument in any case. The article below from IBM outlines some of thoughts currently in play around multi-threading vs. multi-process programming, and in this particular article comes down on the multi-process side of the fence.

Further Reading

"Why thread-based application parallelism is trumped in the multicore era" by Vasudevan Thiyagarajan
http://www.ibm.com/developerworks/java/library/j-nothreads/index.html?ca=drs-

There pro's and cons to both types of concurrency. Threading has fewer overheads and easier communication, for instance, whilst multi-process programming has easier and greater isolation of tasks and minimal shared state. Regardless of the arguments, PHP doesn't support threads and they aren't currently on the roadmap for the 5.x series of releases. If threads do make an appearance it will likely be as part of a major language overhaul in PHP 6 or 7. With the development community currently lacking an appetite for such overhauls (plans for PHP6 have been discussed for several years and pretty much abandoned), such developments are likely several years away at best. Even then, implementing threading in a successful manner isn't a given. Python programmers often complain about the way threads are implemented and their performance characteristics (or lack thereof) in their own language, causing some to default to multi-process programming, even though threads are available. So for the moment, if we want to do concurrent or parallel processing we need to do it without threads. Luckily there are several ways to work around this in and around PHP, with the best way depending on what you need to achieve.

The most common way is by creating and controlling new PHP processes. With this method, when your PHP script needs to carry out concurrent programming it can fire up multiple copies of another PHP script as separate processes (or indeed fork itself multiple times as new processes), pass data to the other scripts, and wait for them to finish running. See Chapters 6 & 7 for details on forking and inter-process communication. Multi-process programming is in some ways easier than multi-threaded programming, you usually don't have to worry about shared state (for instance accidentally overwriting variables/data/memory in use by other threads). However you do still need to think about how to handle interactions between the processes, how to deal with failures in another script (e.g. if the user, the system or an error kills one of your worker scripts, what do you do?), and how to manage/monitor system resources.

An alternative to managing process control yourself is to use a task dispatch & management system to manage it for you. Systems like Gearman (discussed in Chapter 6) will take your tasks

and allocate them to worker scripts, managing the process from starting the scripts to passing task information and monitoring scripts and system resources. There can be a few downsides to such systems in some cases though. You'll need to deploy and support additional software (i.e. the Gearman software) to your users, and the options for timing, error handling and resource management might not exactly meet the needs of your scripts. However for many cases it removes a lot of the programming overhead that multi-process software requires and hence helps speed development (as well as avoiding the re-invention of wheels).

If your performance problems come from processing very large amounts of data, a task typically improved by threading or concurrent processing, see the section on "Big data and PHP" below for a tool that may be able to help.

9.17 Big data and PHP - MapReduce

In 2004 Google opened a window into their world of "big data" processing by releasing a seminal paper on MapReduce, the programming model that they used to handle the web-scale data needed for their search engine. Although reports suggest that Google have now moved on from MapReduce, it is still currently considered state of the art for the rest of us, and a popular open source implementation, called Hadoop, will allow you to run MapReduce with PHP scripts.

Before we continue, lets clarify what we mean by *big* data. Hadoop and other MapReduce algorithms have an inherent overhead in the way they work. This means that they only start producing benefits (at least performance wise) the larger the dataset gets. Your mileage will vary depending on the algorithms you want to apply to your data, and the alternatives methods available, but typically your data set needs to be well into the Gb range before you will start seeing noticeable benefits, and significantly bigger before these may outweigh the engineering effort to implement MapReduce on your task.

If your data falls into this particular definition of "big data" then you can harness the power of Hadoop with PHP. While Hadoop is written in Java, it is language agnostic when writing the map and reduce jobs that drive the data processing. It also uses standard streams (STDIN and friends) for data interchange, which we've seen in earlier chapters are easy to handle natively in PHP. There is even a PHP framework to aid in writing those map and reduce jobs.

The following links should start you along the road of big data processing in PHP if you think that your dataset qualifies as "big".

Further Reading

"MapReduce: Simplified Data Processing on Large Clusters" - the original google paper
http://research.google.com/archive/mapreduce.html

"What is MapReduce" - a very simple explanation of MapReduce
http://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/

"Using Hadoop and PHP" beginners tutorial by Jason Graves
http://collaboradev.com/2010/12/10/using-hadoop-and-php/

HadooPHP - PHP Hadoop framework
https://github.com/dzuelke/HadooPHP

"Big Data Analytics (with Hadoop and PHP)" presentation by David Zuelke
https://speakerdeck.com/dzuelke/big-data-analytics-with-hadoop-and-php-dpc2013-2013-06-06

Criticisms of MapReduce on Wikipedia
http://en.wikipedia.org/wiki/MapReduce#Criticism

9.18 Data caching

At various points in this chapter (and elsewhere) we have mentioned that it is often good practice to cache data where possible. For the uninitiated, this simply means that when we generate some data (be it variables for use in our script, or text or other program output) that we may want to use again later, then instead of re-generating it each time we instead store it somewhere until we need it again.

It should be obvious as to how we can do this, you are likely to be familiar with creating variables, writing files to disk or database, and so on. However while caching data can speed up your programs, it can also come with its own set of problems, such as when to delete the cached data (cache invalidation), how to deal with cache priorities when disk space or memory runs low, how to share cached data between processes and even machines, and other things you (and I) likely haven't even thought of.

> *"There are only two hard things in Computer Science : Naming things, cache invalidation and off-by-one errors"* - saying based on quote by Phil Karlton

While PHP can't help us with the first and last of those, it can help us out a little when working with caches. The following information sources give some details of caching in PHP.

Further Reading

Using Memcached, the high-performance, distributed memory object generic caching system, from PHP
http://www.php.net/manual/en/book.memcached.php

Mysql query caching
http://php.net/manual/en/mysqlnd-qc.quickstart.caching.php

Using output buffering for simple caching
http://www.theukwebdesigncompany.com/articles/php-caching.php

See the section on "Virtual Files - tmpfs" in Chapter 7 of this book for accessing memory based disks

"More - The file system is slow" - by Kevin Schroeder, discusses file system performance and why OS caching may mean that using memory drives aren't necessarily much better for cache uses etc.
http://www.eschrade.com/page/more-on-the-file-system-is-slow/

9.19 Know thy functions

When profiling, you may find some sections of problematic code, often as little as one or two lines, which are running slowly, but for the life of you you can't work out how to re-write them. It's worth knowing that in PHP there are often multiple functions that do the same or similar things, but they are implemented in a different way and so can take (sometimes vastly) different times to achieve the same goal.

Below are two "cheat sheets" for this situation. The first is a web page (that you can also download to run on your own machine) that benchmarks groups of functions that perform similar tasks. This can let you see at a glance which alternative functions may be available and how they might perform. Do be aware that the performance of most functions depends on the code surrounding them, so always profile each one in the context of your own application!

The second link is a Q&A that attempts to list Big O times for many PHP functions. Again this can be used to estimate which function may perform better in your scenario. If you're not familiar with Big O notation, it is a way to describe how the time (or resource) taken by an algorithm increases as the data input size increases. A link to the Wikipedia article on Big O notation is also given.

Further Reading

PHP benchmarking script, grouped by function performed
http://maettig.com/code/php/php-performance-benchmarks.php

Q&A listing Big O times for various functions
http://stackoverflow.com/questions/2473989/list-of-big-o-for-php-functions

Big O notation on Wikipedia
http://en.wikipedia.org/wiki/Big_O_notation

An article that shows how using alternative functions in unusual ways can elicit performance gains
http://www.puremango.co.uk/2010/06/fast-php-array_unique-for-removing-duplicates/

An alternative benchmarking web page (source not availble)
http://www.phpbench.com/

9.20 Outsourcing code to other languages

When you really hit the limits of PHP performance, and more hardware is not an option, one strategy is too move resource heavy code out into better performing languages. There are two ways to do this, either by creating stand-alone programs that you "shell out" to (see chapter 5), or creating PHP extensions (which are generally faster to access than separate processes). If you're going to do this, you'll need to learn or leverage another language, so it's a topic mostly beyond the scope of this book. The following links should give you a good idea of where to start.

Further Reading

"PHP at the Core: A Hackers Guide" on the official PHP documentation. Covers PHP internals with reference to writing extensions.
http://www.php.net/manual/en/internals2.php

"Extension Writing" on Zend Developer Zone - A good introduction to writing extensions
http://devzone.zend.com/303/extension-writing-part-i-introduction-to-php-and-zend/

"Writing Extensions" in Practical PHP
http://www.tuxradar.com/practicalphp/20/0/0

php4delphi - Create extensions using Delphi/Pascal
http://code.google.com/p/php4delphi/

Embed the V8 Javascript engine in PHP to run Javascript code directly from your scripts
http://www.php.net/manual/en/book.v8js.php

Embed PHP into C/C++ (effectively the opposite way around to extensions) using SWIG
http://www.swig.org/

9.21 Other performance tips and tricks

There is a wealth of information about PHP performance and optimisation out there. Much of it revolves around web based scripts, but a lot of it is still relevant to CLI scripts. The following are some good reads in this area, and a quick google will bring you even more. Just to repeat myself once again, optimisations are often dependent on the code they are applied to, so always profile solutions on your own code to see if they make a difference. Also, make sure you read the section below on avoiding premature optimisation.

Further Reading

"PHP performance tips" by Eric Higgins, Google webmaster
https://developers.google.com//speed/articles/optimizing-php

"Common Optimization Mistakes" by Ilia Alshanetsjy
http://ilia.ws/files/Dutch_PHP_Conference_2010_OPM.pdf

"Website Performance: PHP" by Warren Gaebel
http://blog.monitor.us/2012/03/website-performance-php/

"Fast PHP - effective optimisation and bottleneck detection" by Howard Yeend
http://www.puremango.co.uk/2010/04/fast-php/

"A HOWTO on Optimizing PHP" by phpLens
http://phplens.com/lens/php-book/optimizing-debugging-php.php

9.22 Stability and performance of long running processes

It's not just performance that can be affected by the issues raised in this chapter. Stability of your scripts is also affected by the poor management of resources and non-optimal algorithms. By stability, we mean, essentially, "crashes". PHP was initially "designed to die", that is, it was designed for shorter running scripts where the prospect of still being running many minutes later was not a big issue, let alone many hours or days later. That said, if you program carefully there is no reason why you can't create scripts (usually daemons) that can run pretty much indefinitely.

The key areas to think about when designing a stable program are :

- Resource time-outs : Open resources, like database connections and network sockets, can "go away". This can be for a variety of different reasons, such as "lack of activity" time limits on the service you are connected to, network glitches, password/credential changes, session time/resource limits and so on. You may be used to opening a database connection at the start of your script and simply using the resource handle whenever you need to run an SQL query. This typically works fine on the web, if you can connect to the database OK in the first place it means the chances of anything happening during the duration of your script run is minuscule. However, when your script runs for hours or days, the odds mount of something bad occurring. There are several strategies for dealing with this. You can

  - only open connections when needed, and close them immediately afterwards. This is OK for occasional connections, but less OK where the overhead of connecting will cause responsiveness issues for scripts with time-sensitive responses.
  - keep the connection open by occasionally interacting with the service (e.g. via `ticks`). This may be unacceptable to the operator of the service that you are connecting to, and you also still need to check for disconnections for other reasons (e.g. network glitches), although this can be done at the same time as these periodic connections to reduce the chance of a reconnection being needing during "real" interactions.
  - periodically close and re-open connections. This has similar pros and cons to the strategy above
  - check the connection state at each attempt to access the resource. This is usually the safest way, reconnecting immediately if needed, although it usually requires the most effort from you, the programmer. It can also be combined with the two strategies above to reduce the chance that the resource will have "gone away", which is important in scripts with time-sensitive responses.
- Over consumption of memory, and memory leaks : If your script uses too much memory, or uses functions or extensions that "leak" memory, eventually your script will hit a limit. That will either be a limit imposed in the PHP configuration (e.g. in `php.ini`) or the natural limit of the amount of memory available on the machine itself. In either case, this will result in a fatal error causing your script to stop running. Even if you don't reach those limits, over consumption that causes your system to, for instance, start swapping memory to disk, can lead to system wide slowness and instability (and eventually a Ctrl-Alt-Delete three-finger-salute from the user perhaps). Small, persistent uses of memory, such as the repeated creation (without subsequent unsetting) of new variables in the global scope or new static variables in functions, may not be noticeable when profiling your application over even a few hours. However over days or weeks or months, these small additions to the memory footprint may add up to something that causes unexpected crashes. It's important, where possible, to profile over longer periods of time, and to review the use of memory (e.g. dynamic creation of variables and functions) in persistent scopes.
- Fatal error handling : PHP has a range of functions to deal with handling errors, but one type of error can't be handled within PHP. A fatal run-time error, `E_ERROR`, is non-recoverable, and will always cause your script to exit. These errors can be caused by many things, such as memory allocation problems (low memory), missing named functions and so on. On the web, unexpected transient `E_ERROR`s such as a system temporarily out of memory aren't always a problem, the web user hits reload in their browser and the next run may be successful. However if your daemon hits an unexpected `E_ERROR`, then it's game over as your daemon will stop running completely. Strategies for dealing with this include :
  - better testing (e.g. full coverage unit testing) to try and ensure the errors don't occur
  - better resource handling, such as keeping on top of memory management. See the rest of this chapter for information on this!
  - running under external supervision. Use a system like supervisord to monitor and restart your daemon where necessary. This is good as a backup, but no substitution for stopping the fatals from occurring in the first place.

Internal memory leaks and resource persistence issues with PHP itself are sometimes harder to

work around, but these days they are quite rare and it really is worth upgrading to a modern version where possible. The better resource usage and higher performance of modern versions also have the benefit of minimising the impact of inefficiencies in your own code.

9.23 Avoid micro and premature optimisations

Now that I've convinced you of the need to consider performance and instructed you on the finer arts of speed and resource management in PHP, I'm going to try and convince you to put it to the back of your mind, at least for now. It is easy to get caught up with performance and optimisation and let it affect your workflow and productivity. While it is generally positive to follow good practice and consider the performance of your code at all stages of development, premature optimisation is often just that - premature, and micro-optimisations are not usually worth the paper they are written on.

As your author I have an embarrassing confession to make. Having started programming in PHP many years ago, when the performance difference between using single and double quotes around a string was measurable in some contexts (or at least it was "common knowledge" that it was), I still find myself automatically considering what type of quote to use each time I type out a string, not for syntactical reasons but for performance reasons. That's despite the fact that the performance difference between the two now-a-days is definitely extremely, extremely negligible. Old habits die hard though, and many of the micro-optimisations that you will find on the web fall into a similar category. Profile your code, and you will quite likely find that a single verbose database call will make the time saved by calling `isset()` dozens of time to check variables instead of falling back on @ to simply suppress warnings look incredibly small (that's not to say that there aren't other good reasons to do that though!). Developer time in writing and thinking about micro-optimisations is typically much more expensive than the extra processor cycles used to execute un-optimised code.

Premature optimisation typically occurs when developers start thinking about how to scale their code, before they've got any of the code working and deployed. There is no point working out how to scale code that may never see the light of day because the project over-runs and isn't delivered. Unless you are sure of the volume of customers/users/data you will be processing, its usually a better idea to get something up and running, as a proof of concept if nothing else, and then scale it when you know that it will be needed. Indeed, scaling methods vary considerably depending on exactly what your code needs to do, and often this is not finalised until you have something up and running. So even if you deliver on time, your scaling efforts may all be in the wrong direction compared to how your project ends up functioning.

"But, but, but…" you cry, your project is different. It may well be, you might be lucky enough to work on a large scale well funded project that has a known high performance requirement, or be developing a tool that is specifically geared towards high performance uses. Congratulations, you can ignore this section (if you're even at the level where reading this book is useful!). For the rest of us, and this is the majority of PHP projects, concentrate on getting it working and deployed first. Keep performance in mind during development, but don't let it interfere with your work flow too much. Most projects fail for reasons other than poor performance.

Further Reading

"PHP: Require/Include vs Autoloader". An example of an article that espouses an optimisation method that is definitely premature (if worth it ever). The article shows that you can save a *whole 6 milliseconds in total*, in a script that runs the functions *100 times*. That's not even worth thinking about for most people, who would only call it a hand-full of times at most, and in any case for whom 6 milliseconds isn't a concern in the slightest.
http://www.endyourif.com/php-requireinclude-vs-autoloader/

10 Distribution and deployment issues

Now you've written your perfect peice of software, your greatful audience awaits it's delivery and installtion onto their eager machines. Distributing and deploying software comes with its own set of headaches above and beyond those you encountered just writing the damn thing. this chapter covers some of the issues that you may come across when thinking about deployment and distribution of your software, that may vary from the common situation of pushing code to your own server for a PHP website. In particular, we are looking at scenarios where you distribute your software and it is no longer under your direct control and not on your own systems.

10.1 Error handling and logging

You will need to think about how you handle and log errors (and other information) in your scripts when you run with the CLI SAPI. You can use the normal PHP error handling functions and allow PHP to log errors to the file specified in `error_log` in `php.ini`. However bear in mind that you may not be able to control which user your software is run under, and so you need to log to an appropriate location where you can be sure your user has appropriate permissions to write to.

One alternative to the standard "log to a file" method in PHP is to instead send errors and other log information to the system's `syslog`. In Linux systems this is `syslog(3)` (`man syslog` gives more information), in Windows this is the system `event log`. Using `syslog` has several advantages :

- Virtually all users have permission to log to syslog (usually).
- You don't need to manage log rotation or truncation, concurrent accesses/file locking and other routine tasks.
- There is a wide eco-system of tools for exploring and interacting with syslog
- For system daemons and other critical software, system administrators will often expect to find (at least important) logging/error information in the syslog.
- On systems designed to be scalable, syslog is often redirected to a central logging server/system. Writing your logs to syslog means that your software can scale to centrally managed log servers without any changes to your code.

To send your errors automatically to syslog, you simply need to set `error_log` in php.ini to the special string `syslog` rather than a filename. You can also set this from within your script using `ini_set('error_log', 'syslog')`. To log information other than PHP's own errors, or if you've chosen to handle PHP errors internally and need to subsequently log them manually, you can use the `syslog()` function. `syslog()` takes two parameters, a priority level, and the message to

be logged. The priority levels range from `LOG_DEBUG` for debugging messages at the lowest level up to `LOG_EMERG` to describe errors where the system is rendered unusable. The second parameter is a string with the message to log. Before you can call `syslog()`, you need to use `openlog()` to open a connection to the syslog first. The following is a brief example :

```
<?

# Openlog takes an "ident" ('mysoftware') that appears in the logs to help you
# quickly filter for your own software, a set of or'd options (we use LOG_PERR\
OR to also print the logs to STDERR and LOG_PID
# to include the Process ID in the logs), and a "facility" to specify what
# type of software is logging (only LOG_USER is valid in Windows, in Linux
# you can have LOG_DAEMON, LOG_AUTH and so on).

openlog('mysoftware', LOG_PERROR | LOG_PID, LOG_USER);

# We'll log a "notice", a routine notable message

syslog(LOG_NOTICE, 'Script started. All running smoothly.');

# Then when all hell breaks loose, we'll up the anti with an alert level
# message.

syslog(LOG_ALERT, 'BACON ALERT! Script has run out of bacon!. OMG.');

# Finally we can close the log as we've no more messages to log. If we've
# specified syslog in php.ini as the location for error messages, those
# will continue to be logged, only our route for logging custom messages
# is closed here.

closelog();
```

If you run this script in a terminal, you will see the messages being printed to STDERR (usually back to the command line). If you then look at your syslog (on Linux this can often be done by `tail /var/log/syslog`) you should see two entries similar to :

```
Oct  4 19:44:46 dev-machine mysoftware[6205]: Script started. All running smoo\
thly.
Oct  4 19:44:46 dev-machine mysoftware[6205]: BACON ALERT! Script has run out \
of bacon!.
```

In windows, you can use the Event Viewer tool to view the logs, usually found under `Start->Control Panel->System and Maintenance->Administrative Tools->Event Viewer`.

Some other logging tools and further reading in this area are listed below.

Alternative PHP Monitor

Collects error events and stores them in a local SQLite database

Main website : http://code.google.com/p/peclapm/

Main documentation & Installation info :
http://code.google.com/p/peclapm/w/list

Further Reading

Article on using a logging server (Graylog2) for PHP logging
http://jeremycook.ca/2012/10/02/turbocharging-your-logs/

"Logging Best Practices" by Mark van der Velden
http://blog.dynom.nl/archives/Logging-best-practices_20120304_63.html

10.2 Installers and bundling files

In Chapter 3 we looked at different ways to run PHP, but before you can run your PHP scripts you need to get them (and any related assets) onto your target machine. Generic software installer systems are widely available and many platforms have their own software repositories and bundling systems which are well documented, so we won't cover those in any detail here (although we'll mention a few good open source ones at the end). Instead, we'll have a look at a couple of PHP specific options for bundling files and resources ready for distribution and installation.

10.3 Embedded data files at end of PHP script

If we just need to include a chunk of data with a PHP script, we can use a language construct introduced in PHP 5.1, the `__halt_compiler()` instruction. Simply put this at the end of your PHP script, and you can then put anything you want (text strings, binary data, another PHP script etc.) following it. When PHP execution hits the `__halt_compiler()` line, it simply stops. This means that whatever follows isn't executed, and won't throw any syntax errors and so on. Your script can access the data that follows it by opening a file handle to itself and seeking to the `__COMPILER_HALT_OFFSET__` constant which is created when a `__halt_compiler()` instruction is present. The following example shows the "unbundling" of a text file from a PHP file.

```
<?

# Open a pointer to this file using the magic constant __FILE__

$thisfile = fopen(__FILE__, 'r');

# Seek down to the 1st byte after the __halt_compiler(); instruction. This
# is contained in the automatically created __COMPILER_HALT_OFFSET__
# constant.

fseek($thisfile, __COMPILER_HALT_OFFSET__);

# Lets grab everything that follows that ...

$ourtext = stream_get_contents($thisfile);

# and write it out to a new file.

file_put_contents('textfile.txt', $ourtext);

# Our scripts stops here.

__halt_compiler();The additional content starts here.

This is the text file.
It would normally cause a PHP fatal syntax error if
this text was simply dumped into a PHP file.
```

If you run this script, and then look at the created `textfile.txt` file, you will see that it contains the rest of our script starting with the `The additional content starts here` line.

Although we've used plain text here, as it's easier to show in a book, there is nothing to stop you adding binary files, PHP files, or even tar/zip archives containing multiple files which your script can save and expand. However if you are intent on including multiple files or large amounts of data, you might want to consider the Phar format option, discussed next.

10.4 Phar executable bundles

Phar bundles are a native PHP way of pulling together lots of files, both PHP code and ancillary data files, into a single file for distribution. Phar bundles are, at their core, either a zip file, tar file or custom phar format file. You can access the individual files in the bundle using the phar stream wrapper as you would a normal file, without having to unbundle the files before use. For example, if you have a bundle called `mybundle.phar`, you can do :

```php
<?
include('phar://home/rob/mybundle.phar/sample.php');

# or

$data = file_get_contents('phar://home/rob/mybundle.phar/data.csv');

# or

$resource = 'phar://home/rob/mybundle.phar/images/cat.jpg';
$fp = fopen($resouce, 'rb');
header("Content-Type: image/jpeg");
header("Content-Length: " . filesize($resource));
fpassthru($fp);
```

Note that there is a small overhead accessing files in a phar compared to stand alone files, though it is too small to notice in most applications. Phars used in this way are good for easily deploying support files, and can be opened and managed with existing zip/tar aware programs as well as PHP. However it still requires you to deploy at least one php script to access the phar file and control the whole show, and the Phar extension is required to access them. Phar bundles that have the custom phar format have a couple of tricks up their sleeve however, which removes these needs.

With the custom phar format, you can specify PHP "stub" code. This stub code allows you to directly execute the phar file, as if it were just a standard php file. This means that at the command line you can do something like `?> php mybundle.phar -x something -l something-else` to run your application. It also allows PHP to recognise the file as a standard PHP file, negating the need for the phar extension to be available. With the right file associations set, you can even set a phar file to be self executing (see Chapter 2) in the same way as a standard PHP script. The stub code simply acts as an entry point for PHP, telling it where to begin, and is often just an `require()` statement for the main PHP script in the bundle. Even with stub code, the phar file can still be accessed with the `phar://` stream wrapper from external scripts, so can act both as an executable bundle and a simple collection of files at the same time. The only real downside to the custom phar format is that it can only be created with PHP itself (specifically with the phar extension). The zip and tar varieties can be created using most third party archive tools.

Further Reading

"Phar: PHP libraries included with a single file" tutorial by Giorgio Sironi
http://css.dzone.com/articles/phar-php-libraries-single-file

Phar documentation in the PHP Manual
http://www.php.net/manual/en/intro.phar.php

10.5 Generic installers

The following are a couple of opensource generic installers of note. These can be used to install PHP itself and other sofware along with your PHP application, and can of course be used to deploy and run phar bundles (see previous section).

WiX Toolset

Comprehensive set of tools for creating .msi and .exe installers for Windows

Main website : http://wixtoolset.org/

Main documentation & Installation info :
http://wixtoolset.org/documentation/manual/

fpm

Easy to use tool to create .deb, .rpm, solaris and puppet packages/modules.

Main website : https://github.com/jordansissel/fpm

Installation info : https://github.com/jordansissel/fpm#get-with-the-download

Main documentation : https://github.com/jordansissel/fpm/wiki

10.6 Controlling the (PHP) environment

When you deploy scripts on your own servers or PCs, you can control the environment in which they are deployed. However if you are distributing your software more widely you will suddenly find that environmental factors on external machines can make your PHP scripts misbehave or even stop working completely. Some of the PHP specific environmental factors you need to consider include:

- Installed PHP version : Check the current version (using `phpversion()` from within your script) to ensure you script is compatible. Consider deploying PHP itself with your installer,

if doing so it may be worth installing it in a non-standard location, with a non-standard name, and without setting the `php` environment variable, to avoid confusion with any existing versions.

- Installed Extension version : As with the main PHP installation above, you can check versions of critical extensions using `phpversion('extension_name')` and act accordingly
- `php.ini` settings : These are likely to be different to those on your machine. Ideally you will supply your own custom `php.ini` file and use that (e.g. call `php --php-ini my-php.ini myfile.php`, see chapter 2) to ensure everything is set as you require. You can alternatively set important settings from within your script using `ini_set()`, however be aware that this can be disabled from within the existing `php.ini` file.
- Location of common paths : The system temporary directory on Linux is often `/tmp`. But not always. Where possible, use functions like `sys_get_temp_dir()` and `get_env('PATH')` to find relevant folders on the system, or at least allow users to specify them in a configuration file.

You'll also need to check any other non-PHP dependencies you may have, assess whether there are appropriate levels of hardware resource (memory, disk space) available, and check for any network/internet connectivity required.

3v4l.org

Web service which allows you to test chunks of code in 90+ versions of PHP with the click of 1 button. Helpful in determining which versions of PHP your code can support.

Main website : http://3v4l.org/

10.7 Extending your application with plug-ins

When working on your own website or software, if you need additional functionality you typically add it directly into your software. When you distribute software, it's often useful to allow users to provide their own additional functionality, particularly functions that are only relevant to their particular needs. The usual way to do this is via a plug-in/extension/add-on type system. There are a thousand and one ways to implement plug-ins, the following article outlines some of the common design patterns for doing so and examples of how to do so in PHP.

Further Reading

"Handling Plugins In PHP" by Anthony Ferrara
http://blog.ircmaxell.com/2012/03/handling-plugins-in-php.html

10.8 Documentation

Ideally your application will be so easy to use, that your users won't require any documentation, right? When you've finished laughing, we'll continue. You may be familiar with providing end-user documentation and support information on web projects, but a typical software project requires even more. You will also need installation and upgrade docs, information for system administrators, licensing documentation, and so on. If you allow/encourage users to tinker with the code, you may even need to include code documentation and architecture/design documentation.

You should aim to provide all the documentation in two locations, online and included with the software. The copy with the software allows users access when they may not have an internet connection, or when your website is down (or your company has gone bust…). The online documentation provides an easy to update source for corrections or additional information, and it also allows prospective users/customers to assess the software without having to download/purchase it.

Finally, when distributing your software widely, fully comprehensive and high quality documentation can reduce the number of requests for technical/user support that you get. This will usually save you much more time than it takes to write the documentation.

Further Reading

Software Documentation on Wikipedia
http://en.wikipedia.org/wiki/Software_documentation

10.9 Licensing & legal

Warning

Note: All the legal information presented below and elsewhere in this book may be incorrect, depends on your jurisdiction, and was believed correct by the non-legally-trained author only at the time of writing. The information is only provided to give you an overview of the topics that you may wish to consider in respect of your particular project. Always consult a qualified legal professional for legal advice. Never drink coffee just before bed.

If you're coming from a web development environment to, for instance, desktop software development, it is worth taking a moment to think about the consequences and differences in licensing and legal liabilities in your new mode of programming, particularly when programming commercially. With the web, your "product" is usually your website content, not the PHP script itself. Selling software reverses the situation, with your carefully honed PHP code being the saleable commodity and the output belonging to the end users.

The law applies to software written in PHP just like software written in any language, so we won't treat general legal issues in this book.

However there is one legal issue specific to scripted software like PHP, and that is the distribution of the interpreter itself. Every PHP script requires the PHP binaries to run (assuming you haven't compiled your code using HipHop or similar), and most systems don't come with PHP installed by default. Luckily, PHP itself is licensed under a permissive Open Source Initiative (OSI) certified license, imaginatively called the PHP License. With very few restrictions, you are free to distribute the PHP code base and compiled binaries with your application. If you choose to modify or alter them, then its worth checking the license (an easy to read, short 68 line in plain English) to make sure you're still OK. If you're not changing them in anyway, go right ahead! Commercial distribution is also fine, there are no limits on charging or including with paid-for software.

Another important point is that the license only covers PHP itself, and not any of your scripts written in PHP, so you can release those under any license you deem appropriate.

You should also bear in mind that PHP is supplied `as-is`, so on a warranty basis you will have very little (or most likely no) comeback should a bug in PHP cause you or your customers problems. Likewise, you will not be covered by the PHP team against patent violations (in countries where laws allow software patents) that may result when you deploy PHP as part of your product. These downside, however, are common among most programming languages, commercial and open source alike.

If you want to side-step the issue of distributing PHP yourself, remember that most modern operating systems with software repositories will have PHP installable via a couple of clicks or one line of code. If you're distributing your software as a .deb or .rpm package on Linux, you can simply include PHP as a required dependency and let the package manager deal with it for you.

As with all software, you (unfortunately) need to consider all relevant laws, including but not limited to : Copyright, Patents, Trademarks, Data Protection, Trading Standards, Consumer Protection and any regulatory requirements in your field of endeavour. We'll leave that as an exercise for the reader…(or their legal representative!).

10.10 Deploying frameworks

There are many coding frameworks for PHP, and many of them can be used with CLI applications, although there are none that are specifically created for or targeted to non-web programming. Code in various frameworks may assume that it will be called in an http related context, so you may need to do extra work to code around this. When deciding whether to use a Framework, or which one to use, you should bear in mind their applicability (in terms of their focus on the web) and whether your applications performance will suffer from the overhead they may bring. You will usually also need to look at their license, as they will usually have components that need to be distributed with your scripts. That's not to say that they can't be useful in general purpose programming, for instance many frameworks implement the Model-View-Controller (MVC) design pattern which was invented in the 70's at Xerox Parc, long before the web even existed. However there are none that your author can currently recommend specifically with reference to non-web projects.

Further Reading

The PHP Framework Interop Group - standardising interoperability between Frameworks
http://www.php-fig.org/

Example of using Zend Framework with a CLI script
http://stackoverflow.com/questions/2325338/running-a-zend-framework-action-from-command-line

Framework comparison matrix
http://matrix.include-once.org/framework/

11 Where now? or, Thanks & feedback

If you've read this far, thank you. I sincerely hope that this book has held your interest and at the very least informed some areas of your future PHP programming. If it has, I would encourage you to start some coding right now based on one or more topics in this book while they are still fresh in your mind. People far smarter than me have shown that the sooner relevant activity occurs after learning, the easier it is to retain information and techniques over the longer term. If you use any of the techniques presented in the book in "real-life", I would be very interested to know! Finally, if you've not already headed for the keyboard, don't forget to glance through the Appendices that follow. There may be something interesting or useful in there for you (either to us now or to be aware of for the future).

11.1 Giving feedback, and getting help and support

E-mail : author@phpbeyondtheweb.com

Your feedback on this book, good or bad, fundamental or trivial, is solicited and very much welcomed. Tell me what you thought about the book, overall or a particular section. Let me know if any areas weren't covered in enough depth (or in too much detail), if any topics you were expecting weren't present, or any of the information wasn't clear. I intend to keep revising and improving this book over time, as well as adding new topics and example code, and as an existing purchaser you will get free updates as newer versions are produced. Likewise, if you have any problems getting the sample code to run, or issues implementing the techniques discussed please do drop me a line and I'll see if there is any way I can help.

11.2 Giving testimonials

If you would like to provide a short testimonial for this book's website, please e-mail it to author@phpbeyondtheweb.com indicating you are happy for me to use it to promote the book. In return, if I use it I will provide a link back to your website/twitter/email.

11.3 Are you reading a "pirated" copy?

There are times in my life when I have read "unauthorised" copies of books; when I couldn't afford them, when they weren't easily available through legitimate channels (local bookshops, online sales, libraries), or (for e-books) when DRM thwarted my efforts to read them on the devices I had available. So I quite understand if you are in this position, and I don't want to criticise your choice. If you are unable to afford this book, perhaps you could consider purchasing

it in the future when you are able, and in the meantime I would ask that you do me a favour in return - please consider helping me to promote it by recommending it to those that can afford it, have a go at reviewing it online, and otherwise spread the word. If you're reading an unauthorised copy because there was something else (other than cost) blocking your access to a legitimate copy, please let me know and I'll do my very best to remove that obstacle. I want to help everybody who wants to buy and read this book to be able to do so with the minimum of fuss. This is why I won't employ DRM on my work, because for every "pirate" that it stops (and it won't be many), there are likely to be ten times as many legitimate readers who it annoys.

If you are reading an unauthorised copy because you simply don't want to pay for it, then that is between you and your conscience. That is how you have chosen to lead your life, and I have nothing more to say to you.

If you are selling or otherwise making money from knowingly distributing unauthorised copies of this book, then I will do my level (legal) best to stop you. That's just not nice.

Appendix A : Compiling and installing PHP and its extensions

There are a dozen different ways to get PHP, including downloading & compiling it yourself, downloading pre-compiled binaries, using package managers and software repositories, and finding it pre-installed by a forward thinking administrator. On most Linux distributions, PHP can be installed with a one-line command such as `sudo apt-get install php5` or through graphical package managers like the Synaptic Package Manager or the Ubuntu Software Centre. Many common PHP extensions and add-ons are likewise available as pre-built packages or alternatively through the PECL and PEAR systems.

However there are times when it becomes necessary to do a little more work to install PHP, such as :

- when your project has requirements for a specific version of PHP that is different to the one shipped with your OS
- where you need extensions not available as packages,
- or when you want to compile a customised version of PHP specific to your needs.

Like anything involved in computers and software development, compiling PHP can take you down a rabbit-hole of options, customisations, compatibility issues, libraries and dependencies. A whole book could be written about the different possibilities (and possible headaches) involved. Luckily for us, in most use cases and needs, the basics of compiling a standard version are quite straightforward. And like most things in life, it gets easier once you have done it once. The first section below will go over the steps necessary for getting, compiling and installing PHP and it's core extensions. PHP is written in C, and as many of you may not be familiar with the process of compiling C programs, I have tried to explain each step below to give you an idea of what is happening. This makes the process seem a little more verbose, but in reality it is quite straightforward. Go ahead and try it if you don't believe me! The next sections are also worth a read, as they cover installing extras like libraries and extensions from the PECL, PEAR and Packagist repositories.

Compiling and installing PHP itself

Note for Windows devs - These steps are for Linux/Unix type systems, and use free compiler tools almost always included with the OS. For Windows, the proprietary Visual Studio compiler is required, and the steps are somewhat different and beyond the scope of this book. Windows source code, pre-built binaries and instructions for compiling can be found at http://windows.php.net/download/ , with older versions in the archive at http://windows.php.net/downloads/releases/archives/.

Our first step is to download the PHP source code from the PHP website at http://www.php.net/downloads.php . This page lists the current stable release and the previous supported stable release. Newer versions which are still under development are available at http://snaps.php.net/, and older end-of-life versions are available at http://museum.php.net/ . Git users can also pull the source code down from the official mirror at https://github.com/php.

When you have identified which version you want, make a note of the URL of the .tar.gz source code file which we will use later.

```
1 ~$ mkdir php5.4
2 ~$ cd php5.4
3 ~$ wget http://uk3.php.net/get/php-5.4.6.tar.gz/from/uk.php.net/mirror
4         -o php-5.4.6.tar.gz
5 ~$ tar zxvf php-5.4.6.tar.gz
6 ~$ cd php-5.4.6
```

The first two lines creates a directory for our work and steps into it. The directory holds the source code and intermediate files, and can be deleted once PHP is installed if you want. However it is often a good idea to keep it in-case you need/want to re-install or check what version of the file you downloaded later. The third line downloads a copy of the source code file into our directory. Change the URL in the third line to that of the .tar.gz file you want to use, and the `-o` option to the name of the file (otherwise, in the above example, wget will simply call your file "mirror"). The fifth line unpacks the archive into a directory containing the individual source code files, change the name of the file the one you used on line 3. Finally, the last line steps us into the source code directory. Now we start the actual compilation process.

```
1 ~$ ./configure
```

The configure command creates the "setup" for compilation. We use it to provide the settings and arguments we want for our compilation session. For instance, we can specify which core extensions we want to include in our build. If we don't specify any arguments as above, the defaults provided by the PHP dev team are used. This is a good choice if you don't have any particular needs and want a version that is fairly similar/compatible with the versions included with most distributions. You can also install extensions at a later date either individually or by re-compiling PHP from scratch, which we will discuss in the next section. If you want to include an extension at this stage that's not included in the default settings, then this is the place to do it. For example, if you wanted to include the ldap extension, then you would change the command above to `./configure --with-ldap[=DIR]` where `[=DIR]` is the base installation directory of ldap on your system. The exact option to use and any necessary dependencies can be found in the PHP manual, under the "Installing/Configuring" section for the extension in question. For example, details for the ldap extension can be found at http://www.php.net/manual/en/ldap.setup.php. A (slightly out-of-date) list of options that you can pass to the configure command can be found at http://www.php.net/manual/en/configure.about.php or, for a full list of those supported on your system in the current version you are trying to compile, you can issue the command `autoconf` followed by `./configure --help`. More information about the configure command can be found at http://www.airs.com/ian/configure/. Now we will actually compile PHP.

```
~$ make clean
~$ make
~$ sudo make install
```

We compile the binary PHP files using the "make" tool. The first line removes any previously created files and resets us to the start of the "make" process. This is not strictly necessary on your first run at compiling, but it can help if your attempt fails for some reason (such as missing dependencies, incorrect settings, unintended interruptions to the process etc.) so including it by default is often a good choice. The second line does the actual building and compiling of the files. The third line then takes those files and installs them on the system. By default, PHP will usually be installed in the `/usr/bin` directory on Linux. However you can choose where to install it by specifying a "prefix" directory at the `./configure` stage above. Simply add the switch `--prefix=/path/to/dir` where `/path/to/dir` is the directory into which you want PHP to be installed. This is often useful if you want to install multiple versions on the same machine (although be aware there are other considerations when doing that). Note that the `make install` line must be run with higher permissions (`sudo`) to allow it to copy files into "privileged" locations.

If all goes well, congratulations, you have installed PHP! To check that the correct version is installed and available, use `php -v` at the command line and PHP will display the current version number. If you have installed PHP in a location outside of your search path you will need to specify the full path name, e.g. `/path/to/dir/php -v`. To check which extensions and other options were installed, use `php -i` at the command line to run the `phpinfo()` function. As well as extension information (and a lot more besides), this returns a list of the options used with the `./configure` command. This can be useful when re-installing PHP or when trying to clone an installation on another machine (where the same binaries cannot just be re-used).

If all doesn't go well, have a close look at the errors produced. The most common type of errors happen when your system doesn't have the relevant dependencies installed for a particular extension. Often the error message will say this explicitly, but even if it just gives you an obscure error message mentioning the name of an extension, the best advice is to double-check the installation section for that extension in the the PHP manual to find out exactly what dependencies are required. Missing dependencies can often be installed using your systems package manager rather than having to manually compile them. You should also check that you have provided the location of any dependencies at the configure stage above if required.

If all else fails, copy and paste the exact error message into your favourite internet search engine, probably starting with the first error message shown if multiple errors appear. Many, many people have compiled PHP, and most errors have been encountered and documented online. Don't let all this talk of errors put you off having a go at compiling PHP. Errors are more likely to occur the more complicated you make your configuration, and if you're careful about dependencies you can often avoid them altogether. So have a go at a straight forward compilation with the default options to get the hang of things, and take it from there!

More information on installations, with a general focus on web servers but otherwise useful, can be found in the PHP manual at http://www.php.net/manual/en/install.php.

Compiling and installing (extra) core extensions

As we saw in the previous section, the most common way to install core extensions is to enable the relevant flags at the configure stage during compilation of the main PHP installation (note by default, many extensions are automatically enabled). However it's not uncommon to come across the need to install an additional extension later on, for instance as the requirements for your program change from its initial design. There are two ways to approach this. The first, which you'll find recommended a lot online, is to re-do the compilation/installation of PHP from scratch, adding the newly required modules at the configure stage (after issuing `php -i` to remember what configure options you used the first time). While this works perfectly well, compiling the full PHP binaries is a bit of a slog which can take older PCs in particular a long time to complete. There is a short cut however.

Each of the core extensions is actual a separate .so binary, and can be compiled independently. To do this, follow the first steps in the previous section to download and unpack the PHP source code and step into the directory. If you haven't deleted it from when you compiled PHP itself, it should be ready to go. Within the source code is a directory called ext, inside of which are separate directories for each of the extensions.

```
~$ cd ext
~$ ls
```

This will show you all of the core extensions available. For instance, if we want to add the pcntl extension (used in this book for daemon software), we can enter the pnctl directory and compile/install just that extension in a similar manner to how we compiled the whole PHP package in the previous section.

```
~$ cd pcntl
~$ phpize
~$ ./configure
~$ make clean
~$ make
~$ sudo make install
```

The additional command above, `phpize`, is used to prepare the build environment for the extension. This is not necessary when building the full PHP binaries, but it is when building individual extensions. If you find that you don't have phpize on your system, it is often available through your systems package manager in the `php-dev` package (e.g. on Ubuntu it is available as `phpize5` in the `php5-dev` package). More details about phpize and getting it can be found at http://us.php.net/manual/en/install.pecl.phpize.php.

Once the commands above have been run, you should find that a `.so` file (`pcntl.so` in our example) has been compiled and placed in PHP's extension directory. The final step is to tell PHP about it, by adding the following line somewhere in your `php.ini` file:

```
1 extension=pcntl.so
```

If you're not sure where your `php.ini` file is, you can run `php -i | grep "Loaded Configuration File"` on the command line to find out. You can also use `php -i` to check that your extension is now correctly installed and available for use.

Installing multiple versions of PHP

Sometimes (particularly on development machines) you may want to install multiple versions of PHP at the same time, for instance if you are deploying to end users with PHP already installed but who may have different versions. One straight forward way to achieve this is to create a set of virtual machines (I use VirtualBox for this) with a different version of PHP installed in each. In this scenario you can always be sure which version you are running and that the installation and configuration of one version isn't interfering with that of another. The downside is that it can be slow to start up and shut down different VMs (or a hit on resources to run them all at once), and if you are using proprietary OSes like Windows you can incur additional licensing costs. As I noted in a previous section, it is possible to have multiple versions installed and running directly on the same machine. However if you are not careful it can become a nightmare trying to keep the versions and their dependencies separate and making sure you know which version you are using at all times. As such I am not going to delve into it in this book. However below are two articles from respected PHP community members who have done just that, which may give you some pointers on what to do and the pitfalls involved. I would suggest that before you try this, you become intimately familiar with compiling and installing PHP, the file and directory structures and locations that PHP uses, and how to check which versions of PHP and extensions are running.

There are also a couple of relevant tools listed below. The first is php-build, which automatically builds multiple versions of PHP from source, although you still need to exercise care installing and using them simultaneously. The second is 3v4l.org, a web service which allows you to test chunks of code in 90+ versions of PHP at the same time. This may avoid the need for installing multiple versions at all. And the final tool is a library which simulates many functions from newer versions for use with older versions.

Further Reading

Installing multiple versions, using SVN
http://derickrethans.nl/multiple-php-version-setup.html

Installing multiple versions, using GIT
http://mark-story.com/posts/view/installing-multiple-versions-of-php-from-source

php-build

php-build automatically builds multiple PHP versions from source

Main website & documentation : http://www.christophh.net/php-build/

3v4l.org

Web service which allows you test chunks of code in 90+ versions of PHP with the click of 1 button.

Main website : http://3v4l.org/

upgrade.php

Library to emulate newer functions for older versions of PHP.

Main website : http://include-once.org/p/upgradephp/

PEAR and PECL

PEAR (PHP Extension and Application Repository) is a library of code and extensions written in PHP, with an easy to use packaging and distribution system. PECL (PHP Extension Community Library) is essentially the same, but for extensions written in C.

Both PECL and PEAR work in a similar way to package managers such as Debians apt-get. For example to install the Cairo graphics extension from PECL, simply do `sudo pecl install cairo` at the command line. This will download, compile and install Cairo for you and you can then start using it from within your PHP scripts. Similarly, to install the RDF extension from PEAR, use `pear install rdf`.

The `pear` and `pecl` commands are included as standard with PHP, however some package managers put them in the optional `php-dev` or `php-pear` package. On Ubuntu, for instance, use `sudo apt-get install php-pear` to install it.

More information on both as well as the hundreds of extensions and libraries available, can be found at http://pear.php.net and http://pecl.php.net respectively.

Composer

Composer is a dependency manager. While it deals with packages, it is not a package manager like PEAR. Rather than installing packages centrally, it deals with them on a per-project basis, ensuring that the appropriate versions of the relevant packages, and their dependencies, are installed automatically for that project.

The basic Composer workflow happens as follows :

- You install Composer.
- In the base directory of your project, you create a json-formatted file called `composer.json` that specifies which packages (and versions) your project needs
- In that directory, you run Composer. It will fetch and install of the specified packages, and will automatically also install any of the other packages that those you have specified depend on (and so on until all dependencies are satisfied)
- In your PHP code, simply add the function `require 'vendor/autoload.php';` and your libraries will be automatically available when you use them.

Fully comprehensive documentation, aimed at beginners as well as advanced users, is available on the composer website. Composer itself doesn't host any packages, that is the job of package repositories. Packagist is the main, and currently the only, comprehensive public repository, and is the default used by Composer. You can browse the thousands of available packages on the Packagist website. You can of course specify a different repository and indeed create and use your own packages privately if you need to.

Composer Dependency Manager

The easy way to keep libraries consistent and up-to-date on a per-project basis.

Main website : http://getcomposer.org/

Package Repository : https://packagist.org/

Installation info : http://getcomposer.org/doc/00-intro.md#installation-nix

Main documentation : http://getcomposer.org/doc/

Tutorial : http://hassankhan.me/post/58193034824

Symfony2 bundles

If you're using the Symfony2 framework, you can choose from and download over a thousand useful code bundles from knpbundles. These can be installed manually, or often using the Composer dependency system (see the section above). Visit http://knpbundles.com/ for more information and to browse the available code.

Appendix B : File & data format libraries for PHP

This chapter lists extensions/libraries/bindings broken down by category that can be used to work with different file and data formats in PHP. Within each category you will find the name of the library, followed by a list of file types (specified by their common file extension for brevity) that it supports, and the main website for that library. Where there isn't a common file extension (for example streaming data formats), the name of the format is used instead.

The list below is not exhaustive, and not all libraries allow you to both read and write the given formats, or support all features of the format. Check the relevant documentation for the library and try it out before selecting it for your project. If you haven't found a library to help you with the file format you're looking for, try using your favourite search engine! If you know of one or find one that's not listed below, please e-mail me (author@phpbeyondtheweb.com).

Office documents

OpenDocument : odt
http://pear.php.net/manual/en/package.fileformats.opendocument.php

PHPExcel : ods, gnm, gnumeric, xlsx, xls (biff), xls (SpreadsheeML), htm, html, sky, slk, sylk, csv, pdf
https://github.com/PHPOffice/PHPExcel

php-excel-reader : xls (biff)
http://code.google.com/p/php-excel-reader/

Excel Writer (XML) for PHP : xls (SpreadsheeML)
http://sourceforge.net/projects/excelwriterxml/

php-export-data : xls (SpreadsheeML), csv, tsv
https://github.com/elidickinson/php-export-data

Spreadsheet_Excel_Writer : xls (biff)
http://pear.php.net/manual/en/package.fileformats.spreadsheet-excel-writer.php

SimpleExcel : xlsx, htm, html, csv, tsv, json
http://faisalman.github.com/simple-excel-php/

PHPPowerPoint : pptx
http://phppowerpoint.codeplex.com/

PHPWord : docx
http://phpword.codeplex.com/

Cairo extension : pdf, ps, svg
http://www.php.net/manual/en/book.cairo.php

tcPDF : pdf
http://www.tcpdf.org/

DomPDF : pdf
https://github.com/dompdf/dompdf

mPDF : pdf
http://www.mpdf1.com/mpdf/index.php

Haru PDF extension : pdf
http://www.php.net/manual/en/book.haru.php

PDF extension (PDFlib) : pdf
http://www.php.net/manual/en/book.pdf.php

FDF extension : fdf
http://www.php.net/manual/en/book.fdf.php

DOM extension : htm, html
http://www.php.net/manual/en/book.dom.php

Contact_Vcard : vcf, vcard
http://pear.php.net/manual/en/package.fileformats.contact-vcard.php

File_MARC : mrc, marc
http://pear.php.net/manual/en/package.fileformats.file-marc.php

Compression, archiving & encryption

File_Archive : tar, gz, gzip, bz2, tgz, tar.gz, tbz, tar.bz2, zip, ar, deb
http://pear.php.net/manual/en/package.fileformats.file-archive.php

Zlib extension : gz, gzip
http://www.php.net/manual/en/intro.zlib.php

BZip2 extension : bz2
http://www.php.net/manual/en/book.bzip2.php

RAR extension : rar
http://www.php.net/manual/en/book.rar.php

Zip extension : zip
http://www.php.net/manual/en/book.zip.php

RPM extension : rpm
http://www.php.net/manual/en/book.rpmreader.php

LZF compression : lzf
http://www.php.net/manual/en/book.lzf.php

File_cabinet : cab
http://pear.php.net/manual/en/package.fileformats.file-cabinet.php

Phar extension : phar, tar, zip
http://www.php.net/manual/en/book.phar.php

GNU Privacy Guard extension : gpg, pgp
http://www.php.net/manual/en/book.gnupg.php

Graphics

Gmagick extension : art, avi, avs, bmp, cals, cin, cgm, cmyk, cur, cut, dcm, dcx, dib, dpx, emf, epdf, epi, eps, eps2, eps3, epsf, epsi, ept, fax, fig, fits, fpx, gif, gplt, gray, hpgl, html, ico, jbig, jng, jp2, jpc, jpeg, man, mat, miff, mono, mng, mpeg, m2v, mpc, msl, mtv, mvg, otb, p7, palm, pam, pbm, pcd, pcds, pcl, pcx, pdb, pdf, pfa, pfb, pgm, picon, pict, pix, png, pnm, ppm, ps, ps2, ps3, psd, ptif, pwp, ras, rad, rgb, rgba, rla, rle, sct, sfw, sgi, shtml, sun, svg, tga, tiff, tim, ttf, txt, uil, uyvy, vicar, viff, wbmp, wmf, wpg, xbm, xcf, xpm, xwd, yuv
http://www.php.net/manual/en/book.gmagick.php

ImageMagick extension : aai, art, arw, avi, avs, bmp,bmp2,bmp3, cals, cgm, cin, cmyk, cmyka, cr2, crw, cur, cut, dcm, dcr, dcx, dds, dib, djvu, dng, dot, dpx, emf, epdf, epi, eps, eps2, eps3, epsf, epsi, ept, exr, fax, fig, fits, fpx, gif, gplt, gray, hdr, hpgl, hrz, html, ico, info, inline, jbig, jng, jp2, jpc, jpeg, jxr, man, mat, miff, mono, mng, m2v, mpeg, mpc, mpr, mrw, msl, mtv, mvg, nef, orf, otb, p7, palm, pam, clipboard, pbm, pcd, pcds, pcl, pcx, pdb, pdf, pef, pfa, pfb, pfm, pgm, picon, pict, pix, png, png8, png00, png24, png32, png48, png64, pnm, ppm, ps, ps2, ps3, psb, psd, ptif, pwp, rad, raf, rgb, rgba, rfg, rla, rle, sct, sfw, sgi, shtml, sid,mrsid, sparse-color, sun, svg, tga, tiff, tim, ttf, txt, uil, uyvy, vicar, viff, wbmp, wdp, webp, wmf, wpg, x, xbm, xcf, xpm, xwd, x3f, ycbcr, ycbcra, yuv
http://www.php.net/manual/en/book.imagick.php

File_DICOM : dcm
http://pear.php.net/manual/en/package.fileformats.file-dicom.php

Ming extension : swf, flash
http://www.php.net/manual/en/book.ming.php

Cairo extension : pdf, svg, ps
http://www.php.net/manual/en/book.cairo.php

EXIF extension : exif
http://www.php.net/manual/en/book.exif.php

Audio

MP3_id : mp3
http://pear.php.net/manual/en/package.fileformats.mp3-id.php

OGG/Vorbis extension : ogg, oga, ogv, spx
http://www.php.net/manual/en/book.oggvorbis.php

php-reader : mp3, asf, wma, wmv flac, 3gp, 3gpp, avc, dcf, m21, m4a, m4b, m4p, m4v, maf, mj2, mjp, mov, mp4, odf, sdv,qt, abs, mp1, mp2, mpg, mpeg, vob, evo, ogg, oga, ogv, spx
http://code.google.com/p/php-reader/

ID3 extension : mp3
http://www.php.net/manual/en/book.id3.php

ktaglib extension : mp3
http://www.php.net/manual/en/book.ktaglib.php

Multimedia & video

PHP-FFmpeg : 4xm, 8088flex tmv, act voice, adobe filmstrip, audio iff (aiff), american laser games mm, 3gpp amr, amazing studio packed animation file, apple http live streaming, artworx data format, adp, afc, asf, ast, avi, avisynth, avr, avs, beam software siff, bethesda softworks vid, binary text, bink, bitmap brothers jv, brute force & ignorance, brstm, bwf, cri adx, discworld ii bmv, interplay c93, delphine software international cin, cd+g, commodore cdxl, core audio format, crc testing format, creative voice, cryo apc, d-cinema audio, deluxe paint animation, dfa, dv video, dxa, electronic arts cdata, electronic arts multimedia, ensoniq paris audio file, ffm (ffserver live feed), flash (swf), flash 9 (avm2), fli/flc/flx animation, flash video (flv), framecrc testing format, funcom iss, g.723.1, g.729 bit, g.729 raw, gif animation, gxf, icedraw file, ico, id quake ii cin video, id roq, iec61937 encapsulation, iff, ilbc, interplay mve, iv8, ivf (on2), ircam, latm, lmlm4, loas, lvf, lxf, matroska, matroska audio, ffmpeg metadata, maxis xa, md studio, metal gear solid: the twin snakes, megalux frame, mobotix .mxg, monkeyâ€™s audio, motion pixels mvi, mov/quicktime/mp4, mp2, mp3, mpeg-1 system, mpeg-ps (program stream), mpeg-ts (transport stream), mpeg-4, mime multipart jpeg, msn tcp webcam, mtv, musepack, musepack sv8, material exchange format (mxf), material exchange format (mxf), d-10 mapping, nc camera feed, nist speech header resources, ntt twinvq (vqf), nullsoft streaming video, nuppelvideo, nut, ogg, playstation portable pmp, portable voice format, technotrend pva, qcp, raw adts (aac), raw ac-3, raw chinese avs video, raw cri adx, raw dirac, raw dnxhd, raw dts, raw dts-hd, raw e-ac-3, raw flac, raw gsm, raw h.261, raw h.263, raw h.264, raw ingenient mjpeg, raw mjpeg, raw mlp, raw mpeg, raw mpeg-1, raw mpeg-2, raw mpeg-4, raw null, raw video, raw id roq, raw shorten, raw tak, raw truehd, raw vc-1, raw pcm, rdt, redcode r3d, realmedia, redirector, redspark, renderware texture dictionary, rl2, rpl/armovie, lego mindstorms rso, rsd, rtmp, rtp, rtsp, sap, sbg, sdp, sega film/cpk, silicon graphics movie, sierra sol, sierra vmd, smacker, smjpeg, smush, sony openmg (oma), sony playstation str, sony wave64 (w64), sox native format, sun au format, text files, thp, tiertex limited seq, true audio, vc-1 test bitstream, vivo, wav, wavpack, webm, windows televison (wtv), wing commander iii movie, westwood studios audio, westwood studios vqa, xmv, xwma, extended binary text (xbin), yuv4mpeg pipe, psygnosis yop
https://github.com/alchemy-fr/PHP-FFmpeg

Programming, technical and data interchange

File_Fstab : fstab
http://pear.php.net/manual/en/package.fileformats.file-fstab.php

File_Passwd : passwd
http://pear.php.net/manual/en/package.fileformats.file-passwd.php

YAML extension : yaml
http://www.php.net/manual/en/book.yaml.php

Assorted extensions : xml
http://www.php.net/manual/en/refs.xml.php

XSL extension : xsl, xslt
http://www.php.net/manual/en/intro.xsl.php

json extension : json
http://www.php.net/manual/en/book.json.php

native file functions (fopen) : txt
http://www.php.net/manual/

Misc

File_Fortune : fortune
http://pear.php.net/manual/en/package.fileformats.file-fortune.php

Appendix C : Sources of help

Even with excellent books like this on on the market, you will sometimes need a little additional help when you come across a tricky problem with PHP. Below are some potential sources of help.

The PHP manual

The official PHP manual can be found online at http://php.net/docs.php. The manual provides fairly comprehensive documentation in the main on PHP installation, syntax, functions and many extensions. Of particular note are the user comments at the bottom of each page. These are generally helpful and offer real-world advice related to the function or topic of the page. Occasionally some duff advice is given in the comments, however this is usually corrected or mentioned in a subsequent comment, so it's worth reading through all of the comments on a given page.

A handy function of the online manual is that you can do a quick look up of a function or topic by typing it as the first part of a URL. For instance, if you can't remember what the parameters of `strripos()` are, you can simply type http://php.net/strripos into your browser and you will be sent straight to the relevant page. Likewise, if you want a quick refresher on how PHP handles arrays, visit http://php.net/array and you'll go straight to the array page in the language/types section of the manual.

If you don't always have an internet connection, you can also download a copy of the manual from http://www.php.net/download-docs.php. It is available as HTML, Unix style man pages, and Microsoft Compiled HTML Manual (CHM) format.

Official mailing lists

There are a number of official mailing lists at http://php.net/mailing-lists.php covering a wide variety of topics. Of note for getting help are the "General user list" for general queries, and the "Windows PHP users list" for Windows specific questions. Beware if subscribing that some of the lists area quite busy and you will get a large number of emails each day. The archives are available online if you just want to browse or get a feel for the volume generated on each list.

Stack Overflow

If you're not familiar with it, Stack Overflow (http://stackoverflow.com) is a prolific "Question and Answer" site aimed at programmers. Unlike some Q&A sites, you don't need to join or pay to view the answers to questions, adverts are limited and there are literally millions of answered questions on the board. This includes a good chunk of PHP related questions.

All questions and answers are "tagged" with their topics so that you can find the ones relevant to you. To browse questions tagged with PHP, visit http://stackoverflow.com/questions/tagged/php. You can also use the sites search facility, and to narrow you search to only PHP related answers add "[tag]" to your search. For example, if you want to search for questions about the `date` function, which is a common word in English and a common function name in many programming languages, search for "[php] date" to only get PHP specific information.

At the time of writing, there were 458,414 questions tagged with "php", 446 tagged with "php" AND "command-line", and 397 tagged with "php" AND "cli". The moderators are usually very quick at shutting down duplicate questions, so you can see from these numbers that there is a lot of relevant information available.

Other books

While you may think that this is the only book on PHP you will ever need, I have been told that there may be other PHP books available out there. While I can't recommend any specifically (other than those I have already noted in the relevant chapters), if you browse any big name bookseller you will find a plethora of PHP related titles. If do want to purchase another book, and would like to help support the PHP.net project, follow the links from the books section at http://php.net/support.php to a well known book store, and PHP.net will receive a commission for anything that you buy after following their link. Also, any books on PHP I publish after this one are totally and utterly worth buying and your career may be negatively affected if you don't do so.

Newsgroups

PHP has a set of official newsgroups listed and archived at http://news.php.net/ which cover a wide range of PHP topic areas. These may be worth a browse, and sometimes can elicit a response to queries (although some of the internals lists are definitely not for the faint-hearted).

PHP Subredit

The PHP "Subredit" on reddit.com at http://www.reddit.com/r/PHP/ is a mixture of PHP news, opinions, useful links and requests for help. Although usually genuinely interesting with helpful responses to questions, an occasional assortment of trolls and unhelpful/rude people can be found here as well.

PHP news sites

Although not usually good for direct help, PHP news sites and mailing lists can keep you up to date with essentials like security alerts, useful and interesting articles, and announcements of new projects, libraries and tutorials that you may not even know you needed yet! Some of the more popular ones are listed below.

PHP News Sources

PHPDeveloper http://phpdeveloper.org/

Planet PHP http://www.planet-php.net/

TechPortal http://techportal.inviqa.com/

PHP Weekly News http://www.phpweekly.com/

PHP Weekly http://phpweekly.info/

Appendix D : Interesting libraries, tools, articles and projects

Throughout this book you will have seen links to numerous libraries and tools, related to the topic in hand. This Appendix lists many more PHP related projects which didn't fit neatly into other sections of the book but are nevertheless interesting. Many show off the potential of PHP beyond just websites, and others provide libraries for tasks that are often performed with desktop, server or other longer-running software. Some are just cool! If you know of any that are relevant here, let me know and they can be included in future updates of the book.

Alternative programming styles

Pharen

Lisp like language for PHP
http://scriptor.github.io/pharen/

pEigthP

Basic Lisp implementation embedded in PHP
https://github.com/cninja/pEigthP#readme

Syngr

OO standard interface library to PHP core functions
https://github.com/hassankhan/Syngr

Functional Programming in PHP

Tutorial by Patkos Csaba

http://net.tutsplus.com/tutorials/php/functional-programming-in-php/

phpQuery

Like jQuery, but for PHP
http://code.google.com/p/phpquery/

PHP Linq

Mimics C# LINQ extension methods
http://phplinq.codeplex.com/

Phinq

Another C# LINQ emulation library
http://phplinq.codeplex.com/

PHPz

Functional programming library
http://blog.clement.delafargue.name/posts/2013-04-01-delicious-burritos-in-php-with-phpz.html

xhp

Facebook's brand of PHP which brings XML directly into the syntax
https://github.com/facebook/xhp

Machine learning, artificial intelligence and data analysis

Running Monte Carlo Simulations in PHP

Article by J Armando Jeronymo
http://www.sitepoint.com/running-monte-carlo-simulations-in-php/

PHP Clarke and Wright Algorithm

Class to solve truck routing problems
http://www.phpclasses.org/package/8135-PHP-Solve-a-truck-routing-problem-with-Clarke-Wright.html

PHP/ir

Website full of machine learning / information retrieval type algorithms in PHP
http://phpir.com/

cPerceptron

A Simple Perceptron neural network in PHP
https://github.com/gzanitti/cperceptron#readme

neural-network

Advanced multi-layer neural network
https://github.com/infostreams/neural-network

Learning Library for PHP

Assorted ML and AI algorithms
https://github.com/gburtini/Learning-Library-for-PHP

Finite-State Machine Library

Does what it says on the tin
https://github.com/chriswoodford/techne

K-cluster algorithm in php

K-cluster analysis algorithm
http://johntron.com/programming/k-cluster-algorithm-in-php/

SVM Extension

Support Vector Machine problem solver
http://www.php.net/manual/en/book.svm.php

Databases

Gladius DB

SQL database written in PHP
http://gladius.sourceforge.net/index.php

UnQLite Extension

SQLite-like NoSQL database

https://github.com/kjdev/php-ext-unqlite

The Underground PHP and Oracle Manual

Official, free, book from Oracle
http://www.oracle.com/technetwork/topics/php/underground-php-oracle-manual-098250.html

Introduction to Document Databases with MongoDB

Tutorial by Derick Rethans
http://derickrethans.nl/introduction-to-document-databases.html

Handling Hierarchical Data in MySQL and PHP

Tutorial by Voja Janjic
http://www.phpbuilder.com/articles/databases/mysql/handling-hierarchical-data-in-mysql-and-php.html

Natural language

Article readability stats with PHP

Discussion of, and code for, readability algorithms
http://www.maratz.com/blog/archives/2012/07/26/article-readability-stats-with-php/

Term Extractor

Language term identification and extraction

http://code.fivefilters.org/term-extraction

Counting Syllables and Detecting Rhyme in PHP

Tutorial by Cameron McKay
http://cdmckay.org/blog/2012/08/15/counting-syllables-and-detecting-rhyme-in-php/

nlptools

Natural Language analysis library
http://nlptools.atrilla.net/web/

Talking Machine

Learns a language idiom and generates idiomatic text
http://www.phpclasses.org/package/3848-PHP-Learn-an-idiom-and-generate-words-in-that-idiom.html

Graphics and imaging

Instafilter

Instagram like filters for PHP
https://github.com/fbf/instafilter#readme

OpenCV for PHP

Bindings for the OpenCV image recognition/processing library
https://github.com/mgdm/OpenCV-for-PHP#readme

phpColors

Colour manipulation methods
http://mexitek.github.io/phpColors/

Libpuzzle

Image comparison library
http://www.pureftpd.org/project/libpuzzle/php

Unicode

Bringing Unicode to PHP with "Portable UTF-8"

Tutorial by Hamid Sarfraz
http://www.sitepoint.com/bringing-unicode-to-php-with-portable-utf8/

Audio

Create PHP Voice Recognition Apps on the Cheap

Tutorial by Jason Gilmore
http://www.phpbuilder.com/columns/Voice_Recognition/PHP_Voice_Recognition_1-12-2012.php3

Event driven PHP

React PHP

Node.js type framework for PHP
http://reactphp.org/

Phastlight

Another Node.js like framework
https://github.com/phastlight/phastlight/

Database connection pooling with PHP and React

Tutorial by Gonzalo Ayuso
http://gonzalo123.wordpress.com/2012/05/21/database-connection-pooling-with-php-and-react-node-php/

XPSPL

Signal and event processing library
https://github.com/prggmr/XPSPL/

PHP internals

PHP's Source Code For PHP Developers

article by Anthony Ferrara
http://blog.ircmaxell.com/2012/03/phps-source-code-for-php-developers.html

PHP Parser

A full PHP Parser written in PHP

https://github.com/nikic/PHP-Parser#php-parser

PHP Manipulator

Library for analysing and modifying PHP code
https://github.com/schmittjoh/PHP-Manipulator

How to add new (syntactic) features to PHP

Article by Nikita Popov
http://nikic.github.io/2012/07/27/How-to-add-new-syntactic-features-to-PHP.html

Introspection and Reflection in PHP

Article by Octavia Anghel
http://phpmaster.com/introspection-and-reflection-in-php/

Website/service APIs

Stack Exchange API

Stack.php library (unofficial)
http://stackphp.quickmediasolutions.com/

Access Dropbox Using PHP

tutorial by Vito Tardia
http://www.sitepoint.com/access-dropbox-using-php/

Goutte

A web scraping library, ideal for sites without APIs
https://github.com/fabpot/Goutte#readme

Security related

PHP Secure Communications Library

Pure PHP implementations of many security/encryption algorithms
http://phpseclib.sourceforge.net/documentation

Javascript

Phype

Experimental Javascript-based VM for PHP
http://code.google.com/p/phype/

php.js VM

Another javascript based PHP VM
http://phpjs.hertzen.com/

php.js

Library for using PHP-like functions in Javascript
http://phpjs.org/functions/

Servers

Pancake

Accelerated PHP http server written in PHP
https://github.com/pp3345/Pancake

Nanoweb

Another http server written in PHP
http://nanoweb.si.kz/

Programming

PHPCPD

Copy/Paste detector for PHP code
https://github.com/sebastianbergmann/phpcpd/#readme

Binary parsing with PHP

Article by Igor Wesome
https://igor.io/2012/09/24/binary-parsing.html

Financial

Trader

Technical analysis for traders
http://www.php.net/manual/en/book.trader.php

Appendix E : Integrated Development Environments for PHP

The following is a list of IDE's (Integrated Development Environments) aimed at the PHP market. The purpose is to help you explore the options available, none are specifically recommended for, or dedicated to, CLI/non-web programming. The list is broken into two parts, first the open source options and then the commerical offerings.

If you know of one that's not listed below, please let me know (author@phpbeyondtheweb.com).

Opensource

Eclipse PDT

Website : http://projects.eclipse.org/projects/tools.pdt

Netbeans PHP

Website : https://netbeans.org/features/php/

Aptna Studio

Website : http://www.aptana.org/products/studio3

Commercial

Zend Studio

Website : http://www.zend.com/products/studio/

PhpStorm

Website : http://www.jetbrains.com/phpstorm/

PhpED

Website : http://www.nusphere.com/products/phped.htm

phpDesigner

Website : http://www.mpsoftware.dk/phpdesigner.php

Komodo IDE (and free Komodo Edit)

Website : https://www.activestate.com/komodo-ide
Website : http://www.activestate.com/komodo-edit

CodeLobster

Website : http://www.codelobster.com/

PHPEdit

Website : http://www.phpedit.com/en

Rapid PHP

Website : http://www.rapidphpeditor.com/

PHP Studio

Website : http://www.cayoren.com/

Kodiak for iPad

Website http://www.becomekodiak.com/

Visual Studio Add-ons

Website : VS.php http://www.jcxsoftware.com/
Website : PHP Tools for VS http://www.devsense.com/products/php-tools

SublimeText

Note : A (very good) text editor rather than an IDE, but the following article shows how to set it up as such using extensions :
Website : http://www.sublimetext.com/
Article : http://blog.stuartherbert.com/php/2012/02/28/setting-up-sublime-text-2-for-php-development/

Appendix F : Changelog

This book is an evolving project. This section will detail the changes made each time the book is updated, so if you've already read through a previous version of the book, you can see at a glance what has changed and what new information has been included.

Current Version : 1.0

Changes in Version 1.0

- This is the first release, so everything is new!