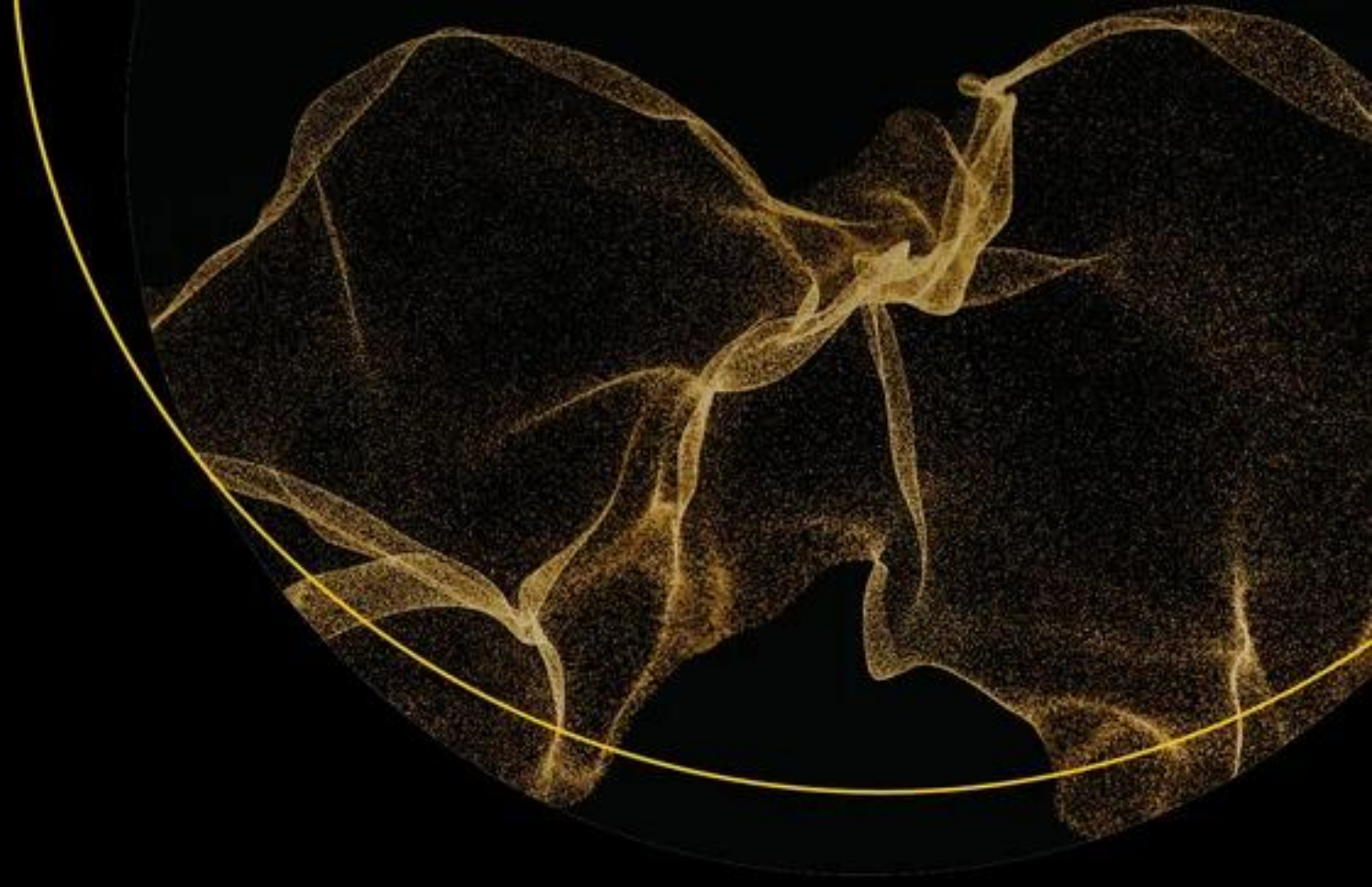

PHP 8 Objects, Patterns, and Practice: Volume 2

Mastering Essential Development Tools

Seventh Edition

Matt Zandstra

Apress®

PHP 8 Objects, Patterns, and Practice: Volume 2

Mastering Essential Development Tools

Seventh Edition

Matt Zandstra

Apress®

PHP 8 Objects, Patterns, and Practice: Volume 2: Mastering Essential Development Tools, Seventh Edition

Matt Zandstra
Brighton, UK

ISBN-13 (pbk): 979-8-8688-0778-7 ISBN-13 (electronic): 979-8-8688-0779-4
https://doi.org/10.1007/979-8-8688-0779-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: James Robinson Prior
Editorial Assistant: Jacob Shmulewitz

Cover designed by eStudioCalamar

Cover image designed by Pawel Czerwinski on Unsplash

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a Delaware LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

To Louise. Still the whole point.

Table of Contents

About the Author

Matt Zandstra has worked as a web programmer, consultant, and writer for over two decades. In addition to this book, he is the author of *Sams Teach Yourself PHP in 24 Hours* (three editions) and a contributor to *DHTML Unleashed*. He has written articles for *Linux Magazine*, *Zend*, IBM DeveloperWorks, and *PHP Architect* magazine and also writes fiction.

Matt was a senior developer/tech lead at Yahoo and API tech lead at LoveCrafts. He now runs an agency that advises companies on their architectures and system management and develops systems primarily with PHP, Python, and Java.

Acknowledgments

I have benefited from the support of many people while working on this edition. But as always, I must also look back to the book's origins. I tried out some of this book's underlying concepts in a talk in Brighton, back when we were all first marveling at the shiny possibilities of PHP 5. Thanks to Andy Budd, who hosted the talk, and to the vibrant Brighton developer community. Thanks also to Jessey White-Cinis, who was at that meeting and who put me in touch with Martin Streicher at Apress.

Once again, this time around, the Apress team has provided enormous support, feedback, and encouragement. I am lucky to have benefited from such professionalism.

I'm delighted that my friend and colleague, Paul Tregoing, agreed again to act as technical reviewer despite many other projects including his own book. This edition has greatly benefited from Paul's knowledge, insight, and attention to detail – many thanks Paul!

Thanks and love to my wife, Louise. The production of this book has coincided with the university careers of my children Holly and Viola who have struggled with their own deadlines and creative blocks. Thanks are due to them for keeping me company at the kitchen table as we found our separate ways together!

I write to music, and in previous editions of this book, I remembered the great DJ, John Peel, champion of the underground and the eclectic. The soundtrack for this edition was largely provided by BBC Radio 3's *Late Junction* and Six Music's *Freak Zone* both played on a loop. Thanks to the DJs and musicians who continue to keep things weird.

Introduction

When I decided to learn to program, I went out to a bookshop on Tottenham Court Road in London and bought myself a book about Perl. Excited, I started building an application before I'd even finished the fourth chapter, which is how I managed to write a working forum application without yet knowing how to define a subroutine. That's another story, though (one involving very very big loops). Once I had finished reading the book and rounded out my understanding, I felt sure I had learned everything I needed to know. I was ready.

It was only then that I began to perceive new gaps in my knowledge. Some of it was relatively easy to fix. I was able to find books on the Unix shell and CGI to address the most obvious chasm. But, even after that, I had questions. Where would I store my code? How would I collaborate with other developers without overwriting their work or having my own work clobbered? How should I source libraries and manage dependencies? What about development environments? What was the best way to deploy my code? How could I test the systems I built?

The answers could be found online – though search in those days was rudimentary. I spent a lot of time on the Usenet search engine DejaNews and pieced together a working practice. In retrospect, it was somewhat suboptimal in all sorts of ways, but it was enough to help me get systems into the world. Over the years, I joined teams and learned from knowledgeable people. I searched out more books. My practice improved.

The coverage gap was not the fault of that Perl book's author. He did a brilliant job within the book's remit. But when I came to pitch a book about coding with objects in PHP, I thought about the extent of that remit. Although I wanted to write about objects and design, I did not want to do so in an absolute vacuum. I wanted to write a *practical* book – a book that helped with the last yard of development too. So I added *practice* to my *PHP*, *objects*, and *patterns* proposal.

PHP 8 Objects, Patterns, and Practice has evolved over the years and grown from a slim volume to a full on doorstopper. When the time came for a seventh edition, I had additions in mind as usual. As well as revising the existing topics covered by the *Practice* section, I wanted to include more on continuous integration, to add new subjects such as PHPStan, Docker, Ansible, and command-line PHP scripting.

Of course, that was impossible. The sixth edition was huge. There was no way we could create a seventh edition that was even larger. Unless, of course, we broke the book into two volumes. So that is what we did. I hope you enjoy the result.

CHAPTER 1

Good (and Bad) Practice

In the previous volume, I focused on coding, concentrating particularly on the role of design in building flexible and reusable tools and applications. Development doesn't end with code, however. It is possible to come away from books and courses with a solid understanding of a language, yet still encounter problems when it comes to running and deploying a project.

In this volume, I will move beyond code to cover some of the tools and techniques that form the underpinnings of a successful development process. This chapter will cover the following:

- *Third-party packages*: Where to get them and when to use them.
- *Deployment*: Pushing your code across servers, applying configuration.
- *Version control*: Bringing harmony to the development process.
- *Documentation*: Writing code that is easy to understand, use, and extend.
- *Unit testing*: A tool for automated bug detection and prevention.
- *Standards*: Why it's sometimes good to follow the herd.
- *Development environments*: Every developer needs a lab of their own. A coder should be able to work safely with a system that resembles the production environment, no matter their hardware or OS.
- *Scripting the command line*: PHP may be known as a web technology, but it can be just as powerful on the command line.
- *Continuous integration*: Using this practice and set of tools to automate project builds and tests as well as to warn of problems as they occur.

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_1

Beyond Code

When I first graduated from working on my own and took a place in a development team, I was astonished at how much stuff other developers seemed to have to know. Good-natured arguments simmered endlessly over issues of vital-seeming importance: Which is the best text editor? Should the team standardize on an integrated development environment? Should we impose a coding standard? How should we test our code? Should we document as we develop? Sometimes, these issues seemed more important than the code itself, and my colleagues seemed to have acquired their encyclopedic knowledge of the domain through some strange process of osmosis.

The books I had read on PHP, Perl, and Java certainly didn't stray from the code itself to any great extent. As I discussed in the previous volume, many books about programming rarely diverge from their tight focus on functions and syntax to take in code design. If design is off topic, you can be sure that wider issues such as version control and testing are rarely discussed. This is not a criticism – if a book professes to cover the main features of a language, it should be no surprise that this is principally what it does.

In learning about code, however, I found that I had neglected many of the mechanics of a project's day-to-day life. I discovered that some of these details were critical to the success or failure of projects I helped develop. In this chapter, and in more detail in coming chapters, I will look beyond code to explore some of the tools and techniques on which the success of your projects may depend.

Borrowing a Wheel

When faced with a challenging but discrete requirement in a project (the need to parse a particular format, perhaps, or to use a novel protocol in talking to a remote server), there is a lot to be said for building a component that addresses the need. It can also be one of the best ways to learn your craft. In creating a package, you gain insight into a problem and file away new techniques that might have wider application.

You invest at once in your project and in your own skills. By keeping functionality internal to your system, you can save your users from having to download third-party packages. Occasionally, too, you may sidestep thorny licensing issues. There's nothing like the sense of satisfaction you can get when you test a component you designed yourself and find that, wonder of wonders, it works – it does exactly what you wrote on the tin.

There is a dark side to all this, of course. Many packages represent an investment of thousands of person-hours: a resource that you may not have on hand. You may be able to address this by developing only the functionality needed specifically by your project, whereas a third-party tool might fulfill a myriad of other needs as well. The question remains, however: If a freely available tool exists, why are you squandering your talents in reproducing it? Do you have the time and resources to develop, test, and debug your package? Might not this time be better deployed elsewhere?

I am one of the worst offenders when it comes to wheel reinvention. Picking apart problems and inventing solutions to them is a fundamental part of what we do as coders. Getting down to some serious architecture is a more rewarding prospect than writing some glue to stitch together three or four existing components. When this temptation comes over me, I remind myself of projects past. Although the choice to build from scratch has never killed a project in my experience, I have seen it devour schedules and murder profit margins. There I sit with a manic gleam in my eye, hatching plots and spinning class diagrams, failing to notice as I obsess over the details of my component that the big picture is now a distant memory.

Now, when I map out a project, I try to develop a feel for what belongs inside the code base and what should be treated as a third-party requirement. For example, your application may generate (or read) an RSS feed, and you may need to validate email addresses and automate mailouts, authenticate users, or read from a standard-format configuration file. All of these needs can be fulfilled by external packages.

In previous versions of this book, I suggested that PEAR (PHP Extension and Application Repository) was the way to go for packages. Times change, though, and the PHP world has very definitely moved to the Composer dependency manager and its default repository, Packagist (`https://packagist.org`). Because Composer manages packages on a per-project basis, it is less prone to the dreaded dependency hell syndrome (where different packages require incompatible versions of the same library). Besides, the fact that all the action has moved to Composer/Packagist means that you're more likely to find what you're looking for there. What's more, many of the PEAR packages are available through Packagist (`https://packagist.org/packages/pear/`).

So, once you have defined your needs, your first stop should be the Packagist site. You can then use Composer to install your package and to manage package dependencies. I will cover Composer in more detail in Chapter 5.

To give you some idea of what's available using Composer and Packagist, here are just a few of the things you can do with the packages you'll find there:

- Cache output with `pear/cache_lite`
- Test the efficiency of your code with the `athletic/athletic` benchmark library
- Abstract the details of database access with `doctrine/dbal`
- Extract RSS feeds with `simplepie/simplepie`
- Access REST APIs with `guzzlehttp/guzzle`
- Parse configuration file formats with `symfony/config`
- Parse and manipulate URLs with `league/uri`

The Packagist website provides a powerful search facility. You may find packages that address your needs there, or you may need to cast your net wider using a search engine. Either way, you should always take time to assess existing packages before setting out to potentially reinvent that wheel.

The fact that you have a need – and that a package exists to address it – should not be the start and end of your deliberations. Although it is preferable to use a package where it will save you otherwise unnecessary development, in some cases, it can add overhead without real gain. You may find that a clean and focused class will get the job done without bloat or that PHP provides a decent built-in solution. Nonetheless, many programmers, myself included, often place too much emphasis on the creation of original code, sometimes to the detriment of their projects.

Note The unwillingness to use third-party tools and solutions is often built-in at the institutional level. This tendency to treat external products with suspicion is sometimes known as the *not invented here* syndrome. As a further note, the technical reviewer and fellow sf fan Paul Tregoing points out that *Not Invented Here* is also the name of a ship in Iain M. Banks' *Culture* series.

This emphasis on authorship may be one reason that there often seems to be more creation than actual use of reusable code.

Effective programmers see original code as just one of the tools available to aid them in engineering a project's successful outcome. Such programmers look at the resources they have at hand and deploy them effectively. If a package exists to take some strain, then that is a win. To steal and paraphrase an aphorism from the Perl world: good coders are lazy.

Playing Nice

The truth of Sartre's famous dictum that "Hell is other people" is proved on a daily basis in some software projects. This might describe the relationship between clients and developers, symptomized by the many ways that lack of communication leads to creeping features and skewed priorities. But the cap fits, too, for happily communicative and cooperative team members when it comes to sharing code.

As soon as a project has more than one developer, version control becomes a critical issue. A single coder may work on code in place, saving a copy of her working directory at key points in development. Introduce another programmer to the mix, and this strategy breaks down in minutes. If the new developer works in the same development directory, then there is a real chance that one programmer will overwrite the work of his colleague when saving, unless both are very careful to always work on different files.

Alternatively, our two developers can each take a version of the code base to work on separately. That works fine until the moment comes to reconcile the two versions. Unless the developers have worked on entirely different sets of files, the task of merging two or more development strands rapidly becomes an enormous headache.

This is where Git, Subversion, and similar tools come in. Using a version control system, you can check out your own version of a code base and work on it until you are happy with the result. You can then update your version with any changes that your colleagues have been making. The version control software will automatically merge these changes into your files, notifying you of any conflicts it cannot handle. Once you have tested this new hybrid, you can save it to the central repository, making it available to other developers.

Version control systems provide you with other benefits. They keep a complete record of all stages of a project, so you can roll back to, or grab a snapshot of, any point in the project's lifetime. You can also create branches, so that you can maintain a public release at the same time as a bleeding-edge development version.

Once you have used version control on a project, you will not want to attempt another without it. Working simultaneously with multiple branches of a project can be a conceptual challenge, especially at first, but the benefits soon become clear. Version control is just too useful to live without. I cover Git in Chapter 17.

Note The current edition of this book was written and edited in plain text (Markdown format) using Git as a collaboration tool.

Giving Your Code Wings

Have you ever seen your code grounded because it is just too hard to build? This is especially a danger for projects that are developed in place. Such projects settle into their context, with passwords and directories, databases, and helper application invocations programmed right into the code. Deploying a project of this kind can be a major undertaking, with teams of programmers picking through source code to amend settings, so that they fit the new environment.

This problem can be eased to some degree by providing a centralized configuration file or class so that settings can be changed in one place. But even then, deployment can be a chore. The difficulty or ease of build will impede or encourage frequent deployments during development.

As with any repetitive and time-consuming task, build should be automated. A deployment tool can determine default values for install locations, check and change permissions, create databases, and initialize variables, among a dizzying range of other tasks. In fact, such a tool should be able to do just about anything you need to get an application from a source directory in a distribution to full deployment.

Cloud products such as Amazon's AWS CodePipeline have made it possible to create test and staging environments as needed. Good deployment solutions are essential in order to take full advantage of these resources. It's no good being able to provision a server on an automated basis if you can't also deploy your system on the fly.

Of course, the most straightforward way to move code is by using a version control system like Git. You can also acquire third-party dependencies (or your own packages) using Composer which, together with the Packagist repository, provides access to thousands of libraries. Powerful as these tools are, they do not cover configuration management, and even for simple use cases, they'd need to be orchestrated. I cover Composer in Chapter 5.

In Chapter 10, I introduce Ansible. This powerful deployment tool can install your code onto multiple servers, typically utilizing Git to acquire core code and Composer for third-party dependencies. Build is about much more than the process of placing file A in location B, however. Ansible can manage your system's secrets and general configuration. It can also run tests and other quality control tools or even manage server-level provisioning.

Standards

I mentioned previously that this book shifted its focus from PEAR to Composer. Is this because Composer is much better than PEAR? I do love lots of things about Composer, and these might (in fact, probably would) swing the decision on their own. The principal reason the book shifted a couple of editions back, though, is that everyone else had shifted. Composer has become the standard for dependency management. That is crucial because it means that when I find a package at Packagist, I am also likely to find all its dependencies and related packages. I'll even find many of the PEAR packages there.

Choosing a standard for dependency management, then, ensures availability and interoperability. But standards apply beyond packages and dependencies to the ways that systems work and to the ways that we code. If we agree on protocols, then our systems and teams can integrate seamlessly with one another. And, as more and more components mix across more and more systems, that is increasingly essential.

Where a definitive way of handling, say, logging, is needed, it is obviously ideal that we adopt the best protocol. But the quality of the recommendation (which will dictate formats, log levels, etc.) is possibly less important than the fact that we all comply with it. It's no good implementing the best standard if you're the only person doing it.

In Chapter 15, I discuss standards in more detail with particular reference to a set of recommendations managed by the PHP-FIG group. These PSRs (PHP Standards Recommendations) cover everything from caching to security. In the chapter, I will focus on PSR-1 and PSR-12, recommendations which address the thorny issue of coding style (where do you like to put your braces? And how do you feel about someone else telling you to change the way you do it?). Then, I'll move on to the absolute boon of PSR-4, which covers autoloading (support for PSR-4 is another area in which Composer excels).

Vagrant and Docker

What operating system does your team use? Some organizations mandate a particular combination of hardware and software, of course. Often, though, there will be a mix. One developer may have a development machine running Fedora. Another might swear by his MacBook, and a third may stick with his Alienware Windows box (he probably likes it for gaming).

Chances are the production system will run on something else entirely – Debian, perhaps.

It can be a pain getting a system to work across multiple platforms, and it can be a risk if none of those platforms resemble the production system. You really don't want to discover issues related to the production OS after you've gone live. In practice, of course, you'll likely deploy to a staging environment first. Even so, wouldn't it be better to catch these problems early?

Vagrant is a technology that uses virtualization to give all team members a development environment that is as close as possible to production. Getting up and running should be as simple as invoking a command or two, and, best of all, everyone can stick with their favorite machines and distributions (I'm a Fedora guy, for the record).

Although Vagrant is a fantastic tool, it is also both monolithic and resource hungry. In order to get an environment up and running, you must create an entire server environment running on a virtual machine. Docker provides a powerful lightweight alternative. Instead of running a single silo running its own kernel, Docker allows you to deploy multiple small containers – one for each of your system's services. Because, behind the scenes, a container runs directly on the host machine's operating system (on Linux, at least), it is easy to deploy, runs fast, and is relatively sparing on resources. By orchestrating such containers, you can build a powerful development environment very quickly.

I cover Vagrant in Chapter 8 and Docker in Chapter 9.

Testing

When you create a class, you are probably pretty sure that it works. You will, after all, have put it through its paces during development. You'll also have run your system with the component in place, checking that it integrates well and that your new functionality is available and performing as expected.

Can you be sure that your class will carry on working as expected, though? That might seem like a silly question. After all, you've checked your code once; why should it stop working arbitrarily? Well, of course, it won't; nothing happens arbitrarily, and if you never add another line of code to your system, you can probably breathe easy. If, on the other hand, your project is active, then it's inevitable that your component's context will change and highly likely that the component itself will be altered in any number of ways.

Let's look at these issues in turn. First, how can changing a component's context introduce errors? Even in a system where components are nicely decoupled from one another, they remain interdependent. Objects used by your class return values, perform actions, and accept data. If any of these behaviors change, the effects on the operation of your class might cause the kind of error that's easy to catch – the kind where your system falls over with a convenient error message that includes a file name and line number. Much more insidious, though, is the kind of change that does not cause an engine-level error but nonetheless confuses your component. If your class makes an assumption based on another class's data, a change in that data might cause it to make a wrong decision. Your class is now in error and without a change to a line of code.

And it's likely that you will go on altering the class you've just completed. Often, these changes will be minor and obvious – so minor, in fact, that you won't feel the need to run through the careful checks you performed during development. You'll have probably forgotten them all, anyhow, unless you kept them in some way (perhaps commented out at the bottom of your class file). Small changes, though, have a way of causing large unintended consequences – consequences that might have been caught had you thought to put a test harness in place.

A test harness is a set of automated tests that can be applied to your system as a whole or to its individual classes. Well deployed, a test harness helps you to prevent bugs from occurring and from *recurring*. A single change may cause a cascade of errors, and the test harness can help you to locate and eliminate these. This means you can make changes with some confidence that you are not breaking anything. It is quite satisfying to make an improvement to your system and then see a list of failed tests. These are all errors that might have propagated within your system and which now it won't have to suffer in production.

Command-Line Scripting

Typically, any project you work on sprouts a thicket of command-line scripts. You'll need to clear down databases, set up test data, run periodic clean up or data population tasks, and send out mail shots. The list tends to just grow and grow.

Since, by definition, you're already working with PHP, the language can be an excellent choice for scripting on the shell. You're likely to have the PHP interpreter to hand, after all. You can achieve pretty much everything with PHP that you can with a shell script, with the added bonus that you're deploying a familiar language. Thanks to Composer, you have access to thousands of powerful libraries, making it easy to build scripts for almost any purpose. What's more, if you are developing a web application in PHP, you can easily integrate your PHP shell scripts into it, giving your command-line utilities seamless access to your software API, and, because you will be using application configuration, you can take easy advantage of system components like databases or web services.

Of course, there are considerations to take into account when building command-line scripts in any language. You need to manage options and positional arguments, for example. Given the potential variations in argument forms and requirements, parsing these can be surprisingly challenging. For more complex scripts, you may want to interactively prompt the user for input or even accept piped data. You also need to consider how to communicate error conditions and usage information to the user as well as managing general output and debug messaging.

I cover all this and more in Chapter 11.

Continuous Integration

Have you ever created a schedule that made everything okay? You start with an assignment: a code commission maybe or a school project. It's big and scary, and failure lurks. But you get out a sheet of paper, and you slice it up into manageable tasks. You determine the books to read and the components to write. Maybe you highlight the tasks in different colors. Individually, none of the tasks is actually that scary, it turns out. And gradually, as you plan, you conquer the deadline. As long as you do a little bit every day, you'll be fine. You can relax.

Sometimes, though, that schedule takes on a talismanic power. You hold it up like a shield to protect yourself from doubt and from the creeping fear that perhaps this time you'll crash and burn. And it's only after several weeks that you realize the schedule is not magic on its own. You actually have to do the work, too. By then, of course, lulled by the schedule's reassuring power, you have let things slide. There's nothing for it but to make a new schedule. This time, it will be less reassuring.

Testing and building are like that, too. You have to run your tests. You have to build your projects and build them in fresh environments regularly; otherwise, the magic won't work.

And if writing tests is a pain, running them can be a chore, especially as they gain in complexity and failures interrupt your plans. Of course, if you were running them more often, you'd probably have fewer failures, and those you did have would stand a good chance of relating to new code that's fresh in your mind.

It's easy to get comfortable in a sandbox. After all, you've got all your toys there: little scriptlets that make your life easy, development tools, and useful libraries. The trouble is your project may be getting too comfortable in your sandbox, too. It may begin to rely on uncommitted code or dependencies that you have left out of your build files. That means it's broken anywhere else but where you work.

The only answer is to build, build, and build again. And do it in a reasonably virgin environment each time.

Of course, it's all very well to advise this; it's quite another matter to do it. Coders as a breed tend to like to code. They want to keep the meetings and the housekeeping to a minimum. That's where continuous integration (CI) comes in. CI is both a practice and a set of tools to make the practice as easy as it possibly can be. Ideally, builds and tests should be entirely automatic or at least launchable from a single command or click. Any problems will be tracked, and you will be notified before an issue becomes too serious. I will talk more about CI in Chapter 12.

Summary

A developer's aim is always to deliver a working system. Writing good code is an essential part of this aim's fulfillment, but it is not the whole story.

In this chapter, I introduced dependency management with Composer and Packagist. I also covered version control. Once you can install your code and the components upon which it depends, you'll need to deploy the system to staging and

production environments, managing configuration so that it works appropriately in each context. I introduced Ansible, a tool designed for the purpose. I also discussed Docker and Vagrant, two approaches to creating production-like development environments. I looked at quality control, taking in both standards and automated testing. Even if your project is web-based, it will likely require scripts for automating development tasks or for performing scheduled work. I briefly explored some issues relating to command-line scripting with PHP. Finally, I introduced CI, a set of tools to automate build and testing.

CHAPTER 2

Generating Documentation with phpDocumentor

Remember that tricky bit of code? The method that called a legacy library and returned an array of objects which were indexed by product IDs. Or was it product names? Even with argument and return type declarations in modern PHP, the behavior of your components can remain obscure from the standpoint of source code alone.

Coding is a messy and complex business, and it's hard to keep track of the way your systems work and what needs doing. The problem becomes worse when you add more programmers to the project. Whether you need to signpost potential danger areas or fantastic features, documentation can help you. For a large code base, documentation or its absence can make or break a project.

This chapter will cover

- *The phpDocumentor application*: Installing phpDocumentor and running it from the command line
- *Documentation syntax*: The DocBlock comment and documentation tags
- *Documenting your code*: Using DocBlock comments to provide information about classes, properties, and methods
- *Creating links in documentation*: Linking to websites and to other documentation elements

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_2

Why Document?

Programmers love and loathe documentation in equal measure. When you are under pressure from deadlines, with managers or customers peering over your shoulders, documentation is often the first thing to be jettisoned. The overwhelming drive is to get results. Write elegant code, certainly (though that can be another sacrifice), but with a code base undergoing rapid evolution, documentation can feel like a real waste of time. After all, you'll probably have to change your classes several times in as many days. Of course, everyone agrees that it's desirable to have good documentation. It's just that no one wants to undermine productivity in order to make it happen.

Imagine a very large project. The code base is enormous, consisting of very clever code written by very clever people. The team members have been working on this single project (or set of related subprojects) for over five years. They know each other well, and they understand the code absolutely. Documentation is sparse, of course. Everyone has a map of the project in their heads, and a set of unofficial coding conventions that provide clues as to what is going on in any particular area. Then the team is extended. The two new coders are given a good basic introduction to the complex architecture and thrown in. This is the point at which the true cost of undocumented code begins to tell. What would otherwise have been a few weeks of acclimatization soon becomes months. Confronted with an undocumented class, the new programmers are forced to trace the arguments to every method, track down every referenced global, and check all the methods in the inheritance hierarchy. And with each trail followed, the process begins again. If, like me, you have been one of those new team members, you soon learn to love documentation.

Lack of documentation costs. It costs in time, as new team members join a project or existing colleagues shift beyond their area of specialization. It costs in errors as coders fall into the traps that all projects set. Methods which are designed for use in a specific context are invoked from the wrong component. A function declares that it returns an array or null – but the logic which determines which will get furnished is unclear, so a new coder makes a best guess. Functionality that already exists is needlessly recreated.

Documentation is a hard habit to get into because you don't feel the pain of neglecting it straightaway. Documentation needn't be difficult, though, if you work at it as you code. This process can be significantly eased if you add your documentation in the source itself as you code. You can then run a tool to extract the comments into neatly formatted web pages. This chapter is about just such a tool.

phpDocumentor (https://www.phpdoc.org/) was originally based on a Java tool called JavaDoc. Both systems extract special comments from source code, building sophisticated application programming interface (API) documentation from both the coder's comments and the code constructs they find in the source.

Installation

Unusually for a PHP tool, Composer is *not* the recommended way to install phpDocumentor. The project maintainers warn that installing with Composer brings with it a high probability of dependency conflicts so, for that reason, probably the easiest way to get up and running is the phar archive.

```
# Get the archive
$ wget https://phpdoc.org/phpDocumentor.phar

# move somewhere central
$ mv phpDocumentor.phar ~/bin/phpdoc

# make it executable
$ chmod 755 ~/bin/phpdoc
```

I downloaded the file `phpDocumentor.phar` and saved it to my local `bin/` directory. I also renamed it to `phpdoc`, partly because that's quicker to type and partly because it is a common practice. I ensured that the archive was runnable using the `chmod` command.

You can also use the official Docker image:

```
$ docker run --rm -v ${PWD}:/data phpdoc/phpdoc:3
```

This assumes you want to run phpDocumentor against your current directory. If that's not the case, change `${PWD}` to point to a more relevant subdirectory.

Generating Documentation

It might seem odd to generate documentation before we have even written any, but phpDocumentor parses the code structures in our source code, so it can gather information about your project before you even start.

I am going to document aspects of an imaginary project called "megaquiz." It consists of two directories, `command` and `quiztools`, which contain class files. These are also the names of packages in the project.

By default, phpDocumentor will run on the current working directory (though you have already seen that you can specify a different directory with Docker's `-v` flag). It is probably more useful, though, to specify your source and target directories.

Let's run it:

```
$ phpdoc \
    --title=megaquiz \
    --target=src/ch15/docs \
    --directory=src/ch15/megaquiz
```

The `--directory` flag denotes the directory whose contents you intend to document (you can also use the single letter flag `-d` with no equals sign for its argument). `--target` (or `-t`) denotes your target directory (the directory to which you wish to write the documentation files). Use `--title` to set a project title.

Here is the command-line output from the previous command:

```
phpDocumentor v3.4.3

Parsing files
 8/8 [============================] 100%
Applying transformations (can take a while)

All done in 0 seconds!
```

Now, in my specified documentation directory at `src/ch15/docs`, I should find my documentation generated as HTML files. I can open `index.html` to find a surprising amount of detail. Because my classes are namespaced, my classes are already organized into a package-like structure.

You can see the `popp\ch15\megaquiz\command` classes, for example, in Figure 2-1.

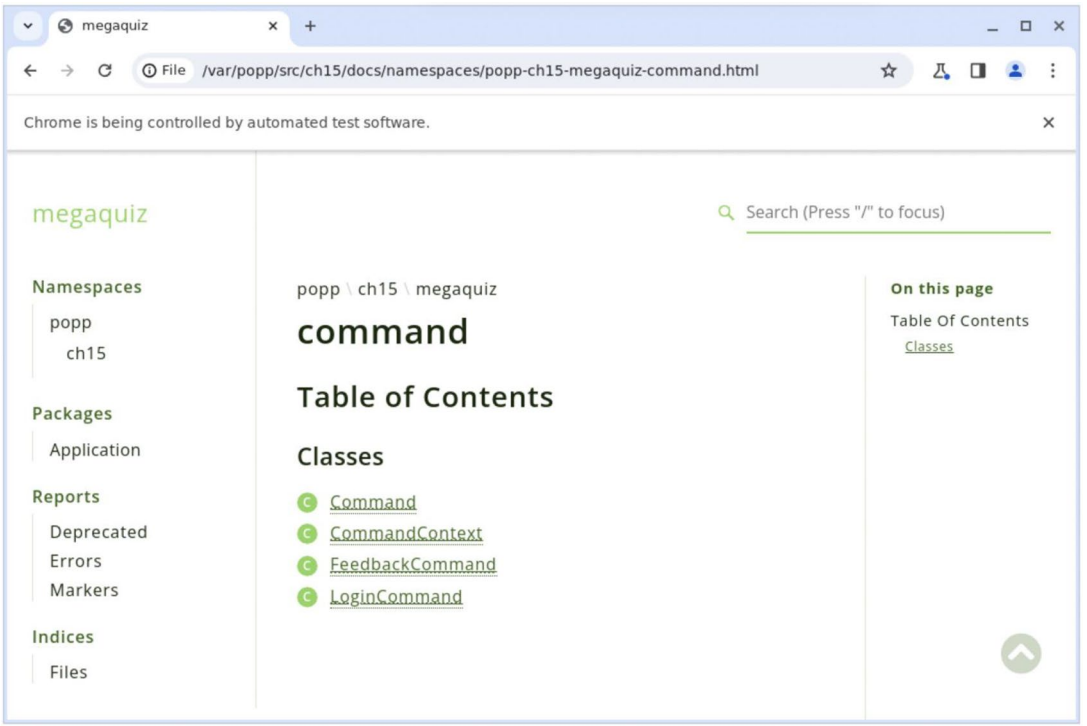

Figure 2-1. *phpDocumentor renders classes by namespace*

Note phpDocumentor supports a `@package` tag which you can use to apply logical package categories to your documented classes. If you're defining namespaces in your project, however, using `@package` as well can represent a needless maintenance overhead and may cause confusion.

As you can see, phpDocumentor shows all the classes in the selected namespace (`popp\ch15\megaquiz\command`). If any had been defined, it would also show any functions, interfaces, or traits. The class names are all hyperlinks. In Figure 2-2, you can see some of the documentation for the `Command` class.

phpDocumentor is smart enough to recognize that `Command` is an abstract class. Notice also that it has reported both the name and the type of the argument required by the `execute()` method as well as its return type.

Because this level of detail alone is enough to provide an easily navigable overview of a large project, it is a huge improvement over having no documentation at all. However, I can improve it further by adding comments to my source code.

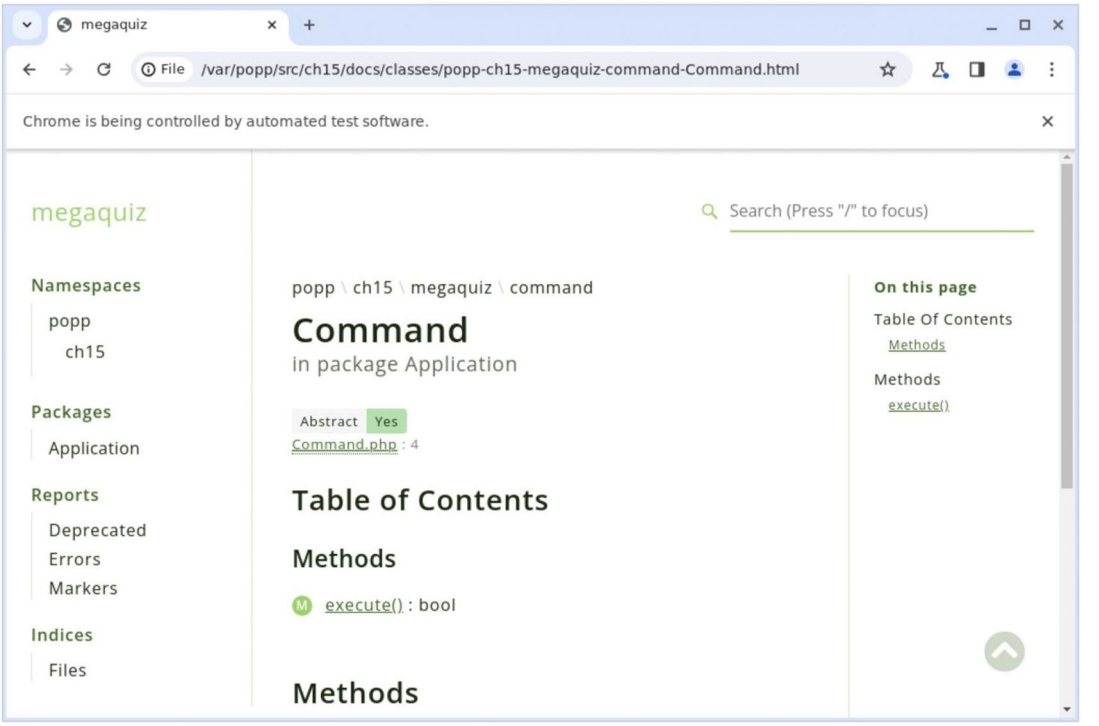

Figure 2-2. *Default documentation for the* `Command` *class*

DocBlock Comments

DocBlock comments are specially formatted to be recognized by a documentation application. They take the form of standard multiline comments. Standard, that is, with the single addition of an asterisk to each line within the comment:

```
/**
 * My DocBlock comment
 */
```

phpDocumentor is designed to expect special content within DocBlocks. This content includes normal text descriptive of the element to be documented (for our purposes, a file, class, method, or property). It also includes special keywords called *tags*. Tags are defined using the at sign (`@`) and may be associated with arguments. So the following DocBlock placed at the top of a class tells phpDocumentor about the author:

```
/**
 * @author Bob Bobson
 */
```

If I add this comment to classes in my project, phpDocumentor will include attribution in its output as you can see in Figure 2-3.

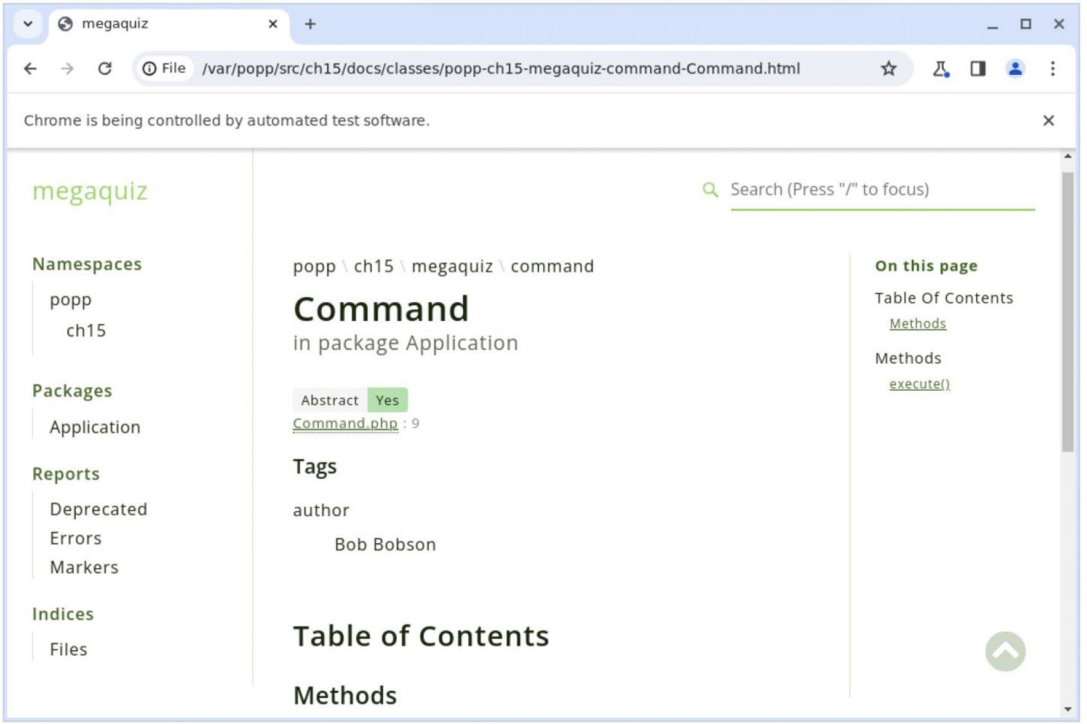

Figure 2-3. *Documentation output that recognizes the* `@author` *tag*

Documenting Classes

Let's add some more tags and text that are useful in class- or file-level DocBlocks. I should identify the class, explain its uses, and add authorship and copyright information.

Here is the `Command` class in its entirety:

```
namespace popp\ch15\megaquiz\command;

/**
 * Defines core functionality for commands.
 * Command classes perform specific tasks in a system via
 * the execute() method
 *
 * @author  Bob Bobson
 * @copyright 2024 Hidden Hat Technologies Ltd
 */
abstract class Command
{
    abstract public function execute(CommandContext $context): bool;
}
```

The DocBlock comment has grown significantly. The first sentence is a one-line summary. This is emphasized in the output and extracted for use in overview listings. The subsequent lines of text contain more detailed description. It is here that you can provide detailed usage information for the programmers who come after you. As we will see, this section can contain links to other elements in the project and fragments of code in addition to descriptive text. I also include the `@author` tag, which you have already seen, and a `@copyright` tag. You can see the effect of my extended class comment in Figure 2-4.

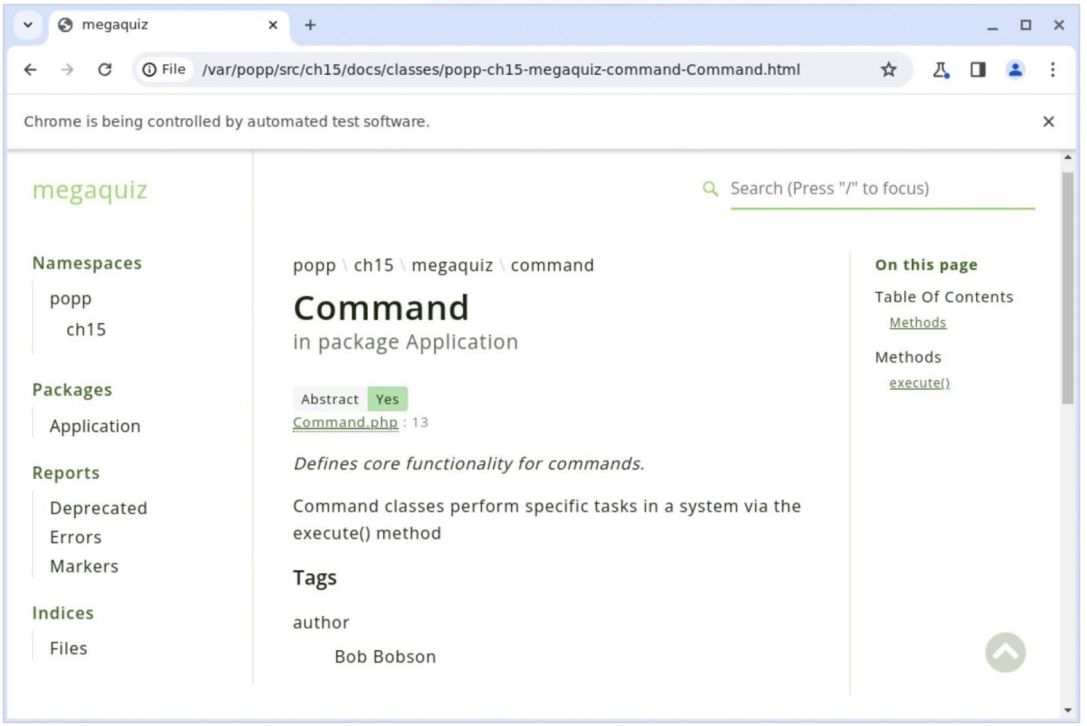

Figure 2-4. *Class details in documentation output*

Notice that I didn't need to tell phpDocumentor that the `Command` class is abstract. This confirms something that we already know, that phpDocumentor interrogates the classes with which it works even without our help. But it is also important to see that DocBlocks are contextual. phpDocumentor understands that we are documenting a class in the previous listing, because the DocBlock it encounters immediately precedes a class declaration.

File-Level Documentation

Although I tend to think in terms of classes rather than of the files that contain them, a file-level comment can be a good place to insert copyright and license information. A file comment should be the first DocBlock in a document. It should not directly precede a coding construct (like a class, for example).

Many open source projects require that every file includes a license notice or a link to one. Page-level DocBlock comments can be used, therefore, for including license information that you do not want to repeat on a class-by-class basis. You can use the `@license` tag for this. `@license` should be followed by a URL, pointing to a license document and a description:

```
/**
 * @license https://opensource.org/license/mit The MIT License
 */
```

Documenting Properties

Once upon a time, all properties were mixed in PHP. That is, a property could potentially contain a value of any type. These days, of course, we can, and usually should, constrain the types of our properties. As you might expect, phpDocumentor will detect and report any property type declarations.

There are still plenty of situations, however, where more information is valuable. You may wish to explain what a property is used for or to provide information about types contained within a collection.

We can document a property, variable, or constant with the `@var` tag. This will accept three arguments which can be placed on a single line. These are *type*, *name*, and *description*.

Here are some properties documented in the `CommandContext` class:

```
class CommandContext
{
    /** @var string  appname The application name  */
    public readonly string $appname;

    /** @var array<string, mixed>   params    Encapsulated keys/values  */
    private array $params = [];

    /** @var string error   An error message  */
    public string $error = "";

    // ...
```

As you can see, I provide a type, a name, and a description for each property. For the `params` array, I also use a special notation to specify the type of both the keys and values in the array. This *generic* notation is a standard way of describing the types that make up collections. Generics, which define and constrain collection types, are supported in many other languages such as TypeScript and Java. Although PHP does not support defining generics within the language, the notation is useful both for anyone reading the documentation as well as other tools that might use the phpdoc – notably IDEs and static analysis tools such as PHPStan.

You can see the documented properties in Figure 2-5.

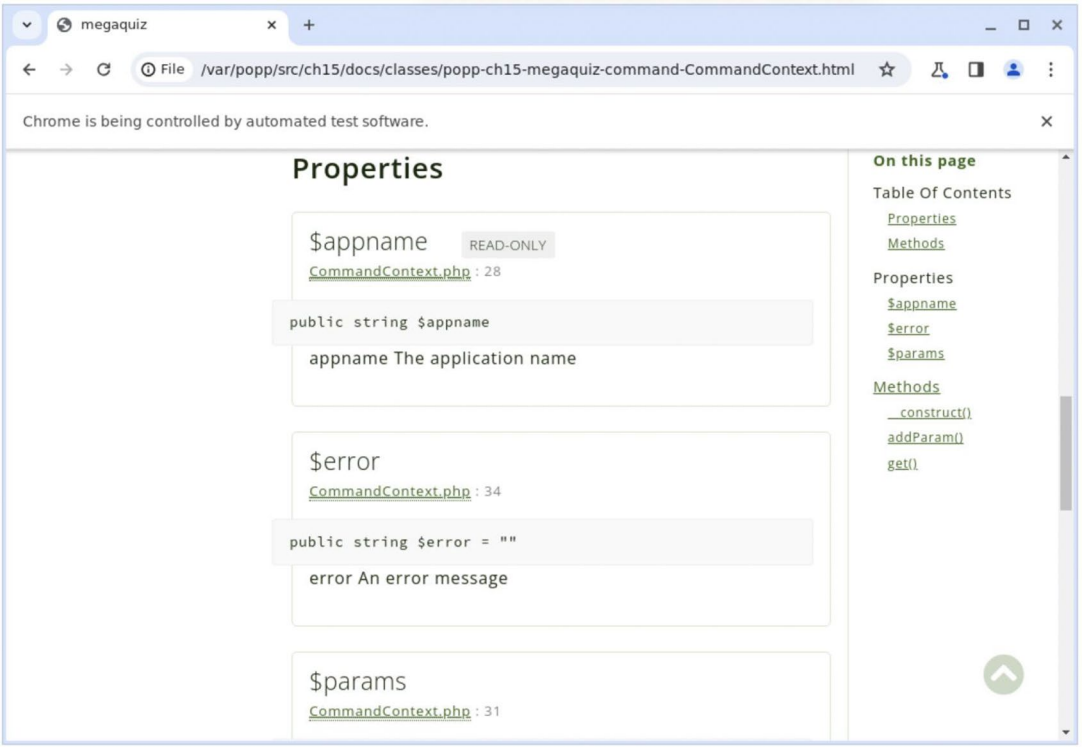

Figure 2-5. *Documenting properties*

Documenting Methods

Together with classes, methods lie at the heart of a documentation project. At the very least, readers need to understand the arguments to a method, the operation performed, and its return value.

As with class-level DocBlock comments, method documentation should consist of two blocks of text: a one-line summary and an optional description. You can provide information about each argument to the method with the `@param` tag. Each `@param` tag should begin a new line and should be followed by the argument name, its type, and a short description.

You can document the method's return type with the `@return` tag. `@return` should begin a new line and should be followed by the return value's type and a short description. I put these elements together here:

```
/**
 * Perform the key operation encapsulated by the class.
 * Command classes encapsulate a single operation. They
 * are easy to add to and remove from a project, can be
 * stored after instantiation and execute() invoked at
 * leisure.
 * @param  $context CommandContext Shared contextual data
 * @return bool     false on failure, true on success
 */
abstract public function execute(CommandContext $context): bool;
```

It may seem strange to add more documentation than code to a document. Documentation in abstract classes is particularly important, though, because it provides directions for developers who need to understand how to extend the class. If you are worried about the amount of dead space the PHP engine must parse and discard for a well-documented project, it is a relatively trivial matter to add code to your build tools to strip out comments on installation.

You can see our documentation's output in Figure 2-6.

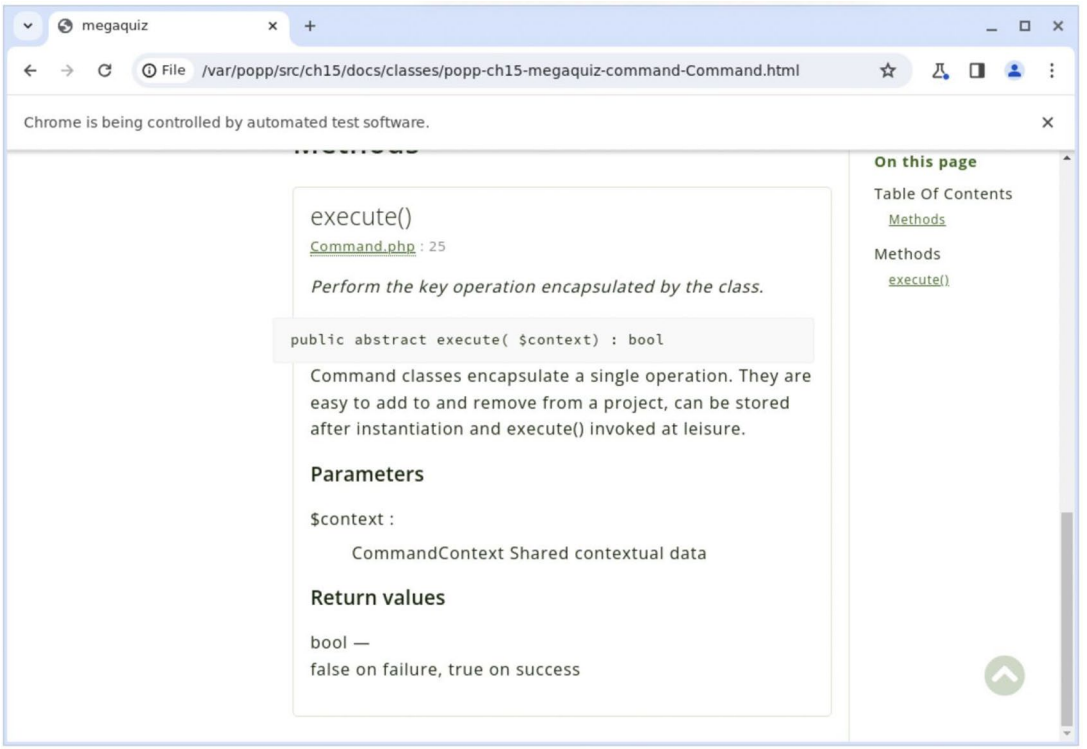

Figure 2-6. *Documenting methods*

Creating Links in Documentation

phpDocumentor generates a hyperlinked documentation environment for you. Sometimes, though, you will want to generate your own hyperlinks, either to other elements within documentation or to external sites. In this section, we will look at the tags for both of these.

As you construct a DocBlock comment, you may want to talk about a related class, property, or method. To make it easy for the user to navigate to this feature, you can use the `@see` tag. `@see` requires a reference to a class using the fully qualified or relative class name (so, for example if you are documenting from the `popp\ch15\megaquiz\command` namespace, you can just use a class name to refer to another class in the same namespace). You can also append double colons followed by an element such as a method (`Command::execute()`) or a property (`CommandContext::$applicationName`).

Once you've referenced your target element, you can add some label text. So, in the following DocBlock comment, I document the `CommandContext` class and emphasize the fact that it is commonly used in the `Command::execute()` method:

```
/**
 * Encapsulates data for passing to, from and between Commands.
 * Commands require disparate data according to context. The
 * CommandContext object is passed to the {@see Command::execute()}
 * method, and contains data in key/value format. The class
 * automatically extracts the contents of the $_REQUEST
 * superglobal.
 *
 * @see Command::execute()  the execute method
 */
class CommandContext
{
    // ...
```

As you can see in Figure 2-7, the `@see` tag resolves to a link. Clicking this will lead you to the `Command::execute()` method. Notice also a new feature. You can apply some tags inline within description text by wrapping them in braces.

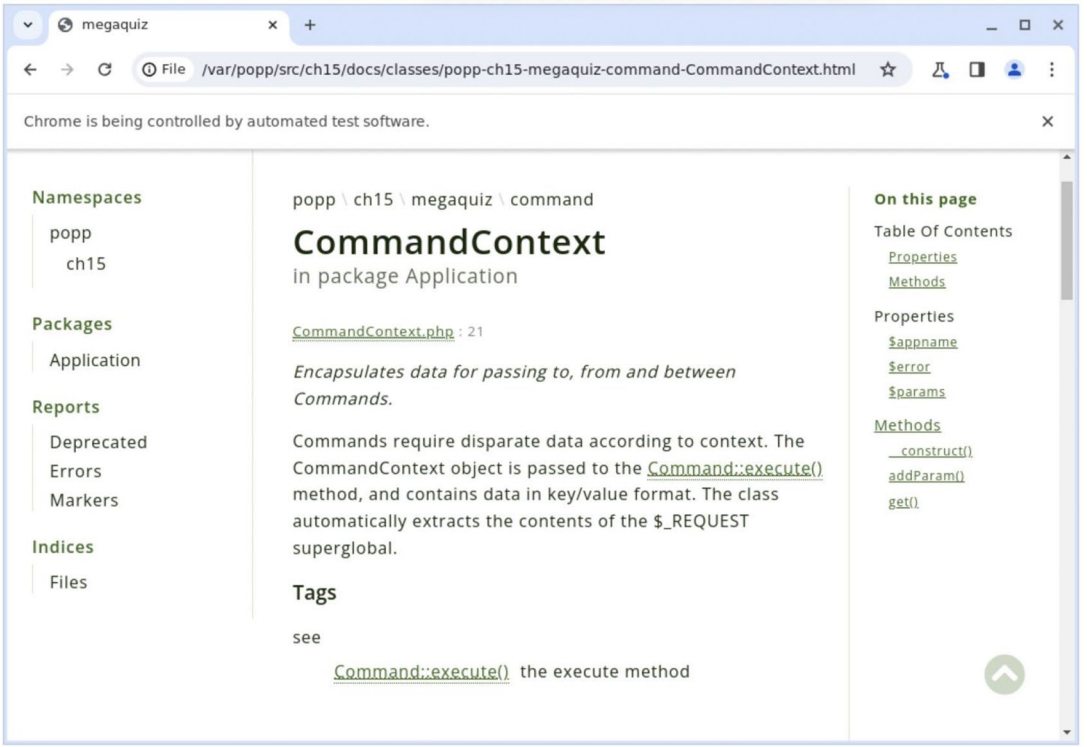

Figure 2-7. *Creating a link with the* `@see` *tag (inline and block)*

You can also create web links using the `@link` tag. Simply combine `@link` with a URL and a description.

```
@link   http://www.example.com    More info
```

Once again, the URL is the target, and the description that follows it is the clickable text. As with `@see`, you can also use `@link` inline.

You may want to make a reciprocal link. `Command` uses `CommandContext` objects, so I can create a link from `Command::execute()` to the `CommandContext` class and a reciprocal link in the opposite direction. I could, of course, do this with two `@see` tags.

`@uses` handles it all with a single tag, however:

```
abstract class Command
{

    /**
     * Perform the key operation encapsulated by the class.
```

```
     * Command classes encapsulate a single operation. They
     * are easy to add to and remove from a project, can be
     * stored after instantiation and execute() invoked at
     * leisure.
     * @param  $context CommandContext Shared contextual data
     * @return bool     false on failure, true on success
     * @uses CommandContext
     * @link https://en.wikipedia.org/wiki/Command_pattern
     */
    abstract public function execute(CommandContext $context): bool;
}
```

So, by adding the `@uses` tag to the `Command::execute()` documentation, I create a link to the `CommandContext` class page. In this `CommandContext` class documentation, a "Used by" link will appear which leads back to `Command::execute()`.

You can see some of this in action in Figure 2-8.

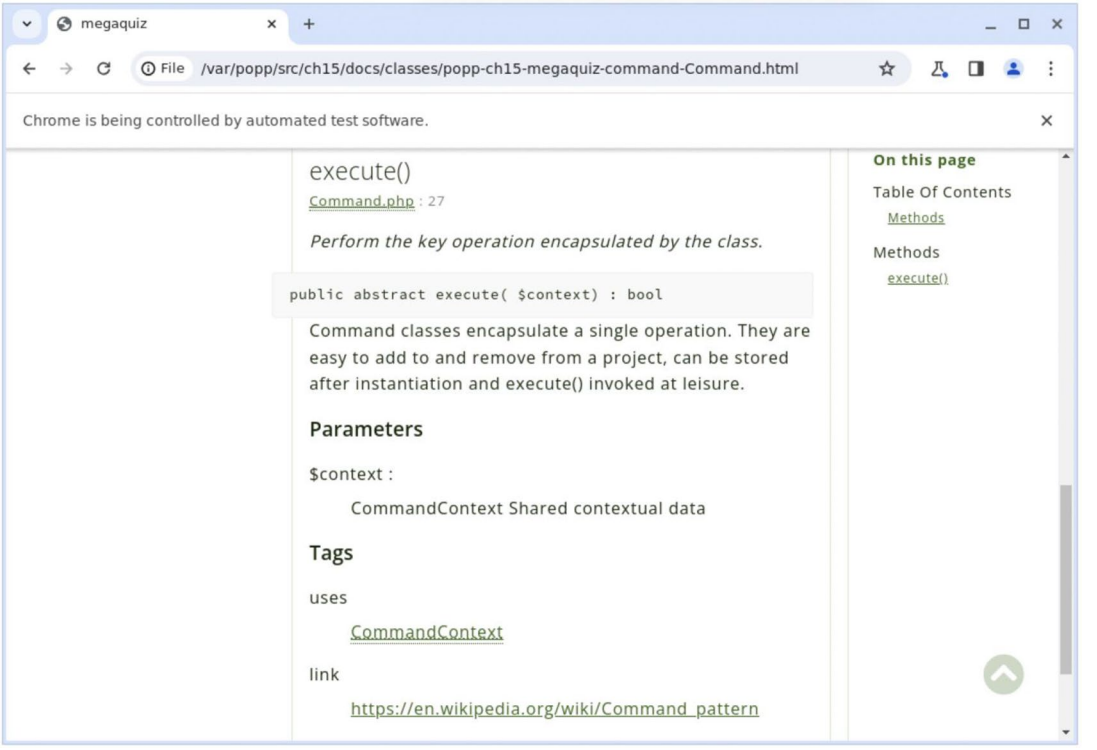

Figure 2-8. *Documentation including* `@link` *and* `@uses` *tags*

Summary

In this chapter, I covered the core features of phpDocumentor. You encountered the DocBlock comment syntax and the tags that can be used with it. I looked at approaches to documenting classes, properties, and methods, and you were provided with enough material to transform your documentation and thus improve collaborative working immeasurably (especially when used in conjunction with build tools and version control). There is a lot more to this application than I have space to cover, though, so be sure to check the phpDocumentor home page at https://www.phpdoc.org.

CHAPTER 3

PHP Standards

Unless you are a lawyer or a health inspector, the topic of standards probably does not make your heart race. However, what standards help us achieve is worth getting excited about. Standards promote interoperability, and that gives us access to a vast array of compatible tools and framework components.

This chapter will cover several important aspects of standards:

- *Why standards*: What are standards and why they matter
- *PHP Standards Recommendations*: Their origins and purpose
- *PSR-1*: The Basic Coding Standard
- *PSR-12*: Extended Coding Style
- *PSR-4*: Autoloading

Why Standards?

Design patterns interoperate. That is built-in at their core. A problem described in a design pattern suggests a particular solution, which in turn generates architectural consequences. These are then well addressed by new patterns. Patterns also help developers to interoperate because they provide a shared vocabulary. Object-oriented systems tend to privilege the principle of playing nice.

As we increasingly share each other's components though, this informal tendency toward interoperability is not always enough. Tools like Composer allow us to mix and match tools in our projects. These components may be designed as stand-alone libraries, or they may be pieces from a wider framework. Either way, once deployed in our system, they must be capable of working beside and in collaboration with any number of other components. By adhering to core standards, we make it less likely that our work will run into compatibility issues.

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_3

In some senses, the nature of a standard is less important than the fact that it is adhered to. Personally, for example, I don't love every aspect of the PSR-12 style guidelines. In most circumstances, including this book, I have adopted the standard. Other developers on my teams will hopefully find my code easier to work with because they will engage with it in a format that is familiar. For other standards, such as autoloading, failure to observe a common standard will result in components that may not work together at all without additional middleware.

Standards are probably not the most exciting aspect of programming. However, there is an interesting contradiction at their core. It may seem that a standard closes down creativity. After all, standards tell you what you can and can't do. You must comply. You might think that this is hardly the stuff of innovation. And yet, we owe the great flowering of creativity that the Internet has ushered into our lives to the fact that every node on this network of networks conforms to open standards. Proprietary systems stuck within walled gardens are necessarily limited in scope and often in longevity – no matter how clever their code or slick their interfaces. The Internet, with its shared protocols, ensures that any site can link to any other site. Most browsers support standard HTML, CSS, and JavaScript. The interfaces we can build within these standards are not always the most impressive we might imagine (though the limitations are much less than they were); still, abiding by them enables us to maximize the reach of our work.

Used well, standards promote openness, cooperation, and, ultimately, creativity. This is true, even if a standard itself enforces some limitations.

What Are PHP Standards Recommendations?

At the 2009 php[tek] conference, a group of framework developers formed an organization they called the PHP Framework Interop Group (PHP-FIG). Since then, developers have come on board from other key components. Their purpose was to build standards, so that their systems could better coexist.

The group vote on standards proposals which progress from Draft through Review and, finally, to Accepted status.

Table 3-1 lists the current standards at the time of this writing.

Table 3-1. *Accepted PHP Standards Recommendations*

PSR Number	Name	Description
1	Basic Coding Standard	Fundamentals such as PHP tags and basic naming conventions
3	Logger Interface	Rules for log levels and logger behaviors
4	Autoloading Standard	Conventions for naming classes and namespaces, as well as their mapping to the file system
6	Caching Interface	Rules for cache management, including data types, cache item lifetime, error handling, etc.
7	HTTP Message Interface	Conventions for HTTP requests and responses
11	Container Interface	A common interface for dependency injection containers
12	Extended Coding Style Guide	Code formatting, including rules for placement of braces, argument lists, etc.
13	Hypermedia Links	Interfaces for describing hypermedia links
14	Event Dispatcher	Definition for event management
15	HTTP Handlers	Common interfaces for HTTP server request handlers
16	Simple Cache	A common interface for caching libraries (a simplification of PSR-6)
17	HTTP Factories	A common standard for factories that create PSR-7-compliant HTTP objects
18	HTTP Client	Interface for sending HTTP requests and receiving HTTP responses
20	Clock	A simple interface for reading the system clock

Why PSR in Particular?

So, why choose one standard and not another? It happens that the PHP Framework Interop Group – the originators of PSRs – has a pretty great pedigree, and the standards themselves therefore make sense. But also, these are the standards that the major frameworks and components are adopting. If you are using Composer to add functionality to your projects, you are already consuming code that complies with PSRs. By using its conventions for autoloading and its style guides, you are likely building code that is ready for collaboration with other people and components.

Note One set of standards is not inherently superior to another. When you choose whether to adopt a standard, your choice may be driven by your judgment of the recommendation's merits. Alternatively, you might make a pragmatic choice based on the context within which you are working. If you're working in the WordPress community, for example, you might want to adopt the style defined in the Core Contributor Handbook at `https://developer.wordpress.org/coding-standards/wordpress-coding-standards/php/`. Such a choice is part of the point of standards, which are all about the cooperation of people and software.

PSRs are a good bet because they are supported by key framework and component projects, including Phing, Composer, PEAR, Symfony, and Zend 2. Like patterns, standards are infectious – you're probably already benefiting from them.

Who Are PSRs For?

Ostensibly, PSRs are designed for the creators of frameworks. The fact that the membership of the PHP-FIG group rapidly widened to include the creators of tools as well as frameworks, however, shows that standards have wide relevance. That said, unless you are creating a logger, you may not need to worry too much about the details of PSR-3 (beyond ensuring any logging tool you use is itself compliant). On the other hand, if you've read Volume 1, chances are you are as likely to be creating tools as you are to be consuming them. So, it's also likely that you'll find something relevant to you either in the present standards or the standards to come.

And then, there are the standards that matter to all of us. Unglamorous as style guides are, for example, they are relevant to every programmer. And while the rules that govern autoloading really apply to those who create autoloaders (and the main game in town is probably Composer's), they also fundamentally affect how we organize our classes, our packages, and our files.

For these reasons, I will focus on coding style and autoloading for the rest of this chapter.

Coding with Style

I tend to find pull request comments like "your braces are in the wrong place" disproportionately irritating. Such input often seems nitpicky and perilously close to *bike-shedding*.

Note In case you have not come across it, the verb "to bike-shed" refers to the tendency in some reviewers to criticize unimportant elements of a project under scrutiny. The implication is that such elements are chosen because they fit within the scope of the commenter's competence. So, given a skyscraper to assess, a particular manager might focus not on the vast and complex tower of glass and steel but on the much easier to comprehend bike shed around the back. Wikipedia has a good history of the term: https://en.wikipedia.org/wiki/Law_of_triviality.

And yet, I have come to see that conforming to a common style can help improve the quality of code. This is mainly a matter of readability (regardless of the reasoning behind a particular rule). If a team abides by the same rules for indentations, brace placement, argument lists, and so on, then a developer can quickly assess and contribute to a colleague's code.

So, in a previous edition of the book, I committed to edit all code examples so that they conformed to PSR-1 and PSR-12. I asked my colleague and technical reviewer Paul Tregoing to hold me to that, too. This was a promise that was so easy to make at the planning stage – and much more effort than I expected. This brings me to the first style guide lesson I learned. If possible, adopt a standard early for your project. Refactoring to a code style will likely tie up resources and make it hard to examine code differences that span The Time of the Great Reformat.

So what changes have I had to apply? Let's start with the basics.

PSR-1 Basic Coding Standard

These are the fundamentals for PHP code. You can find them in detail at https://www.php-fig.org/psr/psr-1/. Let's break them down.

Opening and Closing Tags

First of all, a PHP section should open either with `<?php` or `<?=`. In other words, the short opening tag, `<?`, should not be used nor should any other variation. A section should close with `?>` only (or, as we shall see, no tag at all).

Note PSRs follow a set of definitions for words such as *SHOULD* and *MUST* which determine the degree of compliance a directive should command. While this chapter will rely on the plain English meanings of such words, the absolute intended meanings within the context of PSR are defined at https://www.ietf.org/rfc/rfc2119.txt.

Side Effects

A PHP file should declare classes, interfaces, functions, and the like, or it should perform an action (such as reading or writing to a file or sending output to the browser); however, it should not do both. If you are accustomed to using `require_once()` to include other class files, this will trip you up straightaway because the act of including another file is a side effect. Just as patterns beget patterns, so standards tend to require other standards. The correct way to handle class dependencies is through a PSR-4-compliant autoloader.

So, is it legal for a class you declare to write to a file in one of its methods? That is perfectly acceptable because the effect is not kicked off by the file's inclusion. In other words, it's an execution effect, not a side effect.

So what kind of file might perform actions rather than declare classes? Think of the script that initiates an application.

Here is a listing that performs actions as a direct result of inclusion:

```
namespace popp\ch16\batch01;

require_once(__DIR__ . "/../../../vendor/autoload.php");

$tree = new Tree();
print "loaded " . get_class($tree) . "\n";
```

Here is a PHP file that declares a class with no side effects:

```
namespace popp\ch16\batch01;

class Tree
{
}
```

Note In other chapters, I largely omit `namespace` declarations and `use` directives in order to focus on the code. Since this chapter is about the mechanics of formatting class files, I will include `namespace` and `use` statements where appropriate.

Naming

Classes must be declared in upper camel case, also known as studly caps or PascalCase. In other words, a class name should begin with a capital letter. The rest of the name should be lowercase unless it consists of multiple words. In this instance, each word should begin with an uppercase letter, like this:

```
class MyClassName
```

Properties can be named in any way, although consistency is called for. I tend to use camel case, an approach similar to studly caps, but without the leading capital letter:

```
private string $myPropertyName
```

Methods must be declared in camel case:

```
public function myMethodName()
```

Class constants must be uppercase, with words separated by underscores:

```
public const MY_NAME_IS = 'matt';
```

More Rules and an Example

Classes, namespaces, and files should be declared in accordance with the PSR-4 Autoloading Standard. We will come to that later in the chapter, however. PHP documents must be saved as UTF-8 encoded files (without a byte order mark or BOM).

Note More accurately, PSR-1 states that namespaces and classes MUST follow an autoloading PSR. This would include the now-deprecated PSR-0 which specifies support for PEAR-style class naming.

Finally, for PSR-1, let's get it all wrong – and then put it right. Here is a class file that breaks all the rules:

```
<?
require_once("conf/ConfFile.ini");

class conf_reader {
    const ModeFile = 1;
    const Mode_DB = 2;

    private $conf_file;
    private $confValues = [];

    function read_conf() {
        // implementation
    }
}
?>
```

Can you spot all the issues? First of all, I used a short opening tag. I also failed to declare a `namespace` (though we haven't yet covered this requirement in detail). In naming my class, I used underscores and no capitals, rather than studly caps. And I used two formats for my constant names, neither of which are the required one – all capitals with words separated by underscores. Although both my property names were legal,

I failed to make them consistent; specifically, I used underscores for `$conf_file` and camel case for `$confValues`. In naming my method, `read_conf()`, I used an underscore rather than camel case.

Let's fix those itemized issues.

```
<?php
namespace popp\ch16\batch01;

class ConfReader {
    const MODE_FILE = 1;
    const MODE_DB = 2;

    private $confFile;
    private $confValues = [];

    function readConf() {
        // implementation
    }
}
?>
```

This fixes things up as far as PSR-1 is concerned but we are not done. There is still PSR-12 to consider.

PSR-12 Extended Coding Style

The Extended Coding Style (PSR-12) builds upon PSR-1 and replaces a deprecated standard: PSR-2. Let's jump in and look at some of the rules.

Starting and Ending a PHP Document

We have already seen that PSR-1 requires that PHP blocks open with `<?php` or `<?=`. PSR-12 stipulates that pure PHP files should not have an ending `?>` tag but should end with a single blank line. It's all too easy to end a file with a closing tag and then let an extra new line creep in. This can result in formatting bugs as well as errors when you set HTTP headers (you cannot do this after content has already been sent to the browser).

Table 3-2 describes, in order, the statements that might form a valid PHP document.

Table 3-2. *PHP Statements*

Statement	Example
Opening PHP tag	`<?php`

A file-level DocBlock	`/**`
	`* File doc`
	`*/`

Declare statements	`declare(strict_types=1);`
Namespace declaration	`namespace popp;`
Use import statements (classes)	`use other\Service;`
Use import statements (functions)	`use function other\myFunc;`
Use import statements (constants)	`use const other\MY_CONST;`
The remainder of the code in the file	

A PHP document should follow the structure in Table 3-2 (though any elements that are not necessary for legal PHP code may be omitted). `namespace` declarations should be followed by a blank line, and a block of `use` declarations should be followed by a blank line. Do not put more than one `use` declaration on the same line:

```
namespace popp\ch16\batch01;

use popp\ch10\batch06\PollutionDecorator;
use popp\ch10\batch06\DiamondDecorator;
use popp\ch10\batch06\Plains;

// begin class
```

Note Compound namespaces (with a depth of now more than two) are also allowed by PSR-12. So, a form like this would be legal:

```
use popp\ch10\{
    batch06\PollutionDecorator,
    batch06\DiamondDecorator,
    batch06\Plains,
};
```

Starting and Ending a Class

The `class` keyword, the class name, and `extends` and `implements` must all be placed on the same line. Where a class implements multiple interfaces, each interface name can be included on the same line as the class declaration, or it can be placed indented on its own line. Indentation, by the way, must be four spaces. If you choose to place your interface names on multiple lines, the first item must be placed on its own line rather than directly after the `implements` keyword. Class braces should begin on the line *after* the class declaration and end on their own line (directly after the class contents). So, a class declaration might look something like this:

```
class EarthGame extends Game implements
    Playable,
    Savable
{

    // class body
}
```

However, you could equally place the interface names on a single line:

```
class EarthGame extends Game implements Playable, Savable
{

// class body

}
```

In the case of interfaces, you can specify multiple classes to extend from applying the same rules as for `implements` – that is with all classes on one line or broken into a list with each class on its own line.

Working with Traits

When adding a trait to a class, you must add the `use` statement to the line directly after the class's opening brace. Although PHP allows you to group your traits onto a single line, PSR-12 requires that you place each `use` statement on its own line. If your class provides its own elements in addition to the `use` statements, you must leave a blank line before proceeding with nontrait content. Otherwise, you must close the class block on the line directly after the last `use` statement.

Here is a class that imports two traits and provides a method of its own:

```
namespace popp\ch16\batch01;

class Tree
{
    use GrowTools;
    use TerrainUtil;

    public function draw(): void
    {
        // implementation
    }
}
```

If you declare a block for `as` or `insteadof` statements, it should spread over multiple lines. The opening brace should begin on the same line as the `use` statement. The block should then use one line per statement. Finally, the closing brace should end on its own line, like this:

```
namespace popp\ch16\batch01;

class Marsh
{
    use GrowTools {
        GrowTools::dimension as size;
    }
    use TerrainUtil;

    public function draw(): void
    {
        // implementation
    }
}
```

Declaring Properties and Constants

Properties and constants must have a declared visibility (`public`, `private`, or `protected`). The `var` keyword is not acceptable. We have already covered the format for property and constant names as part of PSR-1.

Starting and Ending a Method

All methods must have a declared visibility (`public`, `private`, or `protected`). The visibility keyword must *follow* `abstract` or `final`, but *precede* `static`. Method arguments with default values should be placed at the end of the argument list.

Single-Line Declarations

Method braces should begin on the line *after* the method name and end on their own line (directly after the method code). A list of method arguments should not begin or end with a space (i.e., they should snuggle in close to the wrapping parentheses). For each argument, the comma should be flush with the preceding argument name (or the default value), but it should then be followed by a space. Let's clarify things with an example:

```
final public static function generateTile(int $diamondCount, bool $polluted
= false): array
{
    // implementation
}
```

Multiline Declarations

A single-line method declaration is not practical in cases where there are many arguments. In this situation, you can break the argument list so that each argument (including type, argument variable, default value, and comma) is placed indented on its own line. In this case, the closing parenthesis should be placed on the line after the argument list, flush with the start of the method declaration. The opening brace should follow the closing parenthesis on the same line, separated by a space. The method body should begin on a new line. Once again, that sounds much more complicated than it is. An example should make it clearer:

```
public function __construct(
    int $size,
    string $name,
    bool $wraparound = false,
    bool $aliens = false
) {
    // implementation
}
```

Return Types

A return type declaration should be on the same line as the closing parenthesis. The colon should directly follow the closing parenthesis. The colon should be separated from the return type by a single space. For multiline declarations, the return type declaration should precede the opening brace on the same line separated by a space.

```
final public static function findTilesMatching(
    int $diamondCount,
    bool $polluted = false
): array {
    // implementation
}
```

PSR-12 does not mandate the use of return type declarations. However, since the introduction of void, mixed, and nullable types, it should be possible to provide a declaration that matches all circumstances.

Lines and Indentation

As mentioned briefly above, you must use four spaces rather than tabs for indentation. It's worth checking your editor settings – you can configure good editors to use spaces rather than a tab when you press the `Tab` key. You should also wrap your text before your line reaches 120 characters (though this is not mandatory). Ideally, lines longer than 80 characters should be split across multiple lines of no more than 80 characters each. Lines must end with Unix line feed characters and not other platform-specific combinations (such as CR in Macs and CR/LF on Windows). Again, check your editor's settings for this, since it will likely use your operating system's default line ending characters.

Calling Methods and Functions

Do not place a space between the method name and the opening parenthesis. You can apply the same rules to the argument list in a method call as you do to the argument list in a method declaration. In other words, for a single-line call, leave no space after the opening parenthesis or before the closing parenthesis. A comma should follow directly after each argument, with a single space falling before the next one. If you need to use multiple lines for a method call, each argument should sit indented on its own line, and the closing parenthesis should fall on a new line:

```
$earthgame = new EarthGame(
    5,
    "earth",
    true,
    true
);
$earthgame::generateTile(5, true);
```

Flow of Control

Flow control keywords (`if`, `for`, `while`, etc.) must be followed by a single space. However, the opening parenthesis must not be followed by a space. Similarly, the closing parenthesis must not be preceded by a space. So, the contents should be snug in their brackets. In contrast to class and (single line) function declarations, the opening brace for the flow control block must begin on the same line as the closing parenthesis. The closing brace should sit on its own line. Here's a quick example:

```
$tile = [];
for ($x = 0; $x < $diamondCount; $x++) {
    if ($polluted) {
        $tile[] = new PollutionDecorator(new DiamondDecorator(new
        Plains()));
    } else {
        $tile[] = new DiamondDecorator(new Plains());
    }
}
```

Notice the space after both `for` and `if`. The `for` and `if` expressions are flush to the parentheses that contain them. In both cases, the closing parenthesis is followed by a space and then the opening brace for the flow control body.

Expressions in parentheses may be split across multiple lines, with each line indented at least once. Where the expressions are broken, the Boolean operators can go either at the beginning or end of each line, but your choice must be consistent.

```
$ret = [];
$count = count($this->tiles);
for (
    $x = 0;
    $x < $count;
    $x++
) {
    if (
        $this->tiles[$x]->isPolluted() &&
        $this->tiles[$x]->hasDiamonds() &&
        ! ($this->tiles[$x]->isPlains())
    ) {
        $ret[] = $x;
    }
}
return $ret;
```

Finishing the ConfReader Class

Remember `ConfReader`? In the previous version, I fixed all issues up to PSR-1 compliance. But that work would not pass muster for PSR-12. In addition to various minor spacing issues, I failed to declare visibility on my constants and methods. For the sake of completeness, let's finish the work off now.

```
namespace popp\ch16\batch01;

class ConfReader
{
    public const MODE_FILE = 1;
    public const MODE_DB = 2;
```

```
    private string $confFile;
    private array $confValues = [];

    public function readConf(): void
    {
        // implementation
    }
}
```

PSR-4 Autoloading

We looked at PHP's support for autoloading in Volume 1. I showed how to use the `spl_autoload_register()` function to automatically require files based on the name of an as yet unloaded class. Although this is powerful, it is also a kind of behind-the-scenes magic. This is fine in a single project but a recipe for great confusion if multiple components come together and all use different conventions for loading class files.

The Autoloading Standard (PSR-4) requires frameworks to conform to a common set of rules, thereby adding some discipline to the magic.

This is great news for developers. It means that we can more or less ignore the mechanics of requiring files and focus instead on class dependencies.

The Rules That Matter to Us

The main purpose of PSR-4 is to define rules for autoloader developers. However, those rules inevitably determine the way we must declare namespaces and classes. Here are some of the basics.

As specified in PSR-1, a fully qualified class name (i.e., the name of a class, including its namespaces) must include an initial "vendor" namespace. So, a class must have at least one namespace.

Let's say that our vendor namespace is `popp`. We can declare a class in this way:

```
namespace popp;

class Services
{
}
```

The fully qualified class name for this class is `popp\Services`.

The initial namespaces in a path must correspond to one or more base directories. We can use this to map a set of sub-namespaces to a starting directory. If, for example, we want to work with the namespace `popp\library` (and nothing else under the `popp` namespace), then we might map that to a top-level directory to spare us from having to maintain an empty `popp/` directory.

Let's set up a `composer.json` file to perform that mapping:

```
{
    "autoload":  {
        "psr-4": {
            "popp\\library\\": "mylib"
        }
    }
}
```

Notice that I don't even need to call the base directory, `"library"`. This is an arbitrary mapping of `popp\library` to the `my\lib` directory. Now I can create a class file under the `mylib` directory:

```
// mylib/LibraryCatalogue.php

namespace popp\library;

use popp\library\inventory\Book;


class LibraryCatalogue
{
    private array $books = [];

    public function addBook(Book $book): void
    {
        $this->books[] = $book;
    }
}
```

In order to be found, the `LibraryCatalogue` class must be placed in a file with exactly the same name (with the obvious addition of the `.php` extension).

After a base directory (`mylib`) has been associated with initial namespaces (`popp\library`), there must then be a direct relation between subsequent directories and sub-namespaces. It happens that I have already referenced a class named `popp\library\inventory\Book` in my `LibraryCatalogue` class. That class file should therefore be placed in the `mylib/inventory` directory:

```
// mylib/library/inventory/Book.php

namespace popp\library\inventory;

class Book
{
    // implementation
}
```

Remember the rule that the initial namespaces in a path must correspond to one *or more* base directories? So far, we have made a one-to-one relationship between `popp\library` and `mylib`. There's actually no reason why we can't map the `popp\library` namespace to more than one base directory. Let's add a directory named `additional` to the mapping; here's the amendment to `composer.json`:

```
{
    "autoload":  {
        "psr-4": {
            "popp\\library\\": ["mylib", "additional"]
        }
    }
}
```

Now I can create the additional/inventory directories and a class to go in them:

```
// additional/inventory/Ebook.php

namespace popp\library\inventory;

class Ebook extends Book
{
    // implementation
}
```

Next, let's create a top-level runner script, `index.php`, to instantiate these classes:

```
require_once("vendor/autoload.php");

use popp\library\LibraryCatalogue;

// will be found under mylib/
use popp\library\inventory\Book;

// will be found under additional/
use popp\library\inventory\Ebook;

$catalogue = new LibraryCatalogue();
$catalogue->addBook(new Book());
$catalogue->addBook(new Ebook());
```

Note You must use Composer to generate the autoload file, `vendor/autoload.php`, and this file must be included in some way before you gain access to the logic you have declared in `composer.json`. You can do this by running the command `composer install` (or by running `composer dump-autoload` if you just want to regenerate the autoload file in an environment that is already installed). You can learn more about Composer in Chapter 18.

Remember the rule about side effects? A PHP file should declare classes, interfaces, functions, and the like, or it should perform an action. However, it should not do both. This script falls into the taking action category. Crucially, it calls `require_once()` to include the autoload code generated using the configuration in the `composer.json` file. Thanks to this, all the classes are located, despite the fact that `Ebook` has been placed in an entirely separate base directory from the rest.

Why would I want to maintain two separate directories for the same core namespace? One possible reason is for unit tests that you want to keep separate from production code. You may also manage plug-ins and extensions that will not ship with every version of your system.

Note Be sure to keep an eye on all the PSR standards at https://www.php-fig.org/psr. This is a fast-moving area, and you'll likely find that standards relevant to you are on their way.

PSR-11 Container Interface

This is something of an aside, since most of us will use a dependency injection container rather than write one. However, I created just such a container in Volume 1 (first implemented Chapter 9 and used extensively in Chapters 12 and 13) so it's worth covering in brief here. You may remember that a dependency injection container supports the Inversion of Control pattern using various means (often including configuration, reflection, and attribute comments) to instantiate or otherwise populate objects. The container acts as a repository for objects which can be accessed via a key (usually, but not always, the object's class name) and which can then be automatically used in the creation and configuration of yet more objects.

The PSR-11 standard is comparatively brief. It mostly consists of a set of interfaces which are made available in the package `Psr\Container`. The most important of these is `ContainerInterface`, which defines the behavior of a compliant IoC container. Two lesser interfaces `ContainerExceptionInterface` and `NotFoundExceptionInterface` should be implemented by any exceptions thrown by the container.

You can add the `Psr\Container` package to a project with

```
$ composer require psr/container
```

The container I created in Chapter 9 of Volume 1 was already close to compliance. Without going back into implementation details, here's what it takes to make it PSR-11 compliant:

```
use Psr\Container\ContainerInterface;

class Container implements ContainerInterface
{
    public function has(string $class): bool
    {
        // ...
    }
```

```
    public function get(string $class): object
    {
        // ...
    }
}
```

There's more to `Container` than the methods shown here, of course, but these are what the standard demands. The `ContainerInterface` interface does not specify a return type for `get()`, but the `bool` return type for the `has()` method is required.

In my implementation of the `get()` method (and the methods it invokes internally), I throw two types of exception. Firstly, if I have not already stored a particular object and I cannot locate a class corresponding to the given string, I throw a `NotFoundException`.

```
use Psr\Container\NotFoundExceptionInterface;

class NotFoundException extends \Exception implements
NotFoundExceptionInterface
{
}
```

If I *can* find a class matching a `get()` invocation but then cannot instantiate an object from it for some reason, I throw a `ContainerException`.

```
use Psr\Container\ContainerExceptionInterface;

class ContainerException extends \Exception implements
ContainerExceptionInterface
{
}
```

These classes simply extend `Exception` and implement their corresponding `Psr\Container` interfaces. Because of this, a client can catch either `ContainerExceptionInterface` or `NotFoundExceptionInterface` and then act accordingly.

Summary

In this chapter, I wrestled a little with the possibility that standards are less than fantastically exciting – and then made a case for their power. Standards get integration issues out of our way, so that we can get on and do amazing things. I looked at PSR-1 and PSR-12, the standards for basic coding and for wider coding style. Next, I went on to discuss PSR-4, the standard for autoloaders. I did not delve into PSR-0, the older autoloading standard which supports old PEAR-style package naming, but you may want to look it up. Finally, I worked through a Composer-based example that showed PSR-4-compliant autoloading in practice.

CHAPTER 4

Refactoring and Standards Tools

As coders, ideally, we work to automate drudgery (the qualification is necessary because we've all encountered systems that have clearly been written to make everyone's lives worse rather than better). In doing so, though, we must often endure our own parade of tedious tasks. While standards make for better, more interoperable code, for example, the need for compliance adds yet another layer of effort to our development routines. And that's before we hunt for misspelled variables, loose method signatures, and all manner of other bug magnets.

Luckily, there are tools available to help with, or even fully automate, the drudgery. In this chapter, I will look briefly at two of the best of these: PHP_CodeSniffer and PHPStan. The chapter will cover

- *Running PHP_CodeSniffer*: Checking standards compliance in your projects
- *PHP Code Beautifier and Fixer*: Automatically correcting PHP_CodeSniffer errors
- *Custom standards*: Writing your own sniffs
- *PHPStan*: Finding deeper issues in your code with this powerful static analysis tool

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_4

PHP_CodeSniffer

Even if Chapter 3 covered every single directive in PSR-12 (which it does not), it would be hard to keep it all in your mind. After all, we have other things to think about – like the design and implementation of our systems. So, given that we have bought into the value of coding standards, how do we comply without using too much of our time or focus? We use a tool, of course.

PHP_CodeSniffer allows you to detect and even repair standards violations – and not just for PSR. You can get it by following the instructions at https://github.com/squizlabs/PHP_CodeSniffer. There are Composer and PEAR options, but here's how you can download the PHP archive files:

```
$ curl -OL https://phars.phpcodesniffer.com/phpcs.phar
$ curl -OL https://phars.phpcodesniffer.com/phpcbf.phar
```

Why two downloads? The first is for the main PHP_CodeSniffer script: `phpcs`, which diagnoses and reports on violations. The second is for an extension: `phpcbf`, or PHP Code Beautifier and Fixer, which can fix a lot of them.

Checking and Fixing Your Code

Let's put the tools through their paces. First, here is a scrappily formatted piece of code:

```
namespace popp\ch17\batch01;
class ebookParser {

    function __construct(string $path , $format=0 ) {
        if ($format>1)
            $this->setFormat( 1 );
    }

    function setformat(int $format) {
        // do something with $format
    }
}
```

Rather than run through the problems here, let's have PHP_CodeSniffer do it for us:

```
$ php phpcs.phar --standard=PSR12 src/ch17/batch01/EbookParser.php
```

```
FILE: /var/popp/src/ch17/batch01/phpcsBroken.php
----------------------------------------------------------------------
FOUND 17 ERRORS AFFECTING 6 LINES
----------------------------------------------------------------------
  5 | ERROR | [x] Header blocks must be separated by a single blank line
  6 | ERROR | [ ] Class name "ebookParser" is not in PascalCase format
  6 | ERROR | [x] Opening brace must not be followed by a blank line
  6 | ERROR | [x] Opening brace of a class must be on the line after the
                  definition
  8 | ERROR | [ ] Visibility must be declared on method "__construct"
  8 | ERROR | [x] Expected 0 spaces between argument "$path" and
                  comma; 1 found
  8 | ERROR | [x] Incorrect spacing between argument "$format" and equals
                  sign; expected 1 but found 0
  8 | ERROR | [x] Incorrect spacing between default value and equals sign
                  for argument "$format"; expected 1 but found 0
  8 | ERROR | [x] Expected 0 spaces before closing parenthesis; 1 found
  8 | ERROR | [x] Opening brace should be on a new line
  9 | ERROR | [x] Inline control structures are not allowed
  9 | ERROR | [x] Expected at least 1 space before ">"; 0 found
  9 | ERROR | [x] Expected at least 1 space after ">"; 0 found
 10 | ERROR | [x] Space after opening parenthesis of function call
                  prohibited
 10 | ERROR | [x] Expected 0 spaces before closing parenthesis; 1 found
 13 | ERROR | [ ] Visibility must be declared on method "setformat"
 13 | ERROR | [x] Opening brace should be on a new line
----------------------------------------------------------------------
PHPCBF CAN FIX THE 14 MARKED SNIFF VIOLATIONS AUTOMATICALLY
----------------------------------------------------------------------

Time: 174ms; Memory: 8MB
```

That's an exhausting number of problems for just a few lines of code. Luckily, as the output indicates, we can fix a lot of these with very little effort by running `phpcbf` (applied to a copy so as to keep my formatting errors for another time):

```
$ php phpcbf.phar --standard=PSR12 src/ch17/batch01/EbookParser.php
```

Here is the command's output:

```
PHPCBF RESULT SUMMARY
----------------------------------------------------------------------
FILE                                                  FIXED  REMAINING
----------------------------------------------------------------------
/var/popp/src/ch17/batch01/EbookParser.php            14     3
----------------------------------------------------------------------
A TOTAL OF 14 ERRORS WERE FIXED IN 1 FILE
----------------------------------------------------------------------

Time: 233ms; Memory: 8MB

   `
```

Now, if I run phpcs again, the situation is much improved:

```
FILE: /var/popp/src/ch17/batch01/EbookParser.php
----------------------------------------------------------------------
FOUND 3 ERRORS AFFECTING 3 LINES
----------------------------------------------------------------------
  6 | ERROR | Class name "ebookParser" is not in PascalCase format
  8 | ERROR | Visibility must be declared on method "__construct"
 15 | ERROR | Visibility must be declared on method "setformat"
----------------------------------------------------------------------

Time: 335ms; Memory: 8MB
```

I'll go ahead and add the visibility declarations and then change the name of the class – a quick job! Now I have a stylishly compliant code file:

```
namespace popp\ch17\batch01;

class EbookParser
{
    public function __construct(string $path, $format = 0)
    {
        if ($format > 1) {
            $this->setFormat(1);
        }
    }
```

```
    public function setformat(int $format)
    {
        // do something with $format
    }
}
```

Managing the Scope of an Analysis

So far, I have invoked the `phpcs` and `phpcbf` commands with a path to an individual file. As you might expect, you can also pass along a path to a directory. PHP_CodeSniffer will work recursively through the directory and generate a report for all valid files found.

```
$ ./phpcs.phar --standard=PSR12 src/ch17/batch02/
```

Depending upon the number of files within a directory, this will often result in a very large report. In such cases, I often pipe the output through the `more` command or redirect to a file. You can reduce the output somewhat by suppressing warnings with the `-n` option. This will limit the output to errors only.

```
$ ./phpcs.phar --standard=PSR12 -n src/ch17/batch02/
```

You can also skip files and directories with the `--ignore` option. This is particularly useful if you'd like to avoid getting reports on third-party code beneath a `vendor/` directory. You can also specify part or all of a file name.

```
$ ./phpcs.phar --standard=PSR12 --ignore=vendor,Blah src/ch17/batch02/
```

Here, I exclude any path containing `vendor` or `Blah`. I could get a little more granular using wildcards. I might want to block a directory named `Blah`, for example, but still check a file name `BlahTools.php`:

```
$ ./phpcs.phar --standard=PSR12 --ignore=vendor,Blah/* src/ch17/batch02/
```

If you're sure that a particular file should be exempt from analysis, you can add a directive to the source:

```
// phpcs:ignoreFile

class DefiantlyBad {
    // I am non-compliant and proud
}
```

Or if you're breaking the rules for good reason in only part of your source file, you can selectively disable and re-enable analysis:

```
namespace popp\ch17\batch02;

class PartiallyBad
{
    public function __construct()
    {
    }

// phpcs:disable
    function intentionalRulebreaking() {}
// phpcs:enable
}
```

I have, throughout, been specifying the PSR12 standard using the `--standard` option. That's because the default standard is PEAR and I have been focusing on PSR standards in this book. I can review the available standards using the `-i` option.

```
$ ./phpcs.phar -i
The installed coding standards are MySource, PEAR, PSR1, PSR2, PSR12, Squiz
and Zend
```

A standard is made up of (usually) multiple *sniffs*. You can get a list of the sniffs for a standard using the `-e` option.

```
$ ./phpcs.phar --standard=PSR12 -e
```

Here's some truncated output:

```
The PSR12 standard contains 60 sniffs

Generic (15 sniffs)
-------------------
  Generic.ControlStructures.InlineControlStructure
  Generic.Files.ByteOrderMark
  Generic.Files.LineEndings
  Generic.Files.LineLength
  Generic.Formatting.DisallowMultipleStatements
```

Armed with that information, I can limit my review to a particular set of sniffs:

```
$ ./phpcs.phar --standard=PSR12 --sniffs=PSR12.Classes.OpeningBraceSpace
src/ch17/batch01/
```

Creating Your Own Sniff

PHP_CodeSniffer is an extremely useful tool when used without much customization. However, teams inevitably negotiate and enforce their own standards and practices in addition to those set by third-party bodies.

A team I worked with, for example, mandated that developers should avoid a range of procedural functions in favor of object-oriented equivalents. So a class that used the `date()` function should be refactored to employ the `DateTime` class. Keeping track of rules like this during code reviews can quickly become a chore as they evolve and multiply.

Luckily, PHP_CodeSniffer supports custom standards and sniffs. Let's create a standard containing a sniff that discourages the use of `date()`.

Defining a Standard

I'm going to call my standard `NoProc`. A minimal setup consists of a directory named after the standard and a file named `ruleset.xml`. So this is my file structure:

```
NoProc/
    ruleset.xml
```

The `ruleset.xml` file can be quite extensive (you can see all the directives it supports at https://github.com/PHPCSStandards/PHP_CodeSniffer/wiki/Annotated-Ruleset). Luckily, though, it only takes a few lines to create a viable standard.

```
<ruleset name="NoProc">
  <description>A standard which discourages use of certain functions where
  OO alternatives should be used.</description>
</ruleset>
```

So, a minimal standard needs a `ruleset` element with a `name` attribute and a `description` sub-element.

So that's it! I have a new standard – albeit one with no rules. I still need to tell phpcs about it by adding the standard to configuration.

```
$ ./phpcs.phar --config-set installed_paths $PWD/NoProc/
```

phpcs tells us where it saved the value.

```
Using config file: /Users/mattz/work/popp7/popp7-repo/CodeSniffer.conf

Config value "installed_paths" added successfully
```

Let's see if phpcs knows about NoProc:

```
$ ./phpcs.phar -i
The installed coding standards are MySource, PEAR, PSR1, PSR2, PSR12,
Squiz, Zend and NoProc
```

A Bad Date File

In order to have something for a sniff to work with, I'll create a scrappy file. In addition to using date(), I'll add all sorts of whitespace and mix in various language elements confusingly named date. I'll call this throwaway file BadDate.php.

```
namespace {
    Date("now");

    $date    =
    /* bloop */ date(DATE_ATOM);

    print_r($date);
}

namespace testClass {
    class

    /** tricksy **/

    date {
        function date(): void {
            print "date!!!";
        }
```

```
    }
}

namespace testEnum {
    enum date {
    }
}

namespace testInterface {
    interface date
    {
    }
}

namespace testTrait {
    trait date
    {
    }
}
```

What an ugly piece of work! All the better to put the parser through its paces, though. This is that rare circumstance in which bad code is good.

Creating the Sniff

A sniff is a class that implements the interface `PHP_CodeSniffer\Sniffs\Sniff`. Here is the interface (stripped of inline documentation):

```
namespace PHP_CodeSniffer\Sniffs;

use PHP_CodeSniffer\Files\File;

interface Sniff
{
    public function register();
    public function process(File $phpcsFile, $stackPtr);
}
```

I'll begin, then, by creating an empty sniff class. There are some simple rules that will simplify the process. By default, the library will look for my sniff under the standard (`NoProc`) directory within a directory named `Sniffs`. You can create subdirectories for different types of sniff. You should name your class so that it (and therefore its class file) contains the substring `Sniff`.

Note Fun fact: it took me the best part of an afternoon to work out why my sniff (a class whose name did not contain `Sniff`) was not being recognized by PHP_CodeSniffer.

Having created an empty class, my file structure looks like this:

```
NoProc/
    ruleset.xml
    Sniffs/
        Dates/
            NoProceduralDateSniff.php
```

Here's my `NoProceduralDateSniff` class template – as yet unimplemented:

```
namespace PHP_CodeSniffer\Standards\NoProc\Sniffs\Dates;

use PHP_CodeSniffer\Sniffs\Sniff;
use PHP_CodeSniffer\Files\File;
use PHP_CodeSniffer\Util\Tokens;

class NoProceduralDateSniff implements Sniff
{
    public function register(): array
    {
    }

    public function process(File $file, $position): void
    {
    }
}
```

The `register()` method should return an array of PHP parser tokens (the elements into which a script is broken into during compilation). As a target file is processed, if one of the tokens returned by `register()` is encountered, the `process()` method will be invoked. This is called with two arguments: a `PHP_CodeSniffer\Files\File` object and an integer index for the current token.

You can see a list of parser tokens at https://www.php.net/manual/en/tokens.php, although PHP_CodeSniffer actually breaks down source code into a more detailed set. This might seem daunting if you're new to it, but, in fact, you can work out what's going on pretty easily by running `phpcs` against a file with the `-vv` option set to crank up its verbosity.

```
$ ./phpcs.phar -vv src/ch17/batch02/BadDate.php
```

Here's a very small sample from the output:

```
*** START PHP TOKENIZING ***
Process token [0]: T_OPEN_TAG => <?php\n
Process token [1]: T_COMMENT => /*·listing·17.20·*/
Process token [2]: T_WHITESPACE => \n
Process token [3]: T_STRING => Date
Process token  4 : T_OPEN_PARENTHESIS => (
Process token [5]: T_CONSTANT_ENCAPSED_STRING => "now"
Process token  6 : T_CLOSE_PARENTHESIS => )
Process token  7 : T_SEMICOLON => ;
```

By running this over a small sample script, you can see how PHP_CodeSniffer breaks it down into tokens. It turns out that the `date` in `date("now")`, the `date` in `function date()`, and the `date` in `class date` are all rendered as the `T_STRING` constant. This is also true of enumerations, interfaces, and traits. That complicates my task a little but not too much since `function` is parsed as a `T_FUNCTION` token, `class` as a `T_CLASS` token, and so on. By matching identifiers for the kinds of elements I am *not* looking for, I can rule them out.

I'll begin with the `register()` method. I can see from the debug output above that a `T_STRING` token should be the sniff's trigger:

```
public function register(): array
{
    return [\T_STRING];
}
```

So now, NoProceduralDateSniff will be activated for each T_STRING token in a target file. Activation, here, means the invocation of process(). Here's my implementation of that:

```
public function process(File $file, $position): void
{
    $tokens = $file->getTokens();
    $content = $tokens[$position]['content'];

    if (strtolower($content) != "date") {
        return;
    }

    $tokenBefore = $file->findPrevious(
        Tokens::$emptyTokens,
        ($position - 1),
        null,
        true
    );
    $tokenCode = $tokens[$tokenBefore]['code'];
    if (
        $tokenCode == T_FUNCTION
        || $tokenCode == T_CLASS
        || $tokenCode == T_INTERFACE
        || $tokenCode == T_TRAIT
        || $tokenCode == T_ENUM
    ) {
        $tokenCodeStr = $tokens[$tokenBefore]['type'];
        return ;
    }
    $error = "Looks like a procedural date() consider using
    DateTime class";
    $file->addError($error, $position, "ProdDate");
}
```

As you can see, most of my work here is with the `PHP_CodeSniffer\Files\File` object which represents the file being parsed. It's worth diving into some of its useful methods for sniff authors.

`File` maintains an array of the tokens that make up the file under review. It makes this available via the `getTokens()` method. The provided `$position` argument contains the index of the current token. Acquiring the token at the `$position` index will render an associative array with, among other fields, a `code` element (corresponding to the value `T_STRING` in this case), a `type` element containing a string representation of the code, and a `content` element (the string "Date" for the first match in my test script).

If the match is *not* for a token with the content "date" (ignoring case), then I know that this is not the token I am looking for, and I dismiss the issue with a `return` statement. Otherwise, some more investigation is needed. In particular, since I'm only interested in a function call, I need to rule out the declarations of methods, classes, interfaces, enumerations, and traits named `date`. Once these declarations are excluded, I am most likely looking at a match. So, as I discovered by studying the output from my verbose parse, I need to look for a preceding `T_FUNCTION`, `T_CLASS`, `T_INTERFACE`, `T_ENUM`, or `T_TRAIT` token in order to exclude the wrong kinds of `date`.

This is where the `findPrevious()` method becomes useful. `findPrevious()` looks for a preceding token. It requires two arguments. First, `$types`, a token (or an array of tokens) to define the search. Then, `$start` the starting point from which to search in the token array. It optionally accepts `$end` which defaults to `null` but which should otherwise contain an end index for the search. Next, it accepts `$exclude`, a Boolean, `false` by default, which inverts the match rule when set to `true`. That means that a search will match anything *other than* one of the specified tokens. Finally, it accepts `$local`, another Boolean. This also defaults to `false`. When this is set to `true`, the search will be limited to the current statement.

Here's my `findPrevious()` call again:

```
$tokenBefore = $file->findPrevious(
    Tokens::$emptyTokens,
    ($position - 1),
    null,
    true
);
```

I call it with a handy array of whitespace tokens made available by the PHP_CodeSniffer\Util\Tokens class and with a starting index. I don't set an end argument, but I *do* invert the search. That means I'm looking for *the first previous token that is not white space*. Once I've acquired this, I can test its type. If I've found one of the declaration tokens, then I can safely return. Otherwise, it looks like I'm in business. In this case, I'm in the business of calling addError(). This method requires an error string, the token index, and a unique self-generated error code. (Out of scope here, it further accepts a data array for interpolating values into the error string, a severity integer, and a Boolean indicating whether the error is automatically fixable.)

Before I move on, it's worth mentioning a couple of other useful File methods. findNext() is identical in signature and function to findPrevious()except that it searches forward and not backward. Similarly, addWarning() is identical to addError() except that it generates a warning rather than an error. Finally getDeclarationName() requires a token stack index pointing to a method, class, interface, enumeration, or trait declaration and returns the name. It will throw an exception if the referenced token is not of a relevant type.

Now that I have my solitary sniff in place, I can try out the standard:

```
$ ./phpcs.phar --standard=NoProc src/ch17/batch02/BadDate.php
FILE: /Users/mattz/work/popp7/popp7-repo/src/ch17/batch02/BadDate.php
----------------------------------------------------------------------
FOUND 2 ERRORS AFFECTING 2 LINES
----------------------------------------------------------------------
 5 | ERROR | Looks like a procedural date() consider using DateTime class
 8 | ERROR | Looks like a procedural date() consider using DateTime class
----------------------------------------------------------------------

Time: 27ms; Memory: 6MB
```

Combining Multiple Standards

Even if I were to round out my NoProc standard, it would really only be useful as an extension to a more complete standard. I can combine NoProc with, for example, PSR12 in a couple of ways. I can do this at runtime by adding both standards to my command invocation:

```
$ ./phpcs.phar --standard=PSR12,NoProc src/ch17/batch02/BadDate.php
```

I can also amend the standard itself so that NoProc always incorporates PSR12. To do this, I'll simply add a rule element to my ruleset.xml file.

```
<ruleset name="NoProc">
  <description>A standard which discourages use of certain functions where
  OO alternatives should be used.</description>
  <rule ref="PSR12"/>
</ruleset>
```

The effect of the two approaches is identical, but the second automates the inclusion of the PSR12 standard.

PHPStan

Standards promote interoperability and good practice. By definition, the tools that enforce them are necessarily limited in scope to the standards they enforce. If you want to find a wider range of potential problems in your code, however, there are other options. Your IDE may well be running an analysis for you as you code, for example. Luckily, for those, like me, who prefer to stick with an editor such as Vim, a set of utilities collectively known as *static analysis* tools can be employed. They are so-called because they read and analyze source code without running it. Among the best and most popular of these is PHPStan.

Installing PHPStan

The easiest way to install PHPStan is with Composer.

```
$ composer require --dev phpstan/phpstan
```

This will install the libraries you need and make a command-line script available at vendor/bin/phpstan.

Running PHPStan

You can run an analysis very simply with the analyse command. Remember my EbookParser example? Now that it's perfectly PSR12 compliant, let's see what PHPStan makes of it.

```
$ php vendor/bin/phpstan analyse --no-progress --no-ansi EbookParser.php
```

I am running the command with the flags `--no-progress` to hide a dynamic progress report and `--no-ansi` to suppress some prettification that won't play well in these pages. I'll not show these flags in future examples. Here's my report:

```
[OK] No errors

Tip of the Day:
PHPStan is performing only the most basic checks.
You can pass a higher rule level through the --level option
(the default and current level is 0) to analyse code more thoroughly.
```

No errors? Great! It looks like my work is done. However, as the tool itself tells us, things aren't that simple.

Rule Levels

PHPStan applies rule levels running from a lenient 0 to a forbiddingly strict 9. Table 4-1 summarizes some of the checks associated with these levels.

Table 4-1. *PHPStan Rule Levels*

Level	Description
0	Basic checks including unknown classes and functions, method and function invocations with the incorrect number of arguments
1	Undefined variables
2	Unknown methods called on objects and classes
3	Property types, return types
4	Redundant code
5	Argument types which don't match parameter declarations
6	More undeclared types
7	Checks on union parameter type declarations
8	Report unchecked use of types that might be null
9	Strict `mixed` type checks

If you're working with a pre-existing code base, you might find an initial report quite shocking, especially if you dial the severity up to 9. I tend to start at about level 6 for my checks.

```
$ php vendor/bin/phpstan analyse --level=6 EbookParser.php
```

At once, I find that my perfectly compliant class is, nonetheless, somewhat imperfect.

```
------ ------------------------------------------------------------------
 Line   EbookParser.php
------ ------------------------------------------------------------------
 8      Constructor of class popp\ch17\batch01\EbookParser has an unused
        parameter $path.
 8      Method popp\ch17\batch01\EbookParser::__construct() has parameter
        $format with no type specified.
 15     Method popp\ch17\batch01\EbookParser::setformat() has no return
        type specified.
------ ------------------------------------------------------------------
```

These are all easy enough to fix. I'll go ahead and set things right.

```
class EbookParserFixed
{
    private int $format;
    private string $path;

    public function __construct(string $path, int $format = 0)
    {
        $this->path = $path;
        if ($format > 1) {
            $this->setFormat(1);
        }
    }

    public function setFormat(int $format): void
    {
        $this->format = $format;
    }
}
```

Of course, sometimes when you fix a bug or an issue, you simply expose another one right behind it. PHPStan immediately spots new problems:

```
$ php vendor/bin/phpstan analyse --level=6 EbookParserFixed.php
------ ------------------------------------------------------------------
 Line   EbookParserFixed.php
------ ------------------------------------------------------------------
 7      Property popp\ch17\batch03\EbookParser::$format is never read, only
        written.
          See: https://phpstan.org/developing-extensions/always-read-
          written-properties
 8      Property popp\ch17\batch03\EbookParser::$path is never read, only
        written.
          See: https://phpstan.org/developing-extensions/always-read-
          written-properties
------ ------------------------------------------------------------------
```

I can fix the issue with `$format` either by removing it or using it some way. I'll create an accessor method:

```
public function getFormat(): int
{
    return $this->format;
}
```

Telling PHPStan to Ignore Errors

My `EbookParser` class is a work in progress. I don't want to remove the currently unused `$path` property. At the same time, though, I would rather not see the error report every time I run PHPStan. I can tell the tool to forgive errors by adding a PHPDoc directive to my source file. To do that, I need to specify an error identifier. I can see the identifier to use by running an analysis with the `-v` flag.

```
$ php vendor/bin/phpstan analyse --level=6 -v EbookParserFixed.php
------ ---------------------------------------------------------------------
 Line   EbookParserFixed.php
------ ---------------------------------------------------------------------
 7      Property popp\ch17\batch03\EbookParserFixed::$format is never read,
        only written.
           property.onlyWritten
          See: https://phpstan.org/developing-extensions/always-read-
          written-properties
 8      Property popp\ch17\batch03\EbookParserFixed::$path is never read,
        only written.
           property.onlyWritten
          See: https://phpstan.org/developing-extensions/always-read-
          written-properties
------ ---------------------------------------------------------------------
```

This slightly more verbose output informs me that the error I need to ignore is `property.onlyWritten`:

```
class EbookParserFixed2
{
    private int $format;
    /** @phpstan-ignore property.onlyWritten */
    private string $path;

    // ...
```

Let's run again with that in place:

```
$ php vendor/bin/phpstan analyse --level=6 EbookParserFixed2.php
```

And now, my known errors are forgiven:

```
[OK] No errors
```

Array Arguments: Correcting Outside the Language

Not everything that PHPStan checks for can be corrected with PHP code alone. Imagine that I decided to refactor the `$path` constructor parameter and property specifying an array rather than a string so that I can manage multiple filepaths. Here's my amendment:

```
class EbookParserFixed3
{
    private int $format;
    /** @phpstan-ignore property.onlyWritten */
    private array $path;

    public function __construct(array $path, int $format = 0)
    {
        $this->path = $path;
        if ($format > 1) {
            $this->setFormat(1);
        }
    }

    // ...
```

When I review this at rule level 6 or higher, I see this error:

```
------ ------------------------------------------------------------------
 Line   EbookParserFixed3.php
------ ------------------------------------------------------------------
 9      Property popp\ch17\batch03\EbookParserFixed3::$path type has no
        value type specified in iterable type array.
          See: https://phpstan.org/blog/solving-phpstan-no-value-type-
          specified-in-iterable-type
 11     Method popp\ch17\batch03\EbookParserFixed3::__construct()
        has parameter $path with no value type specified in iterable
        type array.
          See: https://phpstan.org/blog/solving-phpstan-no-value-type-
          specified-in-iterable-type
------ ------------------------------------------------------------------

[ERROR] Found 2 errors
```

This makes a lot of sense. Modern PHP can be pinned down very well as far as type is concerned. But collections remain untyped. We encountered one solution in Volume 1 – the use of typed collection classes. But if you're sticking with arrays, then you're also stuck as far as enforcing type is concerned. One solution, supported by IDEs and tools such as PHPStan, is the use of PHPDoc. By telling static analysis tools our expectations for array contents, we help them to enforce our rules. So, I can fix the error:

```
class EbookParserFixed4
{
    private int $format;
    /** @phpstan-ignore property.onlyWritten */
    private array $path;

    /**
     * Constructor
     *
     * @param string[] $path   A list of paths to process
     * @param int $format   A format value 0-4
     */
    public function __construct(array $path, int $format = 0)
    {
        $this->path = $path;
        if ($format > 1) {
            $this->setFormat(1);
        }
    }
```

And we get a clean bill of health once again.

Summary

In this chapter, I examined two tools for maintaining code quality in your projects. I covered the use of PHP_CodeSniffer for detecting coding standards violations and the `phpcbf` extension which can automatically correct many violations. I created a custom standard named `NoProc` containing a single sniff for detecting usage of the `date()` function. I introduced PHPStan, a static analysis tool for finding bugs and design issues in PHP code. I showed how to adjust the tool's sensitivity, to disable it for known errors, and to handle collection typing errors with PHPDoc.

CHAPTER 5

Using and Creating Components with Composer

Programmers aspire to produce reusable code. This is one of the great goals in object-oriented coding. We like to abstract useful functionality from the messiness of specific context, turning a particular solution into a tool that can be used again and again. To come at this from another angle, if programmers love the reusable, they hate duplication. By creating libraries that can be reapplied, programmers avoid the need to implement similar solutions across multiple projects.

Even if we avoid duplication in our own code, though, there is a wider issue. For every tool you create, how many other programmers have implemented the same solution? This is wasted effort on an epic scale: Wouldn't it be much more sensible for programmers to collaborate and to focus their energies on making a single tool better, rather than producing hundreds of variations on a theme?

In order to do this, we need to get our hands on existing libraries. But then, the packages we need will likely require other libraries in order to do their work. So, we need a tool which can handle downloading and installing packages, as well as manage their dependencies. That is where Composer comes in; it does all this and more besides.

This chapter will cover several key issues:

- *Installation*: Downloading and setting up Composer
- *Requirements*: Using `composer.json` to get packages
- *Versions*: Specifying versions so as to get the latest code without breaking your system

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_5

- *Packagist*: Configuring your code for public access
- *Private repositories*: Leveraging Composer using a private repository

What Is Composer?

Strictly speaking, Composer is a dependency manager rather than a package manager. This, it seems, is because it handles component relationships on a local basis rather than centrally as Yum and Apt do. If you think that this is an overly fine distinction, you could be right. However we define it, Composer allows you to specify packages. It downloads them to a local directory (`vendor`), finds and downloads all dependencies, and then makes all this code available to your project via an autoloader.

As always, we need to begin by getting the tool.

Installing Composer

You can download Composer at https://getcomposer.org/download/. You will find an installer mechanism there. You can also install a stable phar file like this:

```
$ wget https://getcomposer.org/composer-stable.phar
$ chmod 755 composer-stable.phar
$ sudo mv composer-stable.phar ~/bin/composer
```

I download the archive and run `chmod` to ensure that it is executable. Then, I copy it into a central location so that I can run it easily from anywhere in my system. Now I can test the command:

```
$ composer --version
```

The output confirms both the version and the fact that the command is probably sane.

```
Composer version 2.7.2 2024-03-11 17:12:18
```

Installing a (Set of) Package(s)

Why did I do that funky bit with the parentheses in the title for this section? Because packages inevitably beget packages – sometimes a lot of packages.

Let's begin with a library that stands alone, though. Imagine that we're building an application which needs to communicate with OpenAI. A little bit of research leads me to the `orhanerday/open-ai` package. In order to install this, I need to generate a JSON file named `composer.json` and then define a `require` element:

```
{
    "require": {
        "orhanerday/open-ai": "5.*"
    }
}
```

I begin with a directory that is empty apart from the `composer.json` file. Once I run a Composer command, though, we'll see a change:

```
$ composer update
```

Here's the output:

```
Loading composer repositories with package information
Updating dependencies
Lock file operations: 0 installs, 1 update, 0 removals
  - Upgrading orhanerday/open-ai (5.1 => 5.2)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing orhanerday/open-ai (5.2): Extracting archive
Generating autoload files
1 package you are using is looking for funding.
Use the `composer fund` command to find out more!
No security vulnerability advisories found.
```

So what has been generated? Let's take a look by running `ls`:

```
composer.json  composer.lock  vendor
```

Composer installs packages into `vendor/`. It also generates a file named `composer.lock`. This specifies the exact versions of all packages installed. If you're using version control, you can commit this file. If another developer runs `composer install` with a `composer.lock` file present, package versions will be installed on their system exactly as specified. In this way, the team can stay in sync with one another, and you can be sure that your production environment exactly matches the development and test environments. If a developer runs `composer install` and no `composer.lock` file is present, then the effect is the same as running `composer update` – dependencies will be calculated anew, and a `composer.lock` file will be generated.

You can override the lock file by running `composer update` again. This will generate a new lock file. Typically, you will run this to keep current with new package versions (if you are using ranges or, as I have, wildcards).

Installing a Package from the Command Line

As you have seen, I can create the `composer.json` file using an editor. But you can also have Composer do it for you. This is particularly useful if you need to kick off with a single package. When you invoke `composer require` on the command line, Composer will download the specified package and install it into `vendor/` for you. It will also generate a `composer.json` file, which you can then edit and extend:

```
$ composer require orhanerday/open-ai
```

Output:

```
./composer.json has been created
Running composer update orhanerday/open-ai
Loading composer repositories with package information
Updating dependencies
Nothing to modify in lock file
Writing lock file
Installing dependencies from lock file (including require-dev)
Nothing to install, update or remove
Generating autoload files
```

```
1 package you are using is looking for funding.
Use the `composer fund` command to find out more!
No security vulnerability advisories found.
Using version ^5.1 for orhanerday/open-ai
```

Versions

Composer is designed to support semantic versioning. In essence, this involves defining a package's version with three numbers separated by dots: *major*, *minor*, and *patch*. If you fix a bug, add no functionality, and do not break backward compatibility, you should increment the *patch* number. If you add new functionality but do not break backward compatibility, you should increment the middle *minor* number. If your new version breaks backward compatibility (in other words, if client code would break if this new version were suddenly switched in), then you should increment the first *major* version number.

Note You can read more about the semantic versioning convention at https://semver.org.

You should bear this in mind when specifying versions in your `composer.json` file: if you are too liberal in your ranges or wildcards, you may find that your system breaks on update.

Table 5-1 shows some of the ways that you can specify versions with Composer.

Table 5-1. *Composer and Package Versions*

Type	Example	Notes
Exact	`1.2.2`	Only install the given version.
Wildcard	`1.2.*`	Install the exact specified numbers but find the latest available version matching the wildcard.
Range	`1.0.0-1.1.7`	Install a version no lower than the first number and no higher than the last number.
Comparison	`>1.2.0 <=1.2.2`	Use <, <=, >, and >= to specify complex ranges. You can combine these directives with a space (equivalent to "and") or with || to specify "or".
Tilde (major version)	`~1.3`	The given number is the minimum, and the final number specified can increase. So, for `~1.3`, 1.3 is the minimum, and there can be no match at 2.0.0 or above.
Caret	`^1.3`	Will match up to, but not including, the next breaking change. So, while `~1.3.1` will not match at 1.4 and above, `^1.3.1` will match from 1.3.1 up to, but not including, 2.0.0. This is generally the most useful shortcut.

Note You can further influence the way that composer selects packages by adding stability suffixes to your version constraint strings. By adding `@` followed by one of `dev`, `alpha`, `beta`, and `RC` (running from least to most stable), you will allow composer to consider nonstable versions in its calculations. Composer can work this out by looking at the Git tag names. So, `1.2.*@dev` can match the tag `1.2.2-dev`. You can also use the stability flag `stable` to signal that you do not want to include bleeding-edge code. This will match version tags which are not defined as `dev`, `beta`, and so on.

require-dev

Very often, you need packages during development that are unnecessary in a production context. You will want to run tests locally, for example, but you are unlikely to need PHPUnit available on your public site.

Composer addresses this by supporting a separate `require-dev` element. You can add packages here, just as you can for the `require` element:

```
{
    "require-dev": {
        "phpunit/phpunit": "*"
    },
    "require": {
        "orhanerday/open-ai": "^5.0"
    }
}
```

Now, when we run `composer update`, PHPUnit and all sorts of dependent packages are downloaded and installed:

```
Loading composer repositories with package information
Updating dependencies
Nothing to modify in lock file
Installing dependencies from lock file (including require-dev)
Package operations: 27 installs, 0 updates, 0 removals
```

```
  - Installing orhanerday/open-ai (5.1): Extracting archive
  - Installing sebastian/version (5.0.0): Extracting archive
  - Installing sebastian/type (5.0.0): Extracting archive
  ...
Generating autoload files
25 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
No security vulnerability advisories found.
```

If you're installing in a production context, however, you can pass the `--no-dev` flag to `composer install`, and Composer will download only those packages specified in the `require` element:

```
$ composer install --no-dev
Installing dependencies from lock file
Verifying lock file contents can be installed on current platform.
Package operations: 1 install, 0 updates, 0 removals
  - Installing orhanerday/open-ai (5.1): Extracting archive
Generating autoload files
1 package you are using is looking for funding.
Use the `composer fund` command to find out more!
```

Note As a reminder, when you run the `composer install` command, Composer looks for a file named `composer.lock`. If this file is not present, the command will behave like `composer update` – dependencies will be freshly calculated, and the `composer.lock` file will be generated. This records the exact version of every file you installed under `vendor/`.

If you run `composer install` and a `composer.lock` file is already present alongside `composer.json`, Composer will fetch the package versions it finds there. This is useful because you can commit a `composer.lock` file to your version control repository and be sure that your team will download the same versions of all the packages you have installed. If you need to override `composer.lock`, either to get the latest versions of packages or because you have changed `composer.json`, you should run `composer update` to override the lock file.

Composer and Autoload

We covered autoloading in some detail in Chapter 3. For the sake of completeness, however, it is worth looking at it briefly here. Composer generates a file named `autoload.php`, which handles class loading for the packages it downloads. You can also leverage this functionality for your own code by including `autoload.php` (usually by invoking `require_once`). Once you have done this, any class you declare in your system will be found automatically when accessed in your code, so long as your directories and file names mirror your namespaces and class names.

In other words, a class named `poppbook\megaquiz\command\CommandContext` must be placed in a file named `CommandContext.php` in the `poppbook/megaquiz/command/` directory.

If you want to mix things up (perhaps by omitting a redundant leading directory or two or by adding a test directory to the search path), then you can use the `autoload` element to map a namespace to your file structure, like this:

```
"autoload": {
    "psr-4": {
        "poppbook\\megaquiz\\": ["src", "test"]
    }
}
```

In order to generate the latest `autoload.php` file, I need to run one of `composer install` (will also install anything specified in the lock file) or `composer update` (will also install the latest packages that match the specification in `composer.json`). If you do not want to install or update any packages, you can use `composer dump-autoload` which will only generate autoload files.

Now, so long as `autoload.php` is included, my classes are easily discoverable. Thanks to my `autoload` configuration, the `poppbook\megaquiz\command\CommandContext` class will be found in `src/command/CommandContext.php`. Not only that, because I have referenced more than one target (`test` as well as `src`), I can also create test classes that belong to the `poppbook\megaquiz` namespace under the `test/` directory.

Turn to the "PSR-4 Autoloading" section in Chapter 3 to follow a more in-depth example.

Creating Your Own Package

If you have worked with PEAR in the past, you might expect a section on creating a package to involve an entirely new package file. In fact, we've already been creating a package throughout this chapter. We just have to add some more information and then find a way to make our code available to others.

Adding Package Information

You really do not have to add that much information to make a viable package, but you absolutely need a `name` so that it can be referenced and found. I'll also include `description` and `authors` elements to provide more complete information.

```
"name": "poppbook/megaquiz",
"description": "a truly mega quiz",
"authors": [
    {
        "name": "matt zandstra",
        "email": "matt@getinstance.com"
    }
],
```

These fields should be mostly self-explanatory. The exception might be that leading namespace – `poppbook` in this case – which is separated from the actual package name by a forward slash. This is known as the *vendor name* and it's a PSR-4 requirement. As you might expect, the vendor name becomes a top-level directory under `vendor/` when your package is installed. This is often the organization name used by the package owner in GitHub or Bitbucket.

With all that in place, you are ready to commit your package to your version control host of choice. If you're not sure what that involves, you can learn a lot more about this subject in Chapter 6.

Note Composer supports a `version` field, but it is considered better practice to use a tag in Git to track your package's version. Composer will automatically recognize this.

Remember that you should not push the `vendor` directory (at least not usually – there are some arguable exceptions to that rule). However, it is a good idea to track the generated `composer.lock` file alongside `composer.json`.

Platform Packages

Although you cannot use Composer to install system-wide packages, you *can* specify system-wide requirements, so that your package will only install in a system which is ready for it.

A platform package is specified with a single key, though in a couple of cases the key is further broken down by type, using a dash. I list the available types in Table 5-2.

Table 5-2. *Platform Packages*

Type	Example	Description
PHP	`"php": "8.*"`	The PHP version
Extension	`"ext-xml": ">2"`	A PHP extension
Library	`"lib-iconv": "~2"`	A system library used by PHP
HHVM	`"hhvm": "~2"`	An HHVM version (HHVM is a virtual machine that supports an extended version of PHP)

Let's try it out:

```
{
    "require": {
        "orhanerday/open-ai": "^5.0",
        "ext-xml": "*",
        "ext-gd": "*"
    }
}
```

In the preceding code, I specify that my package requires the `xml` and `gd` extensions. Now, it's time to run `update`:

```
$ composer update
Loading composer repositories with package information
Updating dependencies
Your requirements could not be resolved to an installable set of packages.
```

```
  Problem 1
      - Root composer.json requires PHP extension ext-gd * but it is
        missing from your system. Install or enable PHP's gd extension.

        To enable extensions, verify that they are enabled in your
        .ini files:
            - /usr/local/etc/php/conf.d/docker-php-ext-sodium.ini
              You can also run `php --ini` in a terminal to see which
              files are
              used by PHP in CLI mode.
              Alternatively, you can run Composer with `--ignore-platform-
              req=ext-gd`
              to temporarily ignore these required extensions.
```

It looks as though I was set up for XML; however, GD, an image manipulation package, is not installed on my system, so Composer throws an error. Notice that Composer gave me an option. I could, if I wanted, rerun the command with the flag `--ignore-platform-req=ext-gd`. Of course, my code might not run as expected without the required extension.

Distribution Through Packagist

If you've been working through this chapter, you might have wondered where the packages we have been installing actually come from. It feels a lot like magic, but (as you might expect) there is a package repository behind the scenes. It is called Packagist, and it can be found at https://packagist.org. So long as your code can be found in a public Git repository, it can be made available through Packagist.

Let's give it a shot. I have pushed my `megaquiz` project to GitHub, so now I need to tell Packagist about my repository. Once I have signed up, I simply add the URL of my repository. You can see this in Figure 5-1.

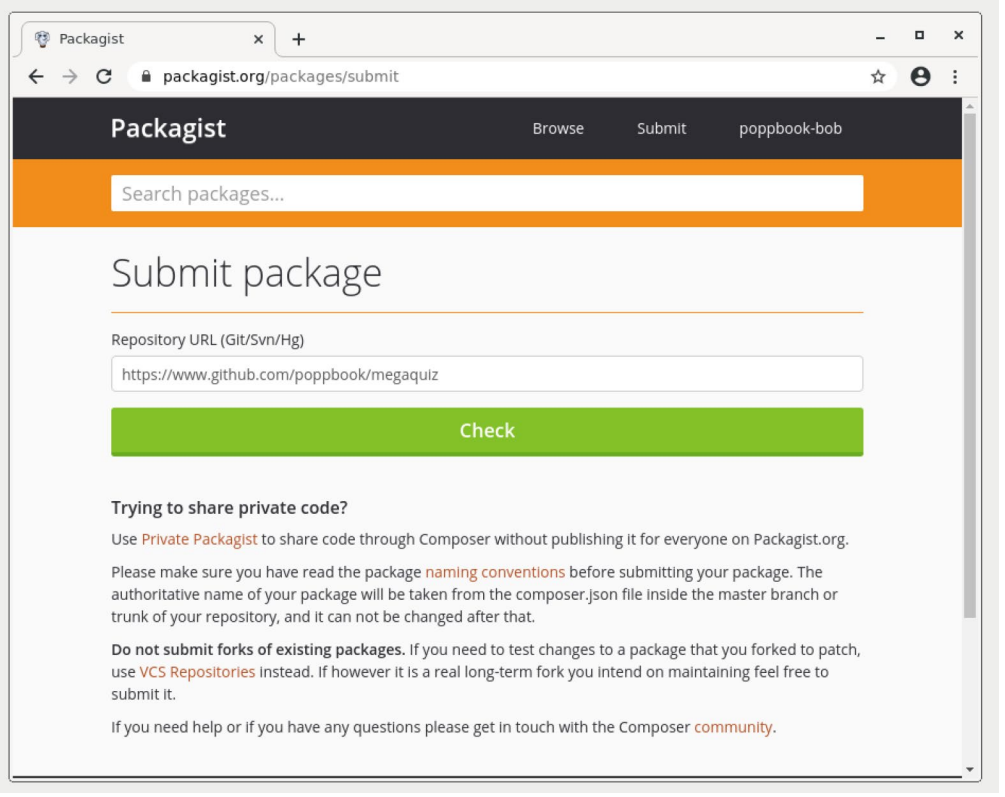

Figure 5-1. *Adding a package to Packagist*

Once I've added `megaquiz`, Packagist locates the repository, checks the `composer.json` file, and displays a control panel. You can see that in Figure 5-2.

Packagist tells me that I have not set license information. I can fix this at any time by adding a `license` element to the `composer.json` file:

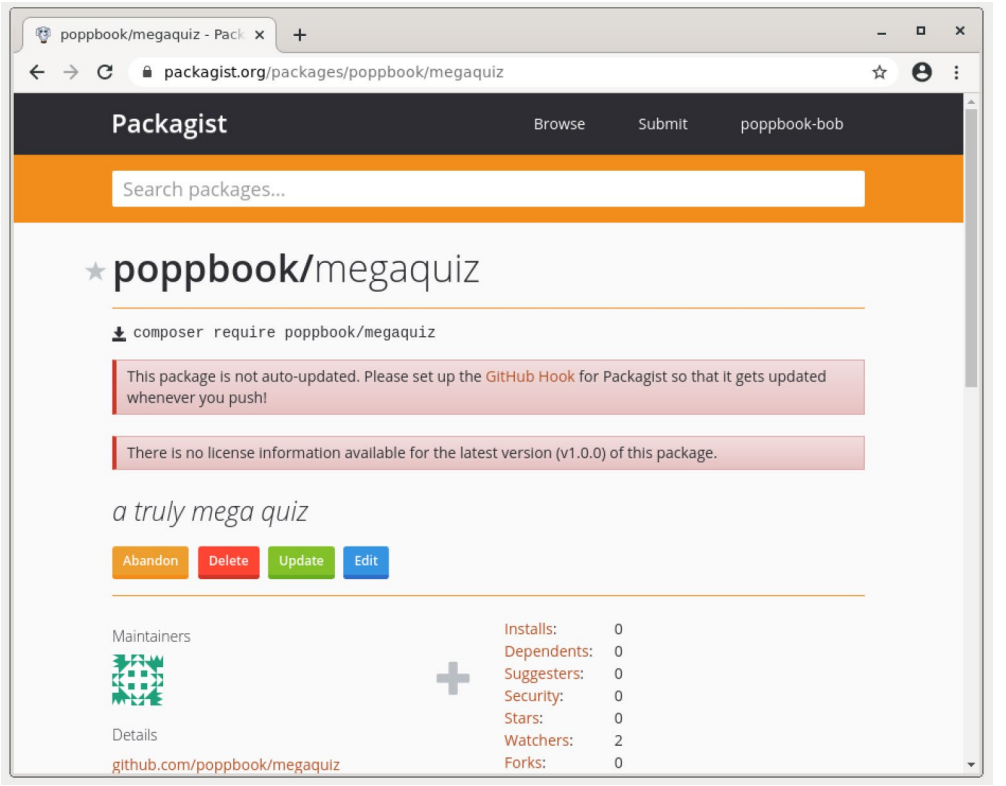

Figure 5-2. *The package control panel*

```
"license": "Apache-2.0",
```

Packagist has also failed to find any version information. I'll fix this by adding a tag to the GitHub repository:

```
$ git tag -a 'v1.0.0' -m 'v1.0.0'
$ git push –tags
```

Note If you think I'm cheating by skimming over this Git stuff, you're right. I cover both Git and GitHub in some detail in Chapter 6.

Now, Packagist knows about my version number. You can see that in Figure 5-3.

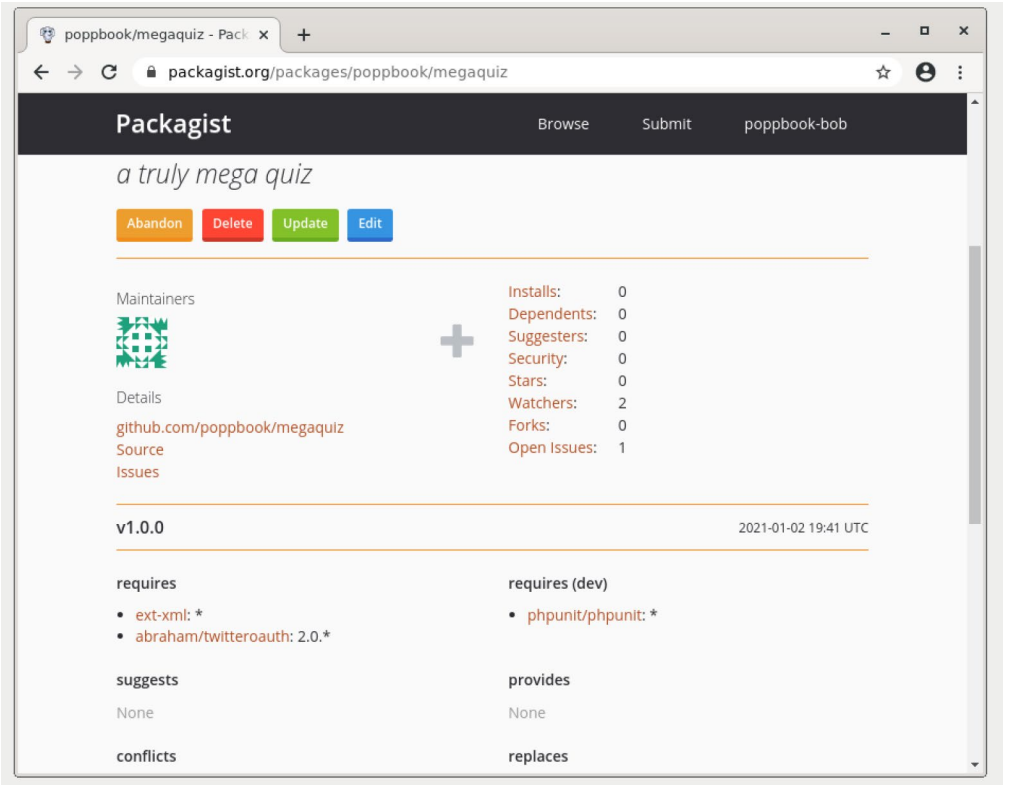

Figure 5-3. *Packagist knows the version*

At this point, anyone can include `megaquiz` from another package. Here is a minimal `composer.json` file:

```
{
    "require": {
        "poppbook/megaquiz": "*"
    }
}
```

I specify the vendor name and the package name. Riskily, I am happy to accept any version at all. Let's go ahead and install:

```
$ composer update
Loading composer repositories with package information
Updating dependencies
Lock file operations: 2 installs, 0 updates, 0 removals
```

```
  - Locking orhanerday/open-ai (5.1)
  - Locking poppbook/megaquiz (v1.0.1)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Downloading poppbook/megaquiz (v1.0.1)
  - Installing orhanerday/open-ai (5.1): Extracting archive
  - Installing poppbook/megaquiz (v1.0.1): Extracting archive
Generating autoload files
1 package you are using is looking for funding.
Use the `composer fund` command to find out more!
No security vulnerability advisories found.
```

Notice that the dependencies I specified when I set up `megaquiz` are also downloaded.

Keeping It Private

Of course, you don't always want to publish your code to the world. Sometimes, you need to share only with a smaller set of authorized users.

Here is a private package named `getinstance/wtnlang-php` which contains a library for a scripting language:

```
{
    "name": "getinstance/wtnlang-php",
    "description": "it's a wtn language",
    "license": "private",
    "authors": [
        {
            "name": "matt zandstra",
            "email": "matt@getinstance.com"
        }
    ],
    "autoload": {
        "psr-4": {
```

```
            "getinstance\\wtnlang\\": ["src/", "test/unit"]
        }
    },
    "require": {
        "aura/cli": "^2.2",
        "monolog/monolog": "^3.5"
    },
    "require-dev": {
        "phpunit/phpunit": "^11.1"
    }
}
```

This is hosted in a private Bitbucket repository, so it's not available via Packagist. So, how would I include it in a project? I simply need to tell Composer where to look. I can do this by creating or adding to the `repositories` element:

```
{
    "repositories": [
        {
            "type": "vcs",
            "url": "git@bitbucket.org:getinstance/wtnlang-php.git"
        }
    ],
    "require": {
        "poppbook/megaquiz": "*",
        "getinstance/wtnlang-php": "dev-develop"
    }
}
```

I could have specified a version for `getinstance/wtnlang-php` in the `require` block, and that would correspond to a tag in the Git repository, but, by using the `dev-` prefix, I can call for a branch. This is very useful during development. So now, so long as I have access to getinstance/wtnlang-php, I can install both my private package and `megaquiz` at once:

```
$ composer update
Loading composer repositories with package information
```

```
Updating dependencies
Nothing to modify in lock file
Installing dependencies from lock file (including require-dev)
Package operations: 7 installs, 0 updates, 0 removals
  - Syncing getinstance/wtnlang-php (dev-develop f33515e) into cache
  - Installing composer/ca-bundle (1.5.0): Extracting archive
  - Installing psr/log (3.0.0): Extracting archive
  - Installing monolog/monolog (3.5.0): Extracting archive
  - Installing aura/cli (2.2.0): Extracting archive
  - Installing getinstance/wtnlang-php (dev-develop f33515e): Cloning
    f33515e3f7 from cache
  - Installing orhanerday/open-ai (5.1): Extracting archive
  - Installing poppbook/megaquiz (v1.0.1): Extracting archive
Generating autoload files
3 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
No security vulnerability advisories found.
```

Summary

You should leave this chapter with a sense of how easy it is to leverage Composer packages to add power to your projects. Through the `composer.json` file, you can also make your code accessible to other users, whether publicly by using Packagist or by specifying your own repository. This approach automates dependency downloads for your users and allows third-party packages to use yours without the need for bundling.

CHAPTER 6

Version Control with Git

All disasters have their tipping point, the moment at which order finally breaks down and events simply spiral out of control. Do you ever find yourself in projects like that? Are you able to spot that crucial moment?

Perhaps it's when you make "just a couple of changes" and find that you have brought everything crashing down around you (and, even worse, you're not quite sure how to get back to the point of stability you have just destroyed). It could be when you realize that three members of your team have been working on the same set of classes and merrily saving over each other's work. Or perhaps it's when you discover that a bug fix that you have implemented twice has somehow disappeared from the code base yet again. Wouldn't it be nice if there were a tool to help you manage collaborative working, allowing you to take snapshots of your projects, roll them back if necessary, and then merge multiple strands of development? In this chapter, we look at Git, a tool that does all that and more.

This chapter will cover the following aspects of working with Git:

- *Basic configuration*: Exploring some tips for setting up Git
- *Importing*: Starting a new project
- *Committing changes*: Saving your work to the repository
- *Updating*: Merging other people's work with your own
- *Branching*: Maintaining parallel strands of development

Why Use Version Control?

If it hasn't already, version control will change your life (if only your life as a developer). How many times have you reached a stable moment in a project, drawn a breath, and plunged onward into development chaos once again? How easy was it to revert to the stable version when it came time to demonstrate your work in progress? Of course, you

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_6

may have saved a snapshot of your project when it reached a stable moment, probably by duplicating your development directory. Now, imagine that your colleague is working on the same code base. Perhaps he has saved a stable copy of the code as you have. The difference is that his copy is a snapshot of his work, not yours. Of course, he has a messy development directory, too. So you have four versions of your project to coordinate. Now, imagine a project with four programmers and a web UI developer. You're looking pale. Perhaps you would like to lie down?

Git exists exclusively to address this problem. Using Git, all of your developers can clone their own copies of the code base from a central repository. Whenever they reach a stable point in their code, they can pull the latest code from the server and merge it with their own recent work. When they are ready, and after they have fixed any conflicts and run all tests, they can push their new stable synthesis back into the shared repository.

Git is a distributed version control system. This means that, once they have acquired a branch, users commit to their own local repository without the need for a network connection. There are a number of benefits to this. It means that day-to-day operations are faster and that you can work easily on planes and trains and in automobiles. Ultimately, however, you can share an authoritative repository with your teammates.

The fact that each developer can merge their work into a central repository means that reconciling multiple strands of development is made vastly easier. Even better, you can check out versions of your code base based on a date or a label. So when your code reaches a stable point, suitable for showing to a client as work in progress, for example, you can tag that with an arbitrary label. You can then use that tag to check out the correct code base when your client swoops into your office looking to impress an investor.

Wait! There's more! You can also manage multiple strands of development at the same time. If this sounds needlessly complicated, imagine a mature project. You have already shipped the first version, and you're well into development of version 2. Does version 1.n go away in the meantime? Of course not. Your users are spotting bugs and requesting enhancements all the time. You may be months away from shipping version 2, so where do you make and test the changes? Git lets you maintain distinct branches of the code base. So you might create a bug fix branch of your version 1.n for development on the current production code. At key points, this branch can be merged back into the version 2 code (the trunk), so that your new release can benefit from improvements to version 1.n.

Note Git is not the only version control system available. You might also like to look into Subversion (https://subversion.apache.org/) or Mercurial (https://www.mercurial-scm.org/). This chapter is necessarily a brief introduction to a large topic. Luckily, however, *Pro Git* by Scott Chacon (Apress, 2014) covers the topic with depth and clarity. Not only that, but a web version is available online at https://git-scm.com/book/en/v2.

Let's get on and look at some of these features in practice.

Getting Git

If you are working with a Unix-like operating system (such as Linux or FreeBSD), you may already have Git installed and ready to use.

Note I show commands that are input at the command line with a leading dollar sign ($) to represent the command prompt to distinguish them from any output they may produce.

Try typing this from the command line:

```
$ git help
```

You should see some usage information that will confirm that you are ready to get started. If you do not already have Git, you should consult your distribution's documentation. You will almost certainly have access to a simple installation mechanism such as Yum or Apt, or you can acquire Git directly from https://git-scm.com/downloads.

Note Technical reviewer Paul Tregoing also recommends Git for Windows (https://gitforwindows.org/) which comes with Git, naturally, but also a set of useful open source tools.

Using an Online Git Repository

You may have noticed by now that this book often goes it alone. I almost never argue that you should reinvent the wheel; rather, you should at least get a sense of what goes into wheel construction before buying one ready-made. For this reason, I'll be covering the mechanics of setting up and maintaining your own central Git repository in the next section. Let's get real, though. You'll almost certainly use a specialized host to manage your repositories. There are a number of these to choose from, though the biggest players are probably Bitbucket (`https://bitbucket.org`), GitHub (`https://github.org`), and GitLab (`https://about.gitlab.com/`).

So, which should you choose? As a rule of thumb, GitHub is the standard for open source products. So, I'll sign up with GitHub for my project. Figure 6-1 shows my next decision, which is the choice between a public and a private repository. I'll opt for a public project (because I'm creating an open source project).

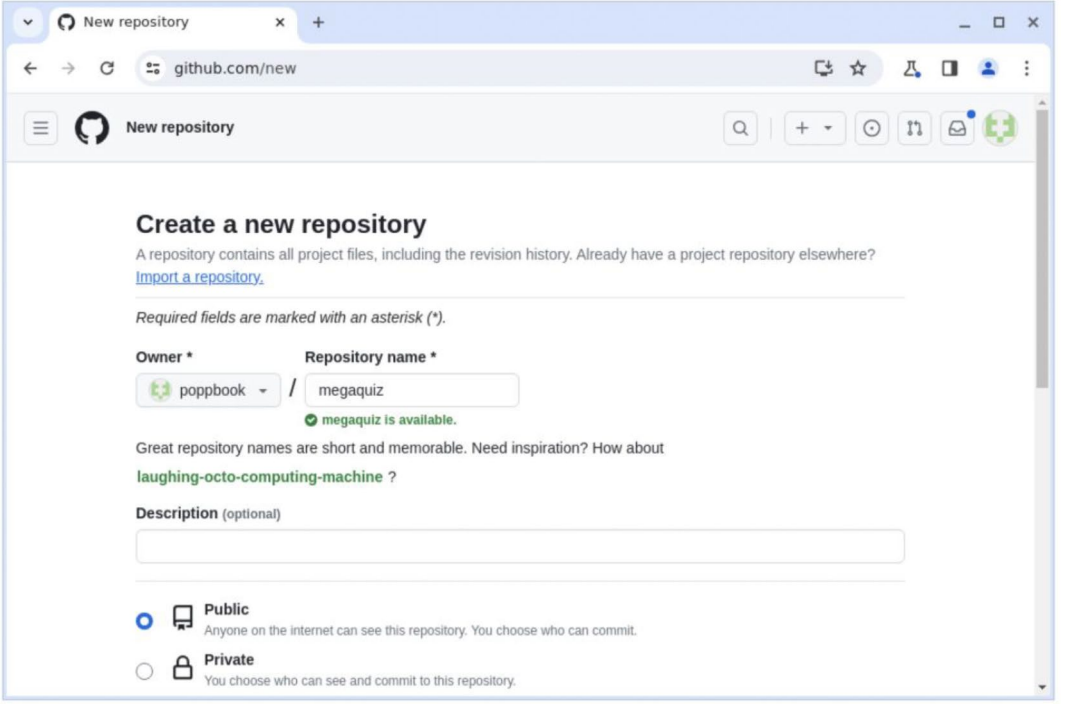

Figure 6-1. *Getting started with a GitHub project*

As you can see, in Figure 6-1, GitHub confirms that the `megaquiz` repository is available within my namespace. At this point, GitHub offers some helpful instructions for importing my project. You can see those in Figure 6-2.

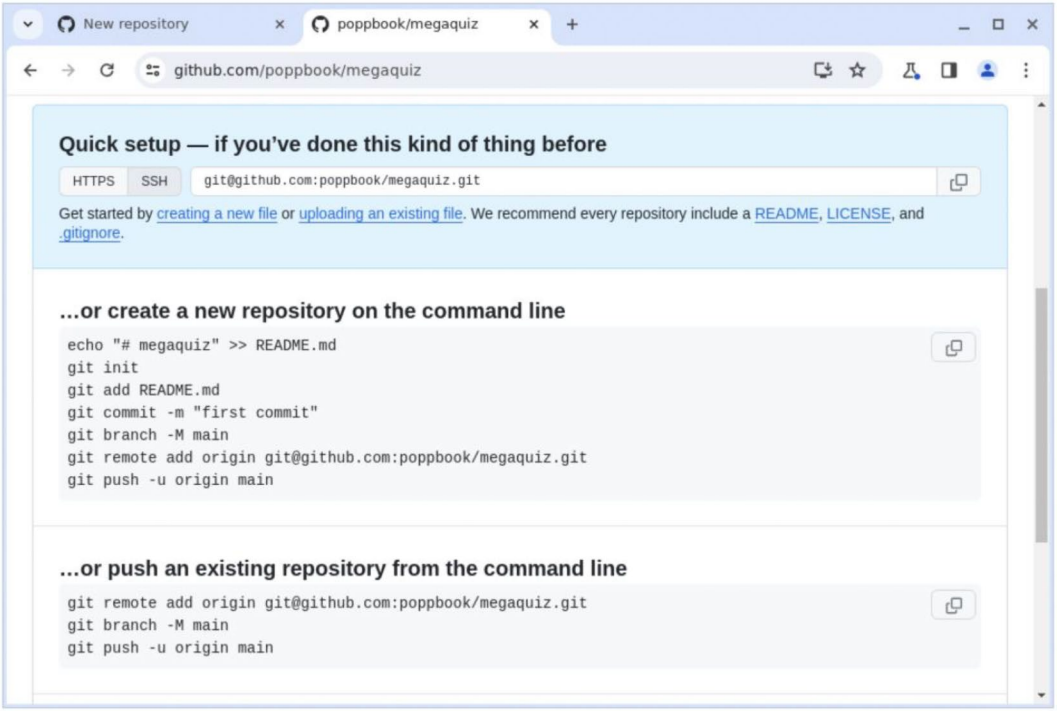

Figure 6-2. *GitHub's import instructions*

I'm not ready to run those commands yet, though. GitHub needs to be able to validate me when I push files to the server. In order to do that, it requires my public key. I describe one way of generating such a key in the next section, "Configuring a Git Server." Once I have a public key, I can add it from the `SSH and GPG keys` link in GitHub's User Settings screen.

You can see GitHub's settings screen for SSH and GPG keys in Figure 6-3.

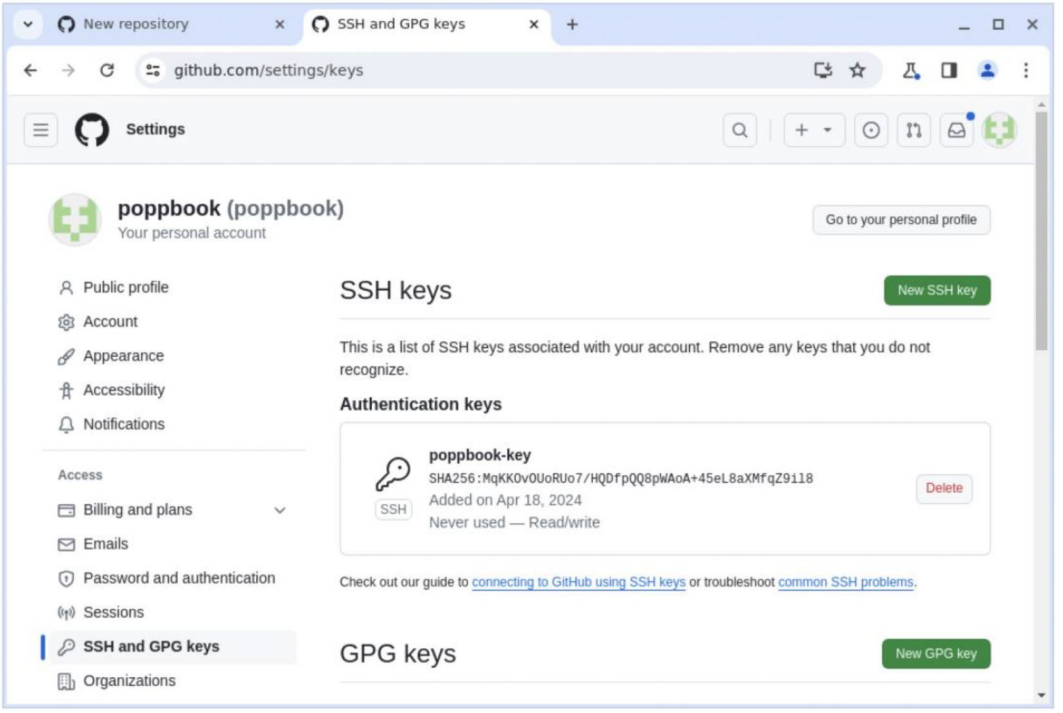

Figure 6-3. *Adding an SSH key*

Now, I'm ready to start adding files to my repository. Before we get into that, though, we should step back and spend some time following the do-it-yourself route.

Configuring a Git Server

Git is different from traditional version control systems in two key ways. First, under the hood, it stores snapshots of files rather than the changes made to files between commits. Second, and more obviously to the user, it operates locally to your system until you choose to push to or pull from a remote repository. This means that you are not dependent on an Internet connection to get on with your work.

You do not need a single remote repository in order to work with Git, but in practice, it almost always makes sense to have a shared source of authority if you are working with a team.

In this section, I look at the steps needed to get a remote Git server up and running. I assume root access to a Linux machine.

Creating the Remote Repository

In order to create a Git repository, I must first create a containing directory. I log in to a freshly provisioned remote server via SSH. I am going to create my repository under `/var/git`. Generally speaking, only the root user can create and modify directories there, so I run the following command using `sudo`:

```
$ sudo mkdir -p /var/git/megaquiz
$ cd /var/git/megaquiz/
```

I create `/var/git`, a parent directory for my repositories and a subdirectory for a sample project called `megaquiz`. Now, I can prepare the directory itself:

```
$ sudo git init --bare
Initialized empty Git repository in /var/git/megaquiz/
```

The `--bare` flag tells Git to initialize a repository without a working directory. Git will complain if you try to push to a repository that has not been created in this way.

At the moment, only the root user can mess around under `/var/git`. I can change this by creating a user and a group named `git` and making it the directory's owner:

```
$ sudo adduser git
$ sudo chown -R git:git /var/git
```

Preparing the Repository for Local Users

Although this is a designated remote server, I should also ensure that local users can commit to the repository. If you're not careful, this can cause ownership and permissions issues (especially if users with `sudo` privileges push code).

```
$ sudo chmod -R g+rws /var/git
```

This gives members of the `git` group write access to `/var/git` and causes all files and directories created here to take on the `git` group. Now, as long as I ensure that they are members of the `git` group, local users will be able to write to the repository.

You can add a local user to the `git` group like this:

```
$ sudo usermod -aG git bob
```

Now, user `bob` is a member of the `git` group.

Providing Access to Users

The owner of the bob user mentioned in the previous section can log in to the server and interact with the repository from his shell. Generally, though, you won't want to provide shell access to all your users. In any case, most users will prefer to take advantage of Git's distributed nature and to work locally with their cloned data.

One way to grant a user SSH access is via public key authentication. To do this, you first need to acquire the user's public SSH key. The user may already have this – on a Linux machine, he will probably find the key in the configuration directory, `.ssh`, in a file named `id_rsa.pub` or `id_ed25519.pub` depending upon the algorithm used to generate the key. Otherwise, he can easily generate a new key. On a Unix-like machine, this is a matter of running the `ssh-keygen` command and copying the value that it generates:

```
$ ssh-keygen -t ed25519 -C "you@example.com"
$ cat ~/.ssh/id_ed25519.pub
```

As the repository administrator, I will have asked you for a copy of this key. Once I have it, I must add it to the `git` user's SSH setup on the repository server. This is merely a matter of pasting the public key into the `.ssh/authorized_keys` file. I may need to create the `.ssh` configuration directory for the first key I set up (I am running these commands from the `git` user's home directory):

```
$ mkdir .ssh
$ chmod 0700 .ssh
```

Now, I can create the `.ssh/authorized_keys` file and paste in the user's key:

```
$ vi .ssh/authorized_keys
$ chmod 0700 .ssh/authorized_keys
```

Note A common cause of SSH access failure is the creation of configuration files with overly liberal permissions. The SSH configuration environment should be readable and writable to the account's owner only. *Pro OpenSSH* by Michael Stahnke (Apress, 2005) covers SSH comprehensively.

Closing Down Shell Access for the Git User

No server should be any more open than it needs to be. You may want to enable your user to access Git commands, but probably not much more.

You can see the shell associated with a user on a Linux server by looking at the file, `/etc/passwd`. Here is the relevant line for the `git` account on my remote server:

```
git:x:1001:1001::/home/git:/bin/bash
```

Git provides a special shell, named `git-shell`, that restricts the user to selected commands only. I can enable this program for logins by editing `/etc/passwd`:

```
git:x:1001:1001::/home/git:/usr/bin/git-shell
```

Now, if I attempt to log in via SSH, I'm told the score and logged out:

```
$ ssh git@poppch19.vagrant.internal
Last login: Mon Apr 15 14:25:05 2024 from 192.168.33.1
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute
access. Connection to 192.168.33.71 closed.
```

Beginning a Project

Now that I have a remote Git server and access to it from my local account, it's time to add my work in progress to the repository at `/var/git/megaquiz`.

Before I start, I take a good look at my files and directories and remove any temporary items I might find.

Failure to do this is a common annoyance. Temporary items to watch for include automatically generated files such as composer packages, build directories, installer logs, and so on.

Note You can specify files and patterns to ignore by placing a file named `.gitignore` in your repository. On a Linux system, the `man gitignore` command should provide examples of file name wildcarding that you can amend to exclude the various lock files and temporary directories created by your build processes, editors, and IDEs. This text is also available online at https://git-scm.com/docs/gitignore.

Before I go any further, I should register my identity with Git – this makes it easier to track who does what in the repository:

```
$ git config --global user.name "poppbook"
$ git config --global user.email "poppbook@getinstance.com"
$ git config --global init.defaultBranch "main"
```

I have also configured Git to default to a branch named `main`. This keeps us in line with GitHub which defaults to `main`. Now that I have established my personal details and ensured that my project is clean, I can set it up and push its code to the server:

```
$ cd /home/mattz/work/megaquiz
$ git init
Initialized empty Git repository in /home/mattz/work/megaquiz/.git/
```

Now, it's time to add my files:

```
$ git add .
```

Git is now tracking all the files and directories under `megaquiz`. Tracked files can be in three states: *unmodified*, *modified*, or *staged*. You can check this by running the command `git status`:

```
$ git status
# On branch main
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
```

```
#       new file:   composer.json
#       new file:   composer.lock
#       new file:   main.php
#       new file:   src/command/Command.php
#       new file:   src/command/CommandContext.php
#       new file:   src/command/FeedbackCommand.php
#       new file:   src/command/LoginCommand.php
#       new file:   src/quizobjects/User.php
#       new file:   src/quiztools/AccessManager.php
#       new file:   src/quiztools/ReceiverFactory.php
#
```

Thanks to my previous `git add` command, all my files are staged for commit. I can go ahead now and execute the `commit` command:

```
$ git commit -m'initial commit'
[main (root-commit) a5ca2d4] initial commit
10 files changed, 1638 insertions(+)
create mode 100644 composer.json
create mode 100644 composer.lock
create mode 100755 main.php
create mode 100755 src/command/Command.php
create mode 100755 src/command/CommandContext.php
create mode 100755 src/command/FeedbackCommand.php
create mode 100755 src/command/LoginCommand.php
create mode 100755 src/quizobjects/User.php
create mode 100755 src/quiztools/AccessManager.php
create mode 100644 src/quiztools/ReceiverFactory.php
```

I add a message via the `-m` flag. If I omitted this, then Git would launch an editor that I can use to add my check-in message.

If you are accustomed to version control systems such as CVS and Subversion, you might think that we're done. And although I could happily continue editing, adding, committing, and branching from here, there is an additional stage I need to consider if I want to share this code using a central repository. As we will see later on in the chapter, Git allows us to manage multiple project branches. Thanks to this feature, I can maintain a branch for each release but also keep my bleeding-edge risky development safely out

of my production code. We have configured Git so that, when we start out, it sets up a single branch named `main` by default. I can confirm the state of my branches with the command `git branch`:

```
$ git branch -a
* main
```

The `-a` flag specifies that Git should show us both local and remote branches (the default is to omit the remote ones). And the output shows the `main` branch.

In fact, I have done nothing yet to associate my local repository with the remote server. It's time to put that right:

```
$ git remote add origin git@poppch19.vagrant.internal:/var/git/megaquiz
```

This command is disappointingly quiet, given the work that it has done. In fact, it is the equivalent of telling Git to "associate the nickname `origin` with the given server location. Furthermore, set up a tracking relationship between the local branch `main` and a remote equivalent."

To confirm all of this, I check with Git that the remote handle `origin` has been set up:

```
$ git remote -v
origin git@poppch19.vagrant.internal:/var/git/megaquiz (fetch)
origin git@poppch19.vagrant.internal:/var/git/megaquiz (push)
```

Of course, if you used a service like GitHub, you would use your equivalent of the `git remote add` step shown in Figure 6-2. In my case, that looks like this:

```
$ git remote add origin git@github.com:poppbook/megaquiz.git
```

Do not run the preceding command, though, unless you really want to push to my GitHub repo! I am sticking to my self-hosted Git repository for now.

I still haven't sent any actual files to my Git server, however, so that's my next step:

```
$ git push origin main
Counting objects: 16, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (16/16), 8.87 KiB | 0 bytes/s, done.
Total 16 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
```

```
To git@github.com:poppbook/megaquiz.git
* [new branch] main -> main
```

Now, I can run the `git branch` command again to confirm that the remote version of the `main` branch has appeared:

```
$ git branch -a
* main
  remotes/origin/main
```

Or to see only the remote branches:

```
$ git branch –r
   origin/main
```

Note I have established what is called a *tracking branch*. This is a local branch that is associated with a remote twin.

Cloning the Repository

For the purposes of this chapter, I have invented a team member named Bob. Bob is working with me on the megaquiz project. Naturally, he wants his own version of the code. I have already added his public key to the Git server, so he is good to go. In the parallel world of GitHub, I have invited Bob to join my project, and he has added his own public key to his account. The effect is the same; Bob can acquire the repository using the command `git clone`:

```
$ git clone git@github.com:poppbook/megaquiz.git
Cloning into 'megaquiz'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 16 (delta 2), reused 16 (delta 2), pack-reused 0
Receiving objects: 100% (16/16), 8.87 KiB | 0 bytes/s, done.
Resolving deltas: 100% (2/2), done.
```

Note Although I cloned over ssh here, Git supports cloning over https if your server also supports it.

Now, both of us can develop locally and, when we're ready, share our code with one another.

Updating and Committing

Bob is, of course, a fine and talented fellow – except, that is, for one common and highly annoying trait: he cannot leave other people's code alone.

Bob is smart and inquisitive, easily excited by shiny new avenues of development, and he's keen to help optimize new code. As a result, everywhere I turn, I seem to see the hand of Bob. Bob has added to my documentation, and he has implemented an idea I mentioned over coffee. I may have to kill Bob. In the meantime, however, I must handle the fact that the code on which I am working needs to be merged with Bob's input.

Here's a file called `quizobjects/User.php`. At the moment, it contains nothing but the barest of bones:

```
namespace poppbook\megaquiz\quizobjects;

class User
{
}
```

I have decided to add some documentation. I begin by adding a file comment to my version of the file:

```
namespace poppbook\megaquiz\quizobjects;

/**
 * @license http://www.example.com Borsetshire Open License
 * @package quizobjects
 */
class User
{
}
```

Remember that a file can have three states: *unmodified*, *modified*, and *staged*. The `User.php` file has now moved from *unmodified* to *modified*. I can see this with the `git status` command:

```
$ git status
# On branch main
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
    directory)
#
#      modified: src/quizobjects/User.php
#
no changes added to commit (use "git add" and/or "git commit -a")
```

`User.php` has been modified, but not yet staged for commit. I can change this state using the command `git add`:

```
$ git add src/quizobjects/User.php
$ git status
# On branch main
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified: src/quizobjects/User.php
#
```

Now, I am ready to commit:

```
$ git commit -m'added documentation' src/quizobjects/User.php
[main 997622c] added documentation
1 file changed, 5 insertions(+)
```

A Git commit only affects my local repository. If I am sure that the world is ready for my change, I must push my code to the remote repository:

```
$ git push origin main
Counting objects: 9, done.
Delta compression using up to 2 threads.
```

```
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 537 bytes | 0 bytes/s, done.
Total 5 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To git@github.com:poppbook/megaquiz.git
    ce5a604..997622c main -> main
```

Meanwhile, working in his own sandbox, Bob is keen as ever, and he has created a class comment:

```
namespace poppbook\megaquiz\quizobjects;

/**
 * @package quizobjects
 */
class User
{
}
```

Now, it's Bob's turn to `add`, `commit`, and `push`. Because the adding and committing parts of this are so commonly run together, Git allows you to combine them into a single command:

```
$ git commit -a -m'my great documentation'
[main 13de456] my great documentation
1 file changed, 4 insertions(+)
```

So, we now have two distinct versions of `User.php`. There's the version I just pushed to the remote repository, and there is Bob's version, committed, but not yet pushed. Let's see what happens when Bob tries to push his local version to the remote repository:

```
$ git push origin main
To git@github.com:poppbook/megaquiz.git
! [rejected]        main -> main (fetch first)
error: failed to push some refs to
'git@github.com:poppbook/megaquiz.git'
hint: Updates were rejected because the remote contains work that you do
```

```
hint: not have locally. This is usually caused by another
repository pushing
hint: to the same ref. You may want to first merge the remote
changes (e.g.,
hint: 'git pull') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

As you can see, Git won't let you push if there's an update to apply. Bob must first pull down my version of the `User.php` file:

```
$ git pull origin main
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 1), reused 5 (delta 1), pack-reused 0 Unpacking
objects: 100% (5/5), done.
From github.com:poppbook/megaquiz
* branch            main     -> FETCH_HEAD
Auto-merging src/quizobjects/User.php
CONFLICT (content): Merge conflict in src/quizobjects/User.php
Automatic merge failed; fix conflicts and then commit the result.
```

Git will happily merge data from two sources into the same file, so long as the changes don't overlap. Git has no means of handling changes that affect the same lines. How can it decide what is to have priority? Should the repository overwrite Bob's changes or the other way around? Should both changes coexist? Which should go first? Git has no choice but to report a conflict and let Bob sort out the problem.

Here's what Bob sees when he opens the file:

```
/**
<<<<<<< HEAD
 * @package  quizobjects
 */
=======
 * @license  http://www.example.com Borsetshire Open License
 * @package  quizobjects
 */
```

```
>>>>>>> f36c6244521dbd137b37b76414e3cea2071958d2

namespace poppbook\megaquiz\quizobjects;

class User
{
}
```

Git includes both Bob's comment and the conflicting changes, together with metadata that tells him which part originates where. The conflicting information is separated by a line of equals signs. Bob's input is signaled by a line of less than symbols followed by "HEAD". The remote changes are included on the other side of the divide.

Note The long list of numbers and letters shown in the conflict message is a commit ID. That is, a SHA-1 hash which references an individual commit. Such references can be used by various Git tools and are also shown in the `git log` command output which provides a record of parent commits from a given starting point. You will also see them in web interfaces that show commit histories. HEAD, here, refers to the tip (the most recent commit) of the currently checked out branch. You can find out the commit ID for HEAD (and much more) by running `git show HEAD`.

Now that Bob has identified the conflict, he can edit the file to fix the collision:

```
/**
 * @license http://www.example.com Borsetshire Open License
 * @package quizobjects
 */

namespace poppbook\megaquiz\quizobjects;

class User
{
}
```

Next, Bob resolves the conflict by staging the file:

```
$ git add src/quizobjects/User.php
$ git commit -m'documentation merged'
[main c99d3f5] documentation merged
```

And now, finally, he can push to the remote repository:

```
$ git push origin main
```

Adding and Removing Files and Directories

Projects change shape as they develop. Version control software must take account of this, allowing users to add new files and remove deadwood that would otherwise get in the way.

Adding a File

You have seen the `add` subcommand many times already. I used it during my project setup to add my code to the empty `megaquiz` repository and, subsequently, to stage files for commit. By running `git add` on an untracked file or directory, you ask Git to track it – and stage it for commit. Here, I add a document called `CompositeQuestion.php` to the project:

```
$ touch src/quizobjects/CompositeQuestion.php
$ git add src/quizobjects/CompositeQuestion.php
```

In a real-world situation, I would probably start out by adding some content to `CompositeQuestion.php`. Here, I confine myself to creating an empty file using the standard `touch` command. Once I have added a document, I must still invoke the `commit` subcommand to complete the addition:

```
$ git commit -m'initial check in'
[main 323bec3] initial check in
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 src/quizobjects/CompositeQuestion.php
```

`CompositeQuestion.php` is now in the local repository.

Removing a File

Should I discover that I have been too hasty and need to remove the document, it should come as no surprise to learn that I can use a subcommand called `rm`:

```
$ git rm src/quizobjects/CompositeQuestion.php
```

Once again, a `commit` is required to finish the job. As usual, I can confirm this by running `git status`:

```
$ git status
# On branch main
# Your branch is ahead of 'origin/main' by 1 commit.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    src/quizobjects/CompositeQuestion.php
#
$ git commit -m'removed Question'
[main 5bf88aa] removed CompositeQuestion
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 src/quizobjects/CompositeQuestion.php
```

Adding a Directory

You can also add and remove directories with `add` and `rm`. Let's say Bob wants to make a new directory available:

```
$ mkdir resources
$ touch resources/blah.gif
$ git add resources/
$ git status
# On branch main
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
#       new file: resources/blah.gif
#
```

Notice how the contents of `resources` are added automatically to the repository. Now, Bob can commit and then push the whole lot to the remote repository in the usual way.

Note Be careful of using `git add` with directories; it is greedy! The command will pick up any files and directories beneath the given directory. It is always a good idea to check the operation with `git status`.

Removing Directories

As you might expect, you can remove directories with the `rm` subcommand. In this situation, however, I must tell Git that I wish it to remove the directory's contents by passing an `-r` flag to the subcommand. Here, I profoundly disagree with Bob's decision to add a `resources` directory:

```
$ git rm -r resources/
```

Renaming Files or Directories

If you're feeling less drastic, you might opt to simply rename a file or directory with `git mv`. This command accepts two arguments: a file, directory, or a symlink and a destination.

```
$ git mv storage/ files
```

Tagging a Release

All being well, a project will eventually reach a state of readiness, and you will want to ship it or deploy it. Whenever you make a release, you should leave a bookmark in your repository, so that you can always revisit the code at that point. As you might expect, you can create a tag in your code with the `git tag` command:

```
$ git tag -a 'v1.0.0' -m'release 1.0.0'
```

By specifying `-a` here, I create an *annotated tag*. This stores rich information about my tag, including information about the commit it references, a tagging message, the date of the tag, and the tagging user. If I were to omit `-m` when creating an annotated tag, Git would launch an editor window and prompt me to provide a message.

I could also have created a *lightweight tag* by omitting `-a` and `-m`. A lightweight tag stores only a commit ID. Either approach would work for tagging a release.

You can see the tags associated with your repository by running `git tag` with no arguments:

```
$ git tag
v1.0.0
```

For long-lived projects you may accrue hundreds of tags. That can make the default `git tag` command unwieldy. You can use wildcards to narrow things down by specifying the `-l` flag and a filter argument. Here, for example, I return all tags that begin with `v1`.

```
$ git tag -l v1*
```

Of course, that's a little redundant in this case since I have only have one tag to list!

We have been working locally up until this point. In order to get the tag onto the remote repository, we must use the `--tags` flag with the `git push` subcommand:

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 159 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:poppbook/megaquiz.git
* [new tag]         v1.0.0 -> v1.0.0
```

Using the `--tags` flag causes all local tags to be pushed to the remote repository.

Of course, any action you take on a GitHub repo can be tracked on the site. You can see my release tag in Figure 6-4.

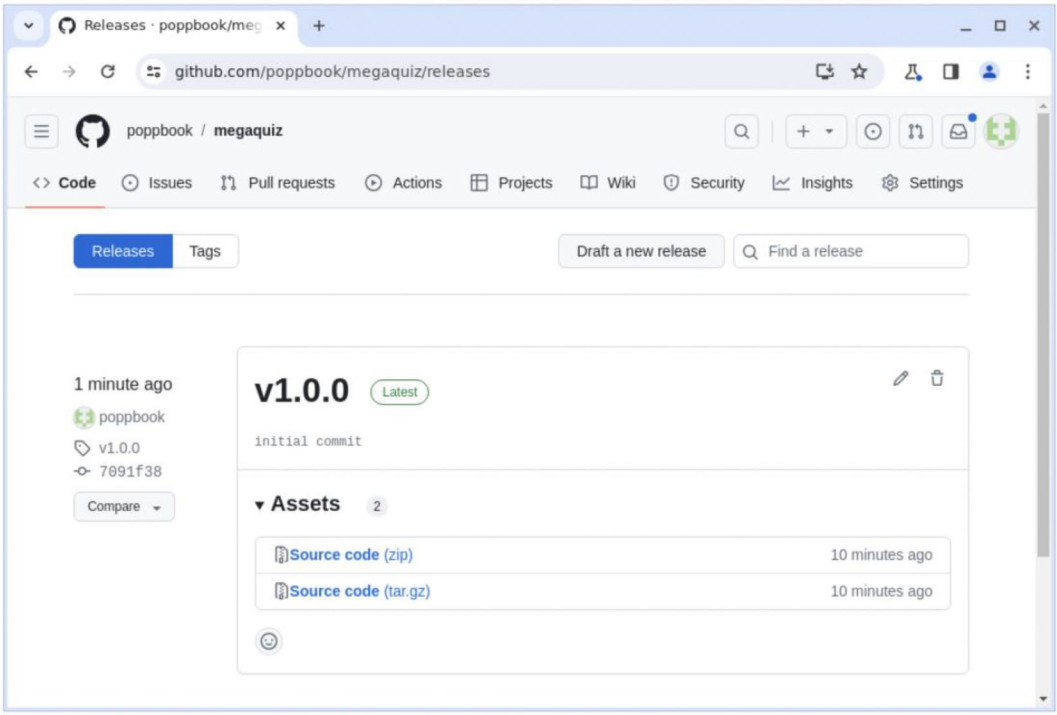

Figure 6-4. *Viewing a tag on GitHub*

Once you can bookmark your code with a tag, it makes sense to wonder how you might go about revisiting old releases. For this, however, you should first spend some time looking at branching – something at which Git is particularly good.

Branching a Project

Once my project has been released, I can pack it away and wander off to do something new, right? After all, it was so elegantly written that bugs are an impossibility, not to mention so thoroughly specified that no user could possibly require any new features!

Meanwhile, back in the real world, I must continue to work with the code base on at least two levels. Bug reports should be trickling in right about now, and the wish list for version 1.2.0 will be swelling with demands for fantastic new features. How do I reconcile these forces? I need to fix the bugs as they are reported, and I need to push on with primary development. I could fix the bugs as part of development and release everything in one go, when the next version is stable. But then, users may have a long wait before they see any problems addressed. This is plainly unacceptable. On the other hand,

I could release as I go. In that scenario, I risk shipping broken code. Clearly, I need two strands to my development. I will continue to add new and risky features to the project's main branch (often called the trunk), but I should now create a branch for my new release on which I can add only bug fixes.

Note This way of managing branches is by no means the only game in town. Developers argue constantly about the best way of organizing branches and managing releases and bug fixes. One of the most popular approaches is git-flow (neatly described at https://danielkummer.github.io/git-flow-cheatsheet/). Under this practice, `main` is the release branch. New code goes on a `develop` branch, and it's merged to `main` at release time. Each unit of active development has its own feature branch, which gets merged into `develop` when stable.

I can both create and switch to a new branch using the `git checkout` command. First, let's take a quick look at the state of my branches:

```
$ git branch -a
* main
  remotes/origin/main
```

As you can see, I have a single branch, `main`, and its remote equivalent. Now, if I invoke `git checkout` with the `-b` option, I will create and switch to a new branch with the latest commit in `main` as its merge base (the originating commit back to which I may, at some point, merge my changes).

```
$ git checkout -b megaquiz-branch1.0
Switched to a new branch 'megaquiz-branch1.0'
```

To track my use of branches, I will use a particular file as an example, `src/command/FeedbackCommand.php`. It seems that I created my bug fix branch just in time. Users have started to report that they are unable to use the feedback mechanism in the system. I locate the bug:

```
//...
$result = $msgSystem->despatch($email, $msg, $topic);
if (! $user)  {
```

```
    $this->context->setError($msgSystem->getError());
//...
```

I should, in fact, be testing `$result` and not `$user`. Here is my edit:

```
//...
$result = $msgSystem->dispatch($email, $msg, $topic);
if (! $result)  {
    $this->context->setError($msgSystem->getError());
//...
```

Because I am working on the branch `megaquiz-branch1.0`, I can commit this change:

```
$ git add src/command/FeedbackCommand.php
$ git commit -m'bugfix'
[megaquiz-branch1.0 6e56ade] bugfix
1 file changed, 1 insertion(+), 1 deletion(-)
```

Of course, this commit is local. I need to use the `git push` command to get the branch onto the remote repository:

```
$ git push origin megaquiz-branch1.0
Counting objects: 9, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 456 bytes | 0 bytes/s, done.
Total 5 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'megaquiz-branch1.0' on GitHub by
visiting:
remote: https://github.com/poppbook/megaquiz/pull/new/megaquiz-branch1.0
remote:
To git@github.com:poppbook/megaquiz.git
* [new branch]     megaquiz-branch1.0 -> megaquiz-branch1.0
```

Now, what about Bob? He will inevitably want to pitch in and fix some bugs. First, he invokes `git fetch`, which acquires any new information from the server. Then, he can look at all available branches with `git branch -a`.

```
$ git fetch
$ git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
  remotes/origin/megaquiz-branch1.0
```

Now, Bob can switch to a local branch which will track the remote one:

```
$ git checkout megaquiz-branch1.0
```

Notice that Bob did not use the `-b` option. That is only used when you want need to create a nonexistent branch.

```
Branch megaquiz-branch1.0 set up to track remote branch
megaquiz-branch1.0 from origin.
Switched to a new branch 'megaquiz-branch1.0'
```

Bob is good to go now. He can add and commit his own fixes, and when he pushes, they will end up on the remote branch.

Meanwhile, I would like to add some bleeding-edge enhancements on the trunk – that is, my `main` branch. Let's look again at the state of my branches from the perspective of my local repository:

```
$ git branch -a
  main
* megaquiz-branch1.0
  remotes/origin/main
  remotes/origin/megaquiz-branch1.0
```

I can switch to an existing branch by invoking `git checkout`:

```
$ git checkout main
Switched to branch 'main'
Your branch is up-to-date with 'origin/main'.
```

When I look now at command/FeedbackCommand.php, I see that my bug fix has magically disappeared. Of course, it's still stored under megaquiz-branch1.0. Later, I can merge the fix into the main branch, so there's no need to worry. Instead, I can focus on adding new code:

```
class FeedbackCommand extends Command
{
    public function execute(CommandContext $context): bool
    {
        // new and risky development
        // goes here
        $msgSystem = ReceiverFactory::getMessageSystem();
        $email = $context->get('email');
        // ...
```

All I have done here is to add a comment to simulate an addition to the code. I can now commit and push this:

```
$ git commit -am'new development on main'
$ git push origin main
```

So I now have parallel branches. Of course, sooner or later, I will want my main branch to benefit from the bug fixes that I have committed on megaquiz-branch1.0.

I can do this on the command line, but first, let's pause to look at a feature supported by GitHub and similar services like Bitbucket. The pull request (often abbreviated to PR) allows me to request a code review before merging a branch. So before megaquiz-branch1.0 hits main, I can ask Bob to check my work. As you can see in Figure 6-5, GitHub detects the branch and gives me the opportunity to issue my pull request.

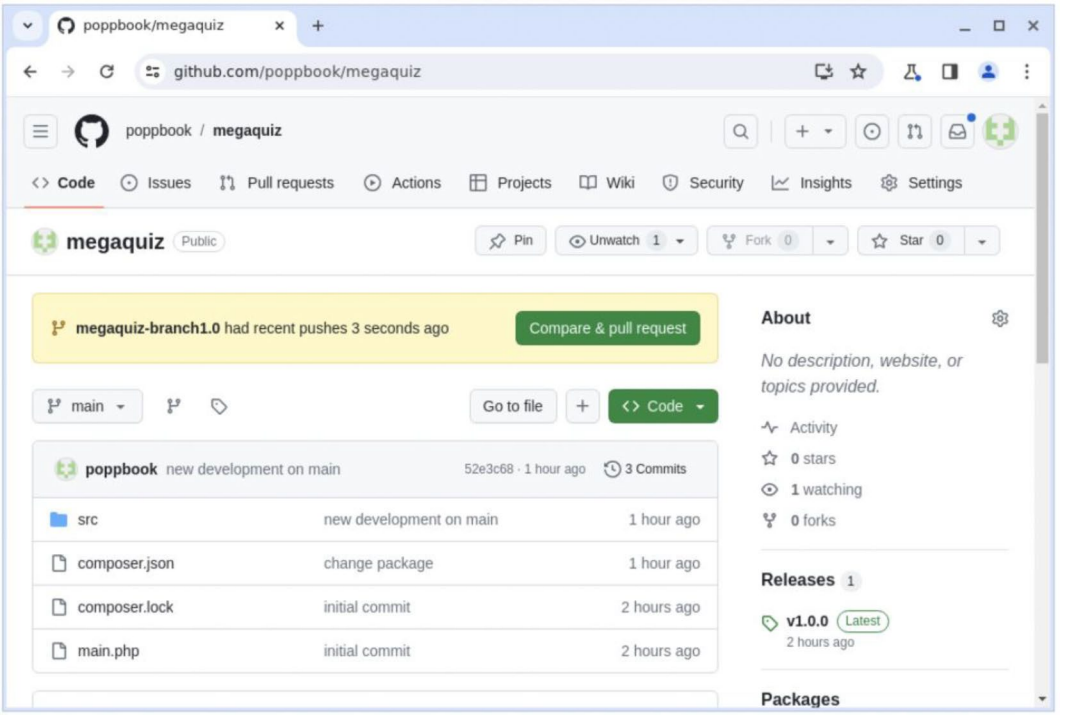

Figure 6-5. *GitHub makes issuing pull requests easy*

I hit the 'Compare & pull request' button and add a comment before submitting the pull request. You can see the result of that in Figure 6-6.

Now, Bob can examine my changes and add any comments he may have. GitHub shows him exactly what has changed. You can see Bob's comment in Figure 6-7.

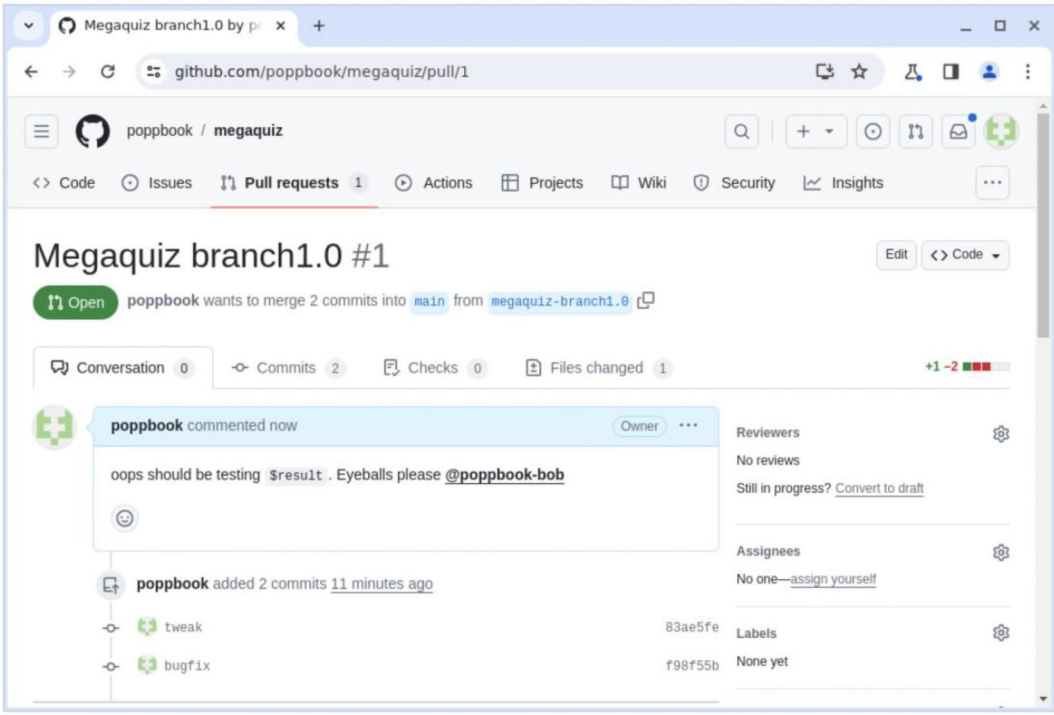

Figure 6-6. *Issuing the pull request*

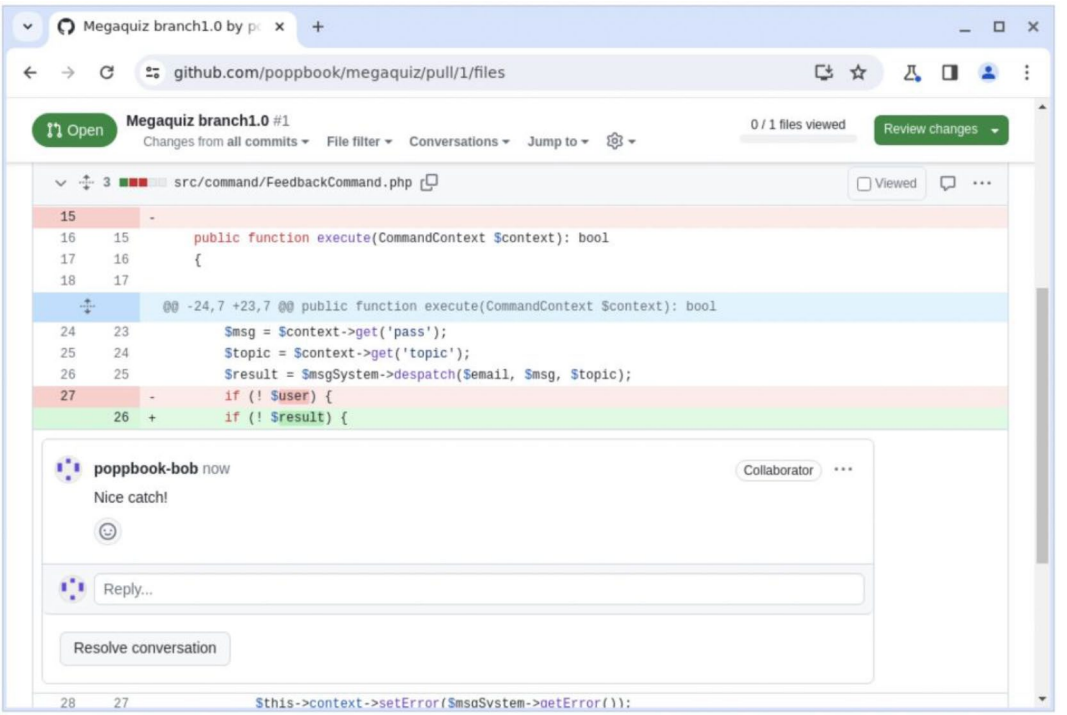

Figure 6-7. *The changes covered by a pull request*

Once Bob approves my pull request, I can merge directly from the browser, or I can return to the command line. This is pretty easy. Git provides a subcommand named `merge`:

```
$ git checkout main
Already on 'main'
```

In fact, I'm already on the `main` branch – but it can't hurt to be sure. Now, I perform the actual merge:

```
$ git merge --no-commit megaquiz-branch1.0
Auto-merging src/command/FeedbackCommand.php
Automatic merge went well; stopped before committing as requested
```

By passing in the `--no-commit` flag, I keep the merge uncommitted – which gives me another chance to check all is well. Once I am satisfied, I can go ahead and commit.

```
$ git commit -m'merge from megaquiz-branch1.0'
[main e1b5169] merge from megaquiz-branch1.0
```

Note To merge or not to merge? The choice is not always as straightforward as it might seem. In some cases, for example, your bug fix may be the kind of temporary work that is supplanted by a more thorough refactoring on the trunk, or it may no longer apply due to a change in specification. This is necessarily a judgment call. Most teams I have worked in, however, tend to merge to the trunk where possible while keeping work on the branch to the bare minimum. New features for us generally appear on the trunk and find their way quickly to users through a "release early and often" policy.

Now, when I look at the version of `FeedbackCommand` on the `main` branch, I confirm that all changes have been merged:

```
public function execute (CommandContext $context): bool
{
    // new and risky development
    // goes here
    $msgSystem = ReceiverFactory::getMessageSystem();
    $email = $context->get('email');
    $msg = $context->get('pass');
    $topic = $context->get('topic');
    $result = $msgSystem->despatch($email, $msg, $topic);
    if (! $result) {
        $this->context->setError($msgSystem->getError());
        return false;
    }
```

The `execute()` method now includes both my simulated `main` development and the bug fix.

Let's recap where we are right now. I am back on the `main` branch. I can check this at any time with `git status`. More specifically, I can also determine the commit ID at the tip of that branch with `git show`:

```
$ git show HEAD
```

This provides lots of information, but I can see that HEAD on main points to a commit ID which starts 52e3c68.

```
commit 52e3c680572f70a3497267b569844b950bbb3007 (HEAD -> main, origin/main)
Author: mattz <matt@getinstance.com>
...
```

If I create a new branch from this point with our old friend `git checkout -b <branchname>`, I will change branches, but until I perform any further commits on the new branch, my new HEAD will point to the same commit – 52e3c68.

Let's confirm this. First, I create the new branch:

```
$ git checkout -b some-new-dev
Switched to a new branch 'some-new-dev'
```

Now that I've switched to some-new-dev, I can get information about the HEAD of the branch:

```
$ git show HEAD
commit 52e3c680572f70a3497267b569844b950bbb3007 (HEAD -> some-new-dev,
origin/main, main)
Author: mattz <matt@getinstance.com>
...
```

So, I am parked at the same commit but I'm on a different branch. Once I start committing to some-new-dev, my HEAD there will point to a new commit ID. But I will be able to find the merge base – the common ancestor – for the two branches with `git merge-base`:

```
$ git merge-base main some-new-dev
52e3c680572f70a3497267b569844b950bbb3007
```

Now, this is not something you're likely to do very often – but it is useful here for illustration purposes. We can see that merging from some-new-dev to main will use the common ancestor – commit ID 52e3c68 – in its merge strategy.

This is useful background to another variation on `git checkout`.

First, let's switch back to main.

```
$ git checkout main
```

I created a branch when I first "released" megaquiz version 1.0, and that's what we have been working with. Remember, however, that I also created a tag at that stage. I promised at the time that I would show you how to access the tag.

The trick here is to create a new branch from `main` with the commit the tag points to at its HEAD.

```
$ git checkout -b v1.0.0-branch v1.0.0
Switched to a new branch 'v1.0.0-branch'
```

This looks like the `git checkout -b <branchname>` example you've already seen, except this time, I have provided an additional argument. This is an optional `<start-point>` value. In this case, that means the commit pointed to by the tag `v1.0.0`. So, as before, I have created a new branch – named `v1.0.0-branch` here – but its HEAD is defined by the provided start point rather than defaulting to the HEAD of the `main` branch as before.

Let's run `git show`:

```
$ git show HEAD
commit 7091f38e28d1c58dfcc7a1343435a1a6082cd869 (HEAD -> v1.0.0-branch,
tag: v1.0.0)
Author: mattz <matt@getinstance.com>
```

So, my new branch has a HEAD which points to the commit referenced by the `v1.0.0` tag – `7091f38`. I can develop from here, push the branch, and share just as you have seen. When ready, if I want, I can merge my new changes back to `main`.

Note Git is an amazingly versatile and useful tool. Like all powerful tools, its use can occasionally lead to unintended consequences. For those moments that you have backed yourself into a corner and need to reset things fast, technical reviewer Paul Tregoing recommends https://dangitgit.com/en (actually, he recommended the swearier version!). The site is full of recipes that might just save your sanity, so it is well worth bookmarking if you work seriously with Git.

Two other Git commands that are worth having in your arsenal are `git stash` and `git stash apply`. When you are up to your ears in local edits but are called to switch branches, your first option is to commit your work in progress. You may

not want to commit rough code, though. You might think that your only choice then is to throw away your local changes or copy them to temporary files. If you run `git stash`, however, all local changes are tucked away for you behind the scenes, and your branch is returned to its state at the last commit. You can go off and do your urgent work and, when you are ready, run `git stash apply` to get your uncommitted work back. It's like magic!

Summary

Git comprises an enormous number of tools, each with a daunting range of options and capabilities. I can only hope to provide a brief introduction in the space available. Nonetheless, if you only use the features that I have covered in this chapter, you should see the benefit in your own work, whether through protection against data loss or improvements in collaborative working.

In this chapter, we took a tour through the basics of Git. I looked briefly at configuration before importing a project. I checked out, committed, and updated code and then showed you how to tag and export a release. I ended the chapter with a brief look at branches, demonstrating their usefulness in maintaining concurrent development and bug fix strands in a project.

There is one issue that I have glossed over here, to some extent. We established the principle that developers should check out their own versions of a project. On the whole, however, projects will not run in place. In order to test their changes, developers need to deploy code locally. Sometimes, this is as simple as copying over a few directories. More often, however, deployment must address a whole range of configuration issues. In upcoming chapters, we will look at some techniques for automating this process.

CHAPTER 7

Testing with PHPUnit

Every component in a system depends, to a greater or lesser extent, on the implementation of its peers for its own continued smooth running. By definition, then, development breaks systems. As you improve your classes and packages, you must remember to amend any code that works with them. For some changes, this can create a ripple effect, affecting components far away from the code you originally changed. Eagle-eyed vigilance and an encyclopedic knowledge of a system's dependencies can help to address this problem. Of course, while these are excellent virtues, systems soon grow too complex for every unwanted effect to be easily predicted, not least because systems often combine the work of many developers. To address this problem, it is a good idea to test every component regularly. This, of course, is a repetitive and complex task, and as such, it lends itself well to automation.

Among the test solutions available to PHP programmers, PHPUnit is perhaps the most ubiquitous and certainly the most fully featured tool. In this chapter, you will learn the following about PHPUnit:

- *Installation*: Using Composer to install PHPUnit
- *Writing tests*: Creating test cases and using assertion methods
- *Handling exceptions*: Strategies for confirming failure
- *Running multiple tests*: Collecting tests into suites
- *Constructing assertion logic*: Using constraints
- *Faking components*: Mocks and stubs
- *Testing web applications*: Testing with and without additional tools

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_7

Functional Tests and Unit Tests

Testing is essential in any project. Even if you don't formalize the process, you must have found yourself developing informal lists of actions that put your system through its paces. This process soon becomes wearisome, and that can lead to a fingers-crossed attitude to your projects.

One approach to testing starts at the interface of a project, modeling the various ways in which a user might negotiate the system. This is probably the way you would go when testing by hand, although there are various frameworks for automating the process. These functional tests are sometimes called acceptance tests because a list of actions performed successfully can be used as criteria for signing off a project phase. Using this approach, you typically treat the system as a black box – your tests remain willfully ignorant of the hidden components that collaborate to form the system under test.

Whereas functional tests operate from without, unit tests work from the inside out. Unit testing tends to focus on classes, with test methods grouped together in test cases. Each test case puts one class through a rigorous workout, checking that each method performs as advertised and fails as it should. The objective, as far as possible, is to test each component in isolation from its wider context. This often supplies you with a sobering verdict on the success of your mission to decouple the parts of your system.

Tests can be run as part of the build process, directly from the command line, or even via a web page. In this chapter, I'll concentrate on the command line.

Unit testing is a good way of ensuring the quality of design in a system. Tests reveal the responsibilities of classes and functions. Some programmers even advocate a test-first approach. You should, they say, write the tests before you even begin work on a class. This lays down a class's purpose, ensuring a clean interface and short, focused methods. Personally, I have never aspired to this level of purity – it just doesn't suit my style of coding. Nevertheless, I attempt to write tests as I go. Maintaining a test harness provides me with the security I need to refactor my code. I can pull down and replace entire packages with the knowledge that I have a good chance of catching unexpected errors elsewhere in the system.

Testing by Hand

In the last section, I said that testing was essential in every project. I could have said instead that testing is *inevitable* in every project. We all test. The tragedy is that we often throw away this good work.

So, let's create some classes to test. Here is a class that stores and retrieves user information. For the sake of demonstration, it generates arrays rather than the `User` objects you'd normally expect to use:

```
class UserStore
{
    private array $users = [];

    public function addUser(string $name, string $mail, string $pass): bool
    {
        if (isset($this->users[$mail])) {
            throw new \Exception(
                "User {$mail} already in the system"
            );
        }

        if (strlen($pass) < 5) {
            throw new \Exception(
                "Password must have 5 or more letters"
            );
        }

        $this->users[$mail] = [
            'pass' => $pass,
            'mail' => $mail,
            'name' => $name
        ];

        return true;
    }
```

```
    public function notifyPasswordFailure(string $mail): void
    {
        if (isset($this->users[$mail])) {
            $this->users[$mail]['failed'] = time();
        }
    }

    public function getUser(string $mail): array
    {
        return ($this->users[$mail]);
    }
}
```

This class accepts user data with the `addUser()` method and retrieves it via `getUser()`. The user's email address is used as the key for retrieval. If you're like me, you'll write some sample implementation as you develop, just to check that things are behaving as you designed them:

```
$store = new UserStore();
$store->addUser(
    "bob williams",
    "bob@example.com",
    "12345"
);
$store->notifyPasswordFailure("bob@example.com");
$user = $store->getUser("bob@example.com");
print_r($user);
```

Here is the output:

```
Array
(
    [pass] => 12345
    [mail] => bob@example.com
    [name] => bob williams
    [failed] => 1715099246
)
```

This is the sort of thing that I might add to the foot of a file as I work on the class it contains. The test validation is performed manually, of course; it's up to me to eyeball the results and confirm that the data returned by `UserStore::getUser()` corresponds with the information I added initially. It's a test of sorts, nevertheless.

Here is a client class that uses `UserStore` to confirm that a user has provided the correct authentication information:

```
class Validator
{
    public function __construct(private UserStore $store)
    {
    }

    public function validateUser(string $mail, string $pass): bool
    {
        if (! is_array($user = $this->store->getUser($mail))) {
            return false;
        }

        if ($user['pass'] == $pass) {
            return true;
        }

        $this->store->notifyPasswordFailure($mail);

        return false;
    }
}
```

The class requires a `UserStore` object, which it saves in the `$store` property. This property is used by the `validateUser()` method to ensure, first of all, that the user referenced by the given email address exists in the store and, second, that the user's password matches the provided argument. If both these conditions are fulfilled, the method returns `true`. Once again, I might test this as I go along:

```
$store = new UserStore();
$store->addUser("bob williams", "bob@example.com", "12345");
$validator = new Validator($store);
```

```
if ($validator->validateUser("bob@example.com", "12345")) {
    print "pass, friend!\n";
}
```

I instantiate a `UserStore` object, which I prime with data and pass to a newly instantiated `Validator` object. I can then confirm a username and password combination.

Once I'm finally satisfied with my work, I could delete these sanity checks altogether or comment them out. This is a terrible waste of a valuable resource. These tests could form the basis of a harness to scrutinize the system as I develop. One of the tools that might help me to do this is PHPUnit.

Introducing PHPUnit

PHPUnit is a member of the xUnit family of testing tools. The ancestor of these is SUnit, a framework invented by Kent Beck to test systems built with the Smalltalk language. The xUnit framework was probably established as a popular tool, however, by the Java implementation, jUnit, and by the rise to prominence of agile methodologies like Extreme Programming (XP) and Scrum, all of which place great emphasis on testing.

You can get PHPUnit with Composer:

```
$ composer require --dev phpunit/phpunit
```

This will install PHPUnit and add a variation on the following to your `composer.json` file:

```
{
    "require-dev":  {
        "phpunit/phpunit":  "^12"
    }
}
```

You will find the `phpunit` script at `vendor/bin/phpunit`.

If you don't want to use Composer, you can download a PHP archive (.phar) file. You can then make the archive executable:

```
$ wget https://phar.phpunit.de/phpunit.phar
$ chmod 755 phpunit.phar
$ sudo mv phpunit.phar /usr/local/bin/phpunit
```

Note I show commands that are input at the command line with a leading $ to represent a command prompt and distinguish them from any output they may produce.

Creating a Test Case

Armed with PHPUnit, I can write tests for the `UserStore` class. Tests for each target component should be collected in a single class that extends `PHPUnit\Framework\TestCase`, one of the classes made available by the PHPUnit package. Here's how to create a minimal test case class:

```
namespace popp\ch20\batch01;

use PHPUnit\Framework\TestCase;

class UserStoreTest extends TestCase
{

    protected function setUp(): void
    {
    }

    protected function tearDown(): void
    {
    }

}
```

I named the test case class `UserStoreTest`. It is often useful to place your test in the same namespace as the class under test. This will give you easy access to the class under test and its peers, and the structure of your test files will likely mirror that of your system. Remember that, thanks to Composer's support for PSR-4, you can maintain separate directory structures for class files in the same package.

Here's how we might do this in a `composer.json` file:

```
 "autoload": {
     "psr-4": {
```

```
            "popp\": ["myproductioncode/", "mytestcode/"]
        }
    }
```

In this code, I have nominated two directories that map to the popp namespace. I can now maintain these in parallel, making it easy to keep my test and production code separate.

Note You can also configure autoloading for your tests with the `autoload-dev` key. This works in tandem with the `autoload` equivalent but is not applied in production mode (i.e., when `composer update` or `composer install` are run with the `--no-dev` flag).

The `setUp()` method is automatically invoked for each test method, allowing us to set up a stable and suitably primed environment for the test. `tearDown()` is invoked after each test method is run. If your tests change the wider environment of your system, you can use this method to reset state. The common platform managed by `setUp()` and `tearDown()` is known as a *fixture*.

In order to test the `UserStore` class, I need an instance of it. I can instantiate this in `setUp()` and assign it to a property. Let's create a test method as well:

```
namespace popp\ch20\batch01;

use PHPUnit\Framework\TestCase;

class UserStoreTest extends TestCase
{
    private UserStore $store;

    protected function setUp(): void
    {
        $this->store = new UserStore();
    }

    protected function tearDown(): void
    {
    }
```

```
    public function testGetUser(): void
    {
        $this->store->addUser("bob williams", "a@b.com", "12345");
        $user = $this->store->getUser("a@b.com");
        $this->assertEquals("a@b.com", $user['mail']);
        $this->assertEquals("bob williams", $user['name']);
        $this->assertEquals("12345", $user['pass']);
    }
}
```

Note Remember that `setUp()` and `tearDown()` are called once for every test method in your class. If you want to include code that will be run once before all the test methods in a class, you can implement the static `setUpBeforeClass()` method. Conversely, for code that should be run after all the test methods in a class, implement `tearDownAfterClass()` (also a static method).

Test methods should be named to begin with the word "test" and should require no arguments. This is because the test case class is manipulated using reflection.

Note Reflection is covered in detail in Volume 1, Chapter 5.

The object that runs the tests looks at all the methods in the class and invokes only those that match this pattern (i.e., methods that begin with "test").

Although it was deprecated in PHPUnit 11, the package allowed you to use annotations rather than method names to specify test methods:

```
/**
 * @test
 */
public function GetUser(): void
{
    // ...
}
```

As of PHPUnit 12, you can still use an attribute for the same purpose.

```
use PHPUnit\Framework\TestCase;
use PHPUnit\Framework\Attributes\Test;

class UserStoreAttributeTest extends TestCase
{
    #[Test]
    public function AddUserShortPass(): void
    {
        // ...
    }

    // ...
```

The attribute form was supported in PHPUnit 11 too.

In the full example above, I tested the retrieval of user information. I don't need to instantiate `UserStore` for each test because I handled that in `setUp()`. Because `setUp()` is invoked for each test, the `$store` property is guaranteed to contain a newly instantiated object.

Within the `testgetUser()` method, I first provide `UserStore::addUser()` with dummy data, and then I retrieve that data and test each of its elements.

There is one additional issue to be aware of here before we can run our test. I am using `use` statements without `require` or `require_once`. In other words, I am relying on autoloading. Finding and including the autoload file is handled automatically if you installed PHPUnit with Composer and if the autoload file for your project was generated in the same context. This may not always be the case, however. I may be running a global PHPUnit command which knows nothing of my local autoload, for example, or I may have downloaded a phar file. In this case, how do I tell my tests how to locate the generated `autoload.php` file? I could put a `require_once` statement in the test class (or a superclass), but that would break the PSR-1 rule that class files should not have side effects. The simplest thing to do is to tell PHPUnit about the `autoload.php` file from the command line:

```
$ phpunit src/ch20/batch01/UserStoreTest.php --bootstrap vendor/
autoload.php
PHPUnit 12.0.10 by Sebastian Bergmann and contributors.
```

```
Runtime:       PHP 8.3.7

...                                                                 3 / 3 (100%)

Time: 00:00.001, Memory: 25.30 MB

OK (3 tests, 5 assertions)
```

For future examples, I will use a version of PHPUnit that was installed with Composer along with the tests and system under test.

Assertion Methods

An assertion in programming is a statement or method that allows you to check your assumptions about an aspect of your system. In using an assertion, you typically define an expectation that something is the case, that `$cheese` is `"blue"` or `$pie` is `"apple"`. If your expectation is confounded, a warning of some kind will be generated. Assertions are such a good way of adding safety to a system that PHP supports them natively inline and allows you to turn them off in a production context.

Note See the manual page at https://php.net/assert for more information on PHP's support for assertions.

PHPUnit supports assertions through a set of methods that can be called either statically or on an instance of a class that extends `PHPUnit\Framework\TestCase`.

In the previous example, I used a `TestCase` method, `assertEquals()`. This method compares its two provided arguments and checks them for equivalence. If they do not match, the test method will be chalked up as a failed test. Having subclassed `PHPUnit\Framework\TestCase`, I have access to a set of assertion methods. Some of these methods are listed in Table 7-1.

Table 7-1. *Some* `PHPUnit\Framework\TestCase` *Assert Methods*

Method	Description
`assertEquals($val1, $val2, $message)`	Fail if `$val1` is not equivalent to `$val2`.
`assertFalse($expression, $message)`	Evaluate `$expression`; fail if it does not resolve to `false`.
`assertTrue($expression, $message)`	Evaluate `$expression`; fail if it does not resolve to `true`.
`assertNotNull($val, $message)`	Fail if `$val` is `null`.
`assertNull($val, $message)`	Fail if `$val` is anything other than `null`.
`assertSame($val1, $val2, $message)`	Fail if `$val1` and `$val2` are not references to the same object or if they are variables of different types or values.
`assertNotSame($val1, $val2, $message)`	Fail if `$val1` and `$val2` are references to the same object or variables of the same type and value.
`assertMatchesRegularExpression ($regexp, $val, $message)`	Fail if `$val` is not matched by the regular expression, `$regexp`.
`assertEqualsCanonicalizing ($val1, $val2, $message)`	Fail if `$val1` is not equivalent to `$val2`. This is usually used for complex values. PHPUnit will attempt to canonicalize the values prior to comparison, sorting arrays and first converting objects to arrays if needed.

Testing Exceptions

Your focus as a coder is usually to make systems that *work* and work well. Often, that mentality carries through to testing, especially if you are testing your own code. The temptation is to test that a method behaves as advertised. It's easy to forget how important it is to test for failure. How good is a method's error checking? Does it throw an exception when it should? Does it throw the right exception? Does it clean up after an error if, for example, an operation is half complete before the problem occurs? It is your role as a tester to check all of this. Luckily, PHPUnit can help.

Here is a test that checks the behavior of the `UserStore` class when an operation fails:

```
public function testAddUserShortPass(): void
{
    try {
        $this->store->addUser("bob williams", "bob@example.com", "ff");
    } catch (\Exception $e) {
        $this->assertEquals("Password must have 5 or more letters",
        $e->getMessage());
        return;
    }

    $this->fail("Short password exception expected");
}
```

If you look back at the `UserStore::addUser()` method, you will see that I throw an exception if the user's password is less than five characters long. My test attempts to confirm this. I add a user with an illegal password in a `try` clause. If the expected exception is thrown, then flow skips to the catch clause, and all is well. If the `addUser()` method does not throw an exception as expected, the execution flow reaches the `fail()` method call.

Another way to test that an exception is thrown is to use an assertion method called `expectException()`, which requires the name of the exception type you expect to be thrown (either `Exception` or a subclass). If the test method exits without the correct exception having been thrown, the test will fail.

Note The `expectException()` method was added in PHPUnit 5.2.0.

Here's a quick reimplementation of the previous test:

```
public function testAddUserShortPassNew(): void
{
    $this->expectException(\Exception::class);
    $this->store->addUser("bob williams", "bob@example.com", "ff");
}
```

So, given that there is a neat way of testing for exceptions, why did I show the older approach at all? In most circumstances, the simplest approach – using `expectException()` – will be the best. However, occasionally, you may want to perform further tests on the exception, on the state of the object under test, or you may want to clean up some side effect. In such cases, it may still make sense to go old school.

Running Test Suites

If I am testing the `UserStore` class, I should also test `Validator`. Here is a cut-down version of a class called `ValidateTest` that tests the `Validator::validateUser()` method:

```
namespace popp\ch20\batch02;

use PHPUnit\Framework\TestCase;

class ValidatorTest extends TestCase
{
    private Validator $validator;

    protected function setUp(): void
    {
        $store = new UserStore();
        $store->addUser("bob williams", "bob@example.com", "12345");
        $this->validator = new Validator($store);
    }

    public function testValidateCorrectPass(): void
    {
        $this->assertTrue(
            $this->validator->validateUser("bob@example.com", "12345"),
            "Expecting successful validation"
        );
    }

}
```

So now that I have more than one test case, how do I go about running them together? The easiest way is to place your test classes in a common root directory. You can then specify this directory, and PHPUnit will run all the tests beneath it. Here, I run `ValidatorTest` along with additional test files that I have placed in the same directory:

```
$ phpunit src/ch20/batch02/
PHPUnit 12.0.10 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.3.7

........                                                          8 / 8 (100%)

Time: 00:00.018, Memory: 8.00 MB

OK (8 tests, 14 assertions)
```

Constraints

In most circumstances, you will use off-the-peg assertions in your tests. In fact, at a stretch, you can achieve an awful lot with `AssertTrue()` alone (although it is considered best practice to use more specialized assertions where possible). As of PHPUnit 3.0, `PHPUnit\Framework\TestCase` included a set of factory methods that return `PHPUnit\Framework\Constraint` objects. You can combine these and pass them to `TestCase::assertThat()` in order to construct your own assertions.

It's time for a quick example. The `UserStore` object should not allow duplicate email addresses to be added. Here's a test that confirms this:

```
// UserStoreTest

public function testAddUserDuplicate(): void
{
    try {
        $this->store->addUser("bob williams", "a@b.com", "123456");
        $this->store->addUser("bob stevens", "a@b.com", "123456");
        $this->fail("Exception should have been thrown");
    } catch (\Exception $e) {
```

```
        $const = $this->logicalAnd(
            $this->logicalNot($this->containsEqual("bob stevens")),
            $this->isArray(),
        );
        $this->assertThat($this->store->getUser("a@b.com"), $const);
    }
}
```

This test adds a user to the `UserStore` object and then adds a second user with the same email address. The test thereby confirms that an exception is thrown with the second call to `addUser()`. In the `catch` clause, I build a constraint object using the convenience methods available to us. These return corresponding instances of `PHPUnit\Framework\Constraint`. Let's break down the composite constraint in the previous example:

```
$this->containsEqual("bob stevens")
```

The `containsEqual()` method returns a `PHPUnit\Framework\Constraint\TraversableContainsEqual` object. When passed to `assertThat()`, this object will generate an error if the test subject does not contain an element matching the given value (`"bob stevens"`).

I can negate this, though, by passing this constraint to another: `PHPUnit\Framework\Constraint\LogicalNot`.

```
$this->logicalNot($this->containsEqual("bob stevens")),
```

Now, the `assertThat` assertion will fail if the test value (which must be traversable) contains an element that matches the string, `"bob stevens"`.

```
$this->isArray()
```

This method returns an instance of the `PHPUnit\Framework\Constraint\IsType` constraint which, as you'd expect, checks type.

Note A less elegant way of checking for an array, `isType('array')`, is deprecated as of PHPUnit 12.

Now, I can combine my singular type check constraint and my composite constraint in a `logicalAnd()` constraint:

```
$this->logicalAnd(
    $this->logicalNot($this->containsEqual("bob stevens")),
    $this->isType('array'),
);
```

In this way, you can build up quite complex logical structures. My finished constraint can be summarized as follows: "Do not fail if the test value is an array and does not contain the string `"bob stevens"`." You could build much more involved constraints in this way. The constraint is run against a value by passing both to `assertThat()`.

You could achieve all this with standard assertion methods, of course, but constraints have a couple of virtues. First, they form nice logical blocks with clear relationships among components (although good use of formatting may be necessary to support clarity). Second, and more important, a constraint is reusable. You can set up a library of complex constraints and use them in different tests. You can even combine complex constraints with one another:

```
$const = $this->logicalAnd(
    $a_complex_constraint,
    $another_complex_constraint
);
```

Table 7-2 shows some of the constraint methods available in a `TestCase` class.

Table 7-2. *Some Constraint Methods*

TestCase Method	Constraint Fails Unless…
`greaterThan($num)`	Test value is greater than `$num`.
`containsEqual($val)`	Test value (traversable) contains an element that matches `$val`.
`identicalTo($val)`	Test value is a reference to the same object as `$val` or, for nonobjects, is of the same type and value.
`greaterThanOrEqual($num)`	Test value is greater than or equal to `$num`.
`lessThan($num)`	Test value is less than `$num`.
`lessThanOrEqual($num)`	Test value is less than or equal to `$num`.
`equalTo($value)`	Test value equals `$value`.
`equalTo($value, $delta)`	Test value equals `$value`. `$delta` defines a margin of error for numeric comparisons.
`stringContains($str, $casesensitive=true)`	Test value contains `$str`. This is case sensitive by default.
`matchesRegularExpression($pattern)`	Test value matches the regular expression in `$pattern`.
`logicalAnd(PHPUnit\Framework\ Constraint $const, [, $const..])`	All provided constraints pass.
`logicalOr(PHPUnit\Framework\ Constraint $const, [, $const..])`	At least one of the provided constraints matches.
`logicalNot(PHPUnit\Framework\ Constraint $const)`	The provided constraint does not pass.

Mocks and Stubs

Unit tests aim to test a component in isolation of the system that contains it to the greatest possible extent. Few components exist in a vacuum, however. Even nicely decoupled classes require access to other objects as method arguments. Many classes also work directly with databases or the file system.

You have already seen one way of dealing with this. The `setUp()` and `tearDown()` methods can be used to manage a fixture (i.e., a common set of resources for your tests, which might include database connections, configured objects, a scratch area on the file system, etc.).

Note Using `setUp()` and `tearDown()` can be memory intensive because these methods are invoked for every test method in a suite. You can mitigate this problem somewhat by placing expensive processes in the static `setUpBeforeClass()` and `tearDownAfterClass()` methods and sharing resources between your test methods.

Another approach is to fake the context of the class you are testing. This involves creating objects that pretend to be the objects that do real stuff. For example, you might pass a fake database mapper to your test object's constructor. Because this fake object shares a type with the real mapper class (extends from a common abstract base or even overrides the genuine class itself), your subject is none the wiser. You can prime the fake object with valid data. Objects that provide a sandbox of this sort for unit tests are known as *stubs*. They can be useful because they allow you to focus in on the class you want to test without inadvertently testing the entire edifice of your system at the same time.

Fake objects can be taken a stage further than this, however. Because the object you are testing is likely to call a fake object in some way, you can prime it to confirm the invocations you are expecting. Using a fake object in this way – telling it how, when, and how many times it should be called – is known as *behavior verification*, and it is what distinguishes a mock object from a stub.

You can build mocks yourself by creating classes hard-coded to return certain values and to report on method invocations. This is a simple process, but it can be time-consuming.

PHPUnit provides access to an easier and more dynamic solution. It will generate mock objects on the fly for you. It does this by examining the class you wish to mock and building a child class that overrides its methods. Once you have this mock instance, you can call methods on it to prime it with data and to set the conditions for success.

Let's build an example. The `UserStore` class contains a method called `notifyPasswordFailure()`, which sets a field for a given user. This should be called by `Validator` when an attempt to set a password fails. Here, I mock up the `UserStore` class so that it both provides data to the `Validator` object and confirms that its `notifyPasswordFailure()` method was called as expected:

```
// ValidatorTest

    public function testValidateFalsePass(): void
    {
        $store = $this->createMock(UserStore::class);
        $this->validator = new Validator($store);

        $store->expects($this->once())
            ->method('notifyPasswordFailure')
            ->with($this->equalTo('bob@example.com'));

        $store->expects($this->any())
            ->method("getUser")
            ->willReturn([
                "name" => "bob williams",
                "mail" => "bob@example.com",
                "pass" => "right"
            ]);

        $this->validator->validateUser("bob@example.com", "wrong");
    }
```

Mock objects created with `TestCase::createMock()` use a *fluent interface*; that is, they have a language-like structure. These are much easier to use than to describe. Such constructs work from left to right, each invocation returning an object reference, which can then be invoked with a further modifying method call (itself returning an object). This can make for easy use but painful debugging.

In the previous example, I called the `TestCase` method, `createMock()`, passing it `UserStore::class`, the full name of the class I wish to mock. This dynamically generates a class and instantiates an object from it. I store this mock object in `$store` and pass it to `Validator`. This causes no error because the object's newly minted class extends `UserStore`. I have fooled `Validator` into accepting an imposter into its midst.

Mock objects generated by PHPUnit have an expects() method. This method requires a matcher object which defines the cardinality of the expectation; that is, it dictates the number of times a method should be called. You will almost certainly use one of a range of convenience methods that TestCase makes available to generate the correct object for this purpose. You can see these methods in Table 7-3.

Table 7-3. *Some Matcher Methods*

TestCase Method	Match Fails Unless…
any()	Zero or more calls are made to the corresponding method (useful for stub objects that return values but don't test invocations)
never()	No calls are made to the corresponding method
atLeastOnce()	One or more calls are made to the corresponding method
atLeast($num)	At least $num calls are made to the corresponding method
atMost($num)	At most $num calls are made to the corresponding method
once()	A single call is made to the corresponding method
exactly($num)	$num calls are made to the corresponding method

Having set up the match requirement, I need to specify a method to which it applies. The expects() method returns an object which provides a method named method() for this purpose. I can simply call that with a method name. This is enough to get some real mocking done:

```
$store->expects($this->once())
    ->method('notifyPasswordFailure');
```

I need to go further, however, and check the parameters that are passed to notifyPasswordFailure(). InvocationMocker::method() returns an instance of the object it was called on. InvocationMocker includes a method name, with(), which accepts a variable list of parameters to match. It also accepts constraint objects, so you can test ranges and so on. Armed with this, you can complete the statement and ensure that the expected parameter is passed to notifyPasswordFailure():

```
$store->expects($this->once())
    ->method('notifyPasswordFailure')
    ->with($this->equalTo('bob@example.com'));
```

You can see why this is known as a fluent interface. It reads a bit like a sentence: "The `$store` object `expects` a single call to the `notifyPasswordFailure()` method `with` parameter `bob@example.com`."

Notice that I passed a constraint to `with()`. Actually, that's redundant; any bare arguments are converted to constraints internally, so I could write the statement like this:

```
$store->expects($this->once())
    ->method('notifyPasswordFailure')
    ->with('bob@example.com');
```

Sometimes, you only want to use PHPUnit's mocks as stubs, that is, as objects that return values to allow your tests to run. In such cases, you can invoke `InvocationMocker::willReturn()` from the call to `method()`. The `willReturn()` method requires the return value (or values if the method is to be called repeatedly) that the associated method should be primed to return.

```
$store->method("getUser")
    ->willReturn([
        "name" => "bob@example.com",
        "pass" => "right"
    ]);
```

I omit the `expects()` stage altogether here since I'm not monitoring behavior and begin by specifying the `getUser()` method. Next, I call `willReturn()` with my expected value.

You can alternatively pass multiple values to `willReturn()`. Each one of these will be returned by your mocked method as it is called repeatedly.

Tests Succeed When They Fail

Although most agree that testing is a fine thing, you grow to really love it generally only after it has saved your bacon a few times. Let's simulate a situation where a change in one part of a system has an unexpected effect elsewhere.

The `UserStore` class has been running for a while when, during a code review, it is agreed that it would be neater for the class to generate `User` objects rather than associative arrays. Here is the new version:

```
namespace popp\ch20\batch03;

class UserStore
{
    private array $users = [];

    public function addUser(string $name, string $mail, string $pass): bool
    {
        if (isset($this->users[$mail])) {
            throw new \Exception(
                "User {$mail} already in the system"
            );
        }

        $this->users[$mail] = new User($name, $mail, $pass);

        return true;
    }

    public function notifyPasswordFailure(string $mail): void
    {
        if (isset($this->users[$mail])) {
            $this->users[$mail]->failed(time());
        }
    }

    public function getUser(string $mail): ?User
    {
        if (isset($this->users[$mail])) {
            return ( $this->users[$mail] );
        }

        return null;
    }
}
```

Here is the simple User class:

```
namespace popp\ch20\batch03;

class User
{
    private string $pass;
    private ?string $failed;

    public function __construct(private string $name, private string $mail,
    string $pass)
    {
        if (strlen($pass) < 5) {
            throw new \Exception(
                "Password must have 5 or more letters"
            );
        }

        $this->pass = $pass;
    }

    public function getMail(): string
    {
        return $this->mail;
    }

    public function getPass(): string
    {
        return $this->pass;
    }

    public function failed(string $time): void
    {
        $this->failed = $time;
    }

}
```

Of course, I amend the UserStoreTest class to account for these changes. Consider this code designed to work with an array:

```
public function testGetUser(): void
{
    $this->store->addUser("bob williams", "a@b.com", "12345");
    $user = $this->store->getUser("a@b.com");
    $this->assertEquals($user['mail'], "a@b.com");
    $this->assertEquals($user['name'], "bob williams");
    $this->assertEquals($user['pass'], "12345");
}
```

It is now converted into code designed to work with an object, like this:

```
public function testGetUser(): void
{
    $this->store->addUser("bob williams", "a@b.com", "12345");
    $user = $this->store->getUser("a@b.com");
    $this->assertEquals($user->getMail(), "a@b.com");
}
```

I also need to update `testAddUserDuplicate()` so that it expects an object rather than an array:

```
public function testAddUserDuplicate(): void
{
    try {
        $ret = $this->store->addUser("bob williams", "a@b.com", "123456");
        $ret = $this->store->addUser("bob stevens", "a@b.com", "123456");
        self::fail("Exception should have been thrown");
    } catch (\Exception $e) {
        self::assertThat($this->store->getUser("a@b.com"), $this
        ->isObject());
        // perform other checks
    }
}
```

That should mean that I'm all set, right? When I come to run my test suite, however, I am rewarded with a warning that my work is not yet done:

```
$ phpunit src/ch20/batch03/
# ...

1) popp\ch20\batch03\ValidatorTest::testValidateCorrectPass
Expecting successful validation
Failed asserting that false is true.

/Users/mattz/work/popp7/popp7-repo/src/ch20/batch03/ValidatorTest.php:27

# ...
```

Although my tests relating to `User` pass, my `ValidatorTest` class has caught some issues related to the fact that I have not updated the `Validator` to account for the new return value. Let's focus on the failure referenced above:

```
public function testValidateCorrectPass(): void
{
    $this->assertTrue(
        $this->validator->validateUser("bob@example.com", "12345"),
        "Expecting successful validation"
    );
}
```

And here is the `Validator::validateUser()` method that has let me down:

```
public function validateUser(string $mail, string $pass): bool
{
    if (! is_array($user = $this->store->getUser($mail))) {
        return false;
    }

    if ($user['pass'] == $pass) {
        return true;
    }
```

```
    $this->store->notifyPasswordFailure($mail);

    return false;
}
```

So, `User::getUser()` now returns an object and not an array. `getUser()` originally returned an array containing user data on success or `null` on failure. I validated users by checking for an array using the `is_array()` function. Now, of course, this condition is never met and the `validateUser()` method will always return `false`. Without the test framework, the `Validator` would have simply rejected all users as invalid without fuss or warning.

It is a relatively quick fix to bring `validateUser()` method into line.

```
public function validateUser(string $mail, string $pass): bool
{
    $user = $this->store->getUser($mail);
    if (is_null($user)) {
        return false;
    }
    $testpass = $user->getPass();
    if ($testpass == $pass) {
        return true;
    }

    $this->store->notifyPasswordFailure($mail);
    return false;
}
```

Now, imagine making the neat little change to `UserStore::getUser()` on a Friday night without a test framework in place. Think about the frantic text messages that would drag you out of your pub, armchair, or restaurant: “What have you done? All our customers are locked out!”

The most insidious bugs don’t cause the interpreter to report that something is wrong. They hide in perfectly legal code, and they silently break the logic of your system. Many bugs don’t manifest themselves where you are working; they are caused there, but the effects pop up elsewhere, days or even weeks later. A test framework can help you catch at least some of these, preventing rather than discovering problems in your systems.

Write tests as you code, and run them often. If someone reports a bug, first add a test to your framework to confirm it. Next, fix the bug so that the test is passed. Bugs have a funny habit of recurring in the same area. Writing tests to prove bugs and then to guard the fix against subsequent problems is known as *regression testing*. Incidentally, if you keep a separate directory of regression tests, remember to name your files descriptively. On one project, our team decided to name our regression tests after Bugzilla ticket numbers. We ended up with a directory containing 400 test files, each with a name like `test_973892.php`. Finding an individual test became a tedious chore!

Writing Web Tests

So long as its individual components are properly independent, a web application should be just as testable as any other system using the techniques this chapter has covered. We saw in Volume 1 that, by deploying a dependency injection container (or, less commonly these days, a service locator), you can support orthogonality in your classes (i.e., minimize hidden dependencies and maximize configurability). A controller method in a typical web application will accept a `Request` object and return a `Response` object. By providing a preconfigured `Request` (often a stub), you can pretty easily run a controller method through its paces. You can then examine the generated `Response` for an expected state.

By using mocks and stubs across the board in addition to your `Request` and `Response` objects, you can isolate the controller method and narrow the number of components that the unit test activates. Alternatively, with a test configuration, you can create a basic functional test that exercises real components in the system.

Approaches like this are great for testing the inputs and output of a web application. There are some distinct limitations, however. This method won't capture the browser experience. Where a web application uses JavaScript, and other client-side cleverness, testing the text generated by your system won't tell you whether the user is interacting with a sane interface.

Luckily, there is a solution.

Introducing Selenium

Selenium (https://www.selenium.dev/) is a set of tools that can be used for automating web browsers. This ecosystem includes (but is in no way limited to) tools and APIs for authoring and running browser tests.

In this brief introduction, I'll create a quick test for a mocked up system. The test will work in conjunction with the Selenium server via an API called php-webdriver.

Getting Selenium

Probably the easiest way to get up and running with Selenium is via the Docker image. If you have Docker installed, you can simply fire a Selenium server up with a single command:

```
$ docker run -d -p 4444:4444 -p 7900:7900 --net=host --shm-size="2g"
selenium/standalone-chrome:latest
```

The `--net host` flag here is only necessary if you will be testing a locally hosted system – that is, you will referencing `localhost` URLs in your test files. Do not use the `--net` option if you are using a Mac (also see the note below you are using an ARM-based Mac).

Note You can find out more about running Selenium with Docker at https://hub.docker.com/r/selenium/standalone-chrome and https://github.com/SeleniumHQ/docker-selenium.

If you're using an ARM-based Mac, you can read about how to run Selenium with Docker at https://www.selenium.dev/blog/2024/multi-arch-images-via-docker-selenium/ which recommends a different `docker run` invocation.

```
$ docker run --rm -it -p 4444:4444 -p 5900:5900 -p 7900:7900
--shm-size 2g selenium/standalone-chromium:latest
```

For any Mac, if you intend to test a locally hosted system (i.e., by using URLs that would typically reference `localhost`), you should amend your tests to use `host.docker.internal` instead of `localhost`.

If, for some reason, you fail to explicitly end a session in your test suite (I'll show you how to do this shortly), you might find that subsequent test runs are held up until the session times out. You can adjust the timeout period by adding an environment variable with the `-e` option to `docker run`. Here, I set the timeout to ten seconds:

```
-e SE_NODE_SESSION_TIMEOUT=10
```

The first time you invoke the `docker run` command for Selenium, you'll see activity as Docker downloads the required images. Thereafter, the command will initialize much faster and with a little less output.

Now, I'm ready to proceed.

PHPUnit and Selenium

Although PHPUnit has provided APIs for working with Selenium in the past, the best solution is currently a third-party library that provides the bindings we need.

Introducing php-webdriver

WebDriver (https://www.selenium.dev/documentation/webdriver/) is the mechanism by which Selenium controls browsers, and it was introduced with Selenium 2. Selenium supports various language libraries for WebDriver. Although php-webdriver is not among them, it is under active development and mirrors the official APIs. This is very handy when you want to look up a technique, since many examples you'll find online will be offered in Java which means they will apply readily to php-webdriver with a little porting of code.

You can add php-webdriver to your project with Composer:

```
$ composer require php-webdriver/webdriver
```

The System Under Test

I will be working with a mocked up version of a venue listings system I created in Volume 1, Chapter 12. You don't need to know anything about that system for this example, however. The mockup consists of two crude scripts `AddVenue.php` and `AddSpace.php` which, between them, simulate the process of creating a venue and then adding a sub-venue (a "space").

Here is `AddVenue.php`:

```
<?php
```

```
$venue_name =  $_REQUEST['venue_name'] ?? null;
?>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Add a Venue</title>
  </head>
  <body>
<?php if (is_null($venue_name)) { ?>
<div>no name provided</div>
<div>
<form method="post">
    <input type="text" value="" name="venue_name" />
    <input type="submit" value="submit" />
</form>
</div>
<?php } else { ?>
<h1>Add a Space for Venue '<?php print $venue_name ?>'</h1>

<div>
'<?php print $venue_name ?>' added (22)  please add name for the space
</div>

[add space]
<form method="post" action="AddSpace.php">
    <input type="text" value="" name="space_name"/>
    <input type="hidden" name="cmd" value="AddSpace" />
    <input type="hidden" name="venue_id" value="22" />
    <input type="hidden" name="venue_name" value="<?php print $venue_
    name ?>" />
    <input type="submit" value="submit" />
</form>
<?php } ?>

</body>
</html>
```

This either presents a form for "creating" a venue or, if a venue name has been provided by a previous submission, a second form for "adding" a space. This form submits values to a second, even simpler, script: `AddSpace.php`.

```
<?php
    $venue_name = $_REQUEST['venue_name'] ?? "-";
    $space_name = $_REQUEST['space_name'] ?? "-";
?>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Here are the venues</title>
  </head>
  <body>

  <h1>Here are the venues</h1>

  <div>
  space '<?php print $space_name ?>' added (47)</td>
  </div>

  <div>
  <?php print $venue_name ?><br />
     <?php print $space_name ?><br />
  </div>

  </body>
</html>
```

This script – also crude – pretends to have added a space and, using `$_REQUEST` elements, constructs a summary.

I will run this script locally using PHP's built-in development server.

```
$ php -S 0.0.0.0:8784 -t .
```

I run the command above from the directory that contains `AddVenue.php` and `AddSpace.php`. Now, I can follow the dummy flow in my browser by navigating to `http://localhost:8784/AddVenue.php`.

The fact that I'm testing a locally hosted system is the reason I included `--net host` in my `docker run` invocation. This tells the created container to use the host system's network.

Remember, though, that you should not use this option if you're running Selenium on a Mac using Docker even if you need to access local URLs. As you will see, there is an alternative syntax you can use within your tests on a Mac to access your `localhost` URLs.

Creating the Test Skeleton

Time to start creating the tests. I'll kick off with a boilerplate test class:

```
namespace popp\ch20\batch04;

use Facebook\WebDriver\Remote\DesiredCapabilities;
use Facebook\WebDriver\Remote\RemoteWebDriver;
use PHPUnit\Framework\TestCase;

class SeleniumTest1 extends TestCase
{
    protected function setUp(): void
    {
    }

    protected function tearDown(): void
    {
    }

    public function testAddVenue(): void
    {
    }
}
```

I specify some of the php-webdriver classes I will be using and then create a bare-bones test class. Now to make this test do something.

Connecting to Selenium

In order to make the connection to Selenium, I need to pass a URL and a configuration array to a class named `RemoteWebDriver`. The URL for the Selenium server is usually `http://127.0.0.1:4444/wd/hub`.

```
namespace popp\ch20\batch04;

use Facebook\WebDriver\Remote\DesiredCapabilities;
use Facebook\WebDriver\Remote\RemoteWebDriver;
use PHPUnit\Framework\TestCase;

class SeleniumTest2 extends TestCase
{
    private RemoteWebDriver $driver;

    protected function setUp(): void
    {
        $host = "http://127.0.0.1:4444/wd/hub";
        $capabilities = DesiredCapabilities::chrome();
        $this->driver = RemoteWebDriver::create($host, $capabilities);
    }

    protected function tearDown(): void
    {
        $this->driver->quit();
    }

    public function testAddVenue(): void
    {
    }
}
```

If you installed php-webdriver with Composer, you can see a full list of capabilities in the class file at `vendor/php-webdriver/webdriver/lib/Remote/WebDriverCapabilityType.php`. For my present purposes, however, I really only need to specify the minimum configuration needed to run the Chrome browser. This is provided by the convenience method `DesiredCapabilities::chrome()`. I pass the host string and the returned `$capabilities` array to the static `RemoteWebDriver::create()` method and store the resulting object reference in the `$driver` property.

The `tearDown()` method invokes `RemoteWebDriver::quit()`. This method closes all browser windows and ends the test session. If you fail to end a session, you'll need to wait for it to time out before you can run a new test suite. Having a browser hang around can be useful if you need to interact with it during development (perhaps to investigate the cause of a failure), but usually, you'll want the session to end promptly so that you can run a new test suite.

If I were to run this test and monitor the session (I will show you how to do this), I would see that Selenium launches a fresh browser window in preparation for further action and then promptly closes it again.

Let's add that further action.

Writing the Test

For this basic test, I will navigate to `AddVenue.php`, add a venue, confirm the expected response, and then add a space. Finally, I will confirm the output generated by `AddSpace.php`.

Here is my test:

```
namespace popp\ch20\batch04;

use Facebook\WebDriver\Remote\DesiredCapabilities;
use Facebook\WebDriver\Remote\RemoteWebDriver;
use Facebook\WebDriver\WebDriverBy;
use PHPUnit\Framework\TestCase;

class SeleniumTest3 extends TestCase
{

    // setUp(), tearDown() etc

    public function testAddVenue(): void
    {
        $this->driver->get("http://localhost:8784/AddVenue.php");
        $venel = $this->driver->findElement(WebDriverBy::name("venue_
        name"));
        $venel->sendKeys("my_test_venue");
        $venel->submit();
```

```
        $tdel = $this->driver->findElement(WebDriverBy::xpath("//div[1]"));
        $this->assertMatchesRegularExpression("/'my_test_venue' added/",
        $tdel->getText());

        $spacel = $this->driver->findElement(WebDriverBy::name("space_
        name"));
        $spacel->sendKeys("my_test_space");
        $spacel->submit();

        $el = $this->driver->findElement(WebDriverBy::xpath("//div[1]"));
        $this->assertMatchesRegularExpression("/'my_test_space' added/",
        $el->getText());
    }

}
```

Note this line:

```
$this->driver->get("http://localhost:8784/AddVenue.php");
```

The Selenium server will access my computer's version of `localhost` here rather than its own thanks to the `--net host` option I set when launching with `docker run`. Once again, you should not use this option on a Mac. Instead, omit `--net host`, and use `host.docker.internal` instead of `localhost` in the test itself:

```
$this->driver->get("http://host.docker.internal:8784/AddVenue.php");
```

Here's what happens when I run this test on the command line:

```
$ phpunit src/ch20/batch04/SeleniumTest3.php
PHPUnit 12.0.10 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.3.7

.                                                                   1 / 1 (100%)

Time: 00:12.159, Memory: 8.00 MB

OK (1 test, 2 assertions)
```

Of course, the command-line output is not all that happens. If you are looking in the right place, you can watch as Selenium launches a browser window and performs its specified operations within it. I have to admit, I find this effect a little eerie! There are various ways you can get to see this. If you have a VNC client, you can connect it to `localhost:5900`. Otherwise, you can point your browser at `http://localhost:7900/?autoconnect=1&resize=scale&password=secret` (or `http://localhost:4444` if you want more controls and information). By default, you may have to provide a password, which is, cryptically, "secret."

Let's run through the code. First, I invoke `WebDriver::get()`, which acquires my starting page. Note that this method expects a full URL (which does not need to be local to the Selenium server host). In this case, I am accessing my mocked up script `AddVenue.php` script running locally using PHP's built-in development server on port 8784 (remember to use `host.docker.internal` rather than `localhost` on a Mac). Selenium will load the specified document into the browser it has launched. You can see this page in Figure 7-1.

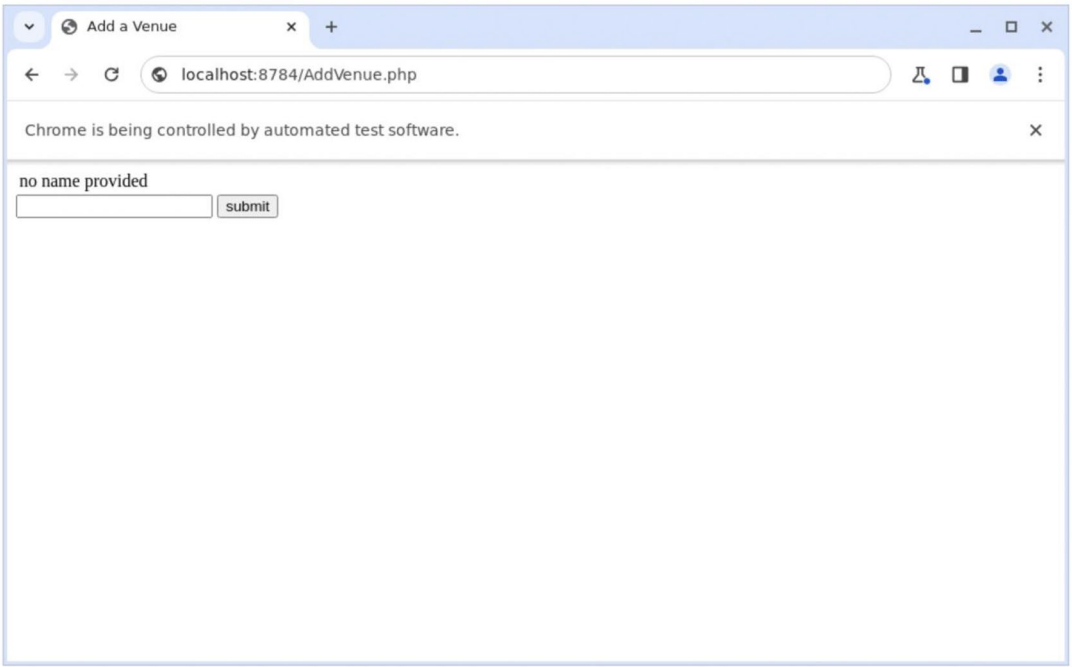

Figure 7-1. *The AddVenue page loaded by Selenium*

Once the page has loaded, I have access to it via the WebDriver API. I can acquire a reference to a page element using the `RemoteWebDriver::findElement()` method. This requires an object of type `WebDriverBy`. The `WebDriverBy` class provides a set of factory methods, each of which returns a `WebDriverBy` object configured to specify a particular means of locating an element. My form element has a `name` attribute set to `"venue_name"`, so I use the `WebDriverBy::name()` method to tell `findElement()` to look for an element this way. Table 7-4 lists all of the available factory methods.

Table 7-4. *`WebDriverBy` Factory Methods*

Method	Description
`className()`	Find elements by CSS class name
`cssSelector()`	Find elements by CSS selector
`id()`	Find an element by its id
`name()`	Find elements by name attribute
`linkText()`	Find elements by their link text
`partialLinkText()`	Find elements by a fragment of link text
`tagName()`	Find elements by their tag
`xpath()`	Find elements that match an Xpath expression

Once I have a reference to the `venue_name` form element, an object of type `RemoteWebElement`, I can use its `sendKeys()` method to set a value. It's important to note that `sendKeys()` does more than just set a value. It also simulates the act of typing into an element. This is useful for testing systems that use JavaScript to capture keyboard events.

With my new value set, I submit the form. The API is smart about this. When I call `submit()` on an element, Selenium locates the containing form and submits it.

Submitting the form, of course, causes a new page to be loaded. So, next I check that all is as I expect. Once again, I use `WebDriver::findElement()`, although this time I pass it a `WebDriverBy` object configured for Xpath. If my search is successful, `findElement()` will return a new `RemoteWebElement` object. If my search fails, on the other hand, the resulting exception will bring down my test. Assuming that all is well, I acquire the element's value using the `RemoteWebElement::getText()` method.

At this stage, I have submitted the form and checked the state of the returned web page. You can see the page in Figure 7-2.

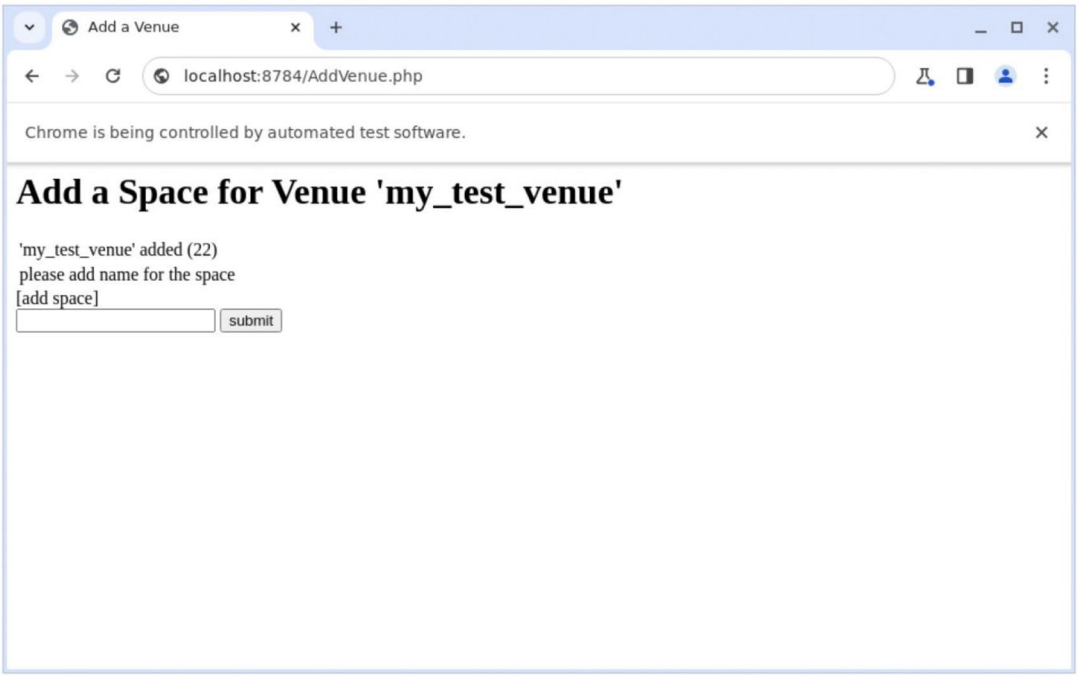

Figure 7-2. *The AddSpace page*

Now, all that remains is to populate the form once again, submit, and check the new page. I use techniques that you have already encountered to achieve this.

Of course, I've only just scratched the surface of Selenium here. But I hope this discussion has been enough to give you an idea of the possibilities. If you want to learn more, there is a complete Selenium manual at `https://www.selenium.dev/documentation/`.

A Note of Caution

It's easy to get carried away with the benefits that automated tests can offer. I add unit tests to my projects, and I use PHPUnit for functional tests, as well. That is, I test at the level of the system, as well as that of the class. I have seen real and observable benefits, but I believe that these come at a price.

Tests add a number of costs to your development. As you build safety into the project, for example, you are also adding a time penalty into the build process that can impact releases. The time it takes to write tests is part of this, but so is the time it takes to

run them. On one system, we may have suites of functional tests that run against more than one database and more than one version control system. Add a few more contextual variables like that, and we face a real barrier to running the test suite. Of course, tests that aren't run aren't useful. One answer to this is to fully automate your tests, so runs are kicked off by a scheduling application like `cron`. Another is to maintain a subset of your tests that can be easily run by developers as they commit code. These should sit alongside your longer, slower test run.

Another issue to consider is the brittle nature of many test harnesses. Your tests may give you confidence to make changes, but as your test coverage increases along with the complexity of your system, it becomes easier to break multiple tests. Of course, this is often what you want. You want to know when expected behavior does not occur or when unexpected behavior does.

Oftentimes, however, a test harness can break because of a relatively trivial change, such as the wording of a feedback string. Every broken test is an urgent matter, but it can be frustrating to have to change 30 test cases to address a minor alteration in architecture or output. Unit tests are less prone than functional tests to problems of this sort because, by and large, they focus on each component in isolation.

The cost involved in keeping tests in step with an evolving system is a trade-off that you simply have to factor in. On the whole, I believe the benefits justify the costs.

You can also do some things to reduce the fragility of a test harness. It's a good idea to write tests with the expectation of change built in, to some extent. I tend to use regular expressions to test output rather than direct equality tests, for example. Testing for a few keywords is less likely to make my test fail when I remove a newline character from an output string. Of course, making your tests too forgiving is also a danger, so it is a matter of using your judgment.

Another issue is the extent to which you should use mocks and stubs to fake the system beyond the component you wish to test. Some insist that you should isolate your component as much as possible and mock everything around it. This works for me in some projects. In others, however, I have found that maintaining a system of mocks can become a time sink. Not only do you have the cost of keeping your tests in line with your system, but you must keep your mocks up to date. Imagine changing the return type of a method. If you fail to update the method of the corresponding stub object to return the new type, client tests may pass in error. With a complex fake system, there is a real danger of bugs creeping into mocks. Debugging tests is frustrating work, especially when the system itself is not at fault.

I tend to play this by ear. I use mocks and stubs by default, but I'm unapologetic about moving to real components if the costs begin to mount up. You may lose some focus on the test subject, but this comes with the bonus that errors originating in the component's context are at least real problems with the system. You can, of course, use a combination of real and fake elements. I routinely use an in-memory database in test mode, for example.

As you may have gathered, I am not an ideologue when it comes to testing. I routinely "cheat" by combining real and mocked components, and because priming data is repetitive, I often centralize test fixtures into what Martin Fowler calls object mothers. These classes are simple factories that generate primed objects for the purpose of testing. Shared fixtures of this sort are anathema to some.

Having pointed out some of the problems that testing may force you to confront, it is worth reiterating a few points that, for my money, trump all objections. Testing accomplishes several things:

- It helps you prevent bugs (to the extent that you find them during development and refactoring).
- It helps you discover bugs (as you extend test coverage).
- It encourages you to focus on the design of your system.
- It lets you improve code design with less fear that changes will cause more problems than they solve.
- It gives you confidence when you ship code.

In every project for which I've written tests, I've had occasion to be grateful for that fact sooner or later.

Summary

In this chapter, I revisited the kinds of tests we all write as developers but all too often thoughtlessly discard. From there, I introduced PHPUnit, which lets you write the same kind of throwaway tests during development but then keep them and feel the lasting benefit! I created a test case implementation, and I covered the available assertion methods. I also examined constraints and explored the devious world of mock objects. Next, I discussed and demonstrated some techniques for testing web applications using PHPUnit and Selenium. Finally, I risked the ire of some by warning of the costs that tests incur and discussing the trade-offs involved.

CHAPTER 8

Vagrant

Where do you run your code?

Maybe you have a development environment you have honed to perfection with a favorite editor and any number of useful development tools. Of course, your perfect setup for writing code is probably very different from the best system on which to run it. And that's a challenge that Vagrant can help you with. Using Vagrant, you get to work on your local machine and run your code on a system that's all but identical to your production server. In this chapter, I will show you how. We will cover the following:

- *Basic setup*: From installation to choosing your first box
- *Logging in*: Investigating your virtual machine with ssh
- *Mounting host directories*: Editing code on your host machine and having it available transparently in your Vagrant box
- *Provisioning*: Writing a script to install packages and configure Apache and MySQL
- *Setting a hostname*: Configuring your box so that you can access it using a custom hostname

The Problem

As always, let's spend a little time defining the problem space. It is relatively easy, these days, to configure a LAMP stack on most desktop or laptop computers. Even so, a personal computer is unlikely to match your production environment. Is it running the same version of PHP? What about Apache and MariaDB? If you're using Elasticsearch, you may need to consider Java or Python, too. The list soon grows. Developing against one set of tools on a particular platform can sometimes be problematic if your production stack is significantly different.

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_8

You might give up and shift your development to a remote machine – there are plenty of cloud vendors who will allow you to spin up a box quickly. But that's not a free option, and, depending upon your editor of choice, a remote system may not integrate well with the development tools you wish to use.

So, it may be worth the effort of matching the packages on your computer as closely as possible with those installed on the production system. The match won't be perfect, but perhaps it will be good enough, and you'll probably catch most issues on the staging server.

What happens, though, when you begin work on a second project with radically different requirements? We have seen that Composer does a great job of keeping dependencies separate, but there are still global packages like PHP, MariaDB, and Apache or Nginx to keep in line.

Note If you decide to develop on remote systems, I recommend making the effort to learn how to use the vim editor. Despite its quirks, it is extremely powerful, and you can be 99% certain that either vim or its more basic ancestor vi will be available on any Unix-like system you encounter.

Virtualization is a potential solution and a good one. It can be a pain installing an operating system, though, and there can be considerable configuration hassles.

If only there were a tool that made creating a production-like development environment on a local virtual machine relatively simple. OK, it's obvious that now I'm going to say that just such a tool exists. Well, one does. It's called Vagrant.

Note There is, of course, another option to consider. Docker provides a lightweight container-based solution to this problem. In a Docker development environment, you break your system down into individual service containers which communicate with one another over a local network. Docker containers (at least when run on a Linux host) operate directly on the host machine's kernel (rather than via a virtualization engine like VirtualBox), making them very easy to deploy. We will examine Docker in more detail in the next chapter. Many development teams have migrated, or are migrating, to Docker for development. Vagrant remains a good option, however, especially when you need to faithfully replicate a development stack in a production environment.

A Little Setup

It is tempting to say that Vagrant gives you a development environment with a single command. That *can* be true – but you do have to install the requisite software first. Given that, and a configuration file that you can check out from your project's version control repository, launching a new environment truly can involve a single command.

Let's get started with the setup first. Vagrant requires a virtualization platform. It supports several, but I will use VirtualBox. My host machine runs Fedora, but you can install VirtualBox on any Linux distribution and on Windows. It is supported on Intel-based Macs but not, unfortunately, on Apple Silicon Macs. You can find the download page at `https://www.virtualbox.org/wiki/Downloads`, together with instructions for your platform.

Once you have VirtualBox installed, you'll need Vagrant, of course. The download page is at `https://developer.hashicorp.com/vagrant/install`. Once we have installed these applications, our next task will be to choose the box we'll run our code on.

Choosing and Installing a Vagrant Box

Probably the easiest way to acquire a Vagrant box is to use the search interface at `https://portal.cloud.hashicorp.com/vagrant/discover`. Since many production systems run Debian, that's what I will look for. You can see the fruits of my research in Figure 8-1.

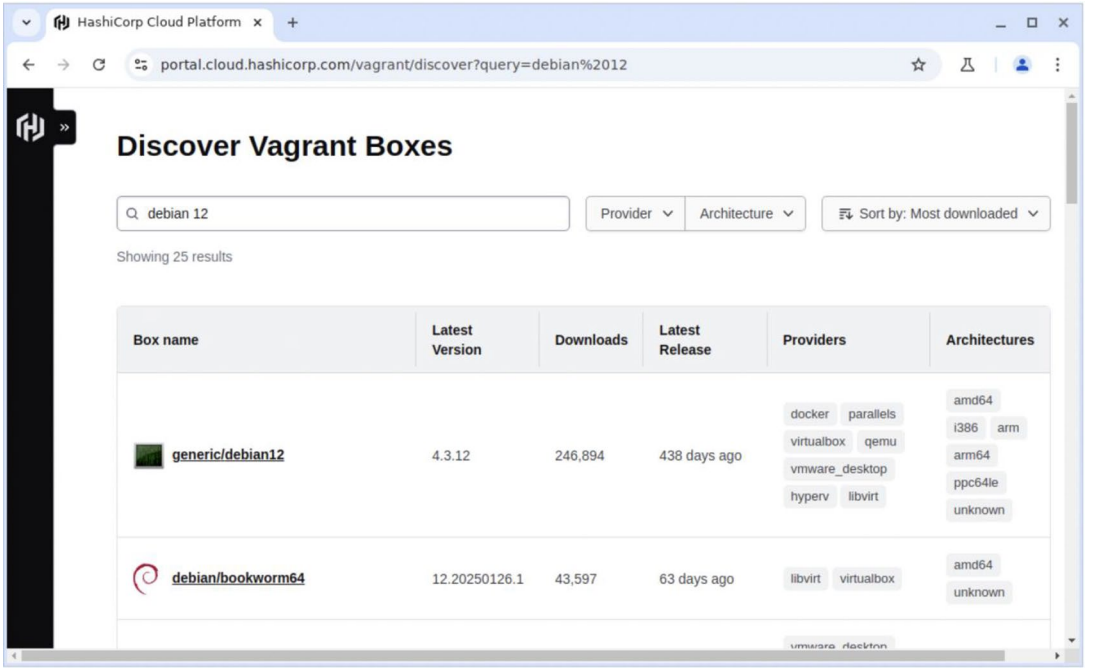

Figure 8-1. *Searching for a Vagrant box*

Debian12 looks about right for my needs. I can click the listing for the box that interests me to get setup instructions. This gives me enough information to get a Vagrant environment running. Usually when you run Vagrant, it will read a configuration file named `Vagrantfile` – but since I am starting from scratch, I need to ask Vagrant to generate one:

```
$ vagrant init generic/debian12
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
```

As you can see, I pass Vagrant the name of the box I want to work with, and it uses this information to generate some minimal configuration.

If I open up the generated `Vagrantfile` document, I can see this (among much other boilerplate):

```
Vagrant.configure("2") do |config|
  # The most common configuration options are documented and
    commented below.
  # For a complete reference, please see the online documentation at
  # https://docs.vagrantup.com.

  # Every Vagrant development environment requires a box. You can
    search for
  # boxes at https://vagrantcloud.com/search.
  config.vm.box = "generic/debian12"
```

At this point, I have only gotten as far as generating configuration. Next, I must run the all-important `vagrant up` command. If you work with Vagrant often, you will soon find this command very familiar. It kicks off your Vagrant session by downloading and provisioning your new box (if necessary), then booting it:

```
$ vagrant up --provider virtualbox
```

Because the `generic/debian12` box supports a number of providers – that is, virtualization engines – I must initially specify `virtualbox`. Once I have created my environment, I can leave that flag out for future runs of `vagrant up`.

In this case, though, I am running this command for the first time with the `generic/debian12` virtual machine, so Vagrant starts by downloading the box:

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Box 'generic/debian12' could not be found. Attempting to find
    and install...
    default: Box Provider: virtualbox
    default: Box Version: >= 0
==> default: Loading metadata for box 'generic/debian12'
    default: URL: https://vagrantcloud.com/generic/debian12
==> default: Adding box 'generic/debian12' (v4.3.12) for provider:
    virtualbox
    default: Downloading: https://vagrantcloud.com/generic/boxes/debian12/
    versions/4.3.12/providers/virtualbox/amd64/vagrant.box
==> default: Successfully added box 'generic/debian12' (v4.3.12) for
    'virtualbox'!
```

```
==> default: Importing base box 'generic/debian12'...
==> default: Checking if box 'generic/debian12' version '4.3.12' is up
    to date...

...
```

Vagrant stores the box (under `~/.vagrant.d/boxes/`) so that you won't have to download it again on your system – even if you run multiple virtual machines. Then, it configures and boots the machine (it provides lots of detail as it does so). Once it has finished running, I can test it out by logging in to my new machine:

```
$ vagrant ssh
$ pwd
/home/vagrant
```

I run the `vagrant ssh` command to log in and then run `pwd` to confirm my working directory on the box. I might also check the operating system:

```
$ cat /etc/debian_version
12.4
```

So, we're in and it all looks sane! What have we won? Well, we have access to a machine that somewhat resembles our production environment. Anything else? Quite a lot, in fact. I said earlier that I would like to edit files on my local machine but run them in a production-like space. Let's set that up.

Time to leave the box again and get back to the host machine:

```
$ exit
```

Mounting Local Directories on the Vagrant Box

Let's put some sample files together. I ran my first `vagrant init` and `vagrant up` commands in a directory I named `infrastructure`. I will resurrect the `woo` project I used in Chapter 7 (a dummy version of the system I developed in Volume 1). Putting all that together, my development environment looks a little like this:

```
ch21/
    infrastructure/
        Vagrantfile
```

```
webwoo/
    AddVenue.php
    index.php
    Main.php
    AddSpace.php
```

Note You'll encounter references to `ch21` in some of the code examples in this chapter. That's because, while this is Chapter 8 of Volume 2, it is also the 21st chapter across both volumes of *PHP 8 Objects, Patterns, and Practice*.

Our challenge is to set up the environment so that we can work with `webwoo` files locally but run them transparently using a stack installed on the Debian box. Depending upon our configuration, Vagrant will attempt to mount directories on the host machine within the guest box.

So, let's instruct Vagrant to mount the `infrastructure` directory as `/vagrant` on the box. That will come in handy when we write a script to provision the box. We will also need to mount the `webwoo` directory, so that its contents can be served.

I open up `Vagrantfile` and add these lines within the configuration section of the document – that is, between `Vagrant.configure("2") do |config|` and `end`:

```
config.vm.synced_folder ".", "/vagrant"
config.vm.synced_folder "../webwoo", "/var/www/poppch21"
```

I can find the best place to put this line by searching the commented boilerplate for the string `synced_folder`. I find a sample configuration line that looks very like my own. With these directives, I am telling Vagrant to mount the `infrastructure` directory on the guest box at `/vagrant` and `webwoo` directory at `/var/www/poppch21`. In order to see that in effect, I need to reboot the box. There's a new command for this (which should be run on the host system and not within the virtual machine):

```
$ vagrant reload
```

The virtual machine shuts down and reboots cleanly. Vagrant mounts the `infrastructure` (`/vagrant`) and `webwoo` (`/var/www/poppch21`) directories. Here's an extract from the command's output:

```
==> default: Mounting shared folders...
    default: /vagrant => /home/mattz/localwork/popp7/src/ch21/
    infrastructure
    default: /var/www/poppch21 => /home/mattz/localwork/popp7/src/
    ch21/webwoo
```

I can log in quickly to confirm that `/var/www/poppch21` is in place:

```
$ vagrant ssh
$ ls /var/www/poppch21/
AddSpace.php  AddVenue.php  index.php  Main.php
```

By the same token, if we were to look at `/vagrant` on the VM, we'd see the contents of the `infrastructure` directory. So, now I can run a sexy IDE on my local machine and have the changes it makes transparently available on the guest box!

Note A note from technical reviewer and Windows user Paul Tregoing: Don't use a VirtualBox shared file system (which underpins Vagrant's synced folder in this example) if running a Windows host. If you do so, you may encounter issues with case sensitivity and lack of symlink support. In this scenario, it's better to run Samba (most distributions install this as `smbd`) on the guest OS and map a network drive on the host for a more seamless experience. There are lots of online guides out there for this.

Of course, placing files on a Debian VM is not the same as running the system. A typical Vagrant box comes without too much preinstalled. The assumption is that the developer will want to customize the environment according to need and circumstance.

The next stage is to provision our box.

Provisioning

Once again, provisioning is directed by the `Vagrantfile` document. Vagrant supports several tools designed for provisioning machines, including Chef (https://www.chef.io/products/chef-infra), Puppet (https://puppet.com), and Ansible (https://www.ansible.com). They're all worth investigating. For the purposes of this example, though, I'm going to use a good old-fashioned shell script.

Note I cover Ansible in Chapter 10.

Once again, I begin with `Vagrantfile`:

```
config.vm.provision "shell", path: "setup.sh"
```

This should be reasonably clear. I'm telling Vagrant to use a shell script to provision my box, and I specify `setup.sh` as the script which should be executed.

What you put in your shell script depends upon your requirements, of course. I'm going to begin by setting a couple of variables.

```
VAGRANTDIR=/vagrant
SERVERDIR=/var/www/poppch21/
```

I can use these variables throughout the setup script. At the time of writing, PHP 8.3 is not available by default on Debian 12. However, it's not particularly difficult to install. Here, I'm adapting the approach recommended by PHP Watch:

```
apt-get -y install apt-transport-https
curl -sSLo /usr/share/keyrings/deb.sury.org-php.gpg https://packages.sury.
org/php/apt.gpg
sh -c 'echo "deb [signed-by=/usr/share/keyrings/deb.sury.org-php.gpg] \
https://packages.sury.org/php/ $(lsb_release -sc) main" > /etc/apt/sources.
list.d/php.list'
apt-get update
```

```
apt-get -y install php8.3 php8.3-cli php8.3-{bz2,curl,mbstring,intl}
apt-get -y install php8.3-fpm
a2enconf php8.3-fpm
```

I use the `apt` package management system to install `apt-transport-https` which supports downloads over HTTPS and then add the Sury PHP package list to the system's list of packages at `/etc/apt/sources.list.d/php.list`. This makes PHP 8.3 available. I run `apt-get update` to update the system. Then, I install PHP 8.3 and various PHP extensions.

Because I'm going to run PHP with Apache using FPM (FastCGI Process Manager), I install the `php8.3-fpm` package. This will add a configuration to the Apache web server which comes installed on this box. I enable this configuration using the `a2enconf`.

Note You can find the original version of this part of the setup script and more explanation at https://php.watch/articles/php-8.3-install-upgrade-on-debian-ubuntu#detailed.

Of course, other distributions will require different strategies for installation. The main takeaway here is that I have installed PHP 8.3 and some PHP extensions. A quick search will provide you with equivalent scripts for your distribution of choice.

I write the script as it currently stands to a file named `setup.sh` which I place in the `infrastructure` directory alongside `Vagrantfile`.

Now, how do I kick off the provisioning process? If the `config.vm.provision` directive and the `setup.sh` script had both been in place when I first ran `vagrant up`, then the provisioning would have been automatic. As it is, I'll now need to run it manually:

```
$ vagrant provision
```

This will spew an awful lot of information onto your terminal as the `setup.sh` script is run within the Vagrant box. Let's see if it worked:

```
$ vagrant ssh
$ php -v
PHP 8.3.6 (cli) (built: Apr 22 2024 10:06:36) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.3.6, Copyright (c) Zend Technologies
    with Zend OPcache v8.3.6, Copyright (c), by Zend Technologies
```

Setting Up the Web Server

Of course, even with Apache installed and configured to work with PHP, the system is not ready to be run. First of all, I need to further configure Apache so that it can serve our code. The easiest way to do this is to create a configuration file that can be copied into place. In the case of Debian, the location in question for this is usually `/etc/apache2/conf-available/`.

Let's call the configuration file `poppch21.conf` and drop it into the infrastructure directory:

```
<VirtualHost *:80>
    ServerName poppch21.vagrant.internal
    ServerAlias poppch21.vagrant.internal
    ServerAdmin matt@getinstance.com
    DocumentRoot /var/www/poppch21
    ErrorLog ${APACHE_LOG_DIR}/poppch21-error_log
    CustomLog ${APACHE_LOG_DIR}/poppch21-access_log common
</VirtualHost>

<directory /var/www/poppch21/>
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</directory>
```

I'll return to that hostname a little later. Leaving aside that tantalizing detail, this is enough to tell Apache about our `/var/www/poppch21` directory and to set up logging. Of course, I'll also have to update `setup.sh` to copy the configuration file at provision time:

```
cp $VAGRANTDIR/poppch21.conf /etc/apache2/sites-available
a2dissite 000-default.conf
a2ensite poppch21.conf
systemctl start apache2
systemctl enable apache2
```

I copy the configuration file into place. Then, I run a utility named `a2dissite` to disable the default configuration – which is a little greedy. By the same token, the `a2ensite` command enables our newly copied configuration. Then, I restart the web server so that the configuration is picked up. I also run `systemctl enable` to ensure that the server will be started at boot time.

After making this change, I can rerun the provision script:

```
$ vagrant provision
```

It's important to note that those parts of the setup script we previously covered will also be rerun. When you create a provisioning script, you must design it so it can be executed repeatedly without serious repercussions. Luckily, `apt-get` detects that my specified packages have already been installed and grumbles harmlessly.

Setting Up MariaDB

For many applications, you'll need to make sure that a database is available and ready for connections. Here's a simple addition to my setup script to install the MariaDB application:

```
apt-get -y install mariadb-server
/usr/bin/mysqladmin -s -u root password 'vagrant' || echo " -- unable to
create pass - probably already done"
domysqldb vagrant poppch21_vagrant vagrant vagrant
```

MariaDB is the modern replacement for MySQL (forked originally from the MySQL source and implementing familiar MySQL tools and commands). I install it with `apt-get`. I run the `mysqladmin` command to create a root password. This will fail after the first run because the password will already be set, so I use the `-s` flag to suppress error messages and print a message of my own if the command fails. Then, I create a database, a user, and a password by running a local function: `domysqldb`. Here it is:

```
function domysqldb {
    ROOTPASS=$1
    DBNAME=$2
    DBUSER=$3
    DBPASS=$4
    MYSQL=mysql
    MYSQLROOTCMD="mysql -uroot  -p$ROOTPASS"
    echo "root command is $MYSQLROOTCMD"

    echo "creating database $DBNAME..."
    echo "CREATE DATABASE IF NOT EXISTS $DBNAME" | $MYSQLROOTCMD || \
        die "unable to create db";
```

```
    echo "DB creation done"
    echo "granting privileges for $DBUSER"
    echo "grant all on $DBNAME.* to $DBUSER@'localhost' identified by
    \"$DBPASS\""
    echo "grant all on $DBNAME.* to $DBUSER@'localhost' identified by
    \"$DBPASS\"" | \
        $MYSQLROOTCMD || die "unable to grand privs for user $DBUSER"
    echo "FLUSH PRIVILEGES" | $MYSQL -uroot -p"$ROOTPASS" || \
        die "unable to flush privs"
    echo "done granting privileges for $DBUSER"
}
```

This simple function creates a database and configures access control by piping command strings to MariaDB. I place it near the top of the `setup.sh` script so that the calling code can find it. With this function in place, I can provision again and then test my database:

```
$ vagrant provision
$ vagrant ssh
$ mysql -u root -pvagrant poppch21_vagrant
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 52
Server version: 10.11.6-MariaDB-0+deb12u1 Debian 12

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [poppch21_vagrant]>
```

We now have a running database and a web server. It's time to see the code in action.

Configuring a Hostname

We have logged in to our new production-like development environment several times, so networking is more or less taken care of. Even though I've configured a web server, I've yet to use it. That's because we still need to support a hostname for our VM. So let's add one to `Vagrantfile` (again, within the `configure` block):

```
config.vm.hostname = "poppch21.vagrant.internal"
config.vm.network :private_network, ip: "192.168.56.10"
```

I invent a hostname and use the `config.vm.hostname` directive to add it. I also configure private networking with `config.vm.network`, assigning a static IP address. You should use private address space for this – an unused IP address beginning with `192.168` should work.

Because this is an invented hostname, we must configure our operating system to handle the resolution. On a Unix-like system, that means editing a system file, `/etc/hosts` on the host machine (not within the Vagrant virtual machine). In this case, I would add the following:

```
192.168.56.10   poppch21.vagrant.internal
```

Note The hosts file on Windows can be found at `c:\Windows\System32\Drivers\etc\hosts`.

Not overly onerous, but we are working toward a one-command install for our team, so it would be good to have a way of automating this step. Fortunately, Vagrant supports plug-ins, and the `hostmanager` plug-in does exactly what we need. To add a plug-in, you simply run the `vagrant plugin install` command:

```
$ vagrant plugin install vagrant-hostmanager
Installing the 'vagrant-hostmanager' plugin. This can take a few
minutes...
Installed the plugin 'vagrant-hostmanager (1.8.10)'!
```

Then, you can explicitly tell the plug-in to update `/etc/hosts`, like this:

```
$ vagrant hostmanager --provider=virtualbox
[vagrant-hostmanager:guest] Updating hosts file on the virtual machine
default...
```

In order to make this process automatic for our team members, we should explicitly enable hostmanager in `Vagrantfile`:

```
config.hostmanager.enabled = true
```

With the configuration changes in place, we should run `vagrant reload` in order to apply them. Then, it's the moment of truth! Will our system run in the browser? As you can see in Figure 8-2, the system should work just fine.

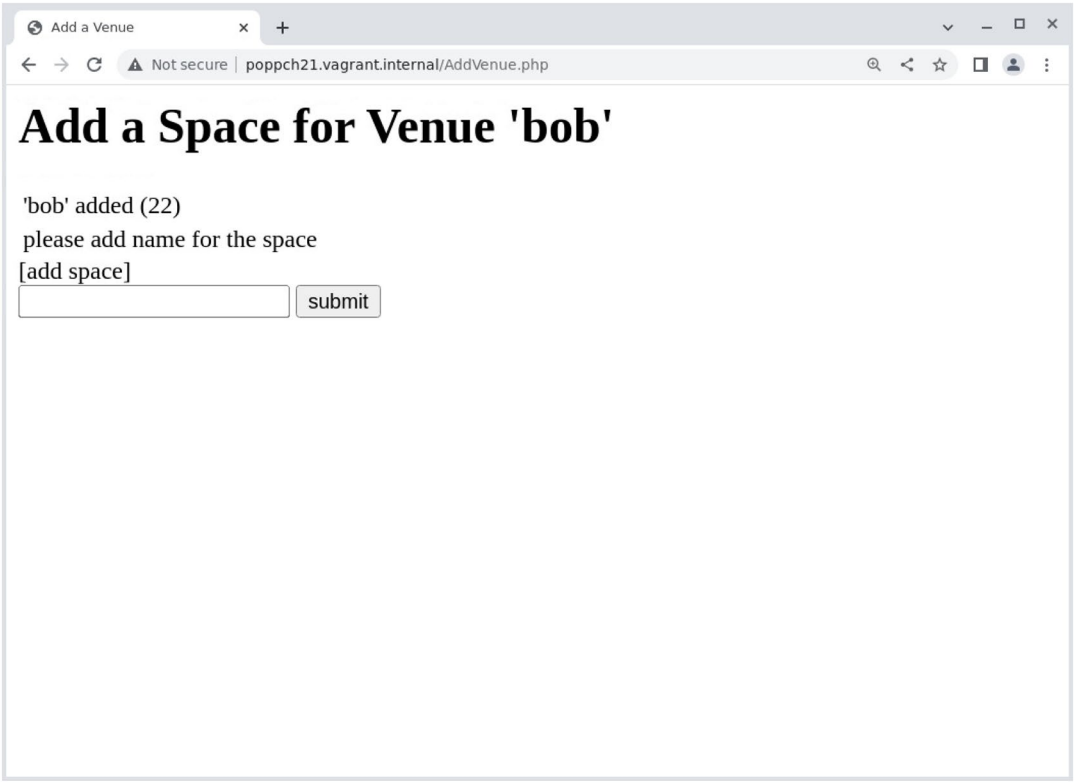

Figure 8-2. *Accessing a configured system on a Vagrant box*

Wrapping It Up

So, we have gone from nothing to a fully working development environment. Given that it took a chapter's worth of effort to get here, it might seem like a bit of a cheat to say that Vagrant is quick and easy. There are two answers to that. First, once you have done this a few times, it becomes a pretty simple matter to spin up yet another Vagrant setup – certainly much easier than trying to juggle multiple dependency stacks by hand.

More importantly, though, the real speed and efficiency gain does not lie with the person who sets Vagrant up. Imagine a new developer coming in to your project expecting days' worth of downloads, configuration file edits, and wiki-clicking.

Imagine telling her, "Install Vagrant and VirtualBox. Check out the code. From the `infrastructure` directory, run 'vagrant up'." And that's it! Compare that with some of the painful onboarding processes you have experienced or heard described.

Of course, we've only scratched the surface in this chapter. As you need to configure Vagrant to do more for you, the official site at `https://www.vagrantup.com` will provide you with all the support you need.

Table 8-1 provides a quick reminder of the Vagrant commands we encountered in this chapter (and a few useful additions).

Table 8-1. *Some Vagrant Commands*

Command	Description
`vagrant up`	Boot the virtual machine and provision if not yet provisioned.
`vagrant reload`	Halt the system and bring it back up (will not run provision again unless run with the –provision flag).
`vagrant plugin list`	List the installed plug-ins.
`vagrant plugin install <plugin-name>`	Install a plug-in.
`vagrant provision`	Run the provision step again (useful if you have updated provision scripts).
`vagrant halt`	Gracefully shut down the virtual machine.
`vagrant suspend`	Stop the virtual machine process and save state.
`vagrant resume`	Resume a previously suspended virtual machine process.
`vagrant init`	Create a new Vagrantfile document.
`vagrant destroy`	Destroy the virtual machine. Don't worry, you can always start again with `vagrant up`!

Summary

In this chapter, I introduced Vagrant, the application that lets you work in a production-like development environment without sacrificing your authoring tools. I covered installation, the choosing of a distribution, and initial setup – including mounting your development directories. Once we had a virtual machine to play with, I moved on to the provisioning process – covering package installation as well as database and web server configuration. Finally, I looked at hostname management, and I showed our system working in the browser!

CHAPTER 9

Docker

Docker is a powerful platform for managing lightweight containers. In plainer English, Docker provides you with tools to run all of your system's components as discrete, fast, interoperable services. Each service can be provided with the environment it needs to do its job (a database service might require MariaDB, for example) often with little or no provisioning required. Together, these services can be used to deploy a reusable tool, a quick demo, a development environment, or a full production-ready stack. The crucial selling point of Docker is that each container packages up an entire system's worth of dependencies, libraries, and components into a self-contained and easily distributable form.

This chapter will cover

- *Getting Docker*: Options for installation
- *Concepts*: Some key terms
- *Acquiring images*: Getting useful images from Docker Hub
- *Generating containers*: How to create and configure powerful services from the command line
- *Building your own images*: Create customized services
- *System management*: Starting, stopping, viewing, and accessing containers
- *Docker Compose*: Taking control to the next level with container orchestration

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_9

What Is Docker?

I love both Vagrant and VirtualBox (the virtualization engine I usually run beneath Vagrant's hood). Still, it is undeniable that a full virtual machine is something of a monolith and that provisioning a working development environment can be quite the project. It can easily take 20 minutes to provision a complex working system from scratch. There is a special agony when provisioning fails at the final step for the fifth time and you face the prospect of tweaking the setup script and beginning yet again from the top. What's more, even when all works as it should, a virtual machine duplicates much of the stack that your operating system is already using, draining a significant fraction of your system's resources.

When running on Linux, Docker (https://docs.docker.com) does not create an entire virtual machine with its own complement of drivers and subsystems. Instead, it allows you to create multiple specialized services or containers, each one running on your host machine's kernel. This makes each container fast and lightweight. Because images already exist to support common services, such as databases, web servers, and cache systems, you can stitch together and initialize a working development environment surprisingly quickly.

Docker on non-Linux systems *does* deploy a virtual machine under the hood. However, even then, you still benefit from Docker's flexibility and convenience and from the way that containers use layered base images, allowing resources to be shared from container to container.

Getting Docker

The Docker documentation site provides a comprehensive overview of installation methods across multiple operating systems and Linux distributions at https://docs.docker.com/engine/install/.

Note In this chapter, I will be working with Docker CE, also known as Docker Engine. If you would prefer a GUI environment, you can install Docker Desktop (https://docs.docker.com/desktop/). As well as a full dashboard experience, this platform also provides the command-line tools I discuss here.

You may be able to install Docker using your distribution's package management system, though you should check the available version. As of this writing, the Docker Engine is at version 28. Alternatively, you might use the convenience script provided at https://get.docker.com. I will be running this on Fedora and Debian distributions.

```
$ curl -fsSL https://get.docker.com -o install-docker.sh
$ sudo sh install-docker.sh
```

By default, you will need to run Docker commands as root in order to access the Unix socket that the Docker daemon creates. This can be tiresome, so you can take a few steps that let you run Docker commands without resorting to `sudo` all the time. The easiest (but not the most secure) solution is to create a `docker` group on your system and add this group to your user.

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
$ newgrp docker
```

If you installed Docker with the `install-docker.sh` script, the `docker` group will already have been added to your system, making the first line of the previous example redundant. However, running it will do no harm.

If you want to avoid running Docker without root privileges, you can configure Docker Engine to run in "rootless mode." The documentation provides instructions for a range of operating systems and distributions at https://docs.docker.com/engine/security/rootless/.

In order to do anything useful at this point, you may need to start the Docker daemon:

```
$ sudo systemctl start docker
```

Hopefully, now, we're ready to run something. I can start by checking my version:

```
$ docker -v
```

It looks like I have the latest version as of this writing:

```
Docker version 28.0.4, build b8034c0
```

Running an Image

Now that I have Docker and confirmed my version, it might be a little more interesting to get and run an image.

```
$ docker run hello-world
```

Here is the output. It's worth including it in full because it nicely summarizes the process.

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
e6590344b1a5: Pull complete
Digest: sha256:7e1a4e2d11e2ac7a8c3f768d4166c2defeb09d2a750b010412b6
ea13de1efb19
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client,
    which sent it to your terminal.
```

I invoked `docker run`, passing it the name of an image, `hello-world`. If you look at the output, you'll see that no `hello-world` existed locally, so Docker searched for it on a remote repository stored, by default on the Docker Hub registry (https://hub.docker.com/). Note that Docker does not just look for `hello-world`. It looks for `hello-world:latest`. Images are tagged – so that you can provide different flavors and versions. The default (and implicit) tag is `latest`.

After my initial run, the `hello-world:latest` image was cached locally so that I won't have to acquire it from the remote server next time. It was used to create a running instance – a container upon which a command was run. It is generally best practice to design images that generate containers as specific services – in this case, one that is

solely responsible for greeting the world! Such services can then be combined to build a system. This is a radically different approach than the one you would take with Vagrant, in which a single VM will tend to encapsulate a complete system.

I can examine local images with `docker image ls`. This can often produce a torrent of output, but, in my newly installed environment, the list is minimal:

```
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
hello-world   latest    74cc54e27dc4   2 months ago    10.1kB
```

Establishing Some Docker Terms

Before we go on, I should pause to itemize some of the terms and concepts I have rushed us through.

Unless they are already locally cached, Docker acquires its images from a *registry* (Docker Hub by default). More specifically, an image will be retrieved from a *repository* which is a namespace within a registry. When we invoke `docker run` or, as you'll see, `docker pull`, we acquire an *image*. An image is a bundle of data that defines a template by which a *container* is generated. A container is an instance of an image. This, then, is broadly analogous to the relationship between classes and objects.

Although it's technically possible to create a single container which runs multiple processes, it's considered best practice to configure each of your containers to support a single service and then to build systems composed of multiple containers as necessary. As you'll see, Docker is designed around this strategy.

Table 9-1 recaps these terms.

Table 9-1. *Some Basic Docker Concepts*

Term	Description
Registry	A server side application (like Docker Hub or GitHub Packages) for storing images.
Repository	A namespaced collection of images within a registry (the Docker Hub default repository is `library`).
Image	According to the Docker documentation, an image is a "standardized package that includes all of the files, binaries, libraries, and configurations to run a container."
Container	An instance of an image, usually designed to run a single service. Multiple containers are often configured to work with one another in order to create an application.

Note Remote image hosting is beyond the remit of this chapter. However, you can read more about working with Docker Hub at https://docs.docker.com/docker-hub/ and more about running your own registry at https://www.docker.com/blog/how-to-use-your-own-registry-2/. Many hosting services such as GitLab and GitHub provide their own registries.

Acquiring an Image with `docker pull`

As we've seen, `docker run` first acquires and then executes a container. Now, let's break this down. We can acquire an image without actually creating a container.

Like most major projects, PHP has an official Docker image (at https://hub.docker.com/_/php). We can cache it locally with `docker pull`.

```
$ docker pull php:8.3-cli
```

Here is my output:

```
8.3-cli: Pulling from library/php
6e909acdb790: Pull complete
31ee84c3cc06: Pull complete
13905c22d489: Pull complete
c1eb1d4cecd3: Pull complete
5dbf261cfd05: Pull complete
7a438ed196d8: Pull complete
291095bacc82: Pull complete
a0afc926b258: Pull complete
82ad65460e66: Pull complete
Digest: sha256:e39867114478af8d8950b679738068deeffa1fa762810aa2
1f6999990411563e
Status: Downloaded newer image for php:8.3-cli
docker.io/library/php:8.3-cli
```

Before we proceed, take a look at that last line. This is the full image name. It incorporates components representing the registry, repository, name, and tag.

Now that I have acquired my image, I can view it locally once again with the `docker image ls` command.

```
$ docker image ls php:8.3-cli
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
php          8.3-cli   cacad2f4349d   3 weeks ago   535MB
```

Let's push on and do something more interesting with our `php` image.

Creating and Invoking a Container with `docker run`

We have already encountered `docker run`, but let's look at the command in a little more detail.

The `docker run` command requires an image name and accepts any additional arguments that the container needs in order to run its primary command. It implicitly pulls an image. That is, it downloads the image and stores it locally. It then uses the image to create and execute a container. A container is configured to invoke a command and exit on completion.

As you might expect, the `php` container runs PHP. In fact, by default, it runs `php -a`. However, it is configured in quite a specific way with regard to the arguments you pass it. If the first argument after the image is a flag (like `-v` or `--version`), it will implicitly pass this, and any subsequent arguments, along to the `php` executable. Otherwise, it will override default behavior and run whatever you pass to it as a standalone command.

Note If you need to learn more about a particular image, the command `docker image inspect <image-reference>` is invaluable. It's like `print_r()` for Docker images! Its cousins `docker container inspect` and `docker network inspect` are similarly useful.

So, if I run a `php:8.3-cli` container with a command, like this:

```
$ docker run php:8.3-cli whoami
```

I will get the output "root." However, if I pass in a flag as the first argument, the behavior changes:

```
$ docker run php:8.3-cli -r 'print "hello\n";'
```

Thanks to the way that this container is configured, the arguments `-r` and 'print "hello";' are passed to PHP. Unsurprisingly, therefore, the output for this is "hello."

It's important to understand that the way that a container will interpret arguments provided to `docker run` may vary somewhat according to implementation. We'll see how when we cover building our own images below.

Note You can read more about `docker run` and its many many flags at https://docs.docker.com/reference/cli/docker/container/run/.

Listing Containers

We have already examined a couple of images. Now, let's look at the container we've created. The command for this is `docker container ls` (though there are various aliases for this command, notably `docker ps`). If I were to run `docker container ls` with no other arguments, I would not see my container. That's because, by default, the command only shows running containers.

A quick look at the documentation at https://docs.docker.com/reference/cli/docker/container/ls/, though, tells me that the `-a` flag will show all containers, running or not. As you might expect, invoking the command with that flag can generate a *lot* of output so you might want to pipe it into `grep` or, as here, use the `--filter` flag.

```
$ docker container ls -a --filter=ancestor=php:8.3-cli
```

This tells Docker to show all containers, running or not, that are derived from the `php:8.3-cli` image.

Here is the output (formatted for readability).

```
CONTAINER ID   IMAGE          STATUS                      NAMES
db3bbe0e040f   php:8.3-cli    Exited (0) 5 seconds ago    focused_solomon
```

As you can see, Docker allocates both an ID and a handy name – "focused_solomon" – to the container. You can allocate your own name with the `--name` option to `docker run` if you wish.

Note The output from `docker container ls` is pretty extensive. So throughout this chapter, I am formatting it to reduce the fields shown. In order to do this, I use the `--format` flag like this: `$ docker ps --filter=ancestor=php:8.3-cli \ --format "table {{.ID}}\t{{.Image}}\t{{.Status}}\t{{.Names}}"`.

Accessing a Container with `docker run`

We have already seen that we can usually pass our own command to a container when calling `docker run`. With the correct flags, we can use this fact to interact with a tool on the container – including a shell.

Here, I use the `bash` shell:

```
$ docker run -it php:8.3-cli /bin/bash
root@1ecf6ab5dbc4:/# ls
bin  boot  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
root@1ecf6ab5dbc4:/# exit
exit
```

I invoked `docker run` with two options. The `-i` (or `--interactive`) option sends any input you generate to the command you provide. In this case, that's the `/bin/bash` shell application rather than PHP. The `-t` (or `--tty`) option attaches a pseudo-TTY – which means that you experience a terminal-like experience. So, I'm able to access the container and work with the command line.

Running a Container in the Background

By default, a container will run in the foreground, hogging your terminal session until you stop it (e.g., with a call to `docker container stop <name-or-id>` run from another terminal). This can be a pain when you're kicking off a long running process. If you call `docker run` with the `-d` (or `--detach`) flag, however, the container will run in the background, and its ID will be output.

Let's try it out:

```
$ docker run -d php:8.3-cli \
    -r 'for ($x=0; ; $x++) { file_put_contents("/tmp/count", "{$x}\n");
    sleep(1); }'
```

So, this tiny script writes a number to a file (`/tmp/count`) once a second until killed. However, the container runs in detached mode, so that I can work with it in other ways. Because `docker run` in detached mode outputs the container ID, I have that to hand. I can also use `docker container ls` (or its synonym, `docker ps`) and see that my container is running.

```
$ docker ps
```

Here is the (reformatted) output:

```
CONTAINER ID   IMAGE          STATUS          NAMES
5ce2ccca805c   php:8.3-cli    Up 10 seconds   sleepy_mestorf
```

Now that I have the container running a long-term process (in a real project, we'd likely be running a web server or a database), I might want to access the container to take a poke about while it's in operation. Since `docker run` creates a *new* container, I'm going to need another way of accessing `sleepy_mestorf`.

Accessing a Container with `docker exec`

The `docker exec` (alias `docker container exec`) command accepts the name or id of a running container and a command argument. It will attempt to run the provided command in the container. Like `docker run`, it supports `-i` and `-t` flags for interactive running and pseudo-TTY operation. That means we can invoke a shell and take a look at the container at work.

```
$ docker exec -it sleepy_mestorf /bin/bash
```

Here's my session:

```
root@5ce2ccca805c:/# cat /tmp/count
94
root@5ce2ccca805c:/#
```

I invoked `docker exec` using the automatically allocated container name, `sleepy_mestorf` (your container would be given a different random name). I could have used the less friendly container ID –`5ce2ccca805c` in this case. Once in, I peek at the `/tmp/count` file to confirm that my little script has been at work.

You can read more about the `docker exec` command at https://docs.docker.com/reference/cli/docker/container/exec/.

Don't forget, incidentally, that, once I'm back on my host machine's command line, I can put an end to `sleepy_mestorf` with `docker container stop`:

```
$ docker container stop sleepy_mestorf
```

This will end the operation of the container and output its identifier.

```
sleepy_mestorf
```

Building Your Own Image

You can go a long way working with off-the-peg images acquired from Docker Hub. Still, it won't be long before you need some customization. You may wish to amend the way that a container is run or to ensure that particular libraries or configurations are bundled into your image.

Let's begin with the first case. I would like to create an image that runs my counter script by default rather than `php -a`.

Here's a script named `counter.php`:

```
print "Arguments: ";
print_r($argv);
print "Beginning the count...\n";
/*
for ($x=0; ; $x++) {
    file_put_contents("/tmp/count", "{$x}\n");
    sleep(1);
}
*/
```

As you can see, I've commented out the counting altogether here and just added some messaging. For now, I don't actually want the container to stick around, just to make some noise and quietly exit. Note that my messaging includes a rundown of any provided arguments.

Now, in order to build my customized image, I need to create a file named `Dockerfile`:

```
FROM php:8.3-cli
WORKDIR /var/myapp
COPY counter.php counter.php
CMD ["php", "counter.php"]
```

The `Dockerfile` is a set of instructions for building an image. You'll always start with a `FROM` instruction, which specifies a base image. If you don't need any specific tool, you might choose a lower level image like `ubuntu`. Unsurprisingly, I'm using `php:8.3-cli` once again for this example.

The `WORKDIR` instruction sets the working directory for subsequent instructions and for the primary process that the container will end up running. If the path does not exist, it will be created.

`COPY` accepts two arguments. The first of these should point to a local (host machine) file or directory. The second should specify a destination within the container.

Finally, `CMD` should define the default command that should be invoked by `docker run`. Because it's more suited for collaboration with another instruction, `ENTRYPOINT` (I'll return to that), I have used the so-called "exec form" here in which the command and any arguments are quoted and placed inside a set of square brackets. I could equally have used the "shell form," in which the command is unquoted. That would have looked like this: `CMD php counter.php`.

The documentation for `Dockerfile` at https://docs.docker.com/reference/dockerfile covers all available instructions.

Now that I have my components ready, it's time to build the image:

```
$ docker build -t mycounter .
```

This command will create an image based on either a directory path or a URL. For the purposes of this chapter, I'll focus on the former use case, so I pass in a reference to my current working directory which contains the `Dockerfile`.

The -t or --tag flag allows you to name and tag your image in the format name:tag. If, as I have, you omit the tag, then Docker will default to latest.

If all goes well, I should have an image named mycounter. Let's check.

```
$ docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
mycounter     latest    6a4b3ef7f909   10 seconds ago   535MB
...
```

And there it is! Let's run the image and see what happens.

```
$ docker run mycounter
```

Here's my output:

```
Arguments: Array
(
    [0] => counter.php
)
Beginning the count...
```

In the Weeds with CMD and ENTRYPOINT

Any image you create will inherit characteristics from the base image you specify. The Dockerfile associated with php:8.3-cli defines its own CMD as ["php", "-a"]. I have overridden this, as you have seen, so that, by default, php counter.php is run instead. What would happen, though, if I were to provide an argument? Let's try it.

```
$ docker run mycounter pwd
```

As you might expect, intuitively, this will cause the pwd command to be run rather than my own php counter.php.

```
/var/myapp
```

If, on the other hand, I provide a flag here, I get a different output. Here's my invocation:

```
$ docker run mycounter -v
```

Because of the magic built in to the php image, this will invoke php with the -v flag:

```
PHP 8.3.8 (cli) (built: Jun 13 2024 05:33:35) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.3.8, Copyright (c) Zend Technologies
```

So, what's going on here? As we have seen, the CMD instruction is easily overridden by providing an argument to docker run. Less easy to casually set aside is another instruction: ENTRYPOINT. This defines a command that is *always* run. Anything specified in CMD (or overridden by providing arguments to docker run) is passed along to the command specified in ENTRYPOINT (use the exec form if you want to configure this behavior into your Dockerfile). Since ENTRYPOINT often just invokes whatever it has been given (typically running /bin/sh -c), this does not usually change much – whatever you define as an argument to docker run will get passed to the shell and executed.

So, this is where the php image performs its cleverness. Its ENTRYPOINT invokes a script that detects an initial flag argument (i.e., an argument beginning with -) and, if a match is found, passes everything along to PHP. That trigger was -v in my example. Otherwise, it attempts to execute whatever argument has been passed (pwd in my example).

Let's rewrite our Dockerfile to override ENTRYPOINT.

```
FROM php:8.3-cli
WORKDIR /var/myapp
COPY counter.php counter.php
ENTRYPOINT ["php", "counter.php"]
CMD []
```

I'll build a new version of the mycounter image using a tag:

```
$ docker image build -t mycounter:entry .
```

I'll take a look at my images just to confirm my new addition:

```
$ docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
mycounter     entry     c5de90beef5d   10 seconds ago   535MB
mycounter     latest    6a4b3ef7f909   5 minutes ago    535MB
...
```

Having confirmed that `mycounter:entry` is there as expected, I can begin to work with it. I have emptied `CMD` so, by default, `counter.php` will be invoked without arguments. However, if I provide additional arguments on the command line, you can see, thanks to the way that I've build `counter.php`, that everything I add is passed along to `counter.php`.

```
$ docker run mycounter:entry ls will not run
Arguments: Array
(
    [0] => counter.php
    [1] => ls
    [2] => will
    [3] => not
    [4] => run
)
Beginning the count...
```

My `docker run` arguments `ls will not run` override the empty `CMD` instruction and end up passed along to the script I defined in the `ENTRYPOINT` instruction. The script dutifully snitches and gives us some insight into the process.

The relationship between `ENTRYPOINT` and `CMD` can be confusing. It's made more so by way that exec and shell modes affect the operations of these instructions. In Table 9-2, I break down the relationships between these instructions and modes.

Table 9-2. *How CMD and ENTRYPOINT Interact in Shell Mode and Exec Mode*

Instruction	Mode	Example	Behavior
ENTRYPOINT	shell	`ls -al`	Implicitly passes argument to `/bin/sh -c`. Will not assign `CMD` values as additional arguments.
ENTRYPOINT	exec	`["ls", "-al"]`	Directly invokes the given command. Will assign `CMD` values (or command-line overrides) as additional arguments.
CMD	shell	`ls -al`	Implicitly generates a shell call like `/bin/sh -c ls -al`. Unless overridden, passes this full statement to `ENTRYPOINT`.
CMD	exec	`["ls", "-al"]`	Unless overridden, passes all arguments to `ENTRYPOINT` without implicit additions.

Mounting a Local Directory

You have seen that you can use the COPY instruction to build an image containing local scripts and configuration. This is useful for setting up a relatively static container. It is less useful, however, for a fast changing file, such as a script under development. To model that situation, here is a very simple script that I have saved to a file named `mytest.php`.

```
print "OUTSIDE IN!\n";
```

If I used COPY to build myself an image, I could run this easily enough. But every time I make a change, I'd have to rebuild my image. Luckily, I have recourse to the `-v` (or `--volume`) option. This sets up a *bind mount* which causes a given directory or file to be mounted within a container. The `-v` option requires a single argument which comprises a local path and a destination path separated by a colon (`:`). Let's try it out.

```
$ docker run --rm -v $PWD:/var/myapp -w /var/myapp php:8.3-cli php
mytest.php
```

Let's begin with `-v`. I map my current working directory on the host environment to a directory (`/var/myapp`) within the container I am initializing. This directory is created if it does not already exist.

I have added a couple more options here which can come in handy. The `-w` (or `--workdir`) option allows you specify your working directory within the container. This is important if the image is not configured by default to use the same directory you have specified with `-v`. The `--rm` option will cause the container to be removed after use. If you don't specify this, the container you create will persist in a stopped state, cluttering up your `docker ps -a` listings.

After all that, you'll not be surprised to learn that running this example results in this output:

```
OUTSIDE IN!
```

While that is not terribly exciting, consider that you can edit `mytest.php` or reference an entirely different PHP file as you wish without having to build a new image.

So, we now have very nearly enough information to build a tiny web development environment with a single command.

A Single Command Development Environment

Although we could use NGINX or Apache for this, I'm going to keep things simple by using PHP's built-in web server.

```
docker run -d \
    -p 3020:8080 \
    -v $PWD:/var/myapp \
    -w /var/myapp \
    php:8.3 \
    php -S 0.0.0.0:8080 -t .
```

Let's run through the aspects of this I have already covered. I use the `-d` option to run the container in a detached state. I configure a bind mount with `-v` so that my current directory is mounted at `/var/myapp` within the container. With `-w`, I set the working directory to `/var/myapp` as well. This is important because, when the primary command (`php -S 0.0.0.0:8080 -t .`), is invoked, I need it to operate upon my directory.

The `-p` (or `--publish`) option describes the mapping of an external port to an internal counterpart. I configured the built-in web server to listen for requests on port 8080 within the container. Thanks to the port mapping, the server can be reached via port 3020 from my host environment. This kind of mapping can come in useful when running multiple containers, since many might otherwise compete for standard ports on the host machine.

Figure 9-1 shows the `mytest.php` script in operation.

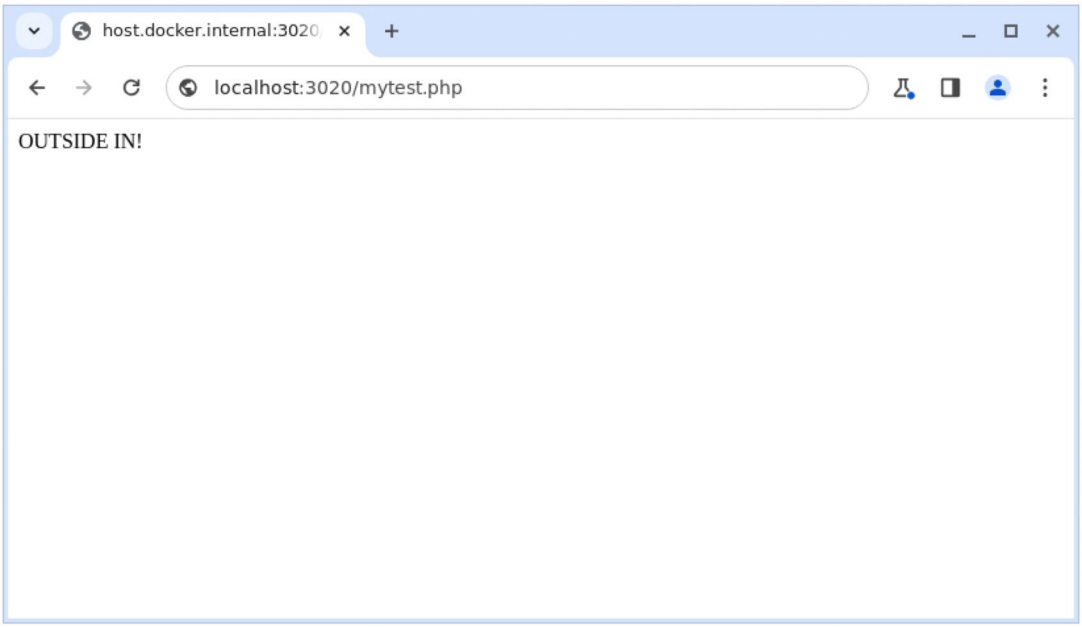

Figure 9-1. *PHP's built-in web server running in a single container*

Building a System Out of Multiple Containers

As we've discussed, a Docker container should run one principle command. Incidentally, this is not the same as running a single process. A web server, for example, will likely spawn multiple subprocesses. How, then, might I create a system that includes a database? To say nothing of all the other components a modern application might require: API, Elasticsearch, and Redis. It soon mounts up.

Let's model this with a second container that we'll use to run a MariaDB instance. I'll begin with some SQL to create a table and tuck it away in a file at `mariadbsetup/1.sql`.

```
CREATE TABLE IF NOT EXISTS `quiz` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
        `quizname` VARCHAR(256) NOT NULL,
        PRIMARY KEY (`id`)
);
INSERT INTO quiz (quizname) values("my lovely quiz");
```

So, I will create a single table named `quiz` and insert a row.

As you might expect, there's an official MariaDB Docker image available. You can read about it at https://hub.docker.com/_/mariadb. Like the PHP image, `mariadb` provides some very useful magic. Let's give it a spin.

```
docker run -d \
    -p 4172:3306 \
    --env MARIADB_ROOT_PASSWORD=megaquiz \
    --env MARIADB_USER=testuser \
    --env MARIADB_PASSWORD=testuser \
    --env MARIADB_DATABASE=megaquiz \
    -v $PWD:/var/myapp \
    -v $PWD/mariadbsetup:/docker-entrypoint-initdb.d \
    -w $PWD \
    mariadb
```

Notice that I am using yet another option to `docker run`. `--env` (or `-e`) sets an environment variable within the container. The MariaDB image is configured to recognize and act on various such variables. I list a few in Table 9-3, and you can see the full set at https://mariadb.com/kb/en/mariadb-server-docker-official-image-environment-variables/.

Table 9-3. *Some Environment Variables Used by the MariaDB Docker Image*

Environment Variable	Description
`MARIADB_ROOT_PASSWORD`	Set the root password
`MARIADB_ALLOW_EMPTY_ROOT_PASSWORD`	If set to a truthy value will run without a root password (inherently insecure)
`MARIADB_DATABASE`	Creates the specified database
`MARIADB_USER`	The non-privileged user name
`MARIADB_PASSWORD`	The non-privileged user's password

For this command, in addition to my current directory, I mount the `mariadbsetup` directory and map it to `/docker-entrypoint-initdb.d`. Any SQL files placed in this directory (in our case a single file named `1.sql`) will be invoked in alphabetical order.

Once that container is running, I can confirm that the database is available.

```
$ echo "select * from quiz" | \
    mysql -u testuser -p -h 127.0.0.1 -P 4172 megaquiz
```

I pipe a simple SQL statement to MariaDB, specifying port 4172. Here's my output:

```
Enter password:
id      quizname
1       my lovely quiz
```

I am prompted for a password. Remember that I set that using the `MARIADB_PASSWORD` environment variable when I ran the `mariadb` image. After that, I see the result of my `SELECT` statement.

Well, that's a great start. I have a web-ready PHP container and a database container. But how can I make them talk to one another? Before I answer that question, perhaps it's time to tidy up.

Removing Images and Containers

It's very easy to let old images and containers accumulate over time (especially if you're using `docker run` without the `--rm` option). This can result in overlong listings, unnecessary resource usage, and port collisions.

I have two containers running that I'll need to stop and remove so that I can generate replacements. Here's a modified listing:

```
CONTAINER ID   IMAGE     PORTS                     NAMES
b58783418150   php:8.3   0.0.0.0:3020->8080/tcp   wonderful_mayer
51f2a35a4c5e   mariadb   0.0.0.0:4172->3306/tcp   eloquent_meninsky
```

The command to stop a running container is `docker container stop`. This command will accept either a container name or an ID (or multiple names/IDs). Once the container is stopped, it can be removed with `docker container rm`. Once again, this requires a name or an ID (and accepts multiple container references).

I'll go ahead and perform those housekeeping actions:

```
$ docker container stop 51f2a35a4c5e
$ docker container rm 51f2a35a4c5e
```

```
$ docker container stop b58783418150
$ docker container rm b58783418150
```

I could have saved myself some time by simply invoking `docker container rm` with the `-f` (or `--force`) option which will implicitly stop the container itself before removing it.

```
$ docker container rm -f 51f2a35a4c5e
$ docker container rm -f b58783418150
```

Remember that I also built two variations of an image named `mycounter`: `mycounter:latest` and `mycounter:entry`. I can filter my overlong image list to find them:

```
$ docker image ls --filter "reference=mycounter"
```

Here's my output.

```
REPOSITORY   TAG       IMAGE ID       CREATED      SIZE
mycounter    entry     a248e671e70d   2 days ago   530MB
mycounter    latest    48f668f05d05   2 days ago   530MB
```

I can delete each of these with `docker image rm`. At least, I can try.

```
$ docker image rm a248e671e70d
```

When I attempt to delete the first of the listed images, I hit a snag. Docker is still managing a stopped container which was generated from this image.

```
Error response from daemon: conflict: unable to delete a248e671e70d
(must be forced) - image is being used by stopped container 22d92508d6e5
```

I could examine the container, but I know that it is disposable so I can rerun the `docker image rm` with the `-f` (or `--force`) option:

```
$ docker image rm -f a248e671e70d
$ docker rmi -f 48f668f05d05
```

Although using `-f` here forces the removal of the images, it doesn't actually remove the containers too. I need to do that myself.

To that end, how about a general cleanup? The `docker image prune` command will remove any images which are not tagged and not used by any container. Or, if you add the `-a` option, it will remove all images which are not used by a container.

The `docker container prune` command will remove all stopped containers.

Now that I've tidied up my docker environment, I can return to getting my containers to talk to one another.

Creating and Using a Named Bridge Network

In order to get my containers to cooperate, I need to create a named network. I can do this with a new command: `docker network create`.

```
$ docker network create quiznet
```

In order to use the newly created `quiznet` network, I can reference it from `docker run` using the `--network` option. I also name the container with the `--name` option. This is important – we'll use the name later.

```
docker run -d \
    --name quizdb \
    --network quiznet \
    -p 4172:3306 \
    --env MARIADB_ROOT_PASSWORD=megaquiz \
    --env MARIADB_USER=testuser \
    --env MARIADB_PASSWORD=testuser \
    --env MARIADB_DATABASE=megaquiz \
    -v $PWD:/var/myapp \
    -v $PWD/mariadbsetup:/docker-entrypoint-initdb.d \
    -w $PWD \
    mariadb
```

It's time to regenerate the PHP container so that it can talk to the database. However, to do that, I'll need call `docker run` on more than the vanilla `php` image. That's because the PHP executable does not come with the `pdo_mysql` extension by default. Luckily, the image *does* provide a handy script for installing extensions: `docker-php-ext-install`. I can run this from the `Dockerfile` during build using a new instruction: `RUN`.

Here is my Dockerfile:

```
FROM php:8.3
WORKDIR /var/myapp
RUN docker-php-ext-install pdo pdo_mysql
CMD ["php", "-S", "0.0.0.0:8080", "-t", "."]
```

Now, let's build the image:

```
$ docker build -t quizimg .
```

I have tagged the image quizimg so that I can reference it when I call docker run:

```
docker run -d \
    --rm \
    --name quizapp \
    --network quiznet \
    -p 3020:8080 \
    -v $PWD:/var/myapp \
    -w /var/myapp \
    quizimg
```

So, the quizapp container is also configured to work with the quiznet network. Other than that, and the fact that the image contains a database-ready PHP executable, it's pretty much identical to the older iteration.

So what do we have now? Here are the two running containers which share the quiznet network:

```
CONTAINER ID   IMAGE     PORTS                    NAMES
5155420d7748   quizimg   0.0.0.0:3020->8080/tcp   quizapp
71939c0b6338   mariadb   0.0.0.0:4172->3306/tcp   quizdb
```

Let's see how they work together. I'll create a script named index.php which will attempt to make a database connection. I'll need to save it in my bind mount directory so that it ends up in the /var/myapp directory of the quizimg container and can be seen by the server. Here's the script:

```
$host = 'quizdb';
$db   = 'megaquiz';
$user = 'testuser';
```

```
$pass = 'testuser';
$dsn = "mysql:host=$host;dbname=$db;charset=utf8mb4";
$pdo = new PDO($dsn, $user, $pass);
$stmt = $pdo->prepare("SELECT * FROM quiz");
$stmt->execute([]);

while($row = $stmt->fetch()) {
    print $row['quizname'] . "<br>";
}
```

There is not much to this script. I construct a DSN (data source name) which references the `megaquiz` database along with the username and password I configured when I created the `quizdb` container. The magic, as far as inter-container communication is concerned, lies with the `$host` variable which contains the name of a container, `quizdb`, on the shared network. Having connected to the database, I make a simple `SELECT` query and output the results. Note that I did not have to specify a port here. The port mapping I defined maps an external port to an internal one. Within my containers, the internal ports are used.

In Figure 9-2, I visit `http://localhost:3020/` and see the results of a successful network connection.

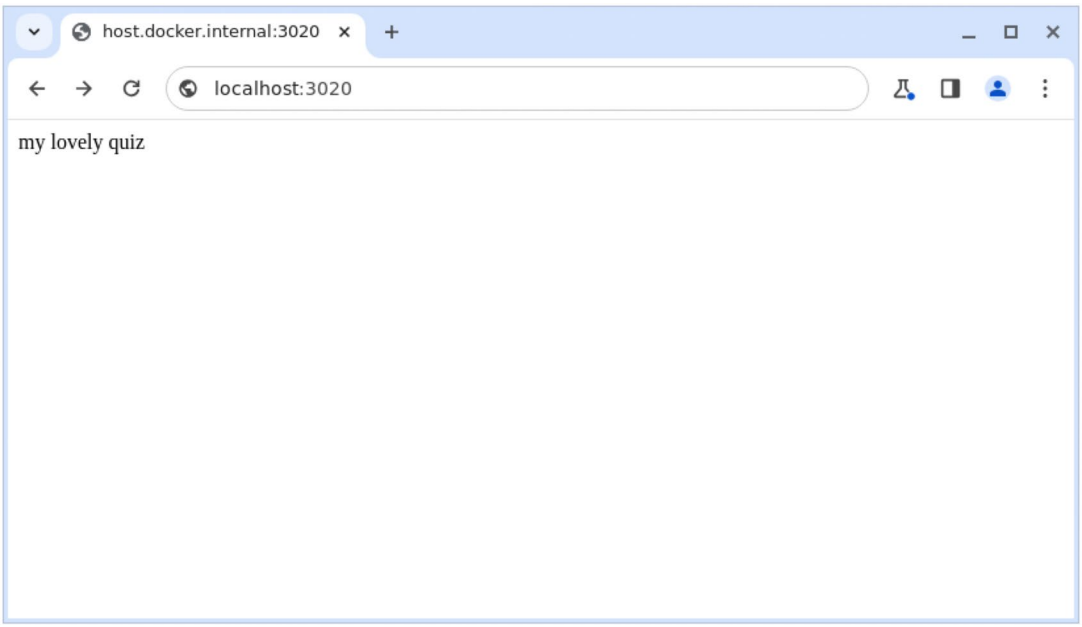

Figure 9-2. *A database query across containers*

Although this is impressive, it's also a massive pain to set up. I must build an image, create a network, and use `docker run` twice with quite complex arguments just to get my simple web app up and running. Of course I could script the process, but even that's a chore. If only there was a tool for orchestrating multiple containers. And, of course, there is! Before I move on to `docker compose`, I'll clean up.

```
$ docker container rm -f quizapp
$ docker container rm -f quizdb
$ docker network rm quiznet
```

Docker Compose

We've covered enough detail to get a development environment working with Docker using commands such as `docker build` and `docker run` in conjunction with `Dockerfile` configuration as needed. However, it has to be admitted that the process became progressively more unwieldy as I piled on more and more steps and options.

Docker Compose is a tool for bringing all of this work together into a single YAML configuration file. In one place, you can define all the services that make up your stack as well as any necessary networks and volumes. What's more, it provides straightforward tools for starting, stopping, and rebuilding your environment, as well as for essential tasks like viewing logs and checking your system's status.

Don't worry, though. The concepts you've already encountered won't be wasted here!

Note YAML (https://yaml.org/) is a compact but human-friendly data serialization language. Like JSON (to which it is related), it allows an author to represent complex data structures in documents which are both easy for humans to understand and for computers to parse.

Resetting the Project

I'll begin again here with a single container. All I'll need for that is a single Compose file and a basic script to prove that my set up works. My initial directory structure will look like this:

```
compose.yaml
web/
    index.php
```

The script `index.php` will just output a cheery message for now.

```
print "docker compose!";
```

The Compose File

As I have discussed, Docker Compose is managed using a YAML configuration file. This should be named `compose.yaml` (the preferred name according to documentation) or `docker-compose.yaml` (the `.yml` extension is also acceptable).

Here's my Compose file:

```
services:
    quizapp:
        image: php:8.3
        command: php -S 0.0.0.0:8080 -c conf/php.ini -t web
        working_dir: /var/myapp
        ports:
            - "3020:8080"
        volumes:
            - .:/var/myapp
```

At its core, a Compose file defines a set of services. Here, I begin with a `quizapp` service. The `image` definition defines the image that will form the basis of a running container. The `command` definition establishes the primary process of the service. `working_dir` is similar to the `-w` option in the `docker run` command or to the `WORKDIR` instruction in `Dockerfile` – it sets (and creates if necessary) the working directory within the container. The `ports` definition establishes port mapping and `volumes` mounts local files or directories within the container. Local paths run relative to the location of the Compose file.

Let's get the system running:

```
$ docker compose up
```

By default, the output to `docker compose up` is often very verbose. It will show you the process by which all images are acquired and the containers are configured. Then, it will track the logs for each of the primary commands. For this minimal example, the output (formatted a little here) is pretty manageable.

```
[+] Running 1/1
- Container batch04-quizapp-1  Created                0.0s
Attaching to quizapp-1
quizapp-1  | [Mon Apr  7 16:44:42 2025] PHP 8.3.19
Development Server (http://0.0.0.0:8080) started
```

If I had run this with a `-d` (or `--detach`) option, the containers would start in the background. As it is, I can only watch in this particular terminal window until I stop the process by hitting `Ctl-C`. In a separate terminal, though, I can confirm the running container using `docker ps` (output edited).

```
CONTAINER ID   IMAGE     STATUS         NAMES
c50fb3aa56eb   php:8.3   Up 6 minutes   batch04-quizapp-1
```

The name of the container is constructed in part from the service name and the name of the Compose file's parent directory. I should now be able to confirm that the web server in the `quizapp` service is running by pointing my browser at `http://localhost:3020` as shown in Figure 9-3.

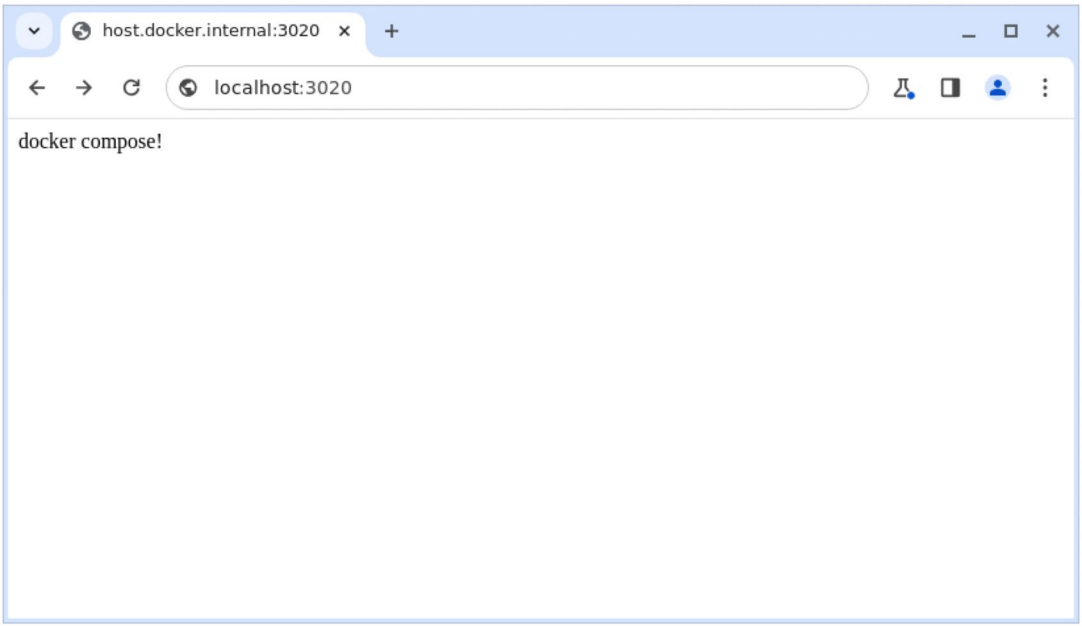

Figure 9-3. *The* `quizapp` *service in operation*

Combining Docker Compose and `Dockerfile`

If I am to recreate my previous example, I will need more than the `php:8.3` base image. I need to add the `pdo_mysql` extension, which means building an image. For this, I'm going to use exactly the same `Dockerfile` as before, but I'll tuck it away under a subdirectory. So, my directory structure now looks like this:

```
compose.yaml
web/
    index.php
dockerdir/
    quizapp/
        Dockerfile
```

Now, instead of specifying the `php:8.3` image, we need to build our own based upon it.

```
services:
    quizapp:
        build: ./dockerdir/quizapp
```

```
    command: php -S 0.0.0.0:8080 -c conf/php.ini -t web
    working_dir: /var/myapp
    ports:
        - "3020:8080"
    volumes:
        - .:/var/myapp
```

So, the only difference here is that I have swapped `image` for a `build` definition. This specifies the path to the directory that contains the `Dockerfile`. Since I'm still running my previous iteration of this configuration, I should first stop it with `Ctl-C`. For good measure, I could then run `docker compose rm` to remove the stopped container.

When I run `docker compose up` after my edit, Docker will first build an image for the server and then create a new container.

Adding a Second Service

Let's add a MariaDB container to the mix. Although I won't need to build my own image for this, I do want to set up the database as before. I'll need to add my SQL file in order to create the `quiz` table and mount the directory. Here's my new directory structure:

```
compose.yaml
web/
    index.php
dockerdir/
    quizapp/
        Dockerfile
    quizdb/
        mariadbsetup/
            01.sql
```

We have already seen the contents of that `01.sql` file. Remember that it contains SQL to create a `quiz` table and insert a row. Now, I'll add the `quizdb` service to `compose.yaml`.

```
services:

    # ...
```

```
    quizdb:
        image: mariadb
        restart: always
        environment:
          MARIADB_ROOT_PASSWORD: quizroot
          MARIADB_DATABASE: megaquiz
          MARIADB_USER: testuser
          MARIADB_PASSWORD: testuser
        volumes:
          - ./dockerdir/quizdb/mariadbsetup:/docker-entrypoint-initdb.d
        ports:
            - "4172:3306"
```

I have included a `restart` definition here. This defines a restart policy (`always`) that causes the container to start again if, for any reason other than a manual intervention, it stops operating. Also new here is the `environment` definition, which I use to set environment variables that the `mariadb` image deploys when configuring MariaDB in the container.

I might choose, this time, to run Docker Compose in the background with `docker compose up -d`. If you go that route, you can run `docker compose stop` from the same directory to stop all services but keep the containers or `docker compose down` to stop and remove the running containers.

While Docker Compose is running in the background, you can tail log output at any time with `docker compose logs -f`. That `-f` flag means "follow," and you can omit it if you just want to grab a snapshot of the logs rather than tail them. You can focus log output by specifying a service. So, if I only wanted to track log output for the `quizdb` service (the name I defined in the `compose.yaml` file), I would run `docker compose logs -f quizdb`.

I can confirm again that `MariaDB` is accessible via port 4172 from the host machine. I can also inspect my running containers with `docker compose ps` (similar to `docker container ls` but pre-filtered for the current Docker Compose configuration).

Although, if I needed to do any advanced network configuration, I could create a `network` definition and set custom drivers, specify pre-existing networks, and so on, none of that is necessary to get my containers linked up. All running services are already visible to one another via their service names. This means that `quizapp` can access the database via the host `quizdb` and vice versa. So, now I can update `index.php` so that it makes a network connection to `quizdb` and runs a query.

As a reminder, here is that script:

```
$host = 'quizdb';
$db   = 'megaquiz';
$user = 'testuser';
$pass = 'testuser';
$dsn = "mysql:host=$host;dbname=$db;charset=utf8mb4";
$pdo = new PDO($dsn, $user, $pass);
$stmt = $pdo->prepare("SELECT * FROM quiz");
$stmt->execute([]);

while($row = $stmt->fetch()) {
    print $row['quizname'] . "<br>";
}
```

I have come full circle now. I can access my script from a browser at `http://localhost:3020` and see it in operation as in Figure 9-2. Although functionally identical to the `docker run` version, however, my system is much easier to manage now. I do not have to worry about individually starting and stopping containers or about creating a network and joining my containers to it.

What About Composer?

Composer is an integral part of most PHP projects, so it's worth considering how one should go about incorporating it. Of course, I'll need a `composer.json` file. This is my final directory layout for the Docker Compose example:

```
compose.yaml
composer.json
web/
    index.php
dockerdir/
    quizapp/
        Dockerfile
    quizdb/
        mariadbsetup/
            01.sql
```

I have just configured `composer.json` to install a sample package or two. How should I go about running `composer install` or `composer update`? One approach for development is to simply run these commands from the host machine. After all, the directory containing the `vendor/` is mounted. This is not a good solution, however, since your host machine's configuration may not match that of your PHP container. Indeed, some developers might not even have PHP or Composer installed on their host machines.

Another approach might be to install composer into your `php` container by adding something like this to your `Dockerfile`.

```
RUN curl -sS https://getcomposer.org/installer | \
            php -- --install-dir=/usr/local/bin \
            --filename=composer \
            && composer install
```

Or, similarly, by applying one of the various other mechanisms described for programmatically installing Composer at https://getcomposer.org/doc/faqs/how-to-install-composer-programmatically.md.

While this approach can be made to work, there is a much neater solution that epitomizes Docker's architectural philosophy.

We can add a `composer` service to the Compose file.

```
services:

    # ...

    composer:
        image: composer
        working_dir: /var/myapp
        command: ["composer", "install"]
        volumes:
            - .:/var/myapp
```

This uses the official Composer image (https://hub.docker.com/_/composer). I mount the base directory, as I do in the `quizapp` service. Then, I define `composer install` as the container's primary command. This will run when I invoke `docker compose up -d`. The container will stop once the command has finished running, and that's fine for my purposes.

While this use of a Composer service is very clean, I have to admit that, in the wild, I have more often seen it invoked in a Dockerfile or form setup script than using this method.

Some Docker Compose Commands

I'll conclude this section with Table 9-4 – an overview of some useful Docker Compose commands. You have seen most, but not all, of them already. Docker Compose commands operate on services defined by a `compose.yaml` (or equivalent) file in the current directory or as referenced by the `-f` option.

Table 9-4. *Some Docker Compose Commands*

Command	Description
docker compose up	Start all services. Build images and create containers as necessary. Use the `-d` option to run in detached mode.
docker compose stop	Stop all containers but keep them.
docker compose down	Stop and remove all containers.
docker compose rm	Remove all stopped containers.
docker compose logs	Show all logs. You can specify a service name to narrow your view and use the `-f` option (or `--follow`) to follow the log output.
docker compose ps	View container statuses.
docker compose restart	Restart all services (or those specified as arguments).

Summary

Docker is a hugely powerful technology, and a single chapter like this can only provide an introduction. However, you should find enough information here to get you up and running with Docker for day-to-day development. Once you get started, I think you'll find containers addictive.

In this chapter, I covered Docker core concepts. I explored techniques for acquiring and building images and generating containers. I joined containers up to one another via a bridge network and built a small working environment with a web server and a database server. Finally, I introduced Docker Compose, a powerful tool for managing your containers.

CHAPTER 10

Automating Build and Deployment with Ansible

If version control is one side of the coin, then automated build is the other. Version control allows multiple developers to work collaboratively on a single project. But that code remains inert until it is properly deployed and combined with configuration in an environment with all dependencies in place. Such dependencies will include the wider software stack – the Linux distribution, the configured web server, a database and schema, the presence of PHP installed with the correct extensions. Additionally, a system will also require numerous libraries, probably installed and updated via Composer, as well as front-end libraries which may be included using tools like NPM. As a project grows in power and complexity, so the number of steps required to install and update an instance will increase.

This requirement holds for development environments, but it is also important to be able to deploy staging and production instances of a system with a minimum of impedance.

In this chapter, I introduce you to Ansible, which can handle all the jobs mentioned so far and many more besides. This chapter will cover the following:

- *Getting and installing Ansible*: Who builds the builder
- *Command-line tools*: An overview of Ansible and its CLI commands
- *The building blocks*: Playbooks, plays, tasks, modules, and inventories explained
- *Deploying to multiple hosts*: From a local directory to 30 servers; Ansible is designed to scale

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_10

- *Checking out a repository*: Getting your code in place
- *Copying and altering files*: Managing configuration
- *Ansible vault*: Keeping your secrets (and letting you store them in a version control system)
- *Variables*: Managing data that changes according to context

What Is Ansible?

Ansible (`https://docs.ansible.com`) is an "automation engine." It is designed, in particular, to manage the provisioning of servers, the deployment of applications, and the management of configuration. In other words, all the essentials that many books and articles about programming wave their figurative hands at. "Those values can be stored in a configuration file. Remember not to check secrets into version control. This is just a detail and we'll leave it to you to work it all out for yourself."

Well, in this chapter, we'll take a crack using Ansible to manage the deployment of a small PHP application across multiple hosts. We'll look at a strategy for managing configuration data across modes – production, staging, and development. We'll store some secrets without exposing them unencrypted either to a version control system or to unauthorized team members.

Getting Ansible

There are a whole bunch of ways of installing Ansible. It's probably easiest, though, to use your operating system's package management system. For example, on my Fedora machine, I use `dnf`:

```
$ sudo dnf install ansible
```

Or on my Mac, I might use Homebrew:

```
$ brew install ansible
```

Note The Ansible documentation site provides exhaustive coverage of many of the ways you can install the system at https://docs.ansible.com/ansible/latest/installation_guide/index.html.

Confirming Your Install

Once you have installed Ansible, it's a good idea to check that it all looks sane. I'll run a command that you'll see a lot more of in this chapter: `ansible-playbook`.

```
$ ansible-playbook --version
```

Here's the output on my Mac – the so-called *control node* which will deploy code on three Linux hosts (the *managed nodes*):

```
ansible-playbook [core 2.16.7]
  config file = None
  configured module search path = ['/Users/mattz/.ansible/plugins/modules',
  '/usr/share/ansible/plugins/modules']
  ansible python module location = /opt/homebrew/Cellar/ansible/9.6.0/
  libexec/lib/python3.12/site-packages/ansible
  ansible collection location = /Users/mattz/.ansible/collections:/usr/
  share/ansible/collections
  executable location = /opt/homebrew/bin/ansible-playbook
  python version = 3.12.3 (main, Apr  9 2024, 16:03:47) [Clang 14.0.0
  (clang-1400.0.29.202)] (/opt/homebrew/Cellar/ansible/9.6.0/libexec/
  bin/python)
  jinja version = 3.1.4
  libyaml = True
```

That seems sane enough, if a tad verbose. Let's cover off what I've installed very quickly.

Command-Line Utilities

Although we'll only use three commands in this chapter, we have actually installed quite a few tools. Table 10-1 summarizes some of the available Ansible commands.

Table 10-1. *Some Ansible Command-Line Utilities*

Utility	Description
`ansible-playbook`	Runs a `playbook` (a deployment script) in conjunction with other files to manage multiple remote hosts.
`ansible-galaxy`	Installs *collections* and *roles* – that is, components that extend Ansible's functionality.
`ansible`	Runs a given command. Typically used for testing or for one-off scenarios.
`ansible-doc`	Provides information on installed modules.
`ansible-pull`	Retrieves a playbook from a version control repository and executes on a target host.
`ansible-console`	A console environment for running Ansible commands.
`ansible-inventory`	Shows Ansible inventory information (i.e., information about a set of managed hosts).
`ansible-vault`	Encrypt and decrypt secrets that can be safely stored in version control and included in application configuration.
`ansible-config`	View configuration.

The (inexhaustive) list of utilities in Table 10-1 might seem daunting, but, luckily, you can go a long way with the few commands we cover in this chapter. Let's get started with an inevitable *Hello, World*.

Hello, Ansible

The command we'll encounter mostly throughout the rest of this chapter is `ansible-playbook`. This command combines a script – or *playbook* – (optionally alongside an inventory of hosts) and runs these instructions on the specified targets.

Note You can read about playbooks at https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html.

Let's create an initial playbook:

```
# 001-playbook.yml

- name: Ansible Says Hello
  hosts: 127.0.0.1
  connection: local

  tasks:
  - name: Send Output
    ansible.builtin.debug:
      msg: Hello, world!
```

A playbook consists of a set of named *plays* which themselves consist of set of playbook keywords (`hosts`, for example, specifies a target host or references a group of hosts specified in an inventory) and tasks. A task (like the `Send Output` example above) invokes a module (a function) which will be applied to the target hosts.

As you can see, playbooks are written in YAML format. YAML (YAML Ain't Markup Language) is a subset of JSON – a data serialization language designed to be compact and human-friendly. You can read more about it at https://yaml.org/.

So, lets run through this example line by line. I have created a playbook (which I've saved in a file named `001-playbook.yml`). This consists of a single play: `Ansible Says Hello`.

I have used two playbook keywords: `hosts` tells Ansible where we will be working and `connection` tells it we will be operating on a local environment.

Note You can read more about playbook keywords at https://docs.ansible.com/ansible/latest/reference_appendices/playbooks_keywords.html.

Then, I move on the `tasks` section of the play. I have defined only one: `Send Output`. This uses the `ansible.builtin.debug` module (https://docs.ansible.com/ansible/latest/collections/ansible/builtin/debug_module.html) which accepts and outputs a `msg` parameter.

A module can be thought of as a function. They are also referred to as "task plug-ins." Modules support parameters and attributes which determine how they are run. They are collected in namespaced collections which can be installed with the `ansible-galaxy` command. You can run a module directly with the `ansible` command, but you'll see them most often as part tasks in playbooks.

Let's run the playbook:

```
$ ansible-playbook 001-playbook.yml
```

And I'm rewarded with quite a lot of output:

```
[WARNING]: No inventory was parsed, only implicit localhost is available
[WARNING]: provided hosts list is empty, only localhost is available. Note
that the implicit localhost does not match 'all'

PLAY [Ansible Says Hello] ******************************************************

TASK [Gathering Facts] *********************************************************
ok: [127.0.0.1]

TASK [Send Output] *************************************************************
ok: [127.0.0.1] => {
    "msg": "Hello, world!"
}

PLAY RECAP *********************************************************************
127.0.0.1                  : ok=2    changed=0    unreachable=0    failed=0
    skipped=0    rescued=0    ignored=0
```

Notably, Ansible has sent out its greeting, along with a lot of other information (that will be more useful when we're running multiple tasks and applying them to many remote hosts).

It's also worth noting those warnings in the output. Ansible is happiest when it's engaging with a list of remote hosts. Let's see how that works.

Inventories: Working with Hosts

An inventory (https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html) is a file which collects together lists of hosts – often organized in named groups. You can create an inventory in YAML format or opt for the even more compact (but less flexible) INI format.

Let's create an inventory file at `inventories/example/hosts.ini`:

```
[myservers]
192.168.1.98
192.168.1.7
192.168.1.82
```

If the script is running on my Mac – my control node – the IP addresses in this inventory file refer to three managed nodes (also known as three battered laptops humming away in various nooks around my house).

Note If you are coding along and you don't have a bunch of servers to play with, you could always spin up some Vagrant machines and use their IPs in your inventory file. I covered Vagrant in Chapter 8.

Here's the equivalent – `hosts.yml` – formatted with YAML:

```
myservers:
  hosts:
    192.168.1.98:
    192.168.1.7:
    192.168.1.82:
```

By default, Ansible connects to servers via OpenSSH. Unless you specify a remote user for your servers, it will use your username (more accurately, it will use the username associated with the "control node"). I will be specifying a remote user for host connections later on.

Before you run a playbook that connects to a remote machine, you need to ensure that you have SSH access to the target servers. Ideally, you will add a public key to your target servers, making Ansible's access relatively frictionless. If your servers are properly configured with your public key, you should be able to access them over SSH without a password. If, when you generated your key pair you specified a passphrase, you can use a tool called `ssh-agent` to store it in memory at the start of your session. You can read more about Ansible and connection methods at https://docs.ansible.com/ansible/latest/inventory_guide/connection_details.html.

Note I covered generating a key pair and adding a public key to a target server in Chapter 6 in the section "Providing Access to Users."

Don't despair if you haven't set up key-based access to your servers. If you are able to connect to your remote hosts using a password, you can specify the `--ask-pass` option when you run `ansible-playbook`. That will cause Ansible to use password authentication, and you will be prompted for your password as needed.

Now, I create a playbook designed to work with either version of this inventory file:

```
# 002-playbook.yml

- name: Pingy ping ping
  hosts: myservers
  remote_user: webuser
  tasks:
    - name: Ping my hosts
      ansible.builtin.ping:

    - name: Run pwd
      ansible.builtin.shell:
```

```
      cmd: pwd
    register: loc

  - name: Print pwd output
    ansible.builtin.debug:
      var: loc.stdout
```

So, my new play here is a little different. Rather than specify a particular IP address, I have referenced a *group* – `myservers`. I no longer need the `connection` keyword here because a remote connection is assumed. I have added a new keyword though: `remote_user` specifies the user I will connect as on each target host.

This time round, I've defined three tasks. The first, `Ping my hosts`, uses the `ansible.builtin.ping` module. This does a lot more than just confirm that the lights are on at the target host as we shall see.

While I'm at it, I define a task named `Run pwd` which invokes `ansible.builtin.shell` (https://docs.ansible.com/ansible/latest/collections/ansible/builtin/shell_module.html). This, as you might expect, invokes a given command – `pwd` in this case. A module generates a return value in JSON format. The `register` task keyword will create a variable to which it will assign the task's output. This can be used later in the play.

In this case, "later in the play" means the `Print pwd output` task which, once again, uses the built-in `ansible.builtin.debug` module. This time, though, instead of `msg`, I use `var` to output a variable value. Remember, I registered `loc` in the previous task. The more specific `loc.stdout` refers to an element in the JSON we expect `ansible.builtin.shell` to have generated. The full output for a single call to `ansible.builtin.shell` might look something like this:

```
{
   "changed": true,
   "cmd": "pwd",
   "delta": "0:00:00.004249",
   "end": "2024-06-09 12:17:08.613542",
   "failed": false,
   "msg": "",
   "rc": 0,
   "start": "2024-06-09 12:17:08.609293",
   "stderr": "",
```

```
        "stderr_lines": [],
        "stdout": "/home/webuser",
        "stdout_lines": [
            "/home/webuser"
        ]
    }
```

Because this value is quite extensive, it makes sense to narrow down our own output by focusing on an individual element: `stdout`, in this case.

Now, I can run the playbook, this time specifying an inventory file.

```
$ ansible-playbook -i inventories/example/hosts.ini 002-playbook.yml
```

I include my inventory file with `-i` flag. I could also have just referenced a directory and Ansible would have happily parsed all contained files.

Here's my output:

```
PLAY [Pingy ping ping] *********************************************************

TASK [Gathering Facts] *********************************************************
ok: [192.168.1.82]
ok: [192.168.1.7]
ok: [192.168.1.98]

TASK [Ping my hosts] ***********************************************************
ok: [192.168.1.82]
ok: [192.168.1.98]
ok: [192.168.1.7]

TASK [Run pwd] *****************************************************************
changed: [192.168.1.82]
changed: [192.168.1.98]
changed: [192.168.1.7]

TASK [Print pwd output] ********************************************************
ok: [192.168.1.98] => {
    "loc.stdout": "/home/webuser"
}
```

```
ok: [192.168.1.7] => {
    "loc.stdout": "/home/webuser"
}
ok: [192.168.1.82] => {
    "loc.stdout": "/home/webuser"
}

PLAY RECAP *********************************************************************
192.168.1.7  : ok=4  changed=1  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
192.168.1.82 : ok=4  changed=1  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
192.168.1.98 : ok=4  changed=1  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

As you can see, each task runs three times – once for each host in the `myservers` group. The `ping` module runs successfully. Although the output does not show the actual results of our calls to `ansible.builtin.ping`, it looks like this in each case:

```
    {
       "changed": false,
       "failed": false,
       "ping": "pong"
   }
```

However, the data here is less important to us than the fact that module ran successfully.

The `Run pwd` task reports a successful execution for each host, but, as discussed, in order to see the output, I use the `Print pwd output` task.

In order to run, the `ping` module, like most others that act on remote servers, requires full access to its target host. Typically, this means that you will have installed your key in the relevant user account for each host. Remember, however, that if you have not set up keys on your target servers but do have password access, you can use the `--ask-pass` option to force password authentication:

```
$ ansible-playbook --ask-pass -i inventories/example/hosts.ini
002-playbook.yml
SSH password:
```

Checking Out a Git Repository

Having demonstrated the use of a playbook with an inventory, I'll bring things back on track with a playbook which will perform an actual deployment.

We really have all we need for this apart from the addition of a new built-in module:

```
# 003-playbook.yml

- name: Deploy code
  hosts: myservers
  remote_user: webuser
  tasks:
   - name: clone
     ansible.builtin.git:
         repo: 'git@github.com:poppbook/megaquiz.git'
         dest: /home/webuser/app
         version: v1.0.1
```

As you might expect, the `ansible.builtin.git` module (https://docs.ansible.com/ansible/latest/collections/ansible/builtin/git_module.html) checks out a Git repository. Reflecting Git's power and complexity, the module accepts many possible arguments, but we only need three to clone or pull our repository. The `repo` parameter accepts the repository's address. `dest` defines the destination directory. `version` requires information about what to check out. This could be a branch name, a SHA-1 hash representing a commit, or, as in this example, a tag.

Let's run it:

```
$ ansible-playbook -i inventories/example/hosts.ini 003-playbook.yml
PLAY [Deploy code] *************************************************
**********

TASK [Gathering Facts] *********************************************
**********
ok: [192.168.1.82]
ok: [192.168.1.7]
ok: [192.168.1.98]

TASK [clone] *******************************************************
**********
```

```
ok: [192.168.1.98]
ok: [192.168.1.82]
ok: [192.168.1.7]

PLAY RECAP *********************************************************************
192.168.1.7   : ok=2  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
192.168.1.82  : ok=2  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
192.168.1.98  : ok=2  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

So, my run was glitch-free. You may find you need to deal with initial gremlins. As I've already discussed, you need either to have configured key-based access to your servers or specify the `--ask-pass` option in order to fall back to password authentication. Also, make sure that your target servers are configured for access to your Git repository.

Having run this playbook with an inventory as above, I confirm that each of my hosts now has the `megaquiz` repo installed in its `/home/webuser/app` directory.

Copying a Configuration File

If deployment simply meant checking out some code, we'd be done by now. At minimum, though, most applications require configuration.

Let's add a couple of tasks which will copy a configuration file into each environment.

```
# 004-playbook.yml

- name: Megaquiz playbook
  hosts: myservers
  remote_user: webuser
  tasks:
   - name: clone
     ansible.builtin.git:
         repo: 'git@github.com:poppbook/megaquiz.git'
         dest: /home/webuser/app
         version: v1.0.0
```

```
  - name: Make sure destination dir exists
    ansible.builtin.file:
        path: /home/webuser/app/conf
        state: directory

  - name: copy
    ansible.builtin.copy:
        src: res/megaquiz1.ini
        dest: /home/webuser/app/conf/megaquiz.ini
```

I introduce two new modules here. `ansible.builtin.file` (https://docs.ansible.com/ansible/latest/collections/ansible/builtin/file_module.html) performs file operations. The `path` parameter specifies a location on the target host. When set to `directory`, the state argument causes any specified directories to be recursively created.

As you'd expect, `ansible.builtin.copy` will copy a file over from the control node to the target hosts. `src` specifies the source file and `dest` should contain the path on the host.

Now, assuming that I have a useful configuration file in `res/megaquiz1.ini`, it will appear in `/home/webuser/app/conf/megaquiz.ini`.

Some More on Variables

We're heading for a section on injecting values into a configuration file. In order to get there, though, we need to step back and cover some more on variables in Ansible.

Declaring Variables with `vars`

You have already seen the `register` task keyword, which declares a variable and assigns the result of the task's module to it. You can also declare variables with the `vars` playbook keyword.

Here's a new playbook which amends the *Hello, World* example to use a variable.

```
# 005-playbook.yml

- name: Ansible Says Something
  hosts: 127.0.0.1
  connection: local
```

```
  vars:
    person: Bob

  tasks:
  - name: Send output
    ansible.builtin.debug:
      msg: Hello, {{ person }}!
```

As you can see, I declare the `person` variable and set it to `Bob`. Notice, though, how I have incorporated the variable into the `ansible.builtin.debug` module's `msg` argument. If you're a Twig user, then that syntax might seem eerily familiar. In fact, this is an example of Jinja2 (https://jinja.palletsprojects.com), the templating language which inspired Twig and which is used by Ansible in various ways.

No prizes for guessing the output: here. We should be saying hello to Bob:

```
TASK [Send output] *************************************************
ok: [127.0.0.1] => {
    "msg": "Hello, Bob!"
}
```

Overriding Variables from the Command Line

The `vars` keyword provides a prominent place to declare a value that might be used in several places. It might seem that its usefulness is limited in that the variable's value is fixed. However, you can also use `-e` (or `--extra-vars`) flag to `ansible-playbook` to override the value of any variables you declare.

```
$ ansible-playbook -e "person=Harry" 005-playbook.yml
```

And now, Bob is Harry:

```
TASK [Send output] *************************************************
ok: [127.0.0.1] => {
    "msg": "Hello, Harry!"
}
```

Placing Variables in Files

Although you could conceivably add more and more `-e` flags to your command-line calls, that approach would soon become unwieldy. A neater approach might be to define a file for your variables.

Here's `vars/vars.yml`:

```
person: Mary
```

We'd expect to add a lot more elements to this list of course.

I can reference my new variables file with the `vars_files` playbook keyword:

```
# 006-playbook.yml


- name: Ansible Says Something
  hosts: 127.0.0.1
  vars_files: vars/vars.yml
  connection: local

  tasks:
  - name: Send output
    ansible.builtin.debug:
      msg: Hello, {{ person }}!
```

And now, we greet someone new:

```
TASK [Send output] *******************************************************
ok: [127.0.0.1] => {
    "msg": "Hello, Mary!"
}
```

Of course, I have come full circle now. My reference to the `vars/vars.yml` file is itself hard-coded. I can overcome that, though, by combining `vars` and `vars_files` like this:

```
# 006_1-playbook.yml

- name: Ansible Says Something
  hosts: 127.0.0.1
  vars:
    myvars: vars
```

```
  vars_files:
    vars/{{ myvars }}.yml
  connection: local

  tasks:
  - name: Send output
    ansible.builtin.debug:
      msg: Hello, {{ person }}!
```

This is functionally identical to the previous example. But by declaring the `myvars` variable and using it in the file path added to `vars_files`, I render it amenable to overriding. I can now change the variable file by invoking the playbook like this:

```
$ ansible-playbook --extra-vars "myvars=altvars" 006_1-playbook.yml
```

This will result in the inclusion of a file at `vars/altvars.yml`.

Note There is much more to variables. See the documentation at https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_variables.html. There is another useful trick to come later in this chapter too!

Interpolating Values into a File

When I copied the `megaquiz.ini` file over to my servers, I made no changes to it. In some circumstances, however, I might not want all values checked in to version control – either because they change according to context or because they are sensitive in nature.

I can use a module called `ansible.builtin.template` (https://docs.ansible.com/ansible/latest/collections/ansible/builtin/template_module.html) to combine a template file with a set of variables.

Here's a new variables file named `vars/vars2.yml`:

```
user: keisha
```

My new configuration file at `res/megaquiz2.ini` looks like this:

```
user={{ user }}
dbname=megaquiz
```

As you can see, I'm using Jinja2 again. When the final configuration file is generated, I will hope to find "keisha" assigned to the `user` element.

Here is my amended playbook:

```
# 007-playbook.yml

- name: Interpolate conf
  hosts: myservers
  vars_files: vars/vars2.yml
  remote_user: webuser
  tasks:

   # ...

   - name: copy / interpolate
     ansible.builtin.template:
         src: res/megaquiz2.ini
         dest: /home/webuser/app/conf/megaquiz.ini
```

This usage of `ansible.builtin.template` is syntactically similar to the `ansible.builtin.copy` example we have already seen. The functional difference, however, is that the module will apply any variables we have set to the template referenced in the `src` argument.

Let's run the playbook.

```
$ ansible-playbook -i inventories/example/hosts.ini  007-playbook.yml
```

The output tells me that all went well, so I log in to `webuser@192.168.1.98` and take a look at `/home/webuser/app/conf/megaquiz.ini`.

```
user=keisha
dbname=megaquiz
```

My interpolation appears to have worked!

Managing Secrets with Ansible Vault

Every project has its secrets – and it's not good policy to push such data to a version control repository, even a private one, without first encrypting it. Of course, you could store these values on your local machine or write them down on the back of an envelope. In the end though, the problem is the same – you either risk exposing your keys or losing them if you can't check them in to a repository of some kind.

Ansible vault (https://docs.ansible.com/ansible/latest/vault_guide/index.html) provides a good solution to this problem. It allows you to both encrypt and decrypt sensitive data, which can then be stored alongside your code in relative safety.

Let's work through an example. I will store a dummy API key in a file named `vars/secrets1.yml`:

```
quizkey: all_89876786
```

This starts out, of course, as a plain unencrypted file. The `ansible-vault` (https://docs.ansible.com/ansible/latest/cli/ansible-vault.html) command will change all that.

```
$ ansible-vault encrypt --vault-id myproject@prompt vars/secrets1.yml
```

Ansible prompts me for a password and confirms my encryption.

```
New vault password (myproject):
Confirm new vault password (myproject):
Encryption successful
```

The `encrypt` subcommand here should be self-explanatory. `--vault-id` is not technically required, but it is a good idea to use it, since it allows the user to manage multiple passwords for a system. So junior developers might have one level of access, while those that require it can be given the access they need to deploy to production.

The argument to `vault-id` consists of two parts split by a `@` symbol. The first part is the ID itself – the name I am using to label the password I want to associate with my encrypted file. The second part is the source. It indicates the source of the password. In this chapter, I'll always use `prompt` for this, but you could also specify a password file (containing only the password) or a script.

Having run `ansible-vault encrypt`, this is what my `vars/secrets1.yml` file looks like:

```
$ANSIBLE_VAULT;1.2;AES256;myproject
373063313232653533383965363865663235613633356532626261303231326433343065
66383430
386335393733613962633131646234646465383864336362Oa3038373732383231376 6
3165666438
653466303065336563386335363738316565643566643839663036346161613339 30
383738326432
363164323739336339Oa313366646438386332386135626166343138333538633234
303162616339
6164323331313739323761373730353161356661613261303933363131646263343 2
```

Of course, I will also need to be able to edit the file:

```
$ ansible-vault edit --vault-id myproject@prompt vars/secrets1.yml
```

I can now use my new password and the secrets file in conjunction with a playbook:

```
# 008-playbook.yml

- name: Whisper the secret
  hosts: 127.0.0.1
  vars_files: vars/secrets1.yml
  connection: local

  tasks:
  - name: Send output
    ansible.builtin.debug:
      msg: The secret key is {{ quizkey }}!
```

So, I did not need to do anything different in the play itself. I can treat a `secrets1.yml` just like any other variables file. Ansible is smart enough to understand that it's dealing with an encrypted file and acquire a password according to the source specification:

```
$ ansible-playbook --vault-id myproject@prompt 008-playbook.yml
```

Here's the relevant fragment of output:

```
TASK [Send output] *************************************************************
ok: [127.0.0.1] => {
    "msg": "The secret key is all_89876786!"
}
```

So, now I can deploy code from a Git repository. I can populate and copy a configuration file. I can manage secrets. Next, let's consolidate all that and even add a new feature or two.

Checking in on Megaquiz

As you know, the `--vault-id` flag to `ansible-vault` and other Ansible commands allows you to specify different passwords according to context. Let's build on that to create a deployment setup that supports three different project modes: *development*, *staging*, and *production*. Here is a potential directory structure:

```
megaquiz.yml

inventories/
    development/
        hosts.ini
    staging/
        hosts.ini
    production/
        hosts.ini

res/
    megaquiz.ini

vars/
    development/
        secrets.yml
    staging/
        secrets.yml
    production/
        secrets.yml
```

In addition to a playbook: `megaquiz.yml`, I create three directories, `inventories/` for inventory files, `res/` for a configuration file template, and `vars/` for encrypted variables files. In the cases of both `inventories/` and `vars/`, I create subdirectories for `development/ staging/` and `production/` in which to store the hosts or secrets files.

For this example, I'm reusing the same set of values in all the `hosts.ini` files. In a real-world environment, these would vary.

Here are the contents of one file:

```
[myservers]
192.168.1.98
192.168.1.7
192.168.1.82
```

The file at `res/megaquiz.ini` is a template that will combine hard-coded base values and variables – from both encrypted and unencrypted sources.

```
dbuser={{ dbuser }}
dbhost=db
quizkey={{ quizkey }}
```

Here is one of the secrets files – `vars/production/secrets.yml` – before encryption (or, later, in edit mode):

```
quizkey: prodkey_3333333
```

As a reminder, here's how I can encrypt the file:

```
$ ansible-vault encrypt --vault-id production@prompt vars/production/
secrets.yml
```

And, at last, here is the `megaquiz.yml` playbook:

```
# megaquiz.yml

- name: Deploy megaquiz
  hosts: myservers
  remote_user: webuser
  vars_files: "vars/{{ MODE }}/secrets.yml"
  tasks:
```

```
  - name: clone
    ansible.builtin.git:
        repo: 'git@github.com:poppbook/megaquiz.git'
        dest: /home/webuser/app
        version: v1.0.1

  - name: Make sure destination dir exists
    ansible.builtin.file:
        path: /home/webuser/app/conf
        state: directory

  - name: copy / interpolate
    ansible.builtin.template:
        src: res/megaquiz.ini
        dest: /home/webuser/app/conf/megaquiz.ini
```

This should be pretty familiar by now. Let's work through it anyway. The playbook contains a single play: "Deploy megaquiz." The `hosts` playbook keyboard specifies `myservers` – the group I use in all my `hosts.ini` files. `vars_files` specifies a path that incorporates a variable, `MODE`, to create a path to a secrets file.

You have seen each of the tasks before. In turn, they clone or checkout the megaquiz repository, create the `conf` directory, and generate a configuration file using the `ansible.builtin.template` module to combine variables with the template in `res/megaquiz.ini`.

This, of course, begs some questions. Where does `MODE` come from? Although the `quizkey` variable appears to be provided by `secrets.yml`, what provides `dbuser` which is also expected by the template?

The answer lies, once again, in the `-e` flag to `ansible-playbook`:

```
$ ansible-playbook --vault-id development@prompt \
    -e "MODE=development dbuser=bloop" \
    -i inventories/development/hosts.ini \
    megaquiz.yml
```

The most important variable here is `MODE`. As you have already seen, I could use this to generate a path to a variable file, but I chose instead to generate a second variable, `dbuser`. This is fine for my current purposes, but it won't scale well. I'd likely move `dbuser` to a file as soon as the need to support more variables emerges.

Before we wrap up by covering a final module, let's take in one more variable-related feature that Ansible supports.

Inventory Variables

So far, I have declared variables using the `var` and `var_files` playbook keywords. I have also used the `-e` or `--extra-vars` command-line flag. We have not yet encountered another common way to manage variables: inventories. This makes sense, of course, because a host or group of hosts will often correspond with a set of variable data.

You can add variables directly to your inventory file or, using a specially named directory, to your inventory environment. Here, for example, I set a variable for each host in my staging inventory:

```
[myservers]
192.168.1.98 dbuser=elbow
192.168.1.7  dbuser=hats
192.168.1.82 dbuser=flimflam
```

I am using INI format here. All I need to in this case is tack my variable key/value pairs onto the end of each host. The YAML equivalent is a little more verbose:

```
myservers:
  hosts:
    192.168.1.98:
      dbuser: elbow2
    192.168.1.7:
      dbuser: hats2
    192.168.1.82:
      dbuser: flimflam2
```

Thanks to this trick, I no longer need to specify the `dbuser` variable when I invoke `ansible-playbook`.

Of course, it's probably more useful (and less work) to set variables at the group level. This is also supported. Here is the INI version:

```
[myservers]
192.168.1.98
```

```
192.168.1.7
192.168.1.82

[myservers:vars]
dbuser=prodish
```

Here's the YAML equivalent:

```
myservers:
  vars:
     dbuser: groupish
  hosts:
    192.168.1.98:
    192.168.1.7:
    192.168.1.82:
```

Ansible supports a variation to these techniques. If you place a directory named `host_vars` in your current working directory or, more often, at the same level as your inventory files, then a YAML file named after the current host will be read, and any contained variables will be set. The same is true for inventory groups, except that the directory should be named `group_vars` and any contained files should be matched to groups and not hosts.

Let's implement this last feature.

Look again at the directory structure I created for my inventories:

```
inventories/
    development/
        hosts.ini
    staging/
        hosts.ini
    production/
        hosts.ini
```

As you know, an inventory can contain groups, which in turn break down into hosts. Ansible provides two implicit groups for every inventory: `all` which contains all the hosts in that inventory and `ungrouped` which contains a list of hosts which do not belong to any explicit groups. I am going to create variable files for the implicit `all` group in each of my inventories. Here is what my new directory structure will look like:

```
inventories/
    development/
        hosts.ini
        group_vars/
            all.yml
    staging/
        hosts.ini
        group_vars/
            all.yml
    production/
        hosts.ini
        group_vars/
            all.yml
```

Here is `inventories/production/group_vars/all.yml`:

```
MODE: production
dbuser: bob
```

Now, I can run the playbook and pick up the required variables without command-line variable arguments. My choice of inventory drives the selection of variables and the construction of a filepath which resolves to the correct secrets file.

```
$ ansible-playbook --vault-id development@prompt -i inventories/
development/hosts.ini megaquiz.yml
```

Of course, were this a real-world PHP project, I'd likely need to run Composer as part of a deploy.

The Composer Module

The `community.general.composer` module (https://docs.ansible.com/ansible/latest/collections/community/general/composer_module.html) supports most Composer operations.

Up until now, we have used built-in modules. This module is part of the `community.general` collection which you may need to install.

```
$ ansible-galaxy collection install community.general
```

Now, in a new task, I can use `community.general.composer` to run `composer install` on each of my projects (so long as Composer is installed on the remote hosts).

```
# megaquiz.yml

# ...

   - name: composer
     community.general.composer:
        command: install
        working_dir: /home/webuser/app
```

At this point, I've arrived at a pretty functional and extensible deployment environment. Of course, there's much more to Ansible. For example, a `composer install` can sometimes be a slow process. It would be nice to make the `composer` task optional. As you might expect, Ansible has us covered.

Conditionals

Ansible allows you to specify that a task is run only when a condition is met (https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_conditionals.html). You can achieve this with the `when` task keyword:

```
# megaquiz.yml

# ...

   - name: composer
     when: docomposer is defined
     community.general.composer:
        command: install
        working_dir: /home/webuser/app
```

The `when` keyword expects a test that uses the Jinja2 templating language (https://jinja.palletsprojects.com/en/3.1.x/templates/#tests). So, `is defined` will resolve to `True` if a variable has been set.

Since, by default, no `docomposer` variable is declared, the play will step over the `composer` task.

```
TASK [composer] ************************************************************
skipping: [192.168.1.98]
skipping: [192.168.1.7]
skipping: [192.168.1.82]
```

However, the task will be run if the `docomposer` variable is set:

```
$ ansible-playbook --vault-id development@prompt \
    -i inventories/development/hosts.ini \
    -e docomposer=yes \
    megaquiz.yml
```

Summary

Serious development rarely happens all in one place. A code base needs to be separated from its installation so that work in progress does not pollute production code that needs to remain functional at all times. Version control allows developers to check out a project and work on it in their own space. This requires that they should be able to configure the project easily for their environments. Then, in order to get new features into the world, the code needs to be functionally checked in a staging instance of a system. Finally, releases need to make their way to production. All this should be made as friction-free as possible.

In this chapter, I have covered some of the basics of Ansible, an automation engine, designed to do all of this and more. I described playbooks, plays, tasks, modules, and inventories – enough to encompass Ansible fundamentals. Then, I moved on to some practical examples, including getting code, adding configuration, managing secrets, and juggling variable data.

Of course, I have only scratched the surface of Ansible's capabilities. Nevertheless, once you are up and running with the tools and features described here, you'll find it easy to add new tasks and modules to your playbooks to achieve pretty much anything you might want to do, from provisioning an entire suite of servers to adding a configuration directive to an installed application.

CHAPTER 11

PHP on the Command Line

Once upon a time, the acronym *PHP* stood for *Personal Home Page*. Even after the language was renamed and the initials resolved to the recursive *PHP Hypertext Preprocessor*, the name remained overtly a webby affair. Nonetheless, PHP works very well on the command line. In fact, if you're sure that your target environment will furnish you with the PHP interpreter, it may be the best possible tool for your shell scripting tasks. Written in the right way, a PHP script created for the command line can do everything that a more traditional shell script can do, with all the power and familiarity of a fully featured object-oriented language and the rich library infrastructure provided by Composer.

This chapter will cover the following:

- *Scripts and autoloading*: Finding library files from your script
- *Acquiring arguments*: Getting essential application information from the command line
- *The whole shebang*: Running scripts without invoking PHP on the command line
- *Errors and interoperability*: Failing informatively
- *Building usage messages*: How to create self-documenting scripts
- *Parsing options*: Extracting options from positional arguments
- *Encapsulating output*: Managing warnings and primary output
- *Console input*: Prompting the user for information
- *Piped input*: Accepting data piped from other commands

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_11

- *Packaging scripts with Composer*: Adding your scripts to the `vendor/bin` directory
- *Distributing a runnable archive*: Generating a Phar package

Why the Command Line?

If you spend a lot of time building PHP systems, your focus is often, almost inevitably, somewhat Web-oriented. That said, any larger project quickly accumulates ancillary coding requirements. There are test fixtures to generate, server and database tasks to automate, scheduled housekeeping scripts, monitoring tools, integrity checkers. The list grows and grows with the complexity of your project.

You could, of course, use shell scripts for many of these tasks. And, sometimes, that's probably the right call. There are good reasons however to consider using PHP itself for command-line scripting jobs. First of all, if your context is a PHP project, then you almost certainly have the PHP interpreter available, so why not take advantage of its power and familiarity? You may feel a lot more comfortable writing more involved scripts in PHP than shell scripts. What's more, you get to leverage the power of Composer and its thousands of libraries. Even better, because those requirements are handled locally by Composer, you don't have to worry quite as much that a script's dependencies might go missing in some project contexts.

Another factor to bear in mind is that you can build PHP command-line utilities that are tightly integrated into your system. That means they will be able to leverage your environment to perform housekeeping tasks from within your application's context. You might, for example, need to prune a database once a month. You can create a PHP script to perform the task as a Cron script – called from the command line – but with operational access to application libraries and configuration.

Of course, by definition, PHP is less seamlessly integrated into the shell than a shell script. Even so, as I'll discuss at the end of the chapter, there are plenty of options (maybe even too many options) for invoking shell commands from within a PHP script where necessary. So, when it comes to creating command-line utilities, PHP might offer you the best of both worlds.

A Dummy Function

In Volume 1, I looked at controller actions in web applications. A controller action offers a thin layer that sits in front of an application's underlying functionality. We might think of a command-line script of any complexity in a similar way. While, for trivial use cases, you can throw the meat of your application directly into your script file, it's often much cleaner to treat the script itself as an orchestrator, managing input from the user, invoking methods in a deeper system, and presenting the result. To simulate that kind of structure, I'll begin here by mocking up some deeper logic for my script to invoke.

Here is a class containing a simple static method for converting string values to their uppercase equivalents. Let's imagine it's a very complicated and involved process!

```
namespace popp\ch24\batch01;

class Converter
{
    public static function upper(string $str): string
    {
        return strtoupper($str);
    }
}
```

I begin my structure, then, with a single class file:

```
src/
    Converter.php
```

I can test this with a couple of lines tucked temporarily under the class (or in a script that is already configured to find the `Converter` class).

```
$txt = file_get_contents(__DIR__ . "/res/test.txt");
print Converter::upper($txt);
```

Of course, even if I were to break those lines into a standalone script file, that hard-coded path to `test.txt` means that it would not be terribly useful.

We will need a separate script that can detect user input and can find the class file at `src/Converter.php`. So, let's get started.

Autoloading

I will create an initial file at `scripts/conv.php`. In order for it to find `src/Converter.php`, I have two choices. I could use `require_once` or a similar statement. That would work, but I will win more flexibility and reserve the ability to access more tools by using Composer. To that end, I'll need to generate and access an `autoload.php` file.

Note I cover Composer in Chapter 5.

I have two files at this point. An empty script in `scripts/conv.php` and my `Converter` class file at `src/Converter.php`:

```
scripts/
    conv.php
src/
    Converter.php
```

In order to connect them up, I'll use a `composer.json` file at the root of my script environment. I am using the `popp\ch24\batch01` namespace:

```
{
    "name": "popp/conv",
    "autoload": {
        "psr-4": {
            "popp\\ch24\\batch01\\": ["./scripts", "./src"]
        }
    }
}
```

This associates both the `scripts/` and `src/` directories with the same namespace. To activate this relationship, I must generate the autoload file:

```
$ composer dump-autoload
```

By running this, I generate a `vendor/` directory containing an `autoload.php` file.

Now, here's what my script environment might look like:

```
composer.json
scripts/
    conv.php
src/
    Converter.php
vendor/
    autoload.php
res/
    text.txt
```

I've also added a directory named `res` which contains a text file I'll be using throughout the chapter to put my code through its paces.

Now that's all in place, my components will be able to find one another. This will allow me to add third-party libraries to my script later and to expand the environment.

I will add a single `require_once` statement to `conv.php` which will configure autoloading.

```
namespace popp\ch24\batch01;

require_once(__DIR__ . "/../vendor/autoload.php");
```

Acquiring Arguments

Now that `conv.php` can access the `Converter` class, I'll create a minimal script that accepts command-line input.

```
namespace popp\ch24\batch01;

require_once(__DIR__ . "/../vendor/autoload.php");

if (count($argv) <= 1) {
    exit("not enough arguments\n");
}

$file = $argv[1];
if (! file_exists($file)) {
    exit("no file at '{$file}'\n");
}

$txt = file_get_contents($file);
print Converter::upper($txt);
```

When PHP is run from the command line, the `$argv` superglobal array is automatically populated. The first element will contain the path with which the script was called. Subsequent elements correspond to any given arguments. So, the script above is already pretty usable.

```
$ php scripts/conv.php res/test.txt
```

The `test.txt` file happens to contain an extract from the poem "Jab*b*erwocky" by Lewis Carroll. Here's the script's rather shouty output.

```
'TWAS BRILLIG, AND THE SLITHY TOVES
    DID GYRE AND GIMBLE IN THE WABE:
ALL MIMSY WERE THE BOROGOVES,
    AND THE MOME RATHS OUTGRABE.
```

The Shebang

Notice that, in calling `conv.php`, I actually invoked PHP and passed it the path to the script. I could have made it possible to call the script directly by changing its file permissions and including a "shebang" (otherwise known as a "hashbang") line. The shebang is so-called because of its leading `#!` characters. This line, which should be the first in the script, tells the shell to pass the current file to a particular command. In this case, of course, that command is `php`. Here's an amended version of `conv.php`.

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

if (count($argv) <= 1) {
    exit("not enough arguments\n");
}

// ...
```

The shebang here invokes `/usr/bin/env` with the argument `php`. The `env` command searches the `$PATH` environment variable for the given executable and attempts to call it. I might more directly have used `#!/usr/bin/php`. This would have worked for me in one of my environments but would have failed, however, in another because PHP is installed in a different location there. It's more portable, therefore, to use `env` which is likely to be found in a common location.

Now, I can change the file permissions on `conv.php` to make it executable and then run the script directly (or, at least, have it seem that way).

```
$ chmod +x ./scripts/conv.php
$ ./scripts/conv.php res/test.txt
```

Error Conditions

You might think that by invoking the `exit()` language function with a useful message, I have adequately handled error conditions. While that's probably true in functional terms, it's not true as far as script interoperability is concerned. Command-line scripts on Unix-like systems exit with a status code that can run from 0 to 255 (however, a PHP script should only ever use the range 0 to 254 because 255 is reserved by the executable). An exit code of 0 signals that no error occurred. An error code of 1 signals a general error. Although, by convention, other numbers have meanings, it's enough for our purposes to switch between zero and nonzero exit codes according to whether or not an error was encountered.

Note Prior to PHP 8.4, `exit()` was a language construct rather than a function.

The problem with calling `exit()` with a string, as I did above, is that, even though it announces a problem, it actually causes the script to terminate with an exit status of `0`. We can confirm that with a simple shell script:

```
$ php scripts/conv.php || echo "EXPECTING THIS TO BE TRIGGERED"
```

Because `conv.php` requires an argument and I have not provided one, the script will fail here.

If you place a well-behaved command into a shell conditional and it does not run successfully (i.e., if it renders a nonzero exit status), then it will be treated as false. So, ideally, the fragment above would have caused the trigger message to be output. In fact, though, all I saw is this:

```
not enough arguments
```

The `conv.php` script *explained* that it failed, but the shell saw it as a successfully completed command.

To improve matters, I can change the way that I run `exit()`. By passing it an integer rather than a string, I can cause my script to end with the given exit status. In doing that, I lose the ability to generate output. This is a good thing, however, because `exit()` sends string output to STDOUT and, in case of error, we should actually output to STDERR so that anyone using our scripts can handle output and error messages separately if they need to.

So, for my output, I can use `fputs()` with the STDERR constant:

```
if (count($argv) <= 1) {
    fputs(STDERR, "not enough arguments\n");
    exit(1);
}

$file = $argv[1];
if (! file_exists($file)) {
    fputs(STDERR, "no file at '{$file}'\n");
    exit(1);
}

$txt = file_get_contents($file);
print Converter::upper($txt);
```

Having made this change, when I run my script as above, the shell recognizes an error condition and outputs its trigger message.

```
not enough arguments
EXPECTING THIS TO BE TRIGGERED
```

Now, when someone needs to use `conv.php` in a shell script, they can stop execution if the command exits with an error condition – making it a good shell citizen!

Usage

Another element of scripting good practice is the *usage* message. This usually combines any available error feedback with information about how to interact with the script. For anything but the most basic script, I'll generally build a function to generate a usage message of some kind very early on.

Here's a `usage()` function for `conv.php`:

```
function usage(?string $msg = null): string
{
    $argv = $GLOBALS['argv'];
    $usage = "usage: $argv[0] <file>\n";
    $usage .= "\n";

    if (! is_null($msg)) {
        $usage .= "{$msg}\n\n";
    }

    return $usage;
}
```

Note Of course, by putting a function in a file which also causes side effects (by generating output), I breach the standard in PSR-1 which forbids this combination. I could fix this by moving `usage()` into a separate file. I covered PHP standards in Chapter 3.

Now, I can add some more information to any errors I generate.

```
if (count($argv) <= 1) {
    fputs(STDERR, usage("not enough arguments\n"));
    exit(1);
}
```

I'll call the latest iteration of my script with no arguments.

```
$ php scripts/conv3.php
```

And now, as well as an error, I get information about how to fix my call.

```
usage: scripts/conv3.php <file>

not enough arguments
```

Handling Arguments and Options

We've already established that the `$argv` array is populated with the script name and any further arguments. While this is useful, there is a pretty significant drawback. The process which populates `$argv` does not distinguish between positional arguments and options.

As a reminder, when I call `conv3.php` like this:

```
$ php scripts/conv3.php res/test.txt
```

This is what `$argv` looks like:

```
Array
(
    [0] => scripts/conv3.php
    [1] => res/test.txt
)
```

Here, `res/test.txt` is what is known as a positional argument.

So, what would happen if I wanted to support a set of arguments that look more like the following?

```
$ php scripts/conv3.php -v -o /tmp/out.txt -c res/test.txt
```

Unfortunately, this places us at the edge of quite an awkward set of problems.

Options

An option is a special argument that modifies the behavior of a command. You will usually expect options to be placed directly after the command path and before any positional arguments. If you're familiar with the Linux shell, you should recognize commands like

```
$ ls -l /tmp
```

This is a command, followed by a short option (`-l`) and a positional argument (the `/tmp` directory).

You can combine short options:

```
$ ls -aR /tmp
```

A command may also support long options (though there is no conventional guarantee that every long option has a short equivalent or vice versa). A long option is often (but not always) preceded by two dashes rather than one.

```
$ ls --all --recursive /tmp
```

Options can conventionally accept arguments in either form. These are equivalent, for example:

```
$ grep -C 2 hats /tmp/products/*
$ grep --context=2 hats /tmp/products/*
```

It's important to note that there is nothing fundamental about the form of options. One dash or two, the presence of an equals sign for arguments, it's all a matter of convention.

Given that PHP will not automatically manage options, this is something of a parsing adventure. Here is a call to a script called `play-args.php` which does nothing but pass `$argv` to `print_r()`:

```
$ php scripts/play-args.php -a -b with-arg --context=20 -de pos1 pos2
```

We can see that, according to the conventions already discussed, `-b` accepts an argument (`with-arg`). The `--context` option also expects or accepts an argument. Again, conventionally, `-d` and `-e` are separate options run together here. We can't know from the example alone whether `-e` accepts an argument (`pos1`) or whether `pos1` is the first positional argument.

PHP does not trouble itself with any of this; it just breaks down each individual block of text into elements:

```
Array
(
    [0] => scripts/play-args.php
    [1] => -a
    [2] => -b
    [3] => with-arg
    [4] => --context=20
    [5] => -de
    [6] => pos1
    [7] => pos2
)
```

As programmers, we're up to this challenge. Thankfully, we do not have to reinvent this particular wheel.

Introducing getopt

PHP provides a function named `getopt()` which offers a pretty usable solution to the problem of acquiring options and disambiguating them from positional arguments. The method requires a string argument which should contain a pattern describing the short options accepted. It also optionally accepts an array argument for long options and an index which will be populated with the start index of the positional arguments in `$argv` once parsing is complete.

Let's define the rules I discussed in the previous example:

```
$options = getopt("ab:de", ["long-a", "context:"], $index);
print_r($options);
```

So here, I define "ab:de" for my short options. This specifies that `getopt()` will recognize `-a`, `-b` (with a required argument), `-d`, and `-e`. For the second argument, I specify `--long-a` and `--context` (with a required argument).

As you can see, for both long and short arguments, adding a colon (`:`) modifier specifies a required argument. If I wanted to specify an *optional* argument in either case, I would use a double colon (`::`) instead.

With my option specifiers in place, the `$options` array will be neatly populated:

```
Array
(
    [a] =>
    [b] => with-arg
    [context] => 20
    [d] =>
    [e] =>
)
```

But what about my positional arguments `pos1` and `pos2`? Remember that the third argument to `getopt()` is populated with the index of the first positional argument after the options have been parsed. Because I know that, it's just a matter of deploying `array_slice()` to extract the argument portion of `$argv`.

```
$options = getopt("ab:de", ["long-a", "context:"], $index);
print_r($options);
$newargs = array_slice($argv, $index);
print_r($newargs);
```

Here is the output from that second `print_r()` which shows the `$newargs` array.

```
Array
(
    [0] => pos1
    [1] => pos2
)
```

The Problem with `getopt()`

In many cases, `getopt()` will be perfectly fine for your needs. It does not require that you load a third-party library, and it gets the job done. However, if you want to throw an error on invalid flags, you're out of luck. This might seem like a minor point, but imagine a mission-critical script that supports a `--dry-run` flag:

```
$ very-dangerous-deletion-script --dry-rune /home/mattz
```

If you're using `getopt()` here, it will happily ignore my `--dry-rune` typo and run the script in anger. This is quite a risky prospect.

The other, less important, problem is that `getopt()` does not help you to automate your usage message – and these can get gnarly as your list of options and arguments grows. As you might expect, there is a Composer package to address these issues. It's a lucky coincidence that I opted to use Composer for autoloading!

Using GetOpt.php

GetOpt.php is a well-featured object-oriented tool for parsing options. While the `getopt()` function is fine for quick and dirty scripts, you'll likely want GetOpt.php or a similar package for any complex or critical application.

Note In fact, GetOpt.php does a lot more than parse options. If you're planning to build a larger system, it's worth checking out the package's support for sub-commands in the documentation at http://getopt-php.github.io/getopt-php/.

You can install the package with Composer:

```
$ composer require ulrichsg/getopt-php
```

Once I have the package installed, I can incorporate some of its key components into my command-line script and instantiate a `GetOpt` object.

```
use GetOpt\GetOpt;
use GetOpt\Option;
use GetOpt\Argument;
use GetOpt\Operand;

$getopt = new GetOpt();
```

Here, aside from creating a `GetOpt` object, I set the script up to work with options, arguments, and operands. Of these, only the term *operand* should be new to you. That's the term the GetOpt.php package uses to describe positional arguments.

Now, I'm ready to define and apply my first option:

```
$getopt->addOption((new Option('a', 'long-a', GetOpt::NO_ARGUMENT))-
>setDescription('about the a flag'));
```

Thanks to good object-oriented design, you can almost read this as plain English. I create an `Option` object and pass it to `GetOpt::addOption()`. The `Option` constructor accepts a string argument representing a short option and another for an equivalent long option. You must provide at least one of these. The constructor also accepts an optional string value which should describe the option's argument requirement. If provided, this should match one of the `GetOpt` string constants concerned with argument requirements. I describe these in Table 11-1.

Table 11-1. *GetOpt Argument Constants*

Constant	Value	Description
`GetOpt\GetOpt::NO_ARGUMENT`	`':noArg'`	No argument (default)
`GetOpt\GetOpt::REQUIRED_ARGUMENT`	`':requiredArg'`	Argument required
`GetOpt\GetOpt::OPTIONAL_ARGUMENT`	`':optionalArg'`	Argument optional
`GetOpt\GetOpt::MULTIPLE_ARGUMENT`	`':multipleArg'`	Accepts multiple arguments (at least one)

The `Option` class supports a `setDescription()` method which requires a string argument. This will become more obviously useful when I come to generate a usage message. So, having created an `Option` object, I call `Option::setDescription()`.

So, I've configured support for the `a` flag alongside its `--long-a` equivalent. Next, I'll build in support for `-b`.

```
$getopt->addOption((new Option('b', null, GetOpt::REQUIRED_ARGUMENT))-
>setDescription('about the b flag'));
```

Note that as I've configured it here, the `-b` option has no long option equivalent. Next, I add support for the short `-d` and `-e` options.

```
$getopt->addOption((new Option('d', null, GetOpt::NO_ARGUMENT))-
>setDescription('about the d flag'));
$getopt->addOption((new Option('e', null, GetOpt::NO_ARGUMENT))-
>setDescription('about the e flag'));
```

Of course, requiring an argument (as I did for `-b`) is not the same as validating it. GetOpt.php supports that too. The `GetOpt::setArgument()` method accepts an `Argument` object which allows you to provide a validation function.

```
$contopt = new Option(null, 'context', GetOpt::REQUIRED_ARGUMENT);
$contopt->setDescription("context scope");
$contopt->setArgument(new Argument(null, 'is_numeric', 'number of lines'));
$getopt->addOption($contopt);
```

The Argument constructor accepts a default (scalar) value, a callable validation routine, and an argument name. All of these are optional. For validation, I have referenced the built-in is_numeric() function, but for more complex requirements (a valid file path, for example), you can easily provide a your own anonymous function which should accept the argument to test and resolve to a Boolean.

Once I have built up my options configuration, I'm ready to tell the GetOpt object to parse $argv.

```
$ret = $getopt->process();
```

I have not handled any error conditions here. In real-world code, I might wrap GetOpt::process() in a try/catch clause to handle one of several error conditions that the method might encounter during parsing (for an unknown option or an invalid argument, for example).

Once the parsing is complete, I can access an options array with GetOpt::getOptions() and an array of positional arguments with GetOpt::getOperands().

```
$options = $getopt->getOptions();
$newargs = $getopt->getOperands();

print_r($options);
print_r($newargs);
```

Independently of processing, I can also generate a usage message:

```
print $getopt->getHelpText();
```

Let's put our options parsing configuration through its paces:

```
$ php scripts/play-args-getopt.php -a -b with-arg --context=20 -de
pos1 pos2
```

Remember, my example includes three outputs: options, positional arguments, and a usage message. Here's the options output:

```
Array
(
    [a] => 1
    [long-a] => 1
```

```
    [b] => with-arg
    [d] => 1
    [e] => 1
    [context] => 20
)
```

Note here that since I defined them as equivalent, both the "a" and "long-a" elements are populated – even though I only provided `-a` on the command line. The classic `getopts()` function has no concept of equivalence, so it will only populate the option it encounters in the argument list.

Here's my list of positional arguments (*operands* in the GetOpts.php lexicon):

```
Array
(
    [0] => pos1
    [1] => pos2
)
```

Finally, a very labor-saving usage message:

```
Usage: scripts/play-args-getopt.php [options] [operands]

Options:
  -a, --long-a             about the a flag
  -b <arg>                 about the b flag
  -d                       about the d flag
  -e                       about the e flag
  --context <output file>  context scope
```

This last is particularly useful because it saves you having to keep a long string in line with your evolving options and arguments – like this:

```
$argv = $GLOBALS['argv'];
$usage  = "\n";
$usage .= sprintf("usage: %s [options] [args]\n", $argv[0]);
$usage .= "Options:\n";
$usage .= sprintf("%6s %-12s %-6s %s\n", "-a", "--long-a", "", "about the
a flag");
```

```
$usage .= sprintf("%6s %-12s %-6s %s\n", "-b", "", "<arg>", "about the
b flag");
$usage .= sprintf("%6s %-12s %-6s %s\n", "-d", "", "", "about the d flag");
$usage .= sprintf("%6s %-12s %-6s %s\n", "-e", "", "", "about the e flag");
$usage .= sprintf("%6s %-12s %-6s %s\n", "", "--context", "<num>", "context
scope");
```

Of course, it isn't hard to create something like this – or something even simpler using a heredoc string – but, in my experience, hand-maintained usage messages quickly fall out of alignment with configuration.

Enforcing Positional Arguments

GetOpt.php can manage operands (positional arguments) as well as options. In my initial example, I instantiated a `GetOpts` object with no constructor arguments. In fact, it can accept two arguments. The first is a string of the kind accepted by `getopts()` and the second is a settings array. I can use the settings array, along with `Operand` objects, to define and enforce positional arguments. Let's start again from the top:

```
$getopt = new GetOpt(null, [GetOpt::SETTING_STRICT_OPERANDS => true]);

// handle options as before ...

$getopt->addOperand(
    new Operand('pos1', Operand::REQUIRED)
);
$getopt->addOperand(
    new Operand('pos2', Operand::REQUIRED)
);
```

The `Operand` class constructor requires a name and accepts a requirement constant – one of `Operand::REQUIRED`, `Operand::OPTIONAL`, or `Operand::MULTIPLE`. I create two `Operand` objects and pass them to my `GetOpt` instance. Since I've decided to be strict about my positional arguments, I have configured `GetOpt` with `GetOpt::SETTING_STRICT_OPERANDS` set to `true`.

Now, if I fail to provide two positional arguments, `$getopt->process()` will throw an exception.

```
PHP Fatal error:  Uncaught GetOpt\ArgumentException\Missing: Operand pos1
is required in ...
```

Furthermore, the usage string will be automatically updated.

```
Usage: scripts/play-args-getopt-args.php [options] <pos1> <pos2>

Options:
  -a, --long-a             about the a flag
  -b <arg>                 about the b flag
  -d                       about the d flag
  -e                       about the e flag
  --context <output file>  context scope
```

Personally, I tend to enforce and validate my own positional arguments, since I prefer the flexibility that affords me. I might, for example, want to relax or alter my argument requirements when certain flags (like `--help`) are present. Nonetheless, operand management is a nice additional feature.

Handling Output

We saw in Volume 1 that, in web programming, it's a good idea to use `Response` objects to manage output. This makes components easier to test and allows for changes in implementation. In a quick command-line script, I would likely go ahead and use `print` or `fputs()`. But for a slightly larger project, I like to create a class to encapsulate output. This can then be passed to other objects in my system which should not be making decisions about output.

Here's a basic implementation:

```
class Output
{
    public int $verbosity = 0;
    private $handle = STDOUT;
```

```
    public function setFileMode(string $path, $destructive = true)
    {
        $mode = $destructive ? "w" : "a";
        $this->handle = fopen($path, $mode);
    }

    public function say(string $str): void
    {
        fputs($this->handle, $str);
    }

    public function warn(string $str): void
    {
        fputs(STDERR, $str);
    }

    public function debug(string $str): void
    {
        if ($this->verbosity > 0) {
            fputs(STDERR, $str);
        }
    }
}
```

This simple class offers three output methods: `say()`, `warn()`, and `debug()`. By default, as you might expect, `say()` outputs a string to `STDOUT` and `warn()` outputs to `STDERR`. The `debug()` method also sends output to `STDERR` but only if the `$verbosity` property is set to a nonzero value. An additional method, `setFileMode()`, accepts a path and alters the output handle so that `say()` will write output to a file.

Of course, a more complete implementation of the `Output` class would include more error checking and would allow for file mode to be disabled as well as set.

You will see `Output` in operation in future examples. In fact, we can get started with that right away.

Updating the Example Script

By way of consolidation, I'll apply what I've covered so far to the basic converter script.

I'll add two new options: `-h` to generate the usage message and `-o` to allow the user to specify an output file. Here's a version of the script that uses GetOpt.php:

```
use GetOpt\GetOpt;
use GetOpt\Option;
use GetOpt\Argument;
use GetOpt\ArgumentException;

// script globals
$output = new Output();
$getopt = new GetOpt();

// convenience functions
function croak(string $msg): void
{
    global $output, $getopt;
    $output->warn(usage($msg));
    exit(1);
}

function usage(?string $msg = null): string
{
    global $getopt;
    $usage = $getopt->getHelpText();
    if (! is_null($msg)) {
        $usage .= "\n{$msg}\n\n";
    }
    return $usage;
}

// configuration
$getopt->addOption((new Option('h', 'help', GetOpt::NO_ARGUMENT))-
>setDescription("this help message"));
$fileopt = (new Option('o', 'output', GetOpt::REQUIRED_ARGUMENT))-
>setDescription('output to file');
```

```
$fileopt->setArgument(new Argument(null, null, 'file'));
$getopt->addOption($fileopt);

// execution
try {
    $ret = $getopt->process();
} catch (ArgumentException $e) {
    croak($e->getMessage());
}

$options = $getopt->getOptions();
$newargs = $getopt->getOperands();

if (isset($options['help'])) {
    $output->say(usage());
    exit(0);
}

if (isset($options['o'])) {
    $output->setFileMode($options['o']);
}

if (! count($newargs)) {
    croak("not enough arguments");
}

$file = $newargs[0];
if (! file_exists($file)) {
    croak("no file at '{$file}'");
}

$txt = file_get_contents($file);
$output->say(Converter::upper($txt));
```

There's nothing genuinely new here. Note the `croak()` convenience function. This sends the usage output to `STDERR` via `Output::warn()` and exits the script with a nonzero status code (hard-coded to `1` here). The `Output` class is also useful for the `-o` flag. I don't have to do anything more complicated than call `Output::setFileMode()` if this option is set (although, once again, a little more error checking would not go amiss).

It's not hard to replicate this script without GetOpt.php. Here are the main points of difference:

```
function usage(?string $msg = null): string
{
    $argv = $GLOBALS['argv'];
    $usage = sprintf("usage: %s <file>\n", $argv[0]);
    $usage .= sprintf("%6s %-12s %-6s %s\n", "-h", "--help", "", "this help
    message");
    $usage .= sprintf("%6s %-12s %-6s %s\n", "-o", "--output", "", "output
    to a file");
    $usage .= "\n";
    if (! is_null($msg)) {
        $usage .= "{$msg}\n\n";
    }
    return $usage;
}

// configuration
$rest_index = null;
$options = getopt("ho:", ['help', 'output:'], $rest_index);
$newargs = array_slice($argv, $rest_index);
```

Everything else is broadly the same for this version of the script, except that unexpected options won't cause an error and, because `getopt()` does not have a concept of equivalence, I need to check for the long and short versions of any options that have both.

From now on, I will stick to using GetOpt.php.

Adding Verbose Mode

With this set up, we already have enough in place to add a `-v` option to enable verbose output.

```
$getopt->addOption((new Option('v', 'verbose', GetOpt::NO_ARGUMENT))-
>setDescription('verbose mode'));

// ...
```

```
if (isset($options['verbose'])) {
    $output->verbosity = 1;
}

// ...

// this will output only if the user has invoked with '-v'
$output->debug("beginning...\n");
```

So, if the user calls the script with `-v` or `--verbose`, the script sets the `Output::$verbosity` property to 1. Thereafter, all calls to `Output::debug()` will be sent to `STDERR`. As a reminder, here's `Output::debug()` again:

```
public function debug(string $str): void
{
    if ($this->verbosity > 0) {
        fputs(STDERR, $str);
    }
}
```

Let's give it a whirl (this time in a script named `conv_go2.php`):

```
$ php scripts/conv_go2.php -v -o /tmp/blah.txt res/test.txt
beginning...
reading file...
```

Because I specified `-v`, my debug messages are played. We don't see any other output because I set `-o` to `/tmp/blah.txt`. I can confirm that my script worked though by examining that file:

```
$ cat /tmp/blah.txt
'TWAS BRILLIG, AND THE SLITHY TOVES
    DID GYRE AND GIMBLE IN THE WABE:
ALL MIMSY WERE THE BOROGOVES,
    AND THE MOME RATHS OUTGRABE.

...
```

Prompted Input

We have dealt with options and positional arguments which are probably the most common forms of script input. Sometimes, though, you want to prompt the user from within a script. There are various ways of doing this, but I will opt for `readline()`:

```
$getopt->addOption((new Option('c', 'console', GetOpt::NO_ARGUMENT))-
>setDescription("console mode"));

// ...

if (isset($options['console'])) {
    $output->debug("reading from console...\n");
    while (($input = readline("I> ")) !== false) {
        $output->say(Converter::upper("{$input}\n"));
    }
} else {
    $output->debug("reading file...\n");
    if (! count($newargs)) {
        croak("not enough arguments");
    }
    $file = $newargs[0];
    if (! file_exists($file)) {
        croak("no file at '{$file}'");
    }
    $txt = file_get_contents($file);
    $output->say(Converter::upper($txt));
}
```

The `readline()` function accepts an optional prompt string and returns a line of user input (with the newline stripped). It will return `false` if there is nothing left to read. If the `-c` flag is set, the script enters console mode and prompts for user input. The script will continue until killed with Ctrl-C or Ctrl-D.

```
$ php ./scripts/conv_go2.php -c
I> The Owl and the Pussy-cat went to sea
THE OWL AND THE PUSSY-CAT WENT TO SEA
I> In a beautiful pea-green boat,
```

```
IN A BEAUTIFUL PEA-GREEN BOAT,
I>
```

Piped Input

One of the beauties of commands on Unix-like systems is the ability to chain them together. I might, for example, use `cat` to join and output SQL queries from two files and pipe the result into the `mariadb` command.

In order to make the converter script capable of reading piped content, I need a function to check whether such input is available.

```
function pipeThere(): bool
{
    $read = [STDIN];
    $write = [];
    $except = [];
    $timeout = 0;
    $streamCount = stream_select($read, $write, $except, $timeout);
    return (bool) $streamCount;
}

// ...

if (pipeThere()) {
    $output->debug("piped input...\n");
    while (!feof(STDIN)) {
        $line = fgets(STDIN);
        $output->say(Converter::upper($line));
    }
} elseif (isset($options['console'])) {
    $output->debug("reading from console...\n");

    // ...
```

```
} else {
    $output->debug("reading file...\n");

    // ...

}
```

The `pipeThere()` function makes use of the built-in `stream_select()` function to detect content waiting to be consumed from `STDIN`. `stream_select()` requires three arrays, one containing readable streams, another containing writeable streams, and a third containing high-priority streams to be checked. A fourth argument should contain a timeout value in seconds. If the timeout value is set to `0`, the function will test and return right away. The function returns an integer representing the number of streams that match the test or `false` if no streams become available for reading or writing. I am only interested in checking the status of `STDIN`, so I add that to the `$read` array and leave the others empty. I only want to perform a quick check so my timeout value is set to `0`.

I call `pipeThere()` at the start of script operation. If the script is offered piped input, `pipeThere()` will return `true`, and I accept input from `STDIN` rather than a file or from console input.

Let's try it out:

```
$ cat res/test.txt | php scripts/conv_go3.php
'TWAS BRILLIG, AND THE SLITHY TOVES
    DID GYRE AND GIMBLE IN THE WABE:
ALL MIMSY WERE THE BOROGOVES,
    AND THE MOME RATHS OUTGRABE.
```

Packaging Up

There are various options for distributing a script. The most obvious, of course, is Composer. This is particularly attractive in the case of my example because I am already using Composer to manage autoloading and to incorporate dependencies. In some circumstances, I may want to make it particularly easy for others to run my script. For such situations, I may choose to generate a binary. Let's look at both options.

Distribution with Composer

Given the ground covered in this chapter, this is what my project directory might look like at this point:

```
composer.json
scripts/
    conv.php
src/
    Converter.php
    Output.php
res/
    test.txt
```

I would like to make the `conv.php` script available to any project that uses mine. Of course, were I to create a Packagist repo named `popp/conv` my script *would* become available. But its location, `vendor/popp/conv/scripts/conv.php`, would not be particularly intuitive. I'd like `conv.php` to appear in a more accessible location such as `vendor/bin`. Users already working with a Composer-installed PHPUnit will be primed to expect a script there.

In order to identify `conv.php` as a script, I must specify a `bin` element in my `composer.json` file:

```
{
    "name": "popp/conv",
    "bin": ["scripts/conv.php"],
    "autoload": {
        "psr-4": {
            "popp\\ch24\\batch03\\": ["scripts/", "src/"]
        }
    },
    "require": {
        "composer-runtime-api": "^2.2",
        "ulrichsg/getopt-php": "^4.0"
    }
}
```

The only new element here is the `bin` element which tells Composer that it should treat `scripts/conv.php` as special. Slightly confusingly, this will have no effect on my project directory. Rather, it will change the file made available to a user who installs `popp/conv` (probably using `composer require`). Since I'm not actually ready to upload my project to Packagist (see Chapter 5 for more on that), how can I test my setup?

Luckily, I can create a test project to install my work in progress locally. My project directory is named `batch03`. I will create a test project in a sibling directory `batch04` containing a single `composer.json` file. Here it is:

```
{
    "repositories": [
        {
            "type": "path",
            "url": "../batch03",
            "options": {
                "symlink": true
            }
        }
    ],
    "require": {
        "popp/conv": "@dev"
    }
}
```

This sets up `popp/conv` as a requirement and, through the `repositories` element, specifies the `../batch03` directory as its location. Notice also the `options` subelement where I specify that I wish to use a symlink rather than a copy of the source directory. After I have run `composer update`, I should find that I have a `vendor/` directory in my client (`batch04/`) directory containing the `popp/conv` package alongside all the dependencies it specifies. Thanks to my `options` specification, the `conv/popp` project will be included as a symlink, which means I can continue developing locally without having to reinstall every time I make a change and wish to test from a client perspective!

Are you lost? Let's ground things with another snapshot of some key files and directories before continuing:

Here again is my project directory. This is where I develop my `popp/conv` package and my little command-line script.

```
batch03/
    composer.json
    scripts/
        conv.php
    src/
        Converter.php
        Output.php
```

I have set up the batch04 directory as a client environment for testing. I generate a vendor/ directory there thanks to configuration in the batch04/composer.json file. This references project conv/popp in the batch03/ directory which it includes as a symlink.

```
batch04
    composer.json
    vendor/
        bin/
            conv.php
        popp
            conv -> ../../../batch03/
        # many other directories
```

If I take a peek inside my new binary file at vendor/bin/conv.php, I can see that rather than copy over (or link to) conv.php the file that has been installed, there is a stub which simply references the scripts/conv.php file within the package.

```
namespace Composer;

$GLOBALS['_composer_bin_dir'] = __DIR__;
$GLOBALS['_composer_autoload_path'] = __DIR__ . '/..' . '/autoload.php';

return include __DIR__ . '/..' . '/popp/conv/scripts/conv.php';
```

Notice also that it creates a couple of handy global variables. Of particular use to me here is $_composer_autoload_path. Remember that I included autoload.php originally using a relative path: __DIR__ . "../vendor/autoload.php". This path will no longer be correct when my script is buried away under vendor/popp/conv/scripts/. Thanks to the stub script that Composer has created, however, I can incorporate $_composer_autoload_path into my require_once statement so that it is correct both when run locally and when run from within vendor/.

I'll head back to batch03 and update my scripts/conv.php file to take advantage of this:

```
namespace popp\ch24\batch03;

require_once($_composer_autoload_path ?? __DIR__ . "/../vendor/
autoload.php");
```

Now, I return to batch04/ where my test install lives and check that my script is sane:

```
$ ./vendor/bin/conv.php
Usage: ./vendor/bin/conv.php [options] [operands]

Options:
  -h, --help            this help message
  -v, --verbose         verbose mode
  -c, --console         console mode
  -o, --output <file>   output to file

not enough arguments
```

Creating a Phar

So, we have seen how we might create a package suitable for distribution to Packagist for inclusion in other projects. But that requires a certain amount of knowledge from script users. You might also want to distribute a binary that can just be downloaded and run. A Phar is just that. It is a bundle of resources combined into a single file (usually with the .phar suffix) that can be run either directly or via PHP. A Phar still requires the presence on the target system of a PHP interpreter, but, if that's in place, it's usually a pretty plug-n-play experience for an end user. In this book, I have discussed or demonstrated the Phar format in the context of phpDocumentor, PHPUnit, and Composer.

PHP itself provides a suite of tools for building your own Phar files. These are documented at https://www.php.net/manual/en/book.phar.php. If you have created your command-line script in a Composer context however, I recommend a tool named phar-composer which takes a Composer environment and bundles it up into Phar format.

You can install phar-composer via Composer, but the recommended approach is to acquire and run phar-composer itself as a Phar.

```
$ curl -JOL https://clue.engineering/phar-composer-latest.phar
$ mv phar-composer-*.phar phar-composer.phar
$ chmod 755 phar-composer.phar
$ ./phar-composer.phar --version
```

As already discussed, my version of the Composer project containing the `scripts/conv.php` command-line tool is in a directory named `batch03/`. I have downloaded composer-phar to a sibling directory at `batch04` which is where I'm based for this example. Before I run the tool, I must invoke `composer update` in `batch03/` in order to generate the project's `vendor/` directory and populate it with the latest dependencies.

```
$ cd ../batch03/
$ composer update
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
  - Locking ulrichsg/getopt-php (v4.0.3)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing ulrichsg/getopt-php (v4.0.3): Extracting archive
Generating autoload files
No security vulnerability advisories found.
```

Now, I can change back to my phar-composer directory and run a new command: `build`.

```
$ cd -
$ php -d phar.readonly=off  ./phar-composer.phar build ../batch03/
conv.phar
```

Why did I include that `-d` option? For security reasons, Phar writing is disabled in the `php.ini` configuration by default. Adding this option ensures the `readonly` directive is toggled off for the duration of the command's execution. Here's the output:

```
[1/1] Creating phar conv.phar
```

```
  - Adding main package "popp/conv"
  - Adding composer base files
  - Adding dependency "ulrichsg/getopt-php" from "vendor/ulrichsg/
    getopt-php/"
  - Setting main/stub
    Using referenced chmod 0755
    Applying chmod 0755

    OK - Creating conv.phar (132.7 KiB) completed after 0s
```

And that should be it! I have generated a file named `conv.phar` which I can share with my team members:

```
$ ./conv.phar -h
Usage: ./conv.phar [options] [operands]

Options:
  -h, --help            this help message
  -v, --verbose
  -c, --console         console mode
  -o, --output <file>
```

Executing Shell Commands

This chapter is less about calling shell commands from PHP than it is about creating utilities in PHP that can be called from the shell. In fact, calling external binaries, "shelling out" as it's sometimes dubbed, is often frowned upon. That's because the practice scales poorly in a web context and sets up dependencies that can be hard to manage. On the other hand, if you're creating a PHP script designed to run on the command line, perhaps to perform some useful task in your project, then shelling out might be just what you need. Such scripts are often quick and dirty utilities, creatures of the shell. It makes perfect sense to build them to invoke other commands. If they're designed for use during development or deployment, then performance and scalability is less likely to be a consideration than it would be in a web component.

PHP is a pragmatic language, and this book is for pragmatic programmers. So let's take a quick look at some of the options for shelling out.

All the examples in this section will work with a shell command stored in the same variable:

```
$command = "wc README.md";
```

The command `wc` calculates and outputs the number of lines, words, and characters in a given file. `wc README.md` outputs the following when run on the command line in my repository (once I've gotten around to creating a `README.md` file, of course):

```
120  511 3249 README.md
```

I'll begin with `passthru()`:

```
$nullOrFalse = passthru($command, $status);
// returns: null for success / false for failure
// command status: captured by $status variable reference
// output: raw passed to STDOUT (good for binary data)
```

The `passthru()` function accepts a command to run and an optional variable which, if provided, will be populated by the exit status of the command. It returns `null` if the given command has been provided or generates a `ValueError` if the command argument is empty. The output of the given command is not captured by `passthru()`. Instead, it is sent to `STDOUT`. Because it is not processed in any way, this command can be used in a web context for generating binary data.

Note Exit codes in Unix-like systems run from 0 (success) to 255. Any nonzero result is an error condition. On the command line, you can acquire the exit code of the last command run with the `$?` variable. The "success" of a program execution function is not related to the command's exit code. The function's return value addresses the successful running of a command; the execution code is the command's own assessment of its success.

Similar to `passthru()`, the `system()` command sends text to the browser line by line, making it better suited for text than binary output in a web context.

```
$nullOrFalse = system($command, $status);
// returns: last line of output (false on failure)
// command status: captured by $status variable reference
// output: text passed to STDOUT
```

`system()` requires a command argument and optionally accepts a variable to be populated with the exit status. The function returns the last line of the command's output.

If you need to capture a command's output into an array, then `exec()` is the function you want.

```
$lastline = exec($command, $output, $status);
// returns: last line of output (false on failure)
// command status: captured by $status variable reference
// output: added to $output array
```

Unsurprisingly, `exec()` requires a command argument. It optionally accepts two further arguments. The first of these it will populate with an array of output lines. The second, again, will contain the exit code. The method returns the last line of any output (or false on failure).

Finally, for a quick hack, backticks will get the job done.

```
$output = `$command`;  // same as `shell_exec()`
// returns: command output (null for error/no output, false if pipe can't
be established)
// command status: not available
// output: returned
```

This is as easy as anything. Simply wrap the command in backticks then assign, or otherwise work with, the output. You won't get the exit status, so this is not an option to go for if you're worried about a failure condition.

This is not an exhaustive list, but it should be enough for most needs. I summarize the program execution functions in Table 11-2.

Table 11-2. *Some Program Execution Functions*

Function or Language Construct	Output	Exit Status	Returns	Notes
`passthru($cmd, $status)`	Sent to STDOUT	`$status` populated	`null` on success	Best for passing through binary data
`system($cmd, $status)`	Sent to STDOUT	`$status` populated	`null` on success	Better suited to text output
`exec($cmd, $output, $status)`	`$output` populated as array	`$status` populated	Last line of output (or `false` on fail)	-
backticks (`)	Returns output	No status	Output	Identical to `shell_exec()`

Note Passing untrusted data to `exec()` and its cousins is a huge security risk. Do not pass along user-provided data to the shell without first checking and sanitizing it (or, preferably, don't pass it along at all).

Summary

In this chapter, I covered much of what you need to build a user-friendly command-line application in PHP. I looked at various ways of leveraging Composer for your script, from autoloading, to accessing tools, to making scripts available for other users. I covered command-line arguments and the thorny issue of parsing options. I examined gathering data piped in from other commands and via user prompts and suggested a strategy for encapsulating output. I used the composer-phar tool to generate a runnable Phar archive for distribution. Finally, I compared some techniques for "shelling out" – invoking shell commands from within your PHP scripts.

CHAPTER 12

Continuous Integration

In previous chapters, you've seen a plethora of tools that are designed to support a well-managed project. Unit testing, documentation, build, and version control are all fantastically useful. But tools, and testing in particular, can be bothersome.

Even if your tests only take a few minutes to run, you're often too focused on coding to run them. Not only that, but you have clients and colleagues waiting for new features. The temptation to keep on coding is always there. But bugs are much easier to fix close to the time they are hatched. That's because you're more likely to know which change caused the problem and are better able to come up with a quick fix.

In this chapter, I introduce continuous integration, a practice that automates the build and test process and brings together some of the tools and techniques you've encountered in recent chapters.

This chapter will cover these topics:

- Defining continuous integration
- Preparing a project for CI
- Looking at Jenkins: A CI server
- Checking out and testing your code with Jenkins
- Introducing GitHub Actions
- Checking out your code and running automated tests with GitHub Actions

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_12

What Is Continuous Integration?

In the bad old days, integration was something you did after you'd finished the fun stuff. It was also the stage at which you realized how much work you still had to do. Integration is the process by which all of the parts of your project are bundled up into packages that can be shipped and deployed. It's not glamorous, and it's actually hard.

Integration is tied up also with QA. You can't ship a product if it isn't fit for purpose. That means tests. Lots of tests. If you haven't been testing much prior to the integration stage, it probably also means nasty surprises. Lots of them.

You know from Chapter 7 that it's best practice to test early and often. Most of us accept that this is the ideal, but how often does the reality match up?

If you practice test-oriented development (a term I prefer to test-first development, because it better reflects the reality of most good projects I've seen), then the writing of tests is less hard than you might think. After all, you write tests as you code anyway. Every time you develop a component, you create code fragments, perhaps at the bottom of the class file, that instantiate objects and call their methods. If you gather up those throwaway scraps of code, written to put your component through its paces during development, you've got yourself a test case. Stick them into a class and add them to your suite.

Oddly, it's often the *running* of tests that people avoid. Over time, tests take longer to run. Failures related to known issues creep in, making it hard to diagnose new problems. Also, you suspect someone else committed code that broke the tests, and you don't have time to hold up your own work while you fix issues that are someone else's fault. Better to run a couple of tests related to your work than the whole suite.

Failing to run tests, and therefore to fix the problems that they could reveal, makes issues harder and harder to address. The biggest overhead in hunting for bugs is usually the diagnosis and not the cure. Very often, a fix can be applied in a matter of minutes, set against perhaps hours searching for the reason a test failed. If a test fails within minutes or hours of a commit, however, you're more likely to know where to look for the problem.

Software build suffers from a similar problem. If you don't install your project often, you're likely to find that, although everything runs fine on your development box, an installed instance falls over with an obscure error message. The longer you've gone between builds, the more obscure the reason for the failure will likely be to you.

It's often something simple: an undeclared dependency upon a library on your system or some class files you failed to check in. These are easy to fix if you're on hand. But what if a build failure occurs when you're out of the office? Whichever unlucky team

member gets the job of building and releasing the project won't know about your setup and won't have easy access to those missing files.

Integration issues are magnified by the number of people involved in a project. You may like and respect all of your team members, but we all know that they are much more likely than you are to leave tests unrun. And then, they commit a week's work of development at 4 p.m. on Friday, just as you're about to declare the project good to go for a release.

Continuous integration (CI) reduces some of these problems by automating the build and test process.

CI is both a set of practices and a set of tools. As a practice, it requires frequent commits of project code (at least daily). With each commit, tests should be run and any packages should be built. You've already seen some of the tools required for CI, in particular PHPUnit and Ansible. Individual tools aren't enough, however. A higher-level system is required to coordinate and automate the process.

Without the higher system, a CI server, it's likely that the practice of CI will simply succumb to our natural tendency to skip the chores. After all, we'd rather be coding.

Having a system like this in place offers clear benefits. First, your project gets built and tested frequently. That's the ultimate aim and benefit of CI. That it's automated, however, adds two further dimensions. The test and build happens in a different thread to that of development. It happens behind the scenes and doesn't require that you stop work to run tests. Also, as with testing, CI encourages good design. In order for it to be possible to automate installation in a remote location, you're forced to consider ease of installation from the start.

I don't know how many times I've come across projects where the installation procedure was an arcane secret known only to a few developers. "You mean you didn't set up the URL rewriting?" asks one old hand with barely concealed contempt. "Honestly, the rewrite rules *are* in the Wiki, you know. Just paste them into the Apache config file." Developing with CI in mind means making systems easier to test and install. This might mean a little more work up front, but it makes our lives easier down the line. Much easier.

So, to start off, I'm going to lay down some of that expensive groundwork. In fact, you'll find that in most of the sections to come, you've encountered these preparatory steps already.

Preparing a Project for CI

First of all, of course, I need a project to integrate continuously. Now, I'm a lazy soul, so I'll look for some code that comes with tests already written. The obvious candidate is the project I created in Chapter 7 to illustrate PHPUnit. I'm going to name it `userthing`, because it's a *thing*, with a `User` object in it.

First of all, here is a breakdown of my project directory:

```
test/
    util/
        ValidatorTest.php
    persist/
        UserStoreTest.php
src/
    util/
        Validator.php
    domain/
        User.php
    persist/
        UserStore.php
```

As you can see, I've tidied up the structure a little, adding some package directories. Within the code, I've supported the package structure with the use of namespaces. I've separated my test directory from the rest of my source code. I'll need to set up my autoload rules so that PHP can locate all the system's classes during testing. I'll add a `composer.json` file at the top level:

```
{
    "autoload": {
        "psr-4": {
            "userthing\\": ["src/", "test/"]
        }
    }
}
```

Now that I have a project, I should add it to a version control system.

CI and Version Control

Version control is essential for CI. A CI system needs to acquire the most recent version of a project without manual intervention (at least once things have been set up).

I will add `userthing` to GitHub.

Note I cover the process of adding a project to GitHub in Chapter 6.

Figure 12-1 shows my GitHub project some time after initial import. We will cover the included `.github/workflows` directory shortly.

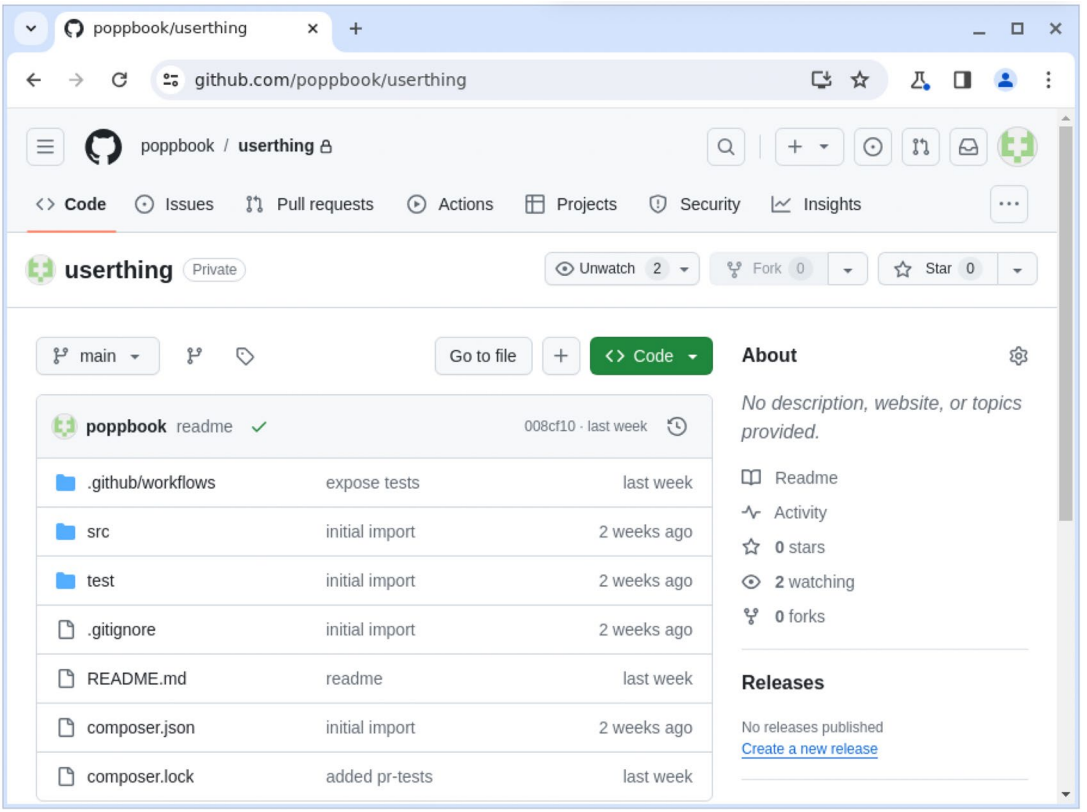

Figure 12-1. *The `userthing` GitHub repository*

Unit Tests

Unit tests are the key to continuous integration. It's no good successfully building a project that contains broken code. The easiest way to install it is via Composer.

```
$ composer require --dev phpunit/phpunit
```

This is the approach I'll take for my example. Because PHPUnit will be installed under the `vendor/` directory, my development directory will remain independent of the wider system.

Here's my complete `composer.json`:

```
{
    "require-dev": {
        "phpunit/phpunit": "^12.1"
    },
    "autoload": {
        "psr-4": {
            "userthing\\": ["src/", "test/"]
        }
    }
}
```

Because I ran `composer require`, I also created a `vendor/` directory which provides me with the `phpunit` script and `autoload.php` which handles autoloading for my application.

Note I covered Composer in Chapter 5.

In Chapter 7, I wrote tests for a version of the `userthing` code I'll be working with in this chapter too. Here, I run them once again (from within my project directory), to make sure my reorganization has not broken anything new:

```
$ ./vendor/bin/phpunit test/
```

After a few false starts and quick fixes, my test run confirms that things look relatively sane:

```
PHPUnit 11.3.1 by Sebastian Bergmann and contributors.
```

```
Runtime:       PHP 8.3.7

.......                                                   7 / 7 (100%)

Time: 00:00.021, Memory: 8.00 MB

OK (7 tests, 7 assertions)
```

I'll add, commit, and push my `composer.json` file along with any other changes. Remember that you should not commit the auto-generated `vendor/` directory. This should be generated afresh during the build process on a target environment.

Getting and Installing Jenkins

So, I have some useful tests that I can use to monitor the basic sanity of my project. Of course, left to myself I'd soon lose interest in running them. In fact, I'd probably revert to the old idea of an integration phase and pull out the tests only when I'm close to a release, by which time their effectiveness as early-warning systems will be irrelevant. What I need is a CI server to build my project and run the tests for me.

Jenkins (formerly named Hudson) is an open source continuous integration server. Although it is written in Java, Jenkins is easy to use with PHP tools. That's because the continuous integration server stands outside of the projects it builds, kicking off and monitoring the results of various commands. Jenkins also integrates well with PHP because it is designed to support plug-ins, and there is a highly active developer community working to extend the server's core functionality.

Note Why Jenkins? Jenkins is very easy to use and extend. It is well established, and it has an active user community. It's free and open source. Plug-ins that support integration with PHP (and that includes most build and test tools you might think of) are available. There are many CI server solutions out there, however. A previous version of this book focused on CruiseControl (http://cruisecontrol.sourceforge.net/), and this remains a good option. We will also shortly examine GitHub Actions which is also a compelling alternative.

Installing Jenkins

The Jenkins site provides good installation instructions at https://www.jenkins.io/doc/book/installing/.

I will opt for the Docker approach, which is a great way to get up and running without worrying too much about your host environment. Here's how you might run a Docker Jenkins container according to the documentation for the official image at https://github.com/jenkinsci/docker/blob/master/README.md:

```
$ docker run \
    -p 8080:8080 \
    -p 50000:50000 \
    --restart=on-failure \
    -v jenkins_home:/var/jenkins_home \
    jenkins/jenkins:lts-jdk17
```

That will get you a container running the Jenkins system on a Debian distribution. However, I will want to install and test the `userthing` application, which means I will need PHP.

Note Although, for the sake of simplicity, I will initially build and run tests within the main Jenkins container, it is a better idea to configure one or more remote Jenkins agents to perform builds under the control of a central node. I will cover this approach later in the chapter.

In this chapter, I'm assuming a basic knowledge of Docker. If you need a refresher, I cover Docker in Chapter 9.

Luckily, it's very easy to add PHP 8.3 to a Debian system. You can find documentation for a popular approach at https://deb.sury.org. I can use those instructions (specifically at https://packages.sury.org/php/README.txt) as the basis of a script which, combined with a `Dockerfile`, can be deployed to create a PHP-capable environment based on the official Jenkins Docker image. Here's the `Dockerfile`:

```
FROM jenkins/jenkins:lts-jdk17

USER root

COPY --chmod=0755 install-php .
RUN ./install-php

USER jenkins
```

Here's that referenced `install-php` script:

```
apt-get update
apt-get install -y curl lsb-release
curl -sSLo /usr/share/keyrings/deb.sury.org-php.gpg https://packages.sury.
org/php/apt.gpg
sh -c 'echo "deb [signed-by=/usr/share/keyrings/deb.sury.org-php.gpg]
https://packages.sury.org/php/ $(lsb_release -sc) main" > /etc/apt/sources.
list.d/php.list'
apt-get update
apt-get install -y php8.3 php8.3-cli composer php8.3-dom php8.3-simplexml
php8.3-mbstring
```

I build myself an image named `jenkins_php` like this:

```
$ docker build . --tag=jenkins_php
```

With that image in place, I can adapt the `docker run` call we've already seen to create a new, PHP-capable Jenkins container from the `jenkins_php` image:

```
$ docker network create jenkins
$ docker run \
    -p 8080:8080 \
    -p 50000:50000 \
    --restart=on-failure \
    -v jenkins_home:/var/jenkins_home \
    --network jenkins \
    jenkins_php
```

Before invoking `docker run`, I create a bridge network named `jenkins`, and I join the new container to it using the `--network` option. This becomes useful later when I need to connect an agent node to the same network. Also, note the `jenkins_home` volume.

This does not exist on my local system, so it will be created as a docker volume. This will persist across containers, so that I can tear down and reinstall my container without losing my Jenkins configuration. This Docker feature came in very handy during the writing of this chapter!

Here is an extract from the `docker run` command's output:

```
Jenkins initial setup is required. An admin user has been created and a
password generated.
Please use the following password to proceed to installation:

73e8df01b1594263a506aee84ff9630e

This may also be found at: /var/jenkins_home/secrets/initialAdminPassword
```

I have configured Jenkins run on the default port of 8080. When I fire up my browser and visit `http://localhost:8080/`, I see something like the screen in Figure 12-2.

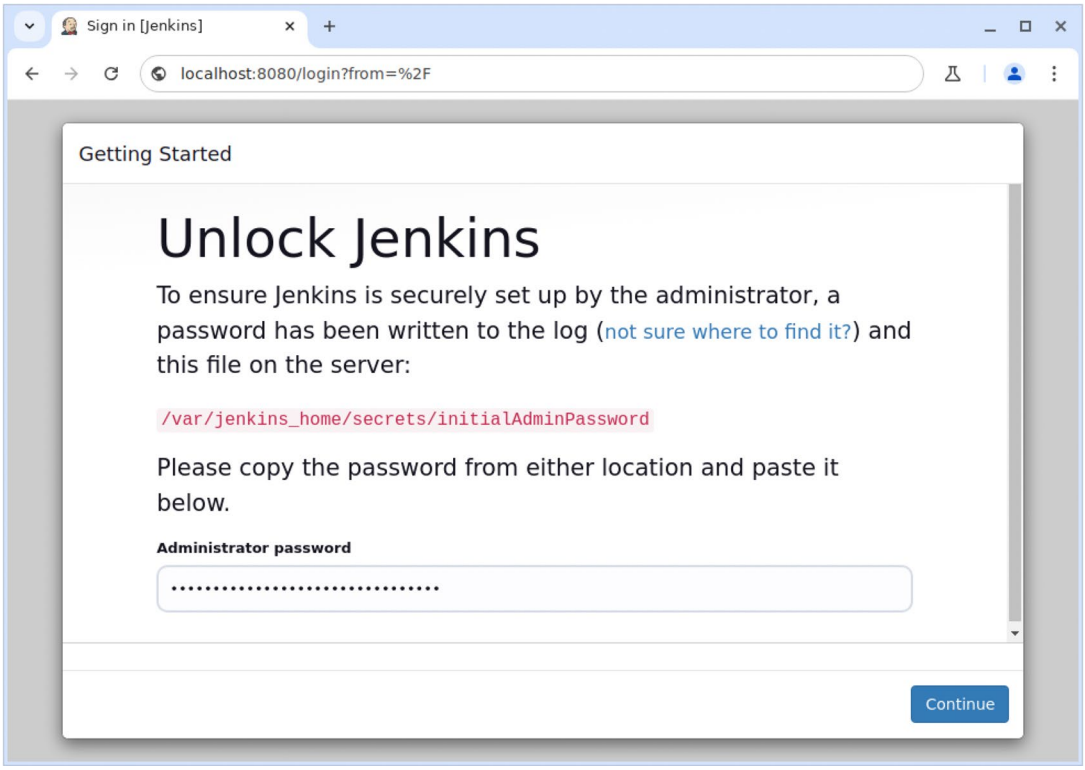

Figure 12-2. *The install screen*

The instructions in Figure 12-2 are pretty self-explanatory so I grab the password from the Docker command's output (yours would be different, of course) and enter it into the provided text field. Then, I'm presented with a choice: install with popular plug-ins or pick my own? I opt for the most popular plug-ins, which I know will get me support for Git, among other things. If you want a slim system, you might choose to select only those plug-ins you need. After that, it's time to create a username and password before finishing up installation.

Installing Jenkins Plug-ins

Jenkins is highly customizable, and although I'm going to stick to basic build and test in this chapter, you'll likely want to perform more operations on your code over time. From within the Jenkins web interface, you can check on what's available by clicking on *Manage Jenkins* in the main screen and then choosing *Plugins* from the panel. From there, you can check your *Installed plugins* or add new ones from *Available plugins*.

You can see the Jenkins plug-in page in Figure 12-3.

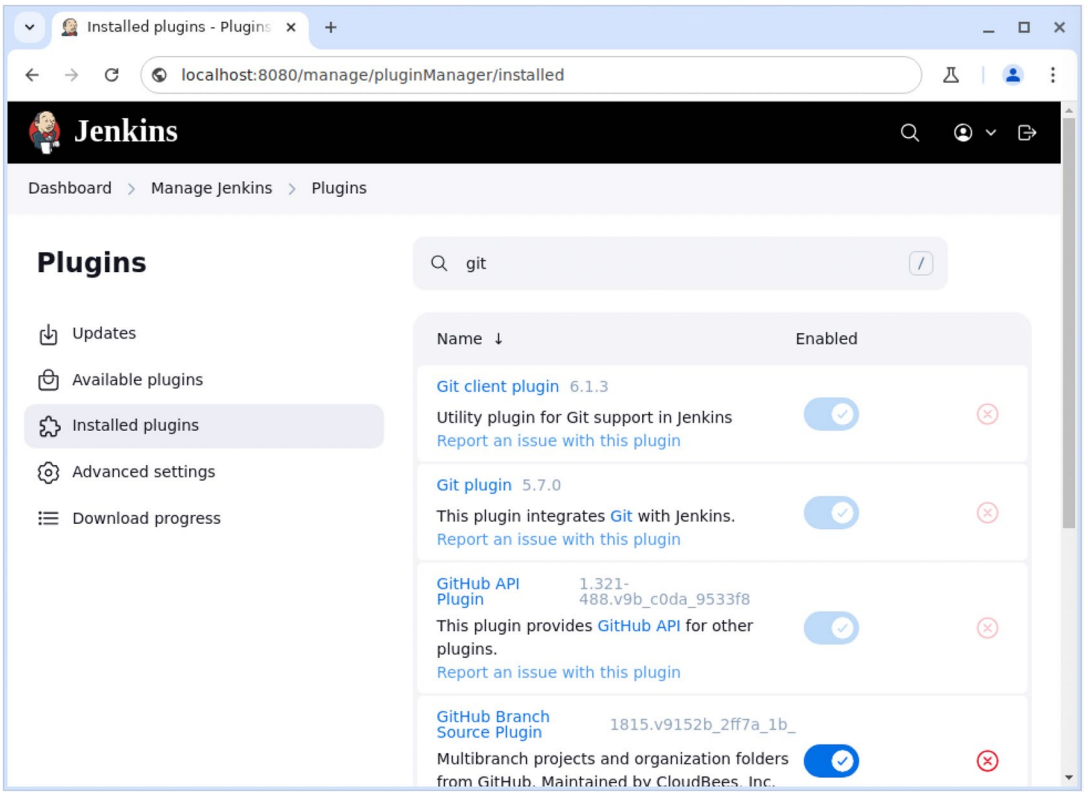

Figure 12-3. *The Jenkins plug-in screen*

Setting Up Git in Jenkins

Before I can use the Git plug-in, I need to ensure that I have access to a Git repository. In Chapter 6, I described the process of generating a public key in order to access a private Git repository. If you've worked through that process, you should already have access to a public and private key pair. As a reminder, here's how you can create a new pair and add the public key to the Git repository.

```
$ ssh-keygen -f ~/.ssh/jenkins_gitub_key
```

This command will create two files: `~/.ssh/jenkins_github_key` for the private key and `~/.ssh/jenkins_github_key.pub` for the public key.

Note Usually, when you run `ssh-keygen`, the output location is important. The `.ssh` directory is where the SSH daemon looks for its credentials. In this chapter, though, I will be acquiring the contents of these files and passing them directly to clients and servers. Neither the locations nor the names of the files really matter very much beyond that unless you also want to use the keys with your `ssh` command.

Now, I can add the public key to my `userthing` GitHub project as shown in Figure 12-4.

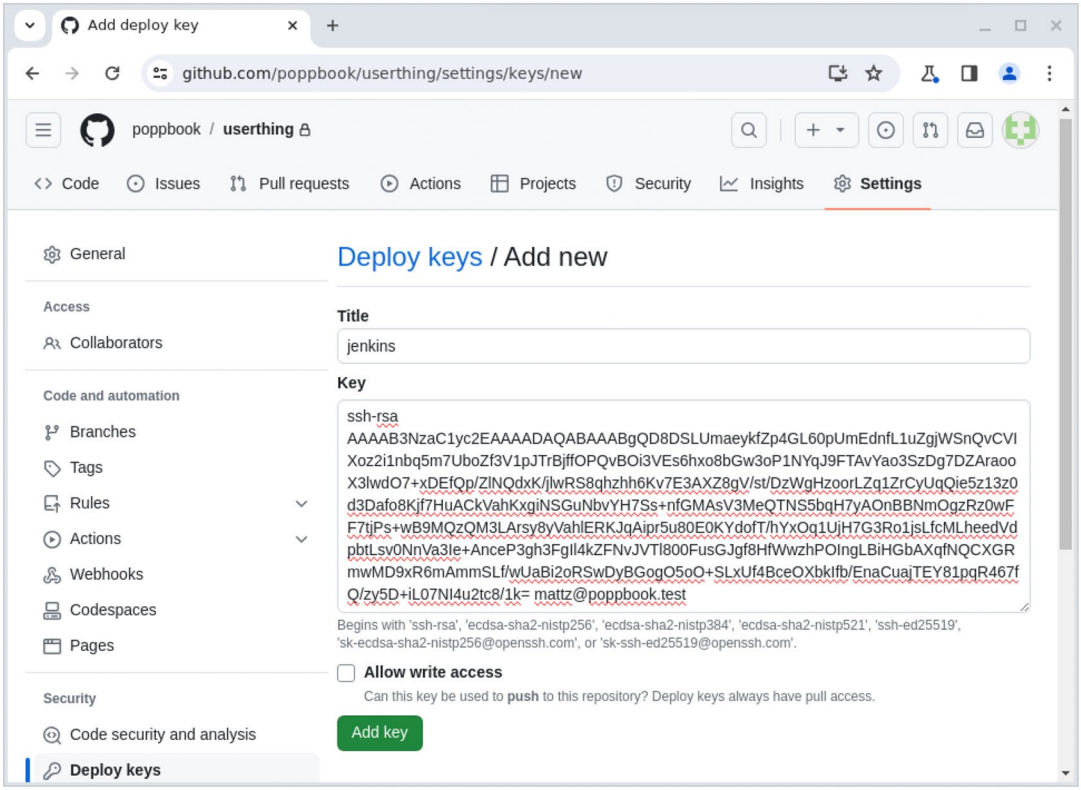

Figure 12-4. *Adding a deploy key to a GitHub project*

GitHub is ready for Jenkins, but I still need to apply the private key to my Jenkins environment. Before I do that, though, I can save myself much frustration by setting up my host key configuration. That is, the information that is usually stored in the `~/.ssh/known_hosts` file when you connect to a new server for the first time via SSH. Failing to have that information in place in Jenkins will likely prevent your Git connection from succeeding – the topic of many forum queries and Stack Overflow wails. You can grab the string you need from the command line by running:

```
$ ssh-keyscan -H github.com
```

Once I have that output, I head to *Manage Jenkins* from the main screen and choose *Security*. I scroll down to *Git Host Key Verification Configuration*. From the *Host Key Verification Strategy* drop-down, I choose *Manually provided keys*. I add the keyscan value there and save.

Time to set up a job with my private key.

From the Jenkins dashboard, I choose *Create Job*, name my job (*userthing* in my case), choose *Freestyle Project*, and hit *OK*. I am presented with a configuration screen. From the left-hand menu, I select *Source Code Management* and click the *Git* radio button to reveal the input fields I need. I add my *Repository URL* (this is the value you would use for a `git clone` operation). I ignore the immediate error message and choose *Add* under *Credentials*.

Note I could also have applied my Git credentials from the *Manage Jenkins* screen. I'll take that approach below when I add another key.

There is only one option on offer by default – *Jenkins*. Most of the fields then presented are self-explanatory. The essential requirements are *Kind* which I set to *SSH Username with private key*, *Username* which requires the GitHub user, *poppbook* in my case, and *Enter directly* for the *Private Key*. I select and paste in the private key I generated earlier. You can see a portion of that interface in Figure 12-5.

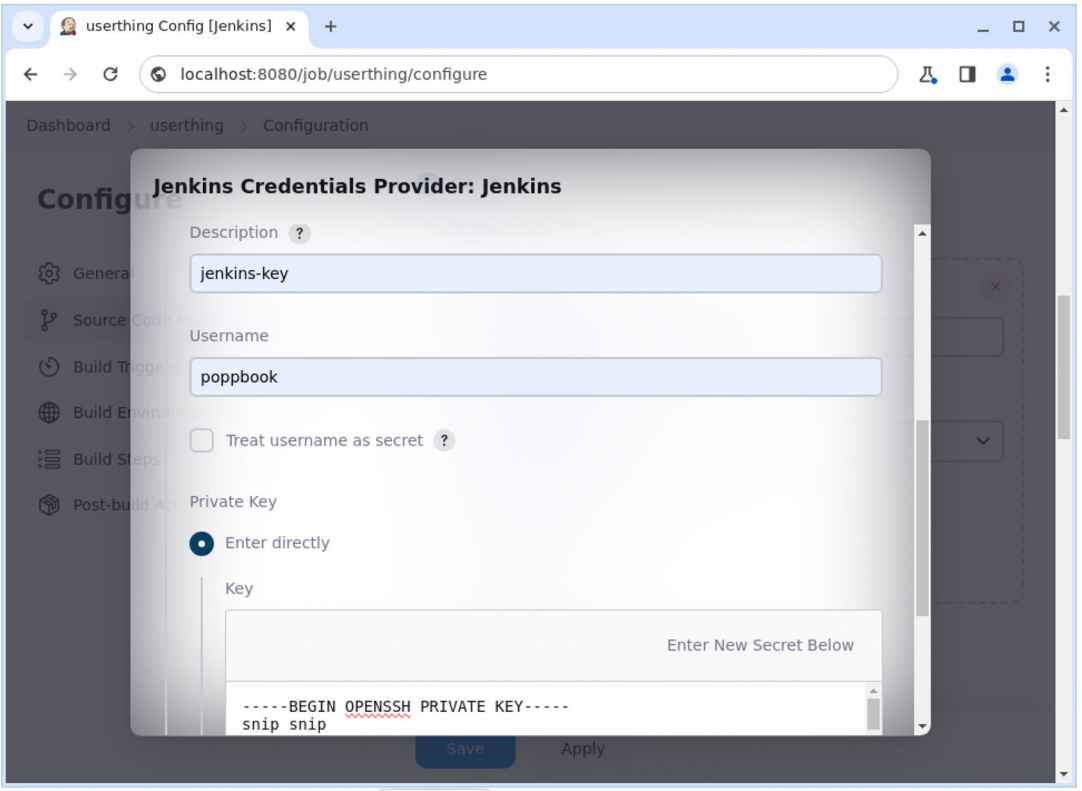

Figure 12-5. *Adding a secret key to the Git section in the Jenkins job configuration screen*

Once I've added that, the connection error message should disappear, and I can proceed with some confidence that Jenkins can access my Git project. Figure 12-6 shows that happy state.

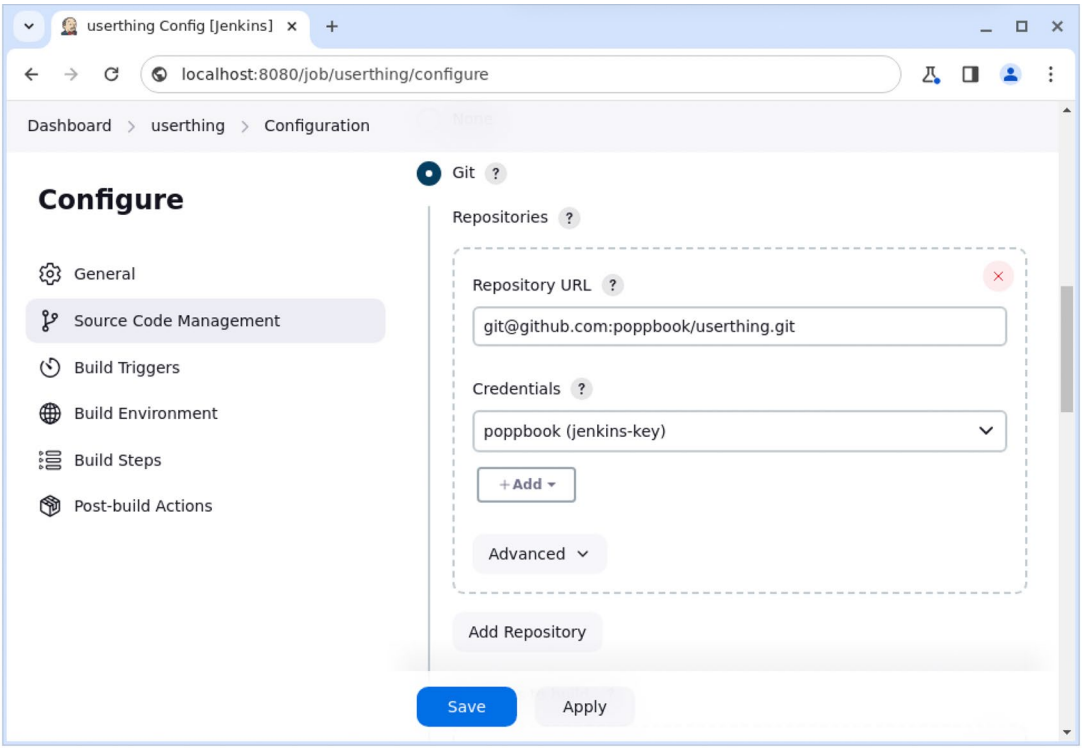

Figure 12-6. *Git configured for a Jenkins job*

Note There are also various plug-ins you can use to manage Git credentials including *SSH Agent*, *OAuth Credentials*, and *Kubernetes Credentials*.

One more step is needed before I can perform an initial build. I must choose the branch. I scroll a little further until I find *Branches to build*. I change the default to *main* (the branch I wish to build and test) and save my configuration for now.

Configuring Composer and PHPUnit

Of course, it's not enough to check out the code. I need to run `composer install` to prepare the build for testing. I have configured the Jenkins environment for this by creating the `jenkins_php` image. Remember, thanks to the `jenkins_home` volume which persists independently of containers, I can regenerate this image with new affordances as I need them, and any containers configured to use it will retain state across iterations.

Note In the longer term, it is not advisable to share your core Jenkins node with your build environment. A more mature configuration would make use of agents. I cover this below.

So, how can I add `composer install` to my build? Jenkins provides a *Build Steps* menu item in the job configuration screen. From there, as you can see in Figure 12-7, I can easily add *Execute shell* build steps. In fact, I configure Jenkins to run both Composer and PHPUnit.

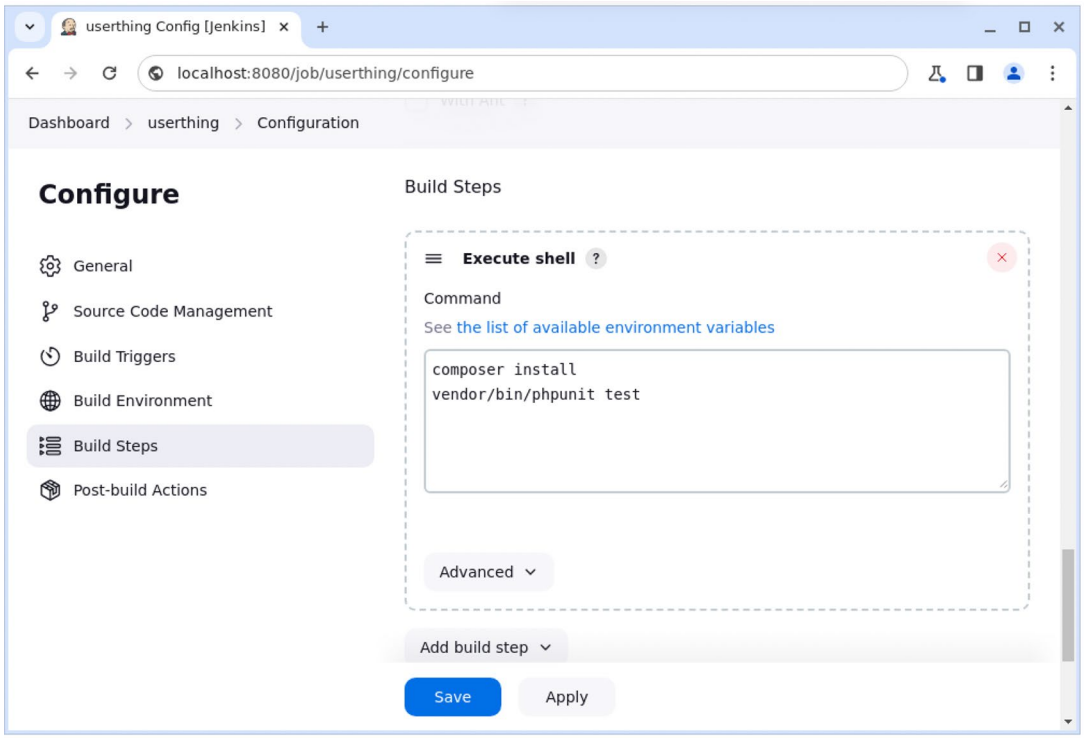

Figure 12-7. *An Execute shell build step*

Now, at last, I can confirm that I have a testable environment.

Running the First Build

In saving the configuration, I return to my job's main screen. From there, I choose *Build Now* to run the build process. This is the moment of truth! A link for the build should appear in the *Build History* area of the screen. I click that and then *Console Output* to confirm that the process went ahead as hoped. You can see some of the output in Figure 12-8.

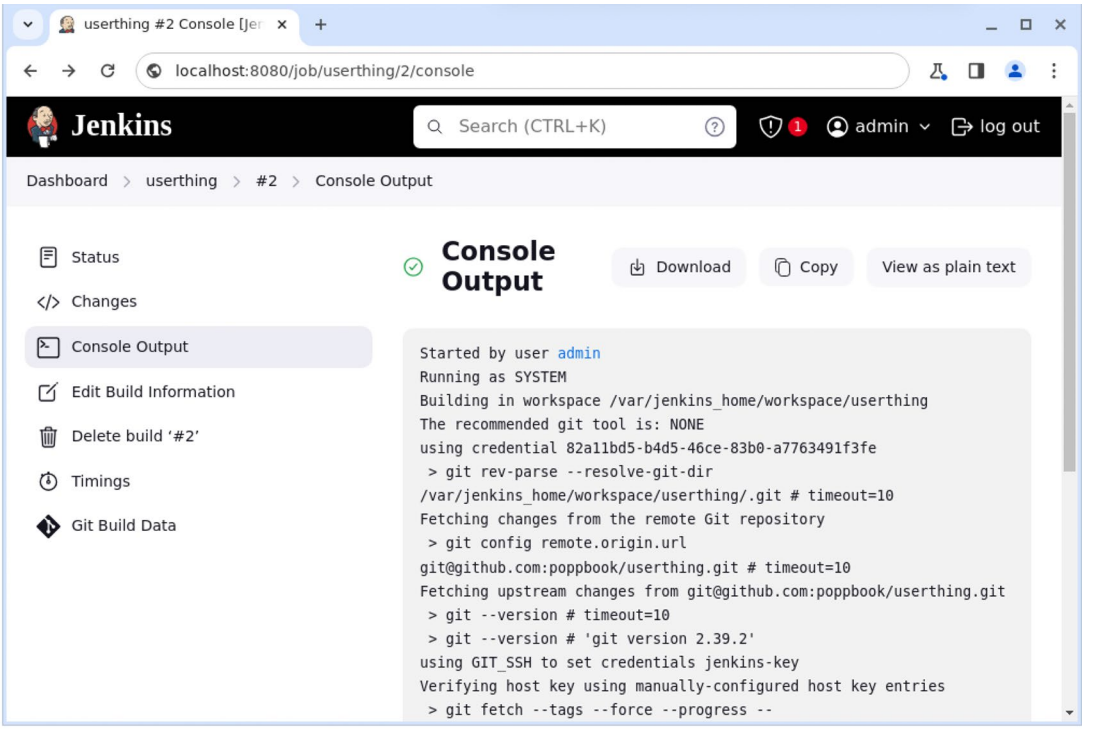

Figure 12-8. *Console output*

So, Jenkins has successfully checked the `userthing` code out from the Git server and run both `composer install` and the PHPUnit tests. Now that the basics are set up, it is easy enough to add additional features such as code coverage reports as needed.

Triggering Builds

All of this automation is almost useless if someone in your team must remember to kick off each build with a manual click. Naturally, Jenkins provides mechanisms by which builds can be automatically triggered.

You can set Jenkins to build, or to poll the version control repository, at specified intervals. These can be set using cron format, which provides fine, although somewhat arcane, control over scheduling. Luckily, Jenkins provides good online help for the format, and there are simple aliases if you don't require precision scheduling. The aliases include `@hourly`, `@midnight`, `@daily`, `@weekly`, and `@monthly`. In Figure 12-9, I configure the build to run once daily, or every time the repository changes, based upon a poll for changes that should take place once an hour.

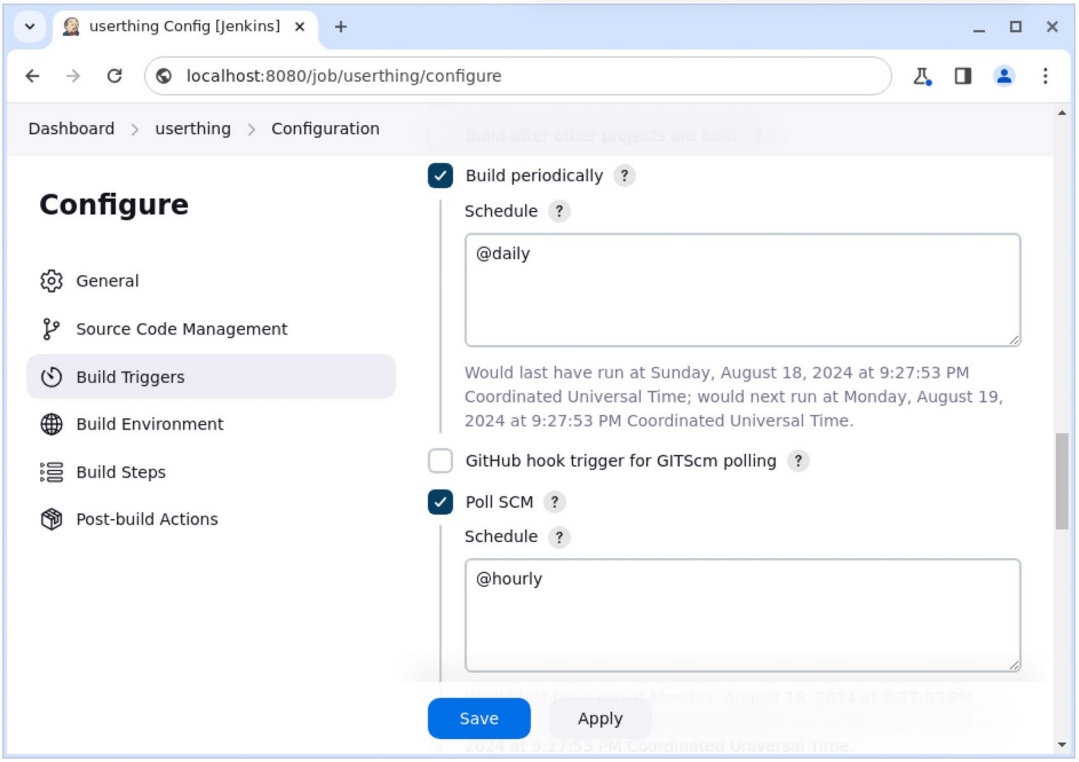

Figure 12-9. *Scheduled builds and SCM polling*

Polling is expensive and scheduling is crude. Luckily, there is a smarter way of doing things. I can set up a webhook in my GitHub configuration area to notify Jenkins when a push (or any other important event) occurs. You can see that configuration in Figure 12-10.

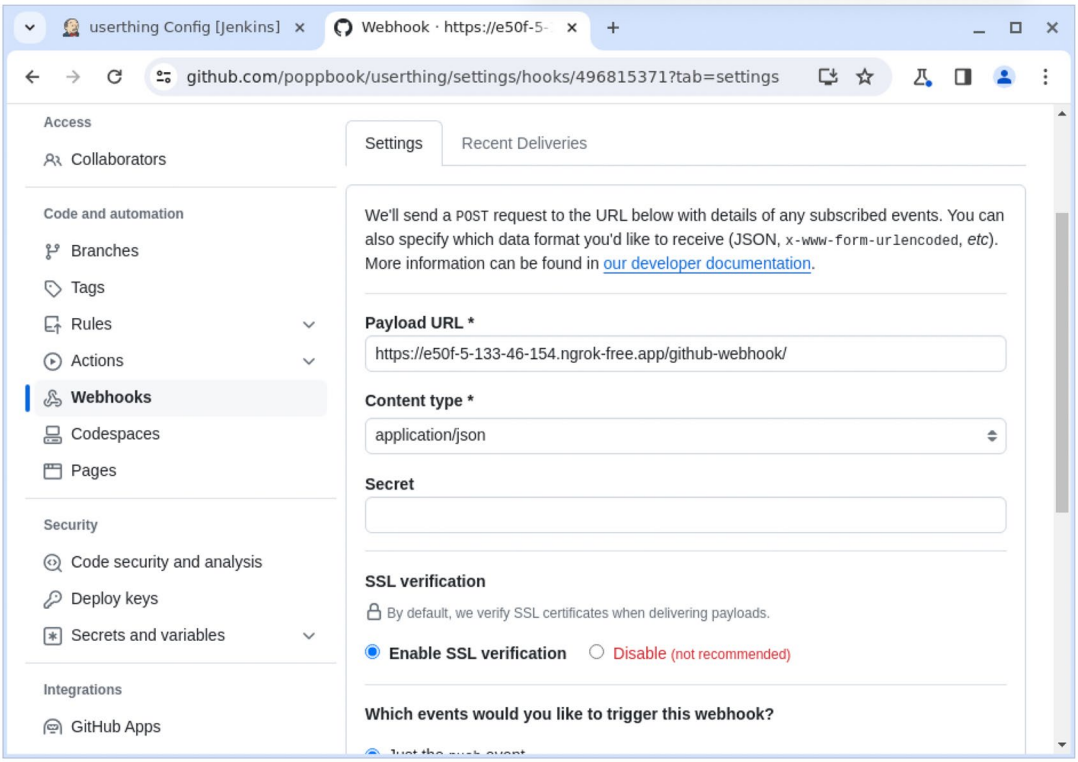

Figure 12-10. *Setting up a webhook in GitHub*

Because my local environment is not publicly available, I have used an API gateway system named Ngrok (`https://ngrok.com/`) to forward incoming requests to my Jenkins host at `http://localhost:8080`. Note that the trailing slash on `/github-webhook/` is required.

Note Ngrok installs a tiny app on your local system which establishes a connection to a remote server. This server listens on a custom, publicly available, URL which can be reached by external services (like GitHub or Bitbucket). When a webhook request (for example) is sent to the public URL, the payload is passed along to the listener on your system. This then invokes a configured local URL. It is a great workaround for development systems that need to be contacted occasionally by external services but which should not become public-facing. You can find installation instructions at `https://ngrok.com/docs/getting-started/`.

I have configured GitHub to send an `application/json` notification to the `/github-webhook/` endpoint.

In my Jenkins job configuration, I swap out the clumsy *Build periodically* and *Poll SCM* options, replacing them with *GitHub hook trigger for GITScm polling* which makes this `/github-webhook/` URL available. This option also has Jenkins poll GitHub but only in response to a webhook notification. This means that builds will be triggered only as needed and without the need for scheduled polling. Figure 12-11 shows this configuration.

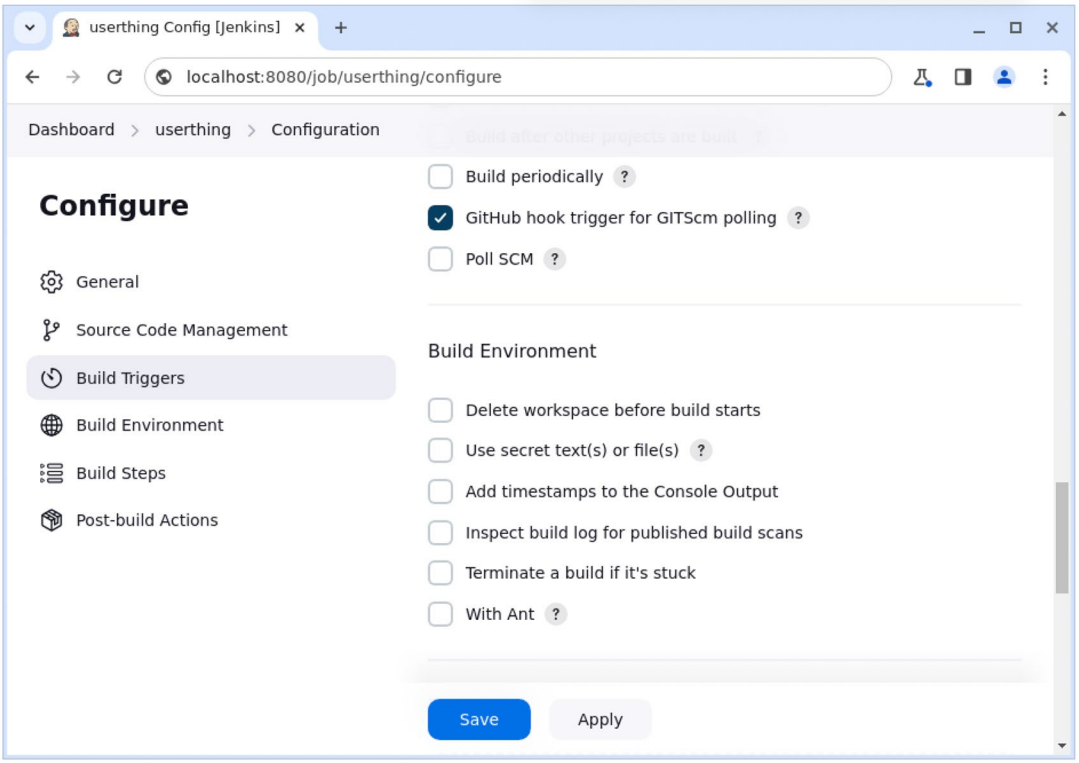

Figure 12-11. *Webhook trigger for GitScm polling*

I can now confirm my trigger by making a trivial change to my `userthing` branch, committing it, and then pushing to GitHub. Within seconds, I see a new build commencing on my Jenkins dashboard!

A Jenkins Agent

You may have spotted a huge problem with the architecture I have stitched together so far. I created an environment which will happily build my PHP project. That's because I am using a custom image that is based on the `jenkins/jenkins:lts-jdk17` image. That works nicely because the image includes PHP 8.3. But what if our team also managed a project which absolutely had to run on PHP 8.2? My clever solution won't stretch very far as I add different projects with wildly differing or even mutually exclusive requirements.

This is where Jenkins agents come in. With this model, the main Jenkins system acts as a controller node managing any number of remote agents, each of which can be configured to run jobs. This fixes my conflicting requirements problem. It also distributes the work that Jenkins has to perform, so that resources can be better managed across multiple servers.

Creating a PHP-Capable Agent Image

Once again, I'm going to take advantage of Docker for my example. I'll begin with a very simple `Dockerfile` (which I'll place in its own directory):

```
FROM jenkins/ssh-agent:jdk17

COPY --chmod=0755 install-php .
RUN ./install-php
```

You have already seen the `install-php` script. It will simply deploy PHP 8.3 and Composer into a new container based on the `jenkins/ssh-agent:jdk17` image. I will place a copy of this script in the directory containing my new `Dockerfile` so that it will be accessible.

Now, I can build myself a PHP-capable Jenkins agent named `php-agent`:

```
$ docker build . -t php-agent
```

That's it for build! But I still need to manage the communication between the main Jenkins node and the new agent.

Another Key Pair

Before I can usefully start the agent, I must create another key.

```
$ ssh-keygen -f ~/.ssh/jenkins_agent_key
```

As before, this command will create two files: `~/.ssh/jenkins_agent_key` for the private key and `~/.ssh/jenkins_agent_key.pub` for the public key. Note that the name of the key is not important. The generated files do not even have to remain under `~/.ssh/`. We will be assigning the keys to configuration in both the main Jenkins node (the private key will go there) and the agent (it will get the public key as an environment variable). This will allow the main node to manage the agent.

So, I must store the private key in Jenkins. From the main screen, I click *Manage Jenkins* and select *Credentials*, then *(Global)*. That takes me to the global credentials management screen. From here, I can click *Add Credentials*. I am presented a screen that may be familiar. I negotiated a version of the same form when I installed my GitHub key. This time, I add values as specified in Table 12-1. Then, I paste in the private key I generated to `~/.ssh/jenkins_agent_key`, and then I click *Create*.

Table 12-1. *Fields for Adding Global Key-Based Credentials*

Field	Value	Required?
Kind	`SSH Username with password`	Yes
Scope	`Global`	Yes (default)
ID	`Jenkins`	No
Description	`Jenkins key`	No
Username	`jenkins`	Yes (important)

Jenkins should now be ready to talk to the agent I configured in the previous section. Mind you, it does not yet exist yet. I need to create the container.

Running the Agent

Remember that I have an image tagged `php-agent`. I need to create a container based on that. The built-in node already has my `jenkins_agent` private key. The new container will need the corresponding public key provided as an environment variable. Because that's verbose, I'll create a tiny script to grab it and incorporate it into the `docker run` invocation:

```
MYPUB=`cat ~/.ssh/jenkins_agent_key.pub`
docker run -d --rm \
    --name=agent1 \
    -p 2200:22 \
    -e "JENKINS_AGENT_SSH_PUBKEY=${MYPUB}" \
    --network jenkins \
    php-agent
```

So, I create a new container based on the `php-agent` image. I name it `agent1`. Here is where my earlier creation of the `jenkins` network comes in handy. I use the `--network` option to join `agent1` to it. Now, the container will be accessible on the network as `agent1`. I pass the container the public key in the form of an environment variable: `JENKINS_AGENT_SSH_PUBKEY`. I map the external port 2200 to the internal ssh port (22).

Of course, at some point, it's likely that you will host your agent and controller nodes on different machines. At this point, you can dispense with the Jenkins bridge network. I covered some of the basics of Docker networking in Chapter 9.

Configuring Jenkins to Speak to the Agent

Once that's running, I have two CI containers in operation – the Jenkins built-in node and a PHP-capable agent. Let's see if `agent1` is accessible.

From the main screen, I head to *Manage Jenkins* and select *Nodes*. Once at the Nodes screen, I can click *New Node*. I name it `agent1` (this name matches the name of my Docker container for convenience – but the match is not necessary here) and select the only *Type* option available: *Permanent Agent* (as shown in Figure 12-12).

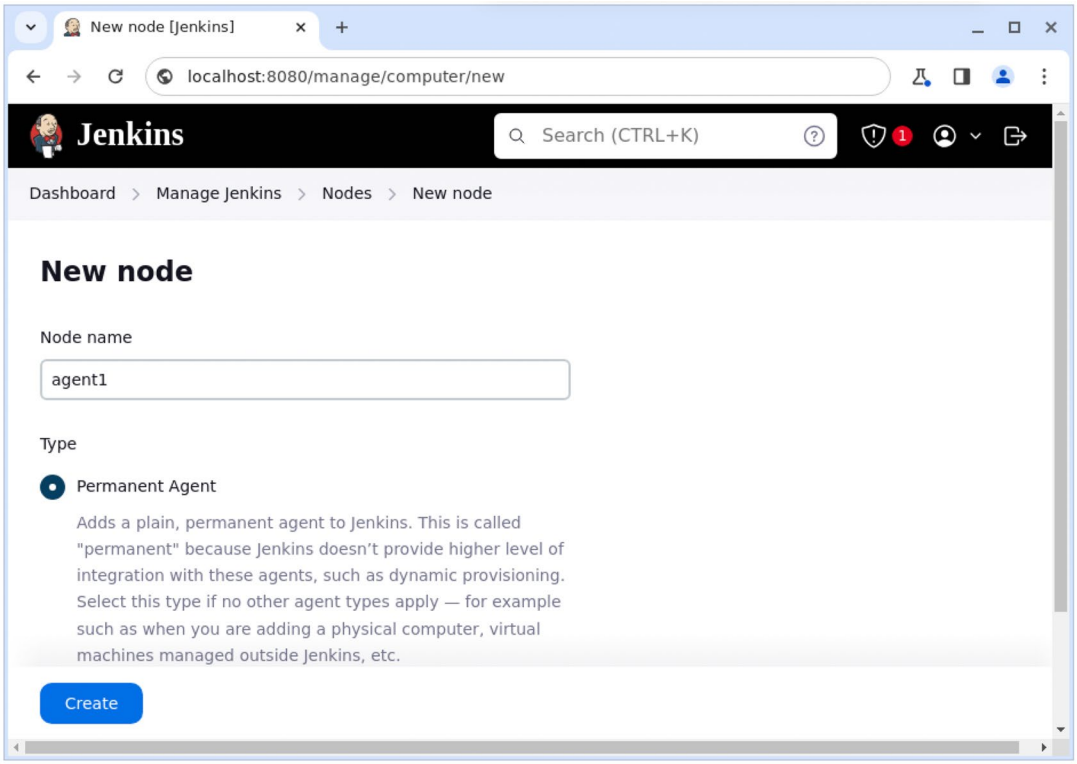

Figure 12-12. *Initial creation of an agent in Jenkins*

Then, I click *Create*. Here's where the real configuration happens. Table 12-2 shows the values I set. You should be able to leave blank or accept the defaults for any fields not specified in the table.

Table 12-2. *Fields for Associating an Agent with Jenkins*

Field	Value	Required?
Name	`agent1` (or your own option)	Yes
Description	Anything you want	No
Remote root directory	`/home/jenkins/`	Yes
Launch method	`Launch agents via SSH`	Yes
Host	`agent1` (the name here is important – it should be the container name if using a bridge network as in this example, otherwise the host on which the agent is running)	Yes
Credentials	Choose your key from the drop-down	Yes
Host Key Verification Strategy	`Manually trusted key Verification Strategy`	Yes

Once I have filled out the fields specified in Table 12-2, I click *Save* to create my agent configuration, and I'm sent back to the Nodes screen. I click *agent1* and arrive at the new agent's management screen. From there, if all is well, I can connect to the agent. I check on the progress of that by viewing the *Log* as shown in Figure 12-13.

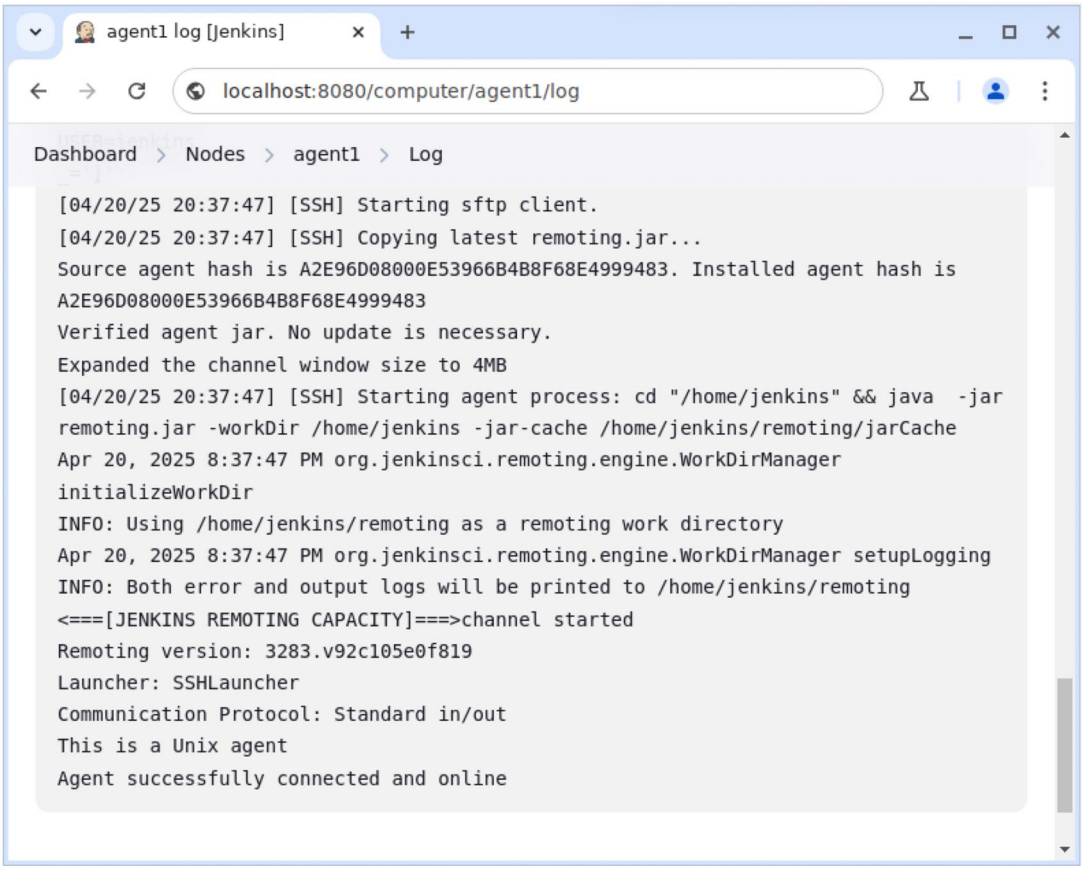

Figure 12-13. *Log showing successful agent connection*

Figure 12-13 shows a successful connection. Back in the real world, of course, you may encounter an error and be forced to troubleshoot. If you run into problems, confirm that the agent is running and is configured with the expected port (the default of `22` within the `jenkins` network in my case). Check that your agent's key is configured with the username `jenkins` (find your key configuration under *Manage Jenkins* ➤ *Credentials*). Return to your agent configuration (located under *Manage Jenkins* ➤ *Nodes*), and double-check the values there. In particular, ensure that you have specified the correct host, port, key, and remote directory. Once you have one agent managed from the built-in node, it should be very easy to add more!

Associating Jobs with the Agent

Now that I have a working agent, I can associate my `userthing` job with it in the job configuration screen (from the main screen, click on the job name, and then select *Configure* from the left-hand menu). The option to force a build to occur on an agent rather than the built-in node can be found in the *General* section. Check *Restrict where this project can be run*, and enter the name of the agent. You can see this configuration in Figure 12-14.

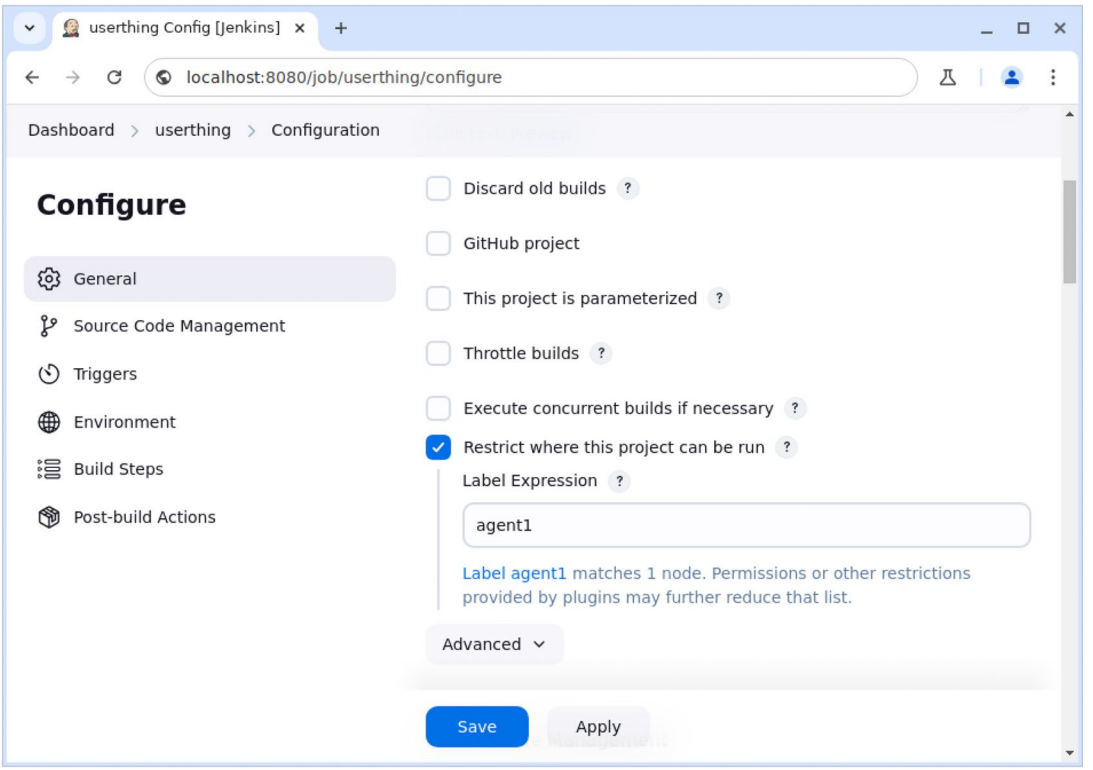

Figure 12-14. *Associating a job with an agent*

If you are running many agents, you might want to bundle groups together using labels. For my purposes, though, I can simply create a one-to-one relationship between the job and the agent using the agent name. From now on, my `userthing` job will build on `agent1`.

Finally, then, I'll kick off a build and watch my agent in action. Thanks to my configuration, it should once again just be a matter of committing and pushing to the `userthing` repository on GitHub. Sure enough, seconds after pushing to the `main` branch, the build appears on my dashboard. You can see the build report in Figure 12-15.

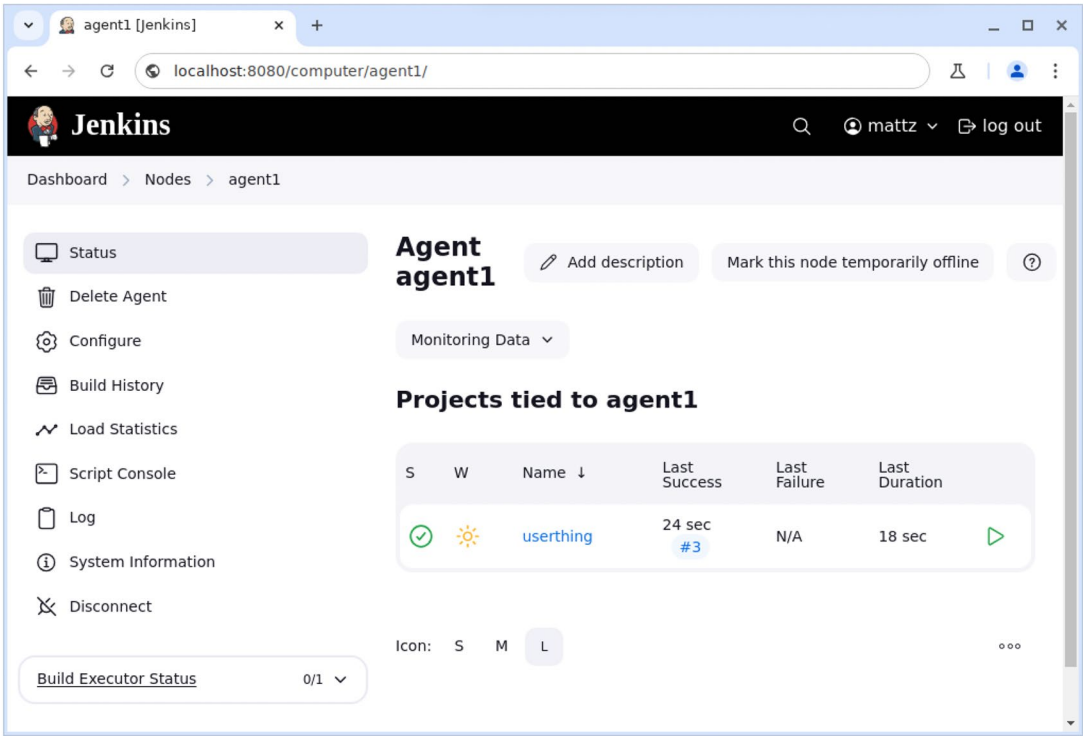

Figure 12-15. *A job run on an agent node*

GitHub Actions

When it comes to continuous integration (and the related topic of continuous delivery), Jenkins is far from the only game in town. These days, the popular Git platforms also offer CI solutions. Since GitHub is the largest of these (and since I happen to have used it to manage the `userthing` repository), I will replicate this chapter's simple example using GitHub Actions.

Why GitHub Actions?

GitHub Actions (https://docs.github.com/en/actions) provide highly configurable workflow systems that you can use to analyze, test, or deploy code from within your repository. These *actions* are individual units of configurable functionality that you can source or even create. They can be stitched into *jobs* within *workflow* scripts which you can define within your code repository.

Although they are less mature than venerable systems like Jenkins, they offer some distinct advantages. In particular, your workflows are, by definition, embedded into your version control repository, making it easy to trigger workflows in response to key events (such as pushes to particular branches, pull requests, or even issues raised) without the song and dance that is necessary to link external tools. Such integration means that you win functionality without the need for a separate CI platform, thereby reducing the number of components in your stack. There are many pre-built actions available (we will encounter `php-actions` a little later, for example), and, because the GitHub Actions uses Docker's containerization at its core, it's also relatively easy to create your own.

There is, of course, a downside. Like most SaaS offerings, GitHub Actions are something of a black box. You are at the mercy of a closed implementation. This means that you may not be able to patch defects and may, at some point, hit arbitrary rate limits or changes in pricing or terms of use. The use of GitHub Actions may also lock you in to GitHub, making it harder to migrate to another version control repository without reimplementing some or all of your CI systems.

Let's take a tour of some of the basics.

The Basics

You can create a workflow script directly within your project. I am already working with `userthing` – a private repository on GitHub. I'll kick things off by creating a file at `.github/workflows/run-tests.yml`. Here it is:

```
name: Run Tests
run-name: Test run intiated by ${{ github.actor }}
on: push
jobs:
  build-test:
    runs-on: ubuntu-latest
```

```
steps:
  - name: envelope
    run: |
        echo "event -- ${{ github.event_name }}"
        echo "on -- ${{ runner.os }}"
        echo "branch/repo --  ${{ github.ref_name }} / ${{ github.
        repository }}"
```

Note The `.github/workflows/` directory should be placed in your project's root directory.

As you may recognize, this is a YML file. The `name` element defines the high-level title. Once you have committed and pushed your workflow file and selected the *Actions* tab in your GitHub repository, you will find the `name` displayed in the left-hand side menu. The value assigned to the `run-name` keyword is applied to individual runs. As you can see, this supports contextual information, so it may vary from run to run. In this case, I reference `github.actor` (the GitHub user who initiated the run). A *context* is an object which provides variable information. Contexts support properties – which can be strings or other objects. The `run-name` string can access two contexts: `github` which contains information about the workflow run and `inputs` which can access custom fields passed in from another workflow. Table 12-3 shows just a few of the many properties provided by `github`. You can find the full list and much more about contexts at https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/accessing-contextual-information-about-workflow-runs.

Table 12-3. *Some `github` Context Object Properties*

Property	Description
`github.actor`	The username associated with a run
`github.event_name`	The name of the event that triggered the workflow
`github.ref_name`	The branch or tag associated with the run
`github.repository`	The full repository name
`github.repositoryUrl`	The repository URL

The on keyword defines the event or events that should trigger a workflow. This can be as simple as a single string like push, or pull_request, or it can become much more complex. To trigger a workflow when a pull request is opened and targets the develop branch, for example, you'd specify:

```
on:
  pull_request:
    types:
      - opened
    branches:
      - 'develop'
```

You can learn more about the gory details of events that trigger workflows at https://docs.github.com/en/actions/writing-workflows/choosing-when-your-workflow-runs/events-that-trigger-workflows.

A workflow is made up of one or more jobs. These will run in parallel by default, but they can be configured run consecutively. I will only define a single job in my Run tests workflow. I name this build-test.

Within the build-test job, then, the runs-on keyword defines the container in which my job will be run. Although you can self-host containers, the standard GitHub-hosted containers include variations on ubuntu, macos, and windows. You can read more about your options at https://docs.github.com/en/actions/using-github-hosted-runners.

A job is made up of multiple steps. A step can run a command or invoke an action. Each step runs in its own process and has access to the workspace (the container defined by runs-on). There are various sub-elements to a step. So far, you've seen name which defines a name for display in a run report and run which invokes a command on the shell (or multiple commands if you use multiline YAML syntax, as I have here).

Well, that's a lot of explanation for what amounts to a glorified *Hello, World* example! Nevertheless, it has set me up for future fragments which will be somewhat less verbose. Let's try things out at this stage, though.

Because I have set the on element to push, all I need to do to get the workflow to play is commit and push the file at .github/workflows/run-tests.yml.

```
$ git add .github/workflows/run-tests.yml
$ git commit -m'added workflow';
$ git push origin main
```

I will see nothing but the standard confirmation of a Git commit and push on the command line. It should be a different matter in the GitHub Web environment, however. From my `userthing` repository screen, when I click the *Actions* tab, I should find that my script has run. In fact, by the time I snapped Figure 12-16, I had already run it a few times!

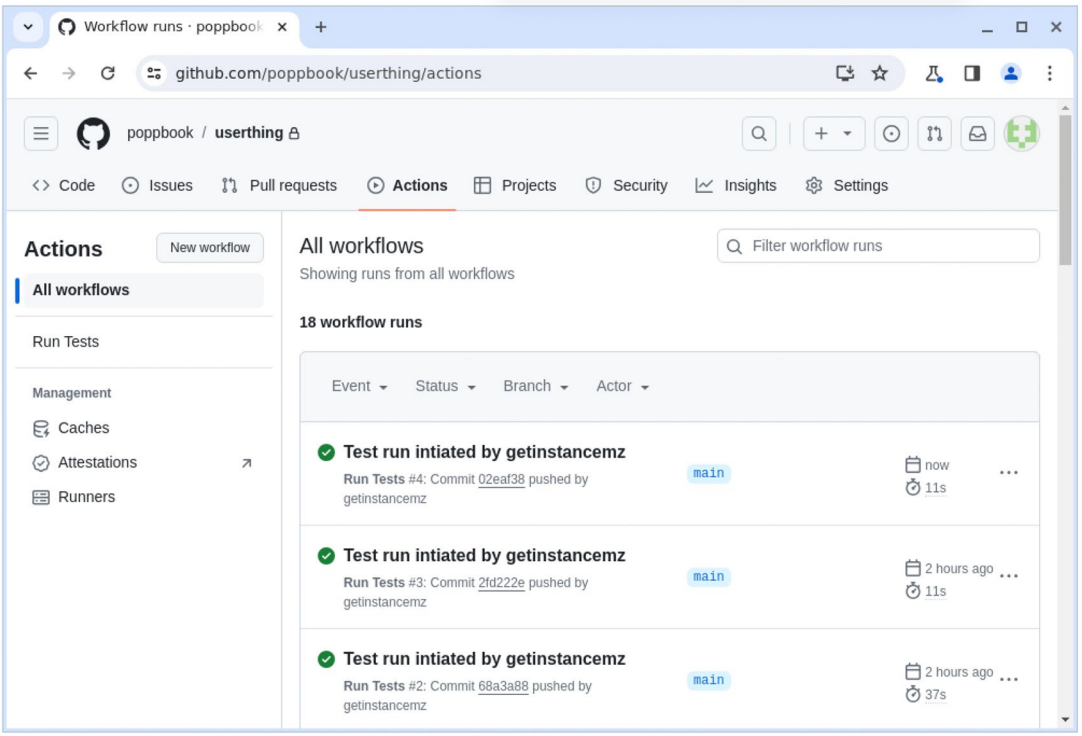

Figure 12-16. *An early version of the "Run Tests" workflow*

From there, I can click my most recent run and, by selecting `Usage`, view my run output. You can see the *Envelope* step in Figure 12-17.

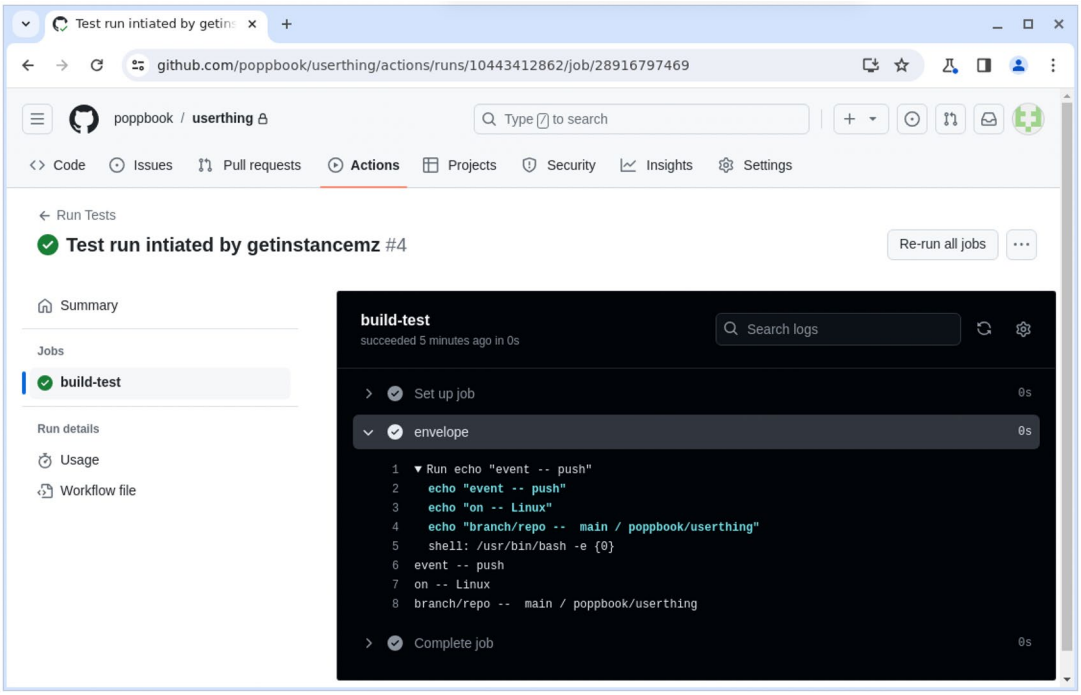

Figure 12-17. *The Envelope step from a "Run Tests" workflow run*

Checking Out the Code

For my next step, I need to check out the repository to my workspace. For this, I'll use my first action:

```
name: Run Tests
run-name: Test run intiated by ${{ github.actor }}
on: push
jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      # ...
      - name: Check out repository code
        uses: actions/checkout@v4
```

An action performs a function in your workflow. It takes the form of a JavaScript file or a Docker container, but from the perspective of a workflow, it is simply a component that can incorporate a `steps` block. You can find actions for many tasks via the GitHub Marketplace at https://github.com/marketplace. The documentation for each action provides a *Usage* panel. As you can see, the `uses` keyword includes the action (comprising an action string and an optional version indicator after the `@` character). If you need to specify any supported inputs to the action, you can use the `with` keyword. Although, as you can see in the documentation at https://github.com/marketplace/actions/checkout, `actions/checkout` does support many inputs, the default behavior is good enough for my purposes. You can see the `actions/checkout` action in operation in Figure 12-18.

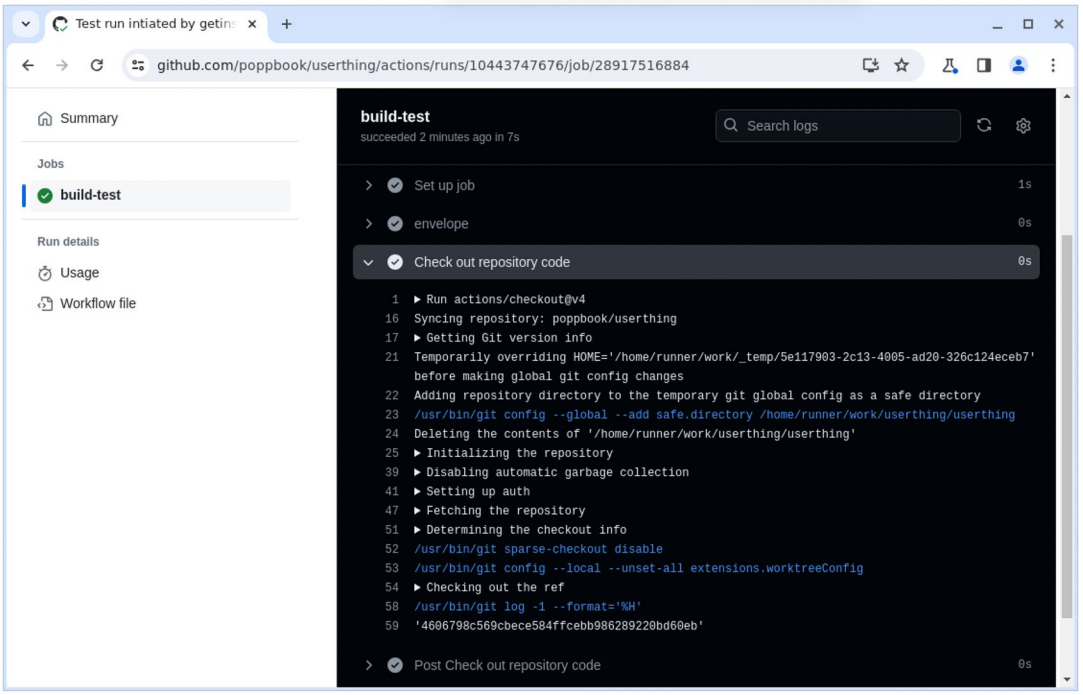

Figure 12-18. *The `actions/checkout` action*

Running Composer

Next, of course, I must run `composer install` on my checked out code. A marketplace search reveals `php-actions/composer`. The documentation tells me that the latest version is `v6`.

```
name: Run Tests
run-name: Test run intiated by ${{ github.actor }}
on: push
jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      # ...
      - name: Composer
        uses: php-actions/composer@v6
```

Note In fact, `php-actions` is a collection of many useful actions for CI on GitHub. Find them at https://github.com/php-actions/.

I add the step to my job then commit and push. Figure 12-19 shows the `Composer` step from my workflow run.

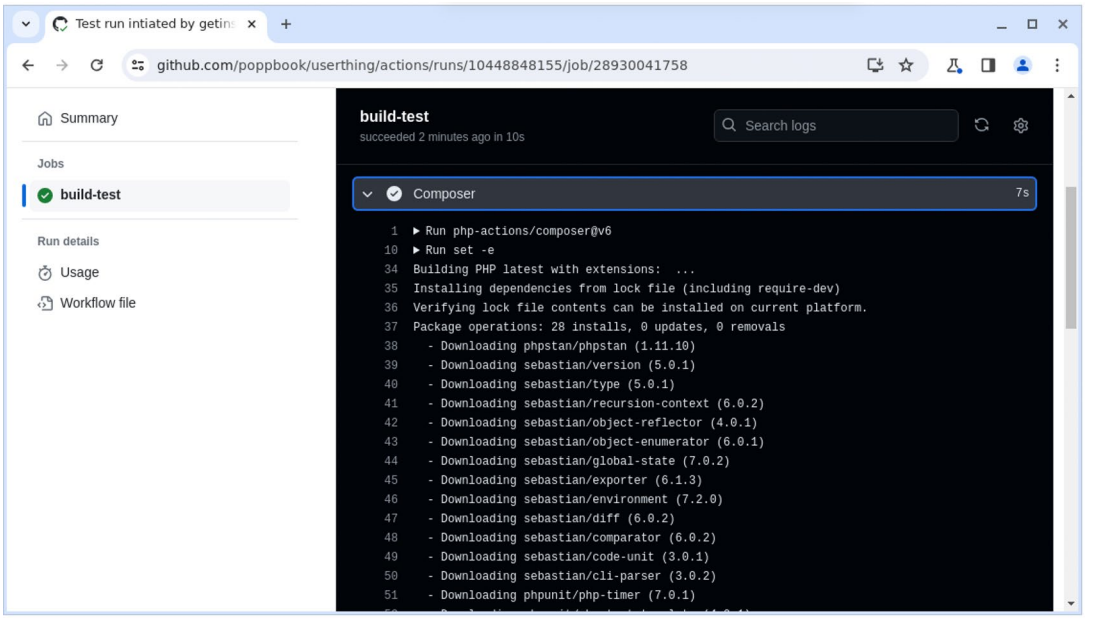

Figure 12-19. *The `php-actions/composer` action*

Running PHPUnit

All I need to do now to catch up with my Jenkins example is add a step for running my tests. As you would expect, there is a `php-actions/phpunit` action.

```
name: Run Tests
run-name: Test run intiated by ${{ github.actor }}
on: push
jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      # ...
      - name: PHPUnit Tests
        uses: php-actions/phpunit@v4
        with:
          bootstrap: vendor/autoload.php
          args: test/
```

This is the first example here of an action that uses the `with` keyword to gather further information. Without `bootstrap`, for example, my classes would not be able to find one another. The action is smart enough to invoke the version of PHPUnit I specified in my composer file – although I could have passed a `version` argument to `with`.

Figure 12-20 shows my tests in action.

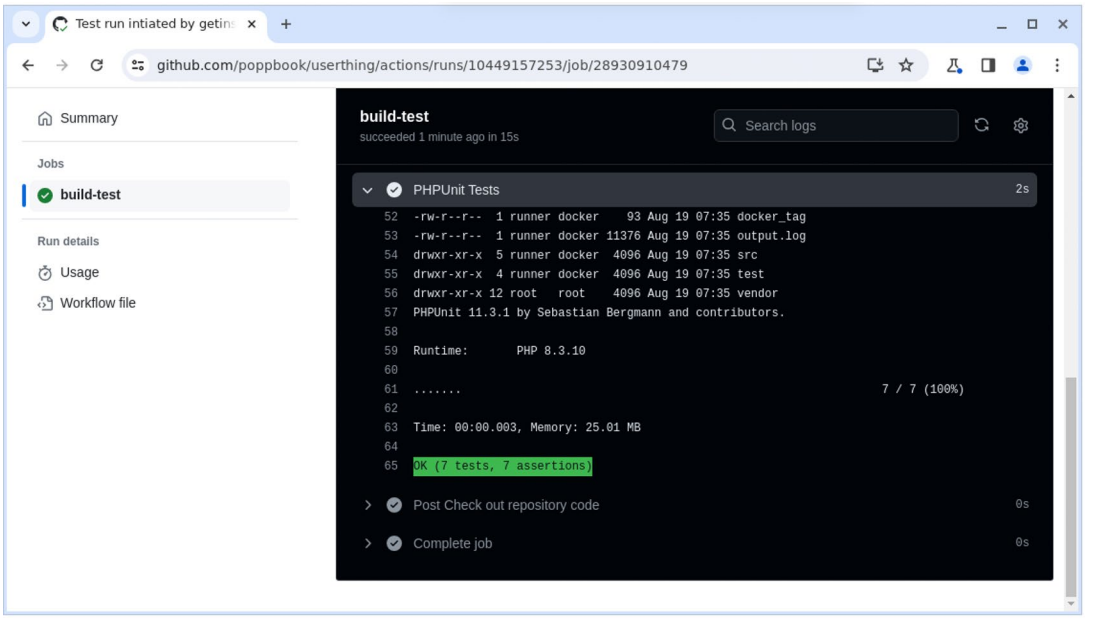

Figure 12-20. *The `php-actions/phpunit` action*

What Next?

Now that you know the basics of both Jenkins and GitHub Actions, the rest is up to you. Once you have automated your PHPUnit tests, it's easy enough to add code coverage reports, for example. Or you might want to check that any pull requests comply with coding standards or pass a baseline PHPStan level. You'll also want to check out the notification options offered by both platforms. If a build fails in obscurity, then, arguably, there was not much point to the automation.

This is a chapter about continuous integration, but you may also have seen references to CI/CD. The CD part of that acronym stands for *continuous delivery*. To grossly simplify a topic that justifies its own book, this means deploying code automatically once all automated checks pass. Of course, you'd want to be satisfied with the quality and extent of your tests before you implemented CD, but thanks to tools such as Ansible, both GitHub Actions and Jenkins are more than capable of driving a CD pipeline.

Summary

In this chapter, I prepared a small project for CI. In addition to the code and some sample unit tests, I configured Composer to support autoloading and install PHPUnit. I created a GitHub repository. Then, I set up Jenkins with Docker and showed you how to add a project to the system and automate building and testing in response to GitHub events. I introduced Jenkins Agents which allow you to separate build environments from the built-in node. Finally, I introduced GitHub Actions and ran through a similar set of build and test steps.

CHAPTER 13

PHP Practice

When I first started learning about programming, picking shiny thick-backed tomes from the shelves in a London bookshop, I discovered worlds of possibility. I soon encountered unexpected practical barriers, however. Just *how* could I get those Perl algorithms to dance on the server? My Java code compiled locally, but I had no idea how it should be packaged and delivered. Of course, the answers were out there online, and there were books that covered them too. I found my way to a book about the Unix shell, for example, when I discovered that for some arcane reason (permissions) I could not write any files from my CGI scripts. Nonetheless, the shiniest, most exciting programming books treated issues such as documentation, testing, and deployment with a somewhat dismissive air, as if these were minor matters that could be left to the reader to sort out. They were right. I sorted them out. But not without much trial, error, and frustration.

The chapters in this volume, therefore, address some of the issues I might have wished were given more prominence back when I was first sorting my classes from my objects. In this chapter, I recap some of these topics:

- *Testing*: Why you should do it.
- *Standards and standards tools*: Keeping your code compliant and bug-free.
- *Inline documentation*: Treating your colleagues (and your future self) with kindness.
- *Development environments*: Using virtualization and containerization to build sandboxes to play in.
- *Version control*: Branch and share code; roll back when things go wrong.
- *Build and deployment*: Manage dependencies and configuration; get your code into the world.

M. Zandstra, *PHP 8 Objects, Patterns, and Practice: Volume 2*, https://doi.org/10.1007/979-8-8688-0779-4_13

- *Command-line scripting*: PHP is not just for the Web.
- *Continuous integration*: Always be building, always be testing.

Practice

The issues that I covered in this volume are often ignored by texts and coders alike. In my own life as a programmer, I discovered that these tools and techniques were at least as relevant to the success of a project as design. There is little doubt that issues such as documentation and automated build are less revelatory in nature than more abstract wonders such as the Composite pattern.

Note Let's just remind ourselves of the beauty of Composite: a simple inheritance tree whose objects can be joined at runtime to form structures that are also trees but are orders of magnitude more flexible and complex. Multiple objects share a single interface by which they are presented to the outside world. The interplay between simple and complex, multiple and singular, has got to get your pulse racing – that's not just software design, it's poetry.

Even if issues such as documentation and build, testing, and version control are more prosaic than patterns, they are no less important. In the real world, a fantastic design will not survive if multiple developers cannot easily contribute to it or understand the source. Systems become hard to maintain and extend without automated testing. Without build tools, no one is going to bother to deploy your work. As PHP's user base widens, so does our responsibility as developers to ensure quality and ease of deployment.

A project exists in two modes. A project is its structures of code and functionality – the logical machine. In a less abstract sense, it is also a set of files and directories, a ground for cooperation, a set of sources and targets, and a subject for transformation. In this sense, a project is a system from the outside as much as it is within the logic of its code. Mechanisms for build, testing, documentation, and version control require the same attention to detail as do the routines such mechanisms support. Focus on the metasystem with as much fervor as you do on the system itself.

Testing

Although testing is part of the framework that one applies to a project from the outside, it is intimately integrated into the code itself. Because total decoupling is not possible, or even desirable, test frameworks are a powerful way of monitoring the ramifications of change. Altering the return type of a method could influence client code elsewhere, causing bugs to emerge weeks or months after the change is made. A test framework gives you half a chance of catching errors of this kind (the better the tests, the better the odds here).

Testing is also a tool for improving object-oriented design. Testing first (or at least concurrently) helps you to focus on a class's interface and think carefully about the responsibility and behavior of every method. I introduced PHPUnit, which is used for testing, in Chapter 7.

Standards and Standards Tools

I am a contrarian by nature. I hate being told what to do. Words like *compliance* instantly invoke a fight-or-flight response in me. But counterintuitive as it may seem, standards drive innovation. That is because they drive interoperability. The rise of the Internet was fueled in part by the fact that open standards are built into its core. Websites can link to one another, and web servers can be reused in any domain because protocols are well known and respected. A solution in a silo may be better than a widely accepted and applied standard, but what if the silo burns down? What if it is bought and the new owner decides to charge for access? What happens when some people decide that the silo next door is better? In Chapter 3, I discussed PSR, PHP Standard Recommendations. I focused, in particular, on standards for autoloading, which have done much to clean up the way that PHP developers include classes. I also looked at PSR-12, the standard for coding style. Programmers have strong feelings about the placement of braces and the deployment of argument lists, but agreeing to abide by a common set of rules makes for readable and consistent code and allows us to use tools to check and reformat our source files.

Larry Wall, the creator of Perl, famously named *laziness* as one of the great virtues of a programmer. So, while I agree that compliance to a shared style is generally a good idea, I also welcome tools that can check for compliance and, as far as possible, fix any issues without the need for manual editing. In Chapter 4, I looked at PHP_CodeSniffer, a set of two tools that do just that. I also covered PHPStan. Like PHP_CodeSniffer, this

static analysis tool checks for compliance with good practice. Even better, it can catch bugs in your system, rooting out variable name typos and type mismatches among very many other code smells.

Inline Documentation

Your code is not as clear as you think it is. A stranger visiting a code base for the first time can be faced with a daunting task. Even you, as author of the code, will eventually forget how it all hangs together. For inline documentation, you should look at phpDocumentor (`https://phpdoc.org/`) which allows you to document as you go and automatically generates hyperlinked output. The output from phpDocumentor is particularly useful in an object-oriented context, as it allows the user to click around from class to class. As classes are often contained in their own files, reading the source directly can involve following complex trails from source file to source file.

The PHPDoc format, which phpDocumentor supports, is also used by tools and IDEs to provide hints for features (such as typed collections) which are not enforced directly by PHP.

See Chapter 2 for more on both phpDocumentor and PHPDoc.

Development Environments

I first taught myself Perl many years ago from several impressively thick books. I thought I had pretty much cracked the whole programming game when I finally understood the code examples and tried them out on my old Mac desktop using MacPerl (long before macOS and Homebrew). To be fair, I *had* made some progress. But the translation from the camel icon on my computer to a Linux server running Apache was another journey altogether. Even when I'd made the transition, this meant installing and running code changes over a horribly slow dial up connection. A laborious process. So, I worked out how to dual boot Linux, and then, I was at last able to approximate a production environment for local development. That was a great step forward. Until, that is, I discovered that the configuration I had created for one project did not play well with the requirements of another. I was forced either to run separate machines or to compromise in complicated ways on my single Linux partition.

The answer was, of course, virtualization. With tools such as VirtualBox, I was able to run completely independent environments on a single machine. Even so, the setup was a

chore in each case. Then, I discovered Vagrant, a platform which sits above virtualization layers such as VirtualBox or VMware and makes choosing, acquiring, and configuring a base box *much* simpler.

Very soon, the first step I took in any project I embarked upon was to create a production-like development environment using Vagrant. The very process of automating this provisioning provided invaluable insight into a new code base and a satisfying productivity boost over less flexible development environments.

I still use and love Vagrant for some projects. Because it uses full virtualization, it allows you to maintain a very good approximation of a full production environment during development. In terms of speed, flexibility, and reliability, though, Docker represents a further step forward. When run on Linux, it runs services in containers directly on the host machine's kernel, which means that, at the cost of some isolation, an environment springs rather than creaks into life.

I covered Vagrant in Chapter 8 and Docker in Chapter 9.

Version Control

Collaboration is hard. Let's face it: people are awkward. Programmers are even worse. Once you've sorted out the roles and tasks on your team, the last thing you want to deal with is clashes in the source code itself. As you saw in Chapter 6, Git (along with similar tools such as CVS and Subversion) enables you to merge the work of multiple programmers into a single repository. Where clashes are unavoidable, Git flags the fact and points you to the source to fix the problem.

Even if you are a solo programmer, version control is a necessity. Git supports branching, so that you can maintain a software release and develop the next version at the same time, merging bug fixes from the stable release to the development branch.

Git also provides a record of every key commit made on your project. This means that you can roll back by date or tag to any moment. This will save your project someday – believe me.

Build and Deployment

Version control without some automation around build and deployment is of limited use. A project of any complexity takes work to deploy. Various files need to be moved to different places on a system, configuration files need to be transformed so that they

incorporate the right values for the target server, database tables need to be set up. I covered two tools designed for installation in this volume.

The first, Composer (see Chapter 5), manages library dependencies. It handles – near flawlessly – intra-package requirements so that you almost never find yourself trapped in so-called "dependency hell." Because it installs most of your project requirements in a local directory, you do not need to worry about clashes between projects or between separate services which are part of the same project.

The second tool I covered was Ansible (see Chapter 10). This is a powerful deployment platform – a tool with enough power and flexibility to automate the installation of the largest and most labyrinthine project across any number of servers. It is particularly good at two perennial problems: managing secrets without exposing them to your version control system and handling system configuration.

Together, Composer and Ansible can transform deployment from a chore to a matter of a line or two at the command prompt.

Command-Line Scripting

While PHP is known primarily as a web programming language, it can also be used to create powerful command-line tools. As a PHP programmer, you will likely have the PHP interpreter at hand in many if not most environments (and if you don't, these days, Docker can always provide it without changing the host machine's configuration). That means you can take advantage of PHP's power, ease of use, and vast repository of libraries to build a command-line tool ranging in scope from a small utility to a fully featured application. What's more, because you're coding in PHP, you can also create command-line utilities to perform actions that integrate tightly with any larger web systems in your project. I covered programming on the command line in Chapter 11.

Continuous Integration

It is not enough to be able to test and build a project; you have to do it *all the time*. This becomes increasingly important as a project grows in complexity and you manage multiple branches. You should build and test the stable branch from which you make minor bug fix releases, an experimental development branch or two, and your main trunk. If you were to try to do all that manually, even with the aid of build and test tools, you'd never get around to any coding. Of course, all coders hate that, so build and testing inevitably get skimped on.

In Chapter 12, I looked at continuous integration, a practice and a set of tools that automate the build and test processes as much as possible.

What I Missed

A few tool categories I have had to omit from this book due to time and space constraints are, nonetheless, supremely useful for any project. In most cases, there is more than one good tool for the job at hand, so, although I'll suggest one or two, you may want to spend some time talking with other developers and digging around with your favorite search engine before you make your choice.

If your project has more than one developer or even just an active client, then you will need a tool to track bugs and tasks. Like version control, a bug tracker is one of those productivity tools that, once you have tried it on a project, you cannot imagine not using. Trackers allow users to report problems with a project, but they are just as often used as a means of describing required features and allocating their implementation to team members.

You can get a snapshot of open tasks at any time, narrowing the search according to product, task owner, version number, and priority. Each task has its own page, in which you can discuss any ongoing issues. Discussion entries and changes in task status can be copied by mail to team members, so it's easy to keep an eye on things without going to the tracker URL all the time.

There are many tools out there. Even after all this time, though, I often return to the venerable Bugzilla (https://www.bugzilla.org). Bugzilla is free and open source and has all the features most developers could need. It is a downloadable product, so you will have to run it on your own server. It still looks a little Web 1.0, but it's none the worse for that. If you do not want to host your own tracker, and you have or prefer your interfaces a little prettier (and have deeper pockets), you might look at the Atlassian's SAAS solution, Jira (https://www.atlassian.com/software/jira). I have also successfully used GitHub's built in issue tracker for some of my projects.

For high-level task tracking and project planning (especially if you're interested in using a Kanban system), you might also look at Trello (https://trello.com).

A tracker is generally just one of a suite of collaboration tools you will want to use to share information around a project. At a price, you can use an integrated solution such as Basecamp (https://basecamp.com/) or Atlassian tools (https://www.atlassian.com/). Or you may choose to stitch together a tools ecosystem using a variety

of applications. To facilitate communication within your team, for example, you will probably need a mechanism for chat or messaging. Perhaps the most popular tool for this at the time of this writing is Slack (https://slack.com). Slack is a multiroomed web-based chat environment. If you're old school like me, you might instantly think of IRC (Internet Relay Chat) – and you'd be right: there's little you can do with Slack that you couldn't do with IRC, except that Slack is browser based, easy to use, and has integration with other services already built-in. Slack is free unless you need premium features. Other options include Mattermost (https://mattermost.com) – which is similar to Slack but can be self-hosted – and Discord (https://discord.com/).

Speaking of old school, you might also consider using a mailing list for your project. My favorite mailing list software is Mailman (https://list.org/), which is free, relatively easy to install, and highly configurable.

For cooperatively editable text documents and spreadsheets, Google Docs (https://docs.google.com/) is probably the easiest solution.

Although inline documentation is important, projects also generate a writhing heap of written material. This can include usage instructions, consultation on future directions, client assets, meeting minutes, and party announcements. During the lifetime of a project, such materials are very fluid, and a mechanism is often needed to allow people to collaborate in their evolution.

A wiki (wiki is apparently derived from the Hawaiian word *wikiwiki* meaning "very fast") is the perfect tool for creating collaborative webs of hyperlinked documents. Pages can be created or edited at the click of a button, and hyperlinks are automatically generated for words that match page names. A wiki is another one of those tools that seems so simple, essential, and obvious that you are sure you probably had the idea first but just didn't get around to doing anything about it. There are a number of wikis to choose from. I have had good experience with DokuWiki, which you can find at https://www.dokuwiki.org/dokuwiki.

For documentation (and for writing in general), though, I have tended, increasingly, to pare back to simple text documents and version control. For formatting, I use Markdown, a lightweight markup language. It is easy to read before rendering and usually is clean and well balanced afterward (though, as with all rendering, you are at the mercy of the renderer). The best starting place for Markdown is https://commonmark.org/. After years of struggling with Word and Word-compatible word processors, I am very grateful that Apress let me use Markdown for this edition of the book!

Note Although I did not omit this tool (see Chapter 6), it is worth mentioning that shifting to a plain text format made it possible for us to make extensive use of Git in the development of this book.

Summary

In this chapter, I wrapped things up, revisiting the core topics that make up the book. Although I haven't tackled any concrete issues such as individual patterns or object functions here, this chapter should serve as a reasonable summary of this book's concerns.

There is never enough room or time to cover all the material that one would like. Nevertheless, I hope that this book and its companion (Volume 1) has served to make one argument: PHP is all grown up. It is now one of the most popular programming languages in the world. I hope that PHP remains the hobbyist's favorite language and that many new PHP programmers are delighted to discover how far they can get with just a little code. At the same time, though, more and more professional teams are building large systems with PHP. Such projects deserve more than a just-do-it approach. Through its extension layer, PHP has always been a versatile language, providing a gateway to hundreds of applications and libraries. Its object-oriented support, on the other hand, gains you access to a different set of tools. Once you begin to think in objects, you can chart the hard-won experience of other programmers. You can navigate and deploy pattern languages developed with reference not just to PHP but to Smalltalk, C++, C#, or Java, too. It is our responsibility to meet this challenge with careful design and good practice. The future is reusable.