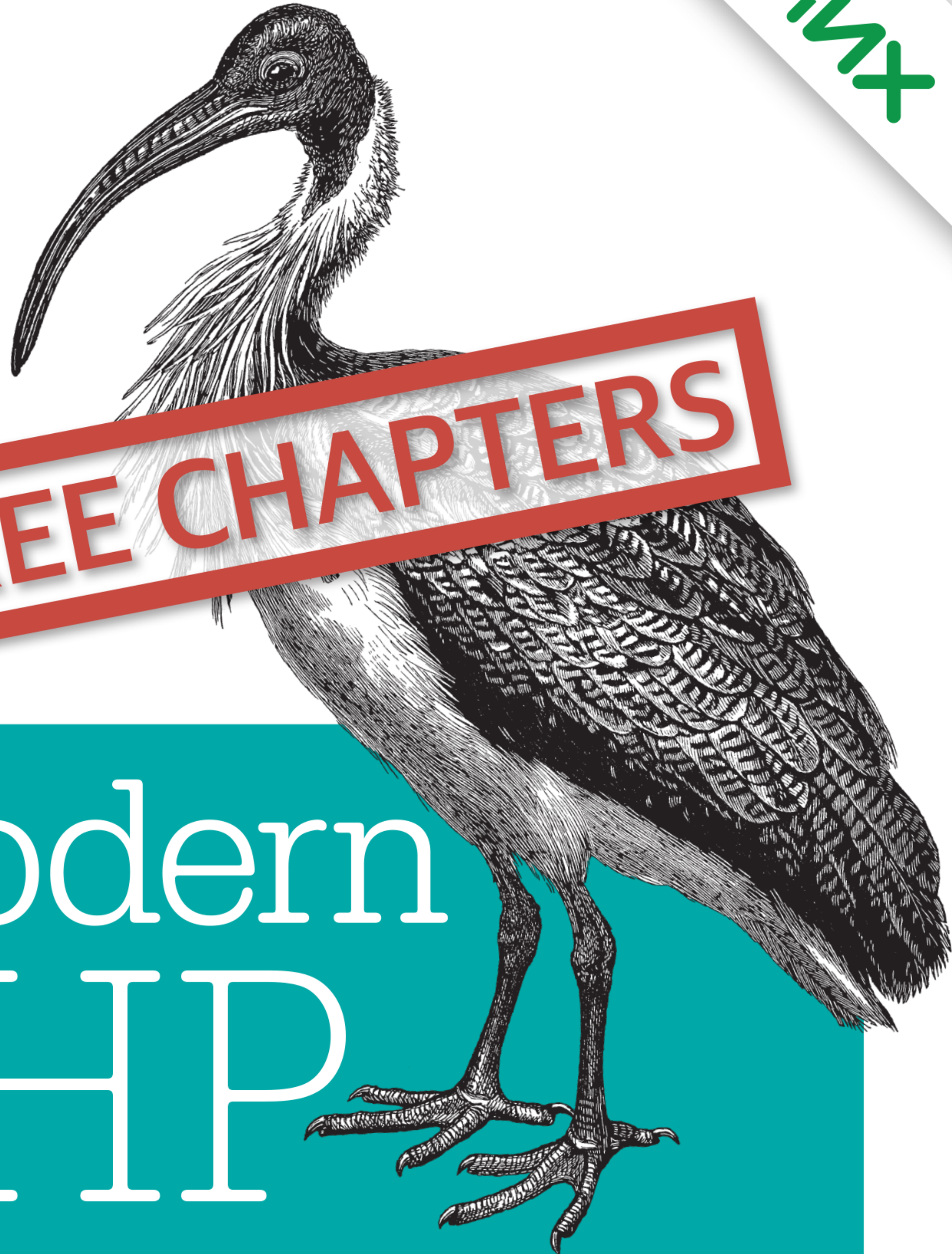
O'REILLY®

Compliments of NGINX

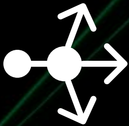


Modern PHP

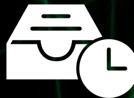
NEW FEATURES AND GOOD PRACTICES

Josh Lockhart

flawless application delivery



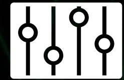
Load Balancer



Content Cache

Web Server

Security Controls



Monitoring & Management

FREE TRIAL

LEARN MORE

NGINX+

Modern PHP

New Features and Good Practices

This is an excerpt of the book *Modern PHP*. The full book is available at oreilly.com and through other retailers.

Josh Lockhart

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Modern PHP

by Josh Lockhart

Copyright © 2015 Josh Lockhart. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Allyson MacDonald
Production Editor: Nicole Shelby
Copyeditor: Phil Dangler
Proofreader: Eileen Cohen
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

February 2015: First Edition

Revision History for the First Edition

2015-02-09: First Release
2016-02-26: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491905012* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Modern PHP*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-90501-2

[LSI]

Table of Contents

Foreword vii

Part I. Deployment, Testing, and Tuning

1. Hosting 1
Shared Server 1
Virtual Private Server 2
Dedicated Server 3
PaaS 3
Choose a Hosting Plan 4

2. Provisioning 5
Our Goal 6
Server Setup 6
  First Login 6
  Software Updates 7
  Nonroot User 7
  SSH Key-Pair Authentication 8
  Disable Passwords and Root Login 10
PHP-FPM 10
  Install 10
  Global Configuration 11
  Pool Configuration 12
Nginx 15
  Install 15
  Virtual Host 15
Automate Server Provisioning 18

Delegate Server Provisioning 18
Further Reading 19
What's Next 19

3. **Tuning** 21
The php.ini File 21
Memory 22
Zend OPcache 23
File Uploads 24
Max Execution Time 25
Session Handling 26
Output Buffering 27
Realpath Cache 27
Up Next 27

4. **Deployment** 29
Version Control 29
Automate Deployment 29
Make It Simple 30
Make It Predictable 30
Make It Reversible 30
Capistrano 30
How It Works 30
Install 31
Configure 31
Authenticate 33
Prepare the Remote Server 33
Capistrano Hooks 34
Deploy Your Application 34
Roll Back Your Application 35
Further Reading 35
What's Next 35

5. **Testing** 37
Why Do We Test? 37
When Do We Test? 38
Before 38
During 38
After 38
What Do We Test? 38
How Do We Test? 39
Unit Tests 39

Test-Driven Development (TDD) 39
Behavior-Driven Development (BDD) 39
PHPUnit 40
Directory Structure 41
Install PHPUnit 42
Install Xdebug 42
Configure PHPUnit 43
The Whovian Class 44
The WhovianTest Test Case 45
Run Tests 47
Code Coverage 48
Continuous Testing with Travis CI 49
Setup 49
Run 50
Further Reading 50
What's Next 51

6. Profiling. 53
When to Use a Profiler 53
Types of Profilers 53
Xdebug 54
Configure 54
Trigger 55
Analyze 55
XHProf 55
Install 56
XHGUI 56
Configure 57
Trigger 57
New Relic Profiler 57
Blackfire Profiler 58
Further Reading 58
What's Next 58

7. HHVM and Hack. 59
HHVM 59
PHP at Facebook 60
HHVM and Zend Engine Parity 61
Is HHVM Right for Me? 62
Install 62
Configure 63
Extensions 64

Monitor HHVM with Supervisord 64
HHVM, FastCGI, and Nginx 66
The Hack Language 67
Convert PHP to Hack 67
What is a Type? 68
Static Typing 69
Dynamic Typing 70
Hack Goes Both Ways 70
Hack Type Checking 71
Hack Modes 71
Hack Syntax 72
Hack Data Structures 74
HHVM/Hack vs. PHP 75
Further Reading 76

8. Community. 77
Local PUG 77
Conferences 77
Mentoring 78
Stay Up-to-Date 78
Websites 78
Mailing Lists 78
Twitter 78
Podcasts 78
Humor 79

Foreword

Back in 1995, software engineer Rasmus Lerdorf needed an easier way to maintain his own web page, and wrote a set of programs he called Personal Home Page, or PHP. He soon open sourced the software, but never anticipated it would become the dominant web-page programming language it is today, Now, PHP powers more than 80% of websites that report the language they use, according to W3Techs. PHP usage extends well beyond personal blogging sites to giants like Facebook, Twitter, and Wikipedia.

The longevity and popularity of PHP has both pros and cons. The positive is that you can easily find lots of PHP tutorials and other resources online. The downside is that many of the existing resources still recommend outdated practices that result in slow and unstable apps. To address this problem, Josh Lockhart has written *Modern PHP*, the go-to resource today's developers have desperately needed to help them modernize their PHP apps.

As you update your approach to developing PHP apps, it also makes sense to reconsider how you're delivering them. The Apache HTTP server debuted when the internet was new, and worked fine at the time. But NGINX is purpose-built to handle the vastly greater demands that your apps now face, with an event-driven design that is much lighter-weight and uses fewer system resources than Apache. The result is higher performance from your PHP applications.

Once your PHP app is deployed to production and gains in popularity, you'll need to scale. NGINX Plus, our enterprise-grade product, is commonly found in the reverse proxy position in front of PHP app servers. NGINX Plus offers a number of advanced features, such as load balancing with health checks and session persistence, that you can use to safely scale out your PHP applications. It also comes with 24x7 support, which is critical for maintaining the production applications businesses depend on.

We hope you enjoy this excerpt from *Modern PHP*. It includes all of Part 3, covering tuning, testing, and deployment of PHP applications.

— Faisal Memon
Product Marketer, NGINX Inc.

PART I

Deployment, Testing, and Tuning

CHAPTER 1
Hosting

So you have a PHP application. Congratulations! However, it doesn't do anyone any good unless your users can, you know, *use* it. You need to host your application on a server and make it accessible to its intended audience. Generally speaking, there are four ways to host PHP applications: shared servers, virtual private servers, dedicated servers, and platforms as a service. Each has its unique benefits and is suitable for different types of applications and budgets.

There are also many web hosting companies, and it can be overwhelming if you are brand new to the web hosting landscape. Some hosting companies provide only shared servers. Other companies provide a mix of shared servers, virtual private servers, and dedicated servers. This chapter will focus less on the companies themselves and more on hosting options.

Shared Server

A shared server is the most affordable hosting option and costs $1–10/month. *You should avoid shared hosting plans*. This is not a commentary on shared hosting companies' quality of service or customer support. There are many good shared hosting companies. Simply put, shared hosting options are not developer-friendly.

A shared server, as its name implies, means that you share server resources with other people. If you purchase a shared hosting plan, your hosting account lives on the same physical machine as many other customers'. If your particular machine has 2 Gb of memory, your PHP application might receive only a fraction of that memory, depending on how many other customer accounts live on the same machine. If another account on the same machine runs a poorly coded script, it can negatively affect your own application. Some shared hosting companies oversell shared servers,

and your PHP application constantly battles for system resources on a crowded machine.

Shared servers are also very difficult to customize. For example, your application may need Memcached or Redis for a fast, in-memory cache. You may want to install Elasticsearch to add search functionality to your application. Unfortunately, shared server software is difficult—if not impossible—to customize. Your applications suffer as a result.

Shared servers rarely provide remote SSH access. Instead, you're often handicapped with (S)FTP access only. This limitation severely restricts your ability to automate PHP application deployment.

If your budget is super-small or your needs extremely modest, a shared server may be sufficient. However, if you're building a business website or a moderately popular PHP application, you're better off using a virtual private server, a dedicated server, or a PaaS.

Virtual Private Server

A virtual private server (VPS) looks, feels, and acts like a bare-metal server. But it's not a bare-metal server. A VPS is a collection of system resources that are distributed across one or many physical machines. A VPS still has its own filesystem, root user, system processes, and IP address. A VPS is allocated a specific amount of memory, CPU, and bandwidth—and they're all yours.

VPSs provide more system resources than a shared server. A VPS provides root SSH access. And a VPS does not limit what software you can install. Great power, though, comes with great responsibility. VPSs give you root access to a virgin operating system. It is your responsibility to configure and secure the operating system for your PHP application. VPSs are ideal for most PHP applications. They provide sufficient system resources (e.g., CPU, memory, and disk space) that scale up or down on demand. A VPS costs $10–100/month based on the amount of system resources needed by your PHP application. If your PHP application becomes super-popular (hundreds of thousands of visitors a month) and a VPS becomes too costly, you might consider upgrading to a dedicated server.

I almost always prefer VPSs for their balance of cost, features, and flexibility. Linode, my favorite hosting company, provides VPS and dedicated hosting plans. Linode isn't the cheapest option, but my personal experience shows Linode is fast and stable, and it comes with a vast treasure of helpful tutorials.

Dedicated Server

A dedicated server is a rack-mounted machine that your hosting company installs, runs, and maintains on your behalf. You configure dedicated servers to your exact specifications. Dedicated servers are real machines that must be transported, installed, and monitored. They cannot be set up and configured as quickly as VPSs. That being said, dedicated servers provide the ultimate performance for demanding PHP applications.

Dedicated servers act much like VPSs. You get root SSH access to a virgin operating system, and you must secure and configure the operating system for your PHP application. The benefit of a dedicated server is cost-effectiveness. Eventually a VPS becomes too costly as you consume more system resources. You save money by investing in your own infrastructure.

A dedicated server costs hundreds of dollars per month depending on the server specifications. It can be unmanaged (i.e., you manage the server yourself) or managed (i.e., you pay extra for your hosting company to manage the server).

PaaS

Platforms as a service (PaaS) are a quick way to launch your PHP application, and—unlike with a virtual private or dedicated server—you don't have to manage a PaaS. All you have to do is log into your PaaS provider's control panel and click a few buttons. Some PaaS providers have a command-line or HTTP API with which you can deploy and manage your hosted PHP applications. Popular PHP PaaS providers include:

- AppFog
- AWS Elastic Beanstalk
- Engine Yard
- Fortrabbit
- Google App Engine
- Heroku
- Microsoft Azure
- Pagoda Box
- Red Hat OpenShift
- Zend Developer Cloud

PaaS pricing varies by provider but is similar to virtual private servers: $10–100/month. You pay for the system resources allocated to your PHP application. System

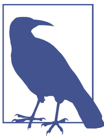
resources can be scaled up or down on demand. I recommend PaaS hosting plans for developers who do not want to manage their own servers.

Choose a Hosting Plan

Choose only what you need when you need it. You can always scale your hosting infrastructure up or down when necessary. For small PHP applications or prototypes, a PaaS provider like Engine Yard or Heroku is the best and quickest solution. If you prefer more control over your server configuration, get a VPS. If your application becomes super-popular and your VPS is buckling beneath the weight of millions of visitors (congratulations, by the way!), get a dedicated server. Whichever hosting option you choose, make sure it provides the latest stable PHP version and extensions required by your PHP application.

CHAPTER 2
Provisioning

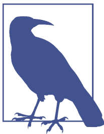
After you choose a host for your application, it's time to configure and provision the server for your PHP application. I'll be honest—provisioning a server is an art, not a science. How you provision your server depends entirely on your application's needs.



If you use a PaaS, your server infrastructure is managed by the PaaS provider. All you have to do is follow the provider's instructions to move your PHP application onto their platform, and you're ready to go.

If you don't use a PaaS, you must provision either a VPS or dedicated server to run your PHP application. Provisioning a server is not as hard as it sounds (stop laughing), but it does require familiarity with the command line. If the command line is alien to you, you're better off with a PaaS like Engine Yard or Heroku.

I don't consider myself a system administrator. However, basic system adminstration is an incredibly valuable skill for application developers that enables more flexible and robust application development. In this chapter, I'll share my system administration knowledge so you can feel comfortable opening a terminal to provision a server for your PHP application. Afterward, I'll suggest a few additional resources for you to continue improving your system administration skills.



In this chapter, I assume you know how to edit a text file using a command-line editor like nano or vim (these are available on most Linux distributions). Otherwise, you'll need an alternative method of accessing and editing files on your server.

Our Goal

First, we need to acquire a virtual private or dedicated server. Next, we need to install a web server to receive HTTP requests. Finally, we need to set up and manage a group of PHP processes to handle PHP requests; these processes must communicate with our web server.

Several years ago, it was common practice to install the Apache web server and the Apache `mod_php` module. The Apache web server spawns a unique child process to handle *each* HTTP request. The Apache `mod_php` module embeds a unique PHP interpreter inside each spawned child process—even processes that serve only static assets like JavaScript, images, or stylesheets. This is a lot of overhead that wastes system resources. I see fewer and fewer PHP developers use Apache nowadays because there are more efficient solutions.

Today, I prefer the nginx web server. Nginx sits in front of (and forwards PHP requests to) a collection of PHP-FPM processes. This is the solution I demonstrate in this chapter.

Server Setup

First, let's set up a virtual private server (VPS). I absolutely adore Linode. It isn't the cheapest VPS provider, but it's one of the most reliable. Head over to Linode's website (or your preferred vendor) and purchase a new VPS. Your vendor will ask you to choose a Linux distribution and a root password for your new server.

Many VPS providers, like Linode and Digital Ocean, bill by the hour. This means you can fire up and play with a VPS at virtually zero cost.

First Login

The first thing you should do is log in to your new server. Do that now. Open a terminal on your local machine and `ssh` into your server. Be sure you swap in your own machine's IP address:

```
ssh root@123.456.78.90
```

You may be asked to confirm the authenticity of your new server. Type **`yes`** and press Enter:

```
The authenticity of host '123.456.78.90 (123.456.78.90)' can't be established.
RSA key fingerprint is 21:eb:37:f3:a5:d3:c0:77:47:c4:15:3d:3c:dc:3c:d1.
Are you sure you want to continue connecting (yes/no)?
```

Next, you'll be prompted for the root user's password. Type the password and press Enter:

```
root@123.456.78.90's password:
```

You are now logged into your new server!

Software Updates

The very next thing you should do is update your operating system's software with these commands.

```
# Ubuntu
apt-get update;
apt-get upgrade;

# CentOS
yum update
```

These commands spit out a lot of information as software updates for your operating system are downloaded and applied. This is an important first step because it ensures you have the latest updates and security fixes for your operating system's default software.

Nonroot User

Your new server is not secure. Here are a few good practices to harden your new server's security.

Create a *nonroot* user. You should log in to your server as this nonroot user in the future. The root user has unlimited power on your server. It is God. It can run any command without question. *You should make it as difficult as possible to access your server as the root user.*

Ubuntu

Create a new nonroot user named `deploy` with the command in Example 2-1. Enter a user password when prompted, and follow the remaining on-screen instructions.

Example 2-1. Create nonroot user on Ubuntu

```
adduser deploy
```

Next, assign the `deploy` user to the `sudo` group with this command:

```
usermod -a -G sudo deploy
```

This gives the `deploy` user `sudo` privileges (i.e., it can perform privileged tasks with password authentication).

CentOS

Create a new nonroot user named `deploy` with this command:

```
adduser deploy
```

Give the `deploy` user a password with this command. Enter and confirm the new password when prompted:

```
passwd deploy
```

Next, assign the `deploy` user to the `wheel` group with this command:

```
usermod -a -G wheel deploy
```

This gives the `deploy` user `sudo` privileges (i.e., it can perform privileged tasks with password authentication).

SSH Key-Pair Authentication

On your local machine, you can log into your new server as the nonroot `deploy` user like this:

```
ssh deploy@123.456.78.90
```

You'll be prompted for the `deploy` user's password, and then you'll be logged in to the server. We can make the login process more secure by disabling password authentication. Password authentication is vulnerable to brute-force attacks in which bad guys try to guess your password over and over in quick succession. Instead, we'll use *SSH key-pair authentication* when we `ssh` into our server.

Key-pair authentication is a complex subject. In basic terms, you create a pair of "keys" on your local machine. One key is private (this stays on your local machine), and one key is public (this goes on the remote server). They are called a *key pair* because messages encrypted with the public key can be decrypted only by the related private key.

When you log in to the remote machine using SSH key-pair authentication, the remote machine creates a random message, encrypts it with your public key, and sends it to your local machine. Your local machine decrypts the message with your private key and returns the decrypted message to the remote server. The remote server then validates the decrypted message and grants you access to the server. This is a dramatic simplification, but you get the point.

If you log in to your remote server from many different computers, you probably do not want to use SSH key-pair authentication. This would require you to generate public/private SSH key pairs for each local computer and copy each key pair's public key to your remote server. In this case, it's probably preferable to continue using password authentication with a secure password. However, if you are only accessing your remote server from a single local computer (as many developers often do), SSH key-

pair authentication is the way to go. You can create an SSH key-pair on your local machine with this command:

```
ssh-keygen
```

Follow the subsequent on-screen instructions and enter the requested information when prompted. This command creates two files on your local machine: *~/.ssh/id_rsa.pub* (your public key) and *~/.ssh/id_rsa* (your private key). The private key should stay on your local computer and remain a secret. Your public key, however, must be copied onto your new server. We can copy the public key with the `scp` (secure copy) command:

```
scp ~/.ssh/id_rsa.pub deploy@123.456.78.90:
```

Be sure you include the trailing `:` character! This command uploads your public key to the `deploy` user's home directory on your remote server. Next, log in to your remote server as the `deploy` user. After you log in to your remote server, make sure the *~/.ssh* directory exists. If it does not exist, create the *~/.ssh* directory with this command:

```
mkdir ~/.ssh
```

Next, create the *~/.ssh/authorized_keys* file with this command:

```
touch ~/.ssh/authorized_keys
```

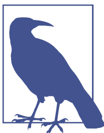
This file will contain a list of public keys that are allowed to log into this remote server. Execute this command to append your recently uploaded public key to the *~/.ssh/authorized_keys* file:

```
cat ~/id_rsa.pub >> ~/.ssh/authorized_keys
```

Finally, we need to modify a few directory and file permissions so that only the `deploy` user can access its own *~/.ssh* directory and read its own *~/.ssh/authorized_keys* file. Assign these permissions with these commands:

```
chown -R deploy:deploy ~/.ssh;
chmod 700 ~/.ssh;
chmod 600 ~/.ssh/authorized_keys;
```

We're done! On your local machine, you should now be able to `ssh` into the remote server without entering a password.



You can only `ssh` into your remote server without a password from the local machine that has your private key!

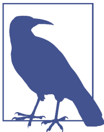
Disable Passwords and Root Login

Let's make the remote server even more secure. We'll disable password authentication for all users, and we'll prevent the root user from logging in—period. Remember, the `root` user can do anything, so we want to make it as difficult as possible to access our server as the root user.

Log in to the remote server as the `deploy` user and open the */etc/ssh/sshd_config* file in your preferred text editor. The is the SSH server's configuration file. Find the `PasswordAuthentication` setting and change its value to `no`; uncomment this setting if necessary. Find the `PermitRootLogin` setting and change its value to `no`; uncomment this setting if necessary. Save your changes and restart the SSH server with this command to apply your changes:

```
# Ubuntu
sudo service ssh restart

# CentOS
sudo systemctl restart sshd.service
```

You're done. You've secured your server, and it's time to install additional software to run your PHP application. From this point forward, all instructions should be completed on the remote server as the nonroot `deploy` user.



Server security is an ongoing task that should be constantly monitored. I recommend you implement a firewall in addition to my previous instructions. Ubuntu users can use UFW. CentOS users can use iptables.

PHP-FPM

PHP-FPM (PHP FastCGI Process Manager) is software that manages a pool of related PHP processes that receive and handle requests from a web server like Nginx. The PHP-FPM software creates one master process (usually run by the operating system's `root` user) that controls how and when HTTP requests are forwarded to one or more child processes. The PHP-FPM master process also controls when child PHP processes are created (to answer additional web application traffic) and destroyed (if they are too old or no longer necessary). Each PHP-FPM pool process lives longer than a single HTTP request, and it can handle 10, 50, 100, 500, or more HTTP requests.

Install

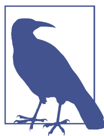
The simplest way to install PHP-FPM is with your operating sytem's native package manager, as demonstrated by the following commands.

See ??? for a detailed PHP-FPM installation guide.

```
# Ubuntu
sudo apt-get install software-properties-common python-software-properties;
sudo add-apt-repository ppa:ondrej/php;
sudo apt-get update;
sudo apt-get install php7.0-fpm php7.0-cli php7.0-curl \
     php7.0-gd php7.0-json php7.0-mcrypt \
     php7.0-mysql php7.0-opcache php7.0-intl;

# CentOS
sudo yum install https://dl.fedoraproject.org/pub/epel/ \
         epel-release-latest-7.noarch.rpm;
sudo yum install http://rpms.remirepo.net/enterprise/remi-release-7.rpm;
sudo yum install yum-utils;
sudo yum-config-manager --enable remi-php70;
sudo yum update;
sudo yum install php70-php-fpm php70-php-cli php70-php-gd \
         php70-php-json php70-php-mbstring php70-php-mcrypt \
         php70-php-mysqlnd php70-php-opcache php70-php-pdo \
         php70-php-intl;
sudo ln -s /usr/bin/php70 /usr/bin/php;
```

Global Configuration

On Ubuntu, the primary PHP-FPM cofiguration file is */etc/php/7.0/fpm/php-fpm.conf*. On CentOS, the primary PHP-FPM configuration file is */etc/opt/remi/php70/php-fpm.conf*. Open this file in your preferred text editor.



PHP-FPM configuration files use the INI file format. Learn more about the INI format on Wikipedia.

These are the most important *global* PHP-FPM settings that I recommend you change from their default values. These two settings might be commented out by default; uncomment them if necessary. These settings prompt the master PHP-FPM process to restart if a specific number of its child processes fail within a specific interval of time. These settings are a basic safety net for your PHP-FPM processes that can resolve simple issues. They are not a solution to more fundamental problems caused by bad PHP code.

`emergency_restart_threshold = 10`
: The maximum number of PHP-FPM child processes that can fail within a given time interval until the master PHP-FPM process gracefully restarts

`emergency_restart_interval = 1m`
: The length of time that governs the `emergency_restart_threshold` setting

Read more about PHP-FPM global configuration at *https://php.net/manual/install.fpm.configuration.php*.

Pool Configuration

Elsewhere in the PHP-FPM configuration file is a section named `Pool Definitions`. This section contains configuration settings for each PHP-FPM pool. A PHP-FPM pool is a collection of related PHP child processes. One PHP application typically has its own PHP-FPM pool.

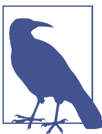
On Ubuntu, the `Pool Definitions` section contains this one line:

```
include=/etc/php/7.0/fpm/pool.d/*.conf
```

CentOS includes the pool definition files at the top of the primary PHP-FPM configuration file with this line:

```
include=/etc/opt/remi/php70/php-fpm.d/*.conf
```

This line prompts PHP-FPM to load individual pool definition files located in the `pool.d/` (Ubuntu) or `php-fpm.d/` (CentOS) directory. Navigate into this directory, and you should see one file named *www.conf*. This is the configuration file for the default PHP-FPM pool named `www`. Open this file in your preferred text editor.



Each PHP-FPM pool configuration begins with a `[` character, the pool name, and a `]` character. The default PHP-FPM pool configuration, for example, begins with `[www]`.

Each PHP-FPM pool runs as the operating system user and group that you specify. I prefer to run each PHP-FPM pool as a unique nonroot user to help me identify each PHP application's PHP-FPM processes on the command line with the `top` or `ps aux` commands. This is a good habit, too, because each PHP-FPM pool's processes are inherently sandboxed by the permissions available to their operating system user and group.

We'll configure the default `www` PHP-FPM pool to run as the `deploy` user and group. If you haven't already, open the `www` PHP-FPM pool configuration file in your preferred text editor. Here are the settings I recommend you change from their default values:

`user = deploy`
: The system user that owns this PHP-FPM pool's child processes. Set this to your PHP application's nonroot operating system user name.

`group = deploy`
: The system group that owns this PHP-FPM pool's child processes. Set this to your PHP application's nonroot operating system group name.

`listen = 127.0.0.1:9000`
: The IP address and port number on which this PHP-FPM pool listens for and accepts inbound requests from nginx. The value `127.0.0.1:9000` instructs this specific PHP-FPM pool to listen for incoming connections on local port `9000`. I use port `9000`, but you can use any nonprivileged port number (any port number greater than 1024) that is not already in use by another system process. We'll revisit this setting when we configure our nginx virtual host.

`listen.allowed_clients = 127.0.0.1`
: The IP address(es) that can send requests to this PHP-FPM pool. For security reasons, I set this to `127.0.0.1`. This means that only the current machine can forward requests to this PHP-FPM pool. This setting might be commented out by default. Uncomment this setting if necessary.

`pm.max_children = 51`
: This value sets the total number of PHP-FPM pool processes that can exist at any given time. There is no correct value for this setting. You should test your PHP application, determine how much memory each individual PHP process uses, and set this to the total number of PHP processes that your machine's available memory can accommodate. Most small to medium-sized PHP applications often use between 5 MB and 15 MB of memory for each individual PHP process (your mileage may vary). Assuming we are on a machine with 512 MB of memory available to this PHP-FPM pool, we can set this value to `512MB total / 10MB per process`, or `51` processes.

`pm.start_servers = 3`
: The number of PHP-FPM pool processes that are available immediately when PHP-FPM starts. Again, there is no correct value for this setting. For most small or medium-sized PHP applications, I recommend a value of `2` or `3`. This ensures that your PHP application's initial HTTP requests don't have to wait for PHP-

FPM to initialize PHP-FPM pool processes. Two or three processes are already ready and waiting.

`pm.min_spare_servers = 2`
: The smallest number of PHP-FPM pool processes that exist when your PHP application is idle. This will typically be in the same ballpark as your `pm.start_servers` setting, and it ensures that new HTTP requests don't have to wait for PHP-FPM to initialize new pool processes to handle new requests.

`pm.max_spare_servers = 4`
: The largest number of PHP-FPM pool processes that exist when your PHP application is idle. This will typically be a bit more than your `pm.start_servers` setting, and it ensures that new HTTP requests don't have to wait for PHP-FPM to initialize new pool processes to handle new requests.

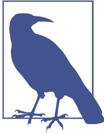
`pm.max_requests = 1000`
: The maximum number of HTTP requests that each PHP-FPM pool process handles before being recycled. This setting helps us avoid accumulating memory leaks caused by poorly coded PHP extensions or libraries. I recommend a value of `1000`, but you should tweak this based on your own application's needs.

`slowlog = /path/to/slowlog.log`
: The absolute filesystem path to a log file that records information about HTTP requests that take longer than *{n}* number of seconds to process. This is helpful for identifying and debugging bottlenecks in your PHP applications. Bear in mind, this PHP-FPM pool's `user` or `group` must have permission to write to this file. The value `/path/to/slowlog.log` is an example; replace this value with your own file path.

`request_slowlog_timeout = 5s`
: The length of time after which the current HTTP request's backtrace is dumped to the log file specified by the `slowlog` setting. The value you choose depends on what you consider to be a slow request. A value of `5s` is a reasonable value to start with.

After you edit and save the PHP-FPM configuration file, restart the PHP-FPM master process with this command:

```
# Ubuntu
sudo service php7.0-fpm restart

# CentOS
sudo systemctl restart php70-php-fpm.service
```



Read more about PHP-FPM pool configuration at *https://php.net/manual/install.fpm.configuration.php*.

Nginx

Nginx (pronounced *in gen ex*) is a web server similar to Apache, but it's much simpler to configure and often uses less system memory. I don't have time to dig into Nginx in detail, but I do want to show you how to install Nginx on your server and forward appropriate HTTP requests to your PHP-FPM pool.

Install

The simplest way to install Nginx is with your operating system's native package manager.

Ubuntu

On Ubuntu, install Nginx with a PPA. This is an Ubuntu-specific term for a prepackaged archive maintained by the Nginx community:

```
sudo add-apt-repository ppa:nginx/stable;
sudo apt-get update;
sudo apt-get install nginx;
```

CentOS

On CentOS, install Nginx using the same EPEL third-party software repository we added earlier. The default CentOS software repositories might not have the latest nginx version:

```
sudo yum install nginx;
sudo systemctl enable nginx.service;
sudo systemctl start nginx.service;
```

Virtual Host

Next, we'll configure an Nginx *virtual host* for our PHP application. A virtual host is a group of settings that tell Nginx our application's domain name, where the PHP application lives on the filesystem, and how to forward HTTP requests to the PHP-FPM pool.

First, we must decide where our application lives on the filesystem. The PHP application files must live in a filesystem directory that is readable and writable by the non-root `deploy` user. For this example, I'll place application files in the */home/deploy/apps/example.com/current* directory. We'll also need a directory to store application

log files. I'll place log files in the */home/deploy/apps/logs* directory. Use these commands to create the directories and assign correct permissions:

```
mkdir -p /home/deploy/apps/example.com/current/public;
mkdir -p /home/deploy/apps/logs;
chmod -R +rx /home/deploy;
```

Place your PHP application in the */home/deploy/apps/example.com/current* directory. The Nginx virtual host configuration assumes your PHP application has a *public/* directory; this is the virtual host document root.

Each Nginx virtual host has its own configuration file. If you use Ubuntu, create the */etc/nginx/sites-available/example.conf* configuration file. If you use CentOS, create the */etc/nginx/conf.d/example.conf* configuration file. Open the *example.conf* configuration file in your preferred text editor.

Nginx virtual host settings live inside a `server {}` block. Here is the complete virtual host configuration file:

```
server {
    listen 80;
    server_name example.com;
    index index.php;
    client_max_body_size 50M;
    error_log /home/deploy/apps/logs/example.error.log;
    access_log /home/deploy/apps/logs/example.access.log;
    root /home/deploy/apps/example.com/current/public;

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    location ~ \.php {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param SCRIPT_NAME $fastcgi_script_name;
        fastcgi_index index.php;
        fastcgi_pass 127.0.0.1:9000;
    }
}
```

Copy and paste this code into the *example.conf* virtual host configuration file. Make sure you update the `server_name` setting and swap the `error_log`, `access_log`, and `root` paths with appropriate values. Here's a quick explanation of each virtual host setting:

`listen`
: The port number on which Nginx listens for inbound HTTP requests. In most cases, this is port `80` for HTTP traffic or port `443` for HTTPS traffic.

`server_name`
: The domain name that identifies this virtual host. Change this to your application's domain name, and ensure the domain name points at your server's IP address. Nginx sends an HTTP request to this virtual host if the request's `Host:` header matches the virtual host's `server_name` value.

`index`
: The default files served if none is specified in the HTTP request URI.

`client_max_body_size`
: The maximum HTTP request body size accepted by Nginx for this virtual host. If the request body size exceeds this value, Nginx returns a `HTTP 4xx` response.

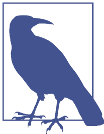
`error_log`
: The filesystem path to this virtual host's error log file.

`access_log`
: The filesystem path to this virtual host's access log file.

`root`
: The document root directory.

There are also two `location` blocks. These tell Nginx how to handle HTTP requests that match specific URL patterns. The first `location / {}` block uses a `try_files` directive that looks for real files that match the request URI. If a file is not found, it looks for a directory that matches the request URI. If a directory is not found, it rewrites the HTTP request URI to */index.php* and appends the query string if available. The rewritten URL, or any request whose URI ends with *.php*, is managed by the `location ~ \.php {}` block.

The `location ~ \.php {}` block forwards HTTP requests to our PHP-FPM pool. Remember how we set up our PHP-FPM pool to listen for requests on port `9000`? This block forwards PHP requests to port 9000, and the PHP-FPM pool takes over.



There are a few extra lines in the `location ~ \.php {}` block. These lines prevent potential remote code execution attacks.

On Ubuntu, we must symlink the virtual host configuration file into the */etc/nginx/sites-enabled/* directory with this command:

```
sudo ln -s /etc/nginx/sites-available/example.conf \
    /etc/nginx/sites-enabled/example.conf;
```

Finally, restart Nginx with this command:

```
# Ubuntu
sudo service nginx restart

# CentOS
sudo systemctl restart nginx.service
```

Your PHP application is up and running! There are many ways to configure Nginx. I've included only the most essential Nginx settings in this chapter because this is a PHP book, not an Nginx book. You can learn more about Nginx configuration at any of these helpful resources:

- *http://nginx.org/*
- *https://github.com/h5bp/server-configs-nginx*
- *https://serversforhackers.com/editions/2014/03/25/nginx/*

Automate Server Provisioning

Server provisioning is a lengthy process. It's also not a fun process, especially if you manually provision many servers. Fortunately, there are tools available that help automate server provisioning. Some popular server provisioning tools are:

- Ansible
- Puppet
- Chef
- SaltStack

Each tool is different, but they all accomplish the same goal—they automatically provision new servers based on your exact specifications. If you are responsible for multiple servers, I strongly encourage you to explore provisioning tools, because they save a ton of time.

Delegate Server Provisioning

There are online services, too, that perform server provisioning on your behalf. An example service is Forge by Taylor Otwell. I was a Forge beta tester, and it really is a helpful service. Forge can provision multiple servers on Linode, Digital Ocean, and other popular VPS providers.

Each server provisioned by Forge is automatically secured using the same security practices I demonstrated earlier. Forge automatically installs an Nginx and PHP-FPM software stack. Forge also simplifies PHP application deployment, SSL certificate installation, CRON task creation, and other mundane or confusing system adminis-

tration tasks. I highly recommend Forge if system administration isn't your cup of tea.

Further Reading

I find system administration fascinating. I don't want to do it as a full-time job, but I enjoy tinkering on the command line. The best system administration learning resource for developers, in my opinion, is Servers for Hackers by Chris Fidao.

What's Next

In this chapter we discussed how to provision a server to run PHP applications. Next we'll talk about how to tune your server to eke out maximum performance for your PHP application.

CHAPTER 3
Tuning

By this point, your PHP application should be running alongside Nginx with its own PHP-FPM process pool. We're not done yet, though. We should *tune* PHP's configuration with settings appropriate for your application and production server. Default PHP installations are like an average dress or suit you find at your local department store; they fit, but they don't fit well. A tuned PHP installation is a custom tailored dress or suit prepared with your exact measurements.

Don't get too excited. PHP tuning is not a universal cure for application performance. Bad code is still bad code. For example, PHP tuning cannot solve poorly written SQL queries or unresponsive API calls. However, PHP tuning is a low-hanging fruit that can improve PHP efficiency and application performance.

The php.ini File

The PHP interpreter is configured and tuned with a file named *php.ini*. This file can live in one of several directories on your operating system. If you run PHP with PHP-FPM, as I demonstrated earlier, you can find the *php.ini* configuration file at */etc/php/7.0/fpm/php.ini*. Oddly enough, this *php.ini* file does not control the PHP interpreter used when you invoke `php` on the command line. PHP on the command line uses its own *php.ini* file often located at */etc/php/7.0/cli/php.ini*. If you built PHP from source, the *php.ini* location is likely beneath the `$PREFIX` directory specified when you configured the PHP source files. I'll assume you're running PHP with PHP-FPM as described, but all of these optimizations are applicable to any *php.ini* file.

Scan your *php.ini* file for best security practices with the PHP Iniscan tool, written by Chris Cornutt.

The *php.ini* file uses the INI format. You can learn about the INI format on Wikipedia.

Memory

My first concern when running PHP is how much memory each PHP process consumes. The `memory_limit` setting in the *php.ini* file determines the maximum amount of system memory that can be used by a single PHP process.

The default value is `128M`, and this is probably fine for most small to medium-sized PHP applications. However, if you are running a tiny PHP application, you can save system resouces by lowering this value to something like `64M`. If you are running a memory-intensive PHP application (e.g., a Drupal website), you may see improved performance with a higher value like `256M`. The value you choose is dictated by the amount of available system memory. Figuring out how much memory to allocate to PHP is more an art than a science. These are the questions I ask myself to determine my PHP memory limit and the number of PHP-FPM processes I can afford:

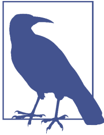
What is the total amount of memory I can allocate for PHP?
: First, I determine how much system memory I can allocate for PHP. For example, I may be working with a Linode virtual machine with 2 GB of total memory. However, other processes (e.g., Nginx, MySQL, or memcache) might run on the same machine and consume memory of their own. I think I can safely set aside 512 MB of memory for PHP.

How much memory, on average, is consumed by a single PHP process?
: Next, I determine how much memory, on average, is consumed by a single PHP process. This requires me to monitor process memory usage. If you live in the command line, then you can run `top` to see realtime stats for running processes. You can also invoke the `memory_get_peak_usage()` PHP function at the tail end of a PHP script to output the maximum amount of memory consumed by the current script. Either way, run the same PHP script several times (to warm caches) and take the average memory consumption. I often find PHP processes consume between 5–20 MB of memory (your mileage may vary). If you are working with file uploads, image data, or a memory-intensive application, this value will obviously be higher.

How many PHP-FPM processes can I afford?
: I have 512 MB of total memory allocated for PHP. I determine that each PHP process, on average, consumes about 15 MB of memory. I divide the total memory by the amount of memory consumed by each PHP process, and I determine I

can afford 34 PHP-FPM processes. This value is an estimate and should be refined with experimentation.

Do I have enough system resources?
: Finally, I ask myself if I believe I have sufficient system resources to run my PHP application and handle the expected web traffic. If yes, awesome. If no, I need to upgrade my server with more memory and return to the first question.



Use Apache Bench or Siege to stress-test your PHP applications under production-like conditions. If your PHP application does not have sufficient resources, it's wise to figure this out *before* you take your application into production.

Zend OPcache

After I figure out my memory allocation, I configure the PHP Zend OPcache extension. This is an *opcode cache*. What's an opcode cache? Let's first examine how a typical PHP script is processed for every HTTP request. First, Nginx forwards an HTTP request to PHP-FPM, and PHP-FPM assigns the request to a child PHP process. The PHP process finds the appropriate PHP scripts, it reads the PHP scripts, it compiles the PHP scripts into an opcode (or bytecode) format, and it executes the compiled PHP opcode to generate an HTTP response. The HTTP response is returned to Nginx, and Nginx returns the HTTP response to the HTTP client. This is a lot of overhead for every HTTP request.

We can speed this up by *caching* the compiled opcode for each PHP script. Then we can read and execute precompiled opcode from cache instead of finding, reading, and compiling PHP scripts for each HTTP request. The Zend OPcache extension is built into PHP 5.5.0+. Here are my *php.ini* settings to configure and optimize the Zend OPcache extension:

```
opcache.memory_consumption = 64
opcache.interned_strings_buffer = 16
opcache.max_accelerated_files = 4000
opcache.validate_timestamps = 1
opcache.revalidate_freq = 0
opcache.fast_shutdown = 1
```

`opcache.memory_consumption = 64`
: The amount of memory (in megabytes) allocated for the opcode cache. This should be large enough to store the compiled opcode for all of your application's PHP scripts. If you have a small PHP application with few scripts, this can be a lower value like 16 MB. If your PHP application is large with many scripts, use a larger value like 64 MB.

`opcache.interned_strings_buffer = 16`
: The amount of memory (in megabytes) used to store interned strings. What the heck is an interned string? That was my first question, too. The PHP interpreter, behind the scenes, detects multiple instances of identical strings and stores the string in memory *once* and uses pointers whenever the string is used again. This saves memory. By default, PHP's string interning is isolated in each PHP process. This setting lets all PHP-FPM pool processes store their interned strings in a shared buffer so that interned strings can be referenced across multiple PHP-FPM pool processes. This saves even more memory. The default value is 4 MB, but I prefer to bump this to 16 MB.

`opcache.max_accelerated_files = 4000`
: The maximum number of PHP scripts that can be stored in the opcode cache. You can use any number between 200 and 100000. I use `4000`. Make sure this number is larger than the number of files in your PHP application.

`opcache.validate_timestamps = 1`
: When this setting is enabled, PHP checks PHP scripts for changes on the interval of time specified by the `opcache.revalidate_freq` setting. If this setting is disabled, PHP does not check PHP scripts for changes, and you must clear the opcode cache manually. I recommend you enable this setting during development and disable this setting during production.

`opcache.revalidate_freq = 0`
: How often (in seconds) PHP checks compiled PHP files for changes. The benefit of a cache is to avoid recompiling PHP scripts on each request. This setting determines how long the opcode cache is considered fresh. After this time interval, PHP checks PHP scripts for changes. If PHP detects a change, PHP recompiles and recaches the script. I use a value of `0` seconds. This value requires PHP to revalidate PHP files on every request *if and only if* you enable the `opcache.validate_timestamps` setting. This means PHP revalidates files on every request during development (a good thing). This setting is moot during production because the `opcache.validate_timestamps` setting is disabled anyway.

`opcache.fast_shutdown = 1`
: This prompts the opcache to use a faster shutdown sequence by delegating object deconstruction and memory release to the Zend Engine memory manager. Documentation is lacking for this setting. All you need to know is *turn this on*.

File Uploads

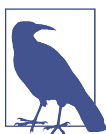
Does your PHP application accept file uploads? If not, turn off file uploads to improve application security. If your application does accept file uploads, it's best to set a maximum upload filesize that your application accepts. It's also best to set a

maximum number of uploads that your application accepts at one time. These are the *php.ini* settings I use for my own applications:

```
file_uploads = 1
upload_max_filesize = 10M
max_file_uploads = 3
```

By default, PHP allows up to 20 uploads in a single request. Each uploaded file can be up to 2 MB in size. You probably don't need to allow 20 uploads at once; I only allow three uploads in a single request, but change this setting to a value that makes sense for your application.

If my PHP applications accept file uploads, they often need to accept files much larger than 2 MB. I bump the `upload_max_filesize` setting to `10M` or higher based on each application's requirements. Don't set this to something too large, otherwise your web server may complain about the HTTP request having too large a body or timing out.

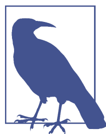


If you accept very large file uploads, be sure your web server is configured accordingly. You may need to adjust the `client_max_body_size` setting in your Nginx virtual host configuration *in addition to* your *php.ini* file.

Max Execution Time

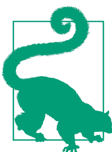
The `max_execution_time` setting in your *php.ini* file determines the maximum length of time that a single PHP process can run before terminating. By default, this is set to 30 seconds. You don't want PHP processes running for 30 seconds. We want our applications to be super-fast (measured in milliseconds). I recommend you change this to 5 seconds:

```
max_execution_time = 5
```



You can override this setting on a per-script basis with the `set_time_limit()` PHP function.

What if my PHP script needs to run a long time? you ask. It shouldn't. The longer PHP runs, the longer your web application visitors must wait for a response. If you have long-running tasks (e.g., resizing images or generating reports), offload those tasks to a separate worker process.

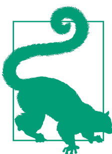


I use the `exec()` PHP function to invoke the `at` bash command. This lets me fork separate nonblocking processes that do not delay the current PHP process. If you use the `exec()` PHP function, it is your responsibility to escape shell arguments with the `escapeshellarg` PHP function.

Assume we need to run a report and generate a PDF file with the results. This task may take 10 minutes to complete. Surely we don't want the PHP request to sit around for 10 minutes. Instead, we create a separate PHP file called *create-report.php* that will chug along for 10 minutes and eventually generate our report. However, our web application will take only milliseconds to spin off a separate background process and return an HTTP response, like this:

```
<?php
exec('echo "create-report.php" | at now');
echo 'Report pending...';
```

The standalone *create-report.php* script runs in a separate background process; it can update a database or email the report recipient upon completion. There is absolutely no reason why the primary PHP script should hold up the user experience for long-running tasks.



If you find yourself spawning a lot of background processes, you may be better served with a dedicated job queue. PHP Resque is a great job queue manager based on the original Resque job queue manager from GitHub.

Session Handling

PHP's default session handler can slow down larger applications because it stores session data on disk. This creates unnecessary file I/O that takes time to read and write. Instead, offload session handling to a faster in-memory data store like Memcached or Redis. This has the added benefit of future scalability. If your session data is stored on disk, this prevents you from scaling PHP across additional servers. If your session data is, instead, stored on a central Memcached or Redis data store, it can be accessed from any number of distributed PHP-FPM servers.

Install the PECL Memcached extension to access a Memcached datastore from PHP. You can now change PHP's default session store to Memcached by adding these lines to your *php.ini* file:

```
session.save_handler = 'memcached'
session.save_path = '127.0.0.2:11211'
```

Output Buffering

Networks are more efficient when sending more data in fewer chunks, rather than less data in more chunks. In other words, deliver content to your visitor's web browser in fewer pieces to reduce the total number of HTTP requests.

This is why you enable PHP output buffering. By default, PHP's output buffer is enabled (except on the command line). PHP's output buffer collects up to 4,096 bytes before flushing its contents back to the web server. Here are my recommended *php.ini* settings:

```
output_buffering = 4096
implicit_flush = false
```

If you change the output buffer size, make sure its value is a multiple of 4 (for 32-bit systems) or 8 (for 64-bit systems).

Realpath Cache

PHP maintains a cache of file paths that are used by your PHP application so it does not have to continually search the include path each time it includes or requires a file. This cache is called the *realpath cache*. If you are running a large PHP application that uses a lot of separate files (Drupal, Composer components, etc.), you can realize better performance by increasing the size of PHP's realpath cache.

The default realpath cache size is `16k`. It's not obvious how to figure out the exact size you need, but here's a trick you can use. First, bump the realpath cache size to something obnoxiously large, like `256k`. Then output the actual realpath cache size at the tail end of a PHP script with `print_r(realpath_cache_size());`. Change your realpath cache size to this actual value. You can set the realpath cache size in your *php.ini* file:

```
realpath_cache_size = 64k
```

Up Next

We've got a server firing on all cylinders, and we're ready to deploy our PHP application into production. In the next chapter we'll discuss several strategies to automate PHP application deployment.

CHAPTER 4
Deployment

We've got a provisioned server running Nginx and PHP-FPM. Now we need to deploy our PHP application to a production server. There are many ways to push code into production. FTP was a popular way to deploy PHP code back when PHP developers first started banging rocks together. FTP still works, but today there are safer and more predictable deployment strategies. This chapter shows you how to use modern tools to automate deployment in a simple, predictable, and reversible way.

Version Control

I assume you are using version control, right? If you are, good job. If you aren't, stop what you are doing and version control your code. I prefer to version control my code with Git, but other version control software like Mercurial works, too. I use Git because it's what I know, and it works seamlessly with popular online repositories like Bitbucket and GitHub.

Version control is an invaluable tool for PHP application developers because it lets us track changes to our codebase. We can tag points in time as a release, we can roll back to a previous state, and we can experiment with new features on separate branches that do not affect our production code. More important, version control helps us automate PHP application deployment.

Automate Deployment

It is important that you automate application deployment so that it becomes a simple, predictable, and reversible process. The last thing you want to worry about is a complicated deployment process. Complicated deployments are scary, and scary things are used less often.

Make It Simple

Instead, make your deployment process a simple one-line command. A simple deployment process is less scary, and that means you're more likely to push code to production.

Make It Predictable

Make your deployment process predictable. A predictable process is even less scary because you know exactly what it is going to do. It should not have unexpected side effects. If it runs into an error, it aborts the deployment process and leaves the existing codebase in place.

Make It Reversible

Make your deployment process reversible. If you accidently push bad code into production, it should be a simple one-line command to roll back to the previous stable codebase. This is your safety net. A reversible deployment process should make you excited—not afraid—to push code into production. If you screw up, just roll back to the previous release.

Capistrano

Capistrano is software that automates application deployment in a simple, predictable, and reversible way. Capistrano runs on your local machine and talks with remote servers via SSH. Capistrano was originally written to deploy Ruby applications, but it's just as useful for any programming language—including PHP.

How It Works

Capistrano deploys your PHP application to a remote server by issuing SSH commands from your local workstation to the remote server. Capistrano organizes application deployments in their own directories on the remote server. Capistrano maintains five or more application deployment directories in case you must roll back to an earlier release. Capistrano also creates a *current/* directory that is a symlink to the current application deployment's directory. Your production server's Capistrano-managed directory structure might look like Example 4-1.

Example 4-1. Example directory structure

```
/
    home/
        deploy/
            apps/
                my_app/
```

```
current/
releases/
    release1/
    release2/
    release3/
    release4/
    release5/
```

When you deploy a new application release to production, Capistrano first retrieves the latest version of your application code from its Git repository. Next, Capistrano places the application code in a new release directory. Finally, Capistrano symlinks the *current/* directory to the new release directory. When you ask Capistrano to roll back to a previous release, Capistrano points the *current/* directory symlink to a previous release directory. Capistrano is an elegant and simple deployment solution that makes PHP application deployments simple, predictable, and reversible.

Install

Install Capistrano on your local machine. Do *not* install Capistrano on your remote servers. You'll need `ruby` and `gem`, too. OS X users already have these. Linux users can install `ruby` and `gem` with their respective package managers. After you install `ruby` and `gem`, install Capistrano with this command:

```
gem install capistrano
```

Configure

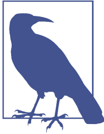
After you install Capistrano, you must initialize your project for Capistrano. Open a terminal, navigate to your project's topmost directory, and run this command:

```
cap install
```

This command creates a file named *Capfile*, a directory named *config/*, and a directory named *lib/*. Your project's topmost directory should now have these files and directories:

```
Capfile
config/
    deploy/
        production.rb
        staging.rb
    deploy.rb
lib/
    capistrano/
        tasks/
```

The *Capfile* file is Capistrano's central configuration file, and it aggregates the configuration files located in the *config/* directory. The *config/* directory contains configuration files for each remote server environment (e.g., testing, staging, or production).



Capistrano configuration files are written in the Ruby language. However, they are still easy to edit and understand.

By default, Capistrano assumes you have multiple environments for your application. For example, you might have separate staging and production environments. Capistrano provides a separate configuration file for each environment in the *config/deploy/* directory. Capistrano also provides the *config/deploy.rb* configuration file, which contains settings common to all environments.

In each environment, Capistrano has the notion of server *roles*. For example, your production environment may have a front-facing web server (the `web` role), an application server (the `app` role), and a database server (the `db` role). Only the largest applications necessitate this architecture. Smaller PHP applications generally use only one machine that runs the web server (nginx), application server (PHP-FPM), and database server (MariaDB).

For this demonstration, I'm only going to use Capistrano's `web` role and ignore its `app` and `db` roles. Capistrano's roles let you organize tasks to be executed only on servers that belong to a given role. This isn't something we're going to worry about here. However, I am going to respect Capistrano's notion of server environments. This demonstration will use the `production` environment, but the following steps are equally applicable to other environments (e.g., `staging` or `testing`).

The config/deploy.rb file

Let's look at the *config/deploy.rb* file. This configuration file contains settings common to all environments (e.g., `staging` *and* `production`). Most of our Capistrano configuration settings go in this file. Open the *config/deploy.rb* file in your preferred text editor and update these settings:

`:application`
: This is the name of your PHP application. It should contain only letters, numbers, and underscores.

`:repo_url`
: This is your Git repository URL. This URL must point to a Git repository, and the repository must be accessible from your remote server.

`:deploy_to`
: This is the absolute directory path on your remote server in which your PHP application is deployed. This would be */home/deploy/apps/my_app* as shown in Example 4-1.

`:keep_releases`
: This is the number of old releases that should be retained in case you want to roll back your application to an earlier version.

The config/deploy/production.rb file

This file contains settings only for your production environment. This file defines the production environment roles, and it lists the servers that belong to each role. We're only using the `web` role, and we have only one server that belongs to this role. Let's use the server we provisioned in Chapter 2. Update the entire *config/deploy/production.rb* file with this content. Make sure you replace the example IP address:

```
role :web, %w{deploy@123.456.78.90}
```

Authenticate

Before we deploy our application with Capistrano, we must establish authentication between our local computer and our remote servers, and between our remote servers and the Git repository. We already discussed how to set up SSH key-pair authentication between our local computer and remote server. You should also establish SSH key-pair authentication between your remote servers and the Git repository.

Use the same instructions we discussed earlier to generate an SSH public and private keypair on each remote server. The Git repository should be given each remote server's public key; both GitHub and Bitbucket let you add multiple public SSH keys to your user account. Ultimately, you must be able to clone the Git repository to your remote servers without a password.

Prepare the Remote Server

We're almost ready to deploy our application. First, we need to prepare our remote server. Log in to your remote server with SSH and create the directory in which we'll deploy our PHP application. This directory must be readable and writable by the `deploy` user. I like to create a directory for my applications in the `deploy` user's home directory, like this:

```
/
    home/
        deploy/
            apps/
                my_app/
```

Virtual host

Capistrano symlinks the *current/* directory to the current application release directory. Update your web server's virtual host document root directory so that it points to Capistrano's *current/* directory. Given this filesystem diagram, your virtual host

document root might become */home/deploy/apps/my_app/current/public/*; this assumes your PHP application contains a *public/* directory that serves as the document root. Restart your web server to load your virtual host configuration changes.

Software dependencies

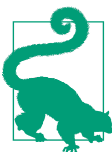
Your remote server doesn't need Capistrano, but it does need Git. It also needs any software required to run your PHP application. You can install Git with these commands:

```
# Ubuntu
sudo apt-get install git;

# CentOS
sudo yum install git;
```

Capistrano Hooks

Capistrano allows us to run our own commands at specific moments (or *hooks*) during application deployment. Many PHP developers manage application dependencies with Composer. We can install Composer dependencies during each Capistrano deployment with a Capistrano hook. Open the *config/deploy.rb* file in your preferred text editor and append this Ruby code:

```
namespace :deploy do
  desc "Build"
  after :updated, :build do
      on roles(:web) do
          within release_path do
            execute :composer, "install --no-dev --quiet --optimize-autoloader"
          end
      end
  end
end
```



If your project uses the Composer dependency manager, make sure Composer is installed on your remote servers.

Our application's dependencies are now installed automatically after each production deployment. You can read more about Capistrano hooks on the Capistrano website.

Deploy Your Application

Now's the fun part! Make sure you've committed and pushed your most recent application code to your Git repository. Then open a terminal on your local computer and

navigate to your application's topmost directory. If you've done everything correctly, you can deploy your PHP application with this one-line command:

```
cap production deploy
```

Roll Back Your Application

On the off chance you deploy bad code to your production environment, you can roll back to a previous release with this one-line command:

```
cap production deploy:rollback
```

Further Reading

I've only scratched the surface. Capistrano has many more features that further streamline your deployment workflow. Capistrano is my favorite deployment tool, but there are many other tools available, including:

- Deployer
- Magallanes
- Rocketeer

What's Next

We've provisioned a server, and we've automated our PHP application deployments with Capistrano. Next we'll discuss how to ensure our PHP applications run as expected. To do this, we'll use *testing* and *profiling*.

CHAPTER 5
Testing

Testing is an important part of PHP application development, but it is often neglected. I think many PHP developers don't test because they consider testing an unnecessary burden that requires too much time for too few benefits. Other developers may not know *how* to test, because there are a large number of testing tools and an overwhelming learning curve.

In this chapter I hope to dispel these misunderstandings. I want you to feel comfortable and excited about testing your PHP code. I want you to consider testing an integral part of your workflow that happens at the beginning, middle, and end of the application development process.

Why Do We Test?

We write tests to ensure that our PHP applications work, and continue to work, according to our expectations. It's as simple as that. How often have you been afraid to deploy an application into production? Before I started testing my code, I was terrified to push a release into production. Would my code work? Would it break? All I could do was cross my fingers and hope for the best. This is no way to code. It's scary and stressful, and it usually ends in frustration. Tests, however, mitigate uncertainty, and they let us write and deploy code with confidence.

Your pointy-haired boss may argue that there isn't enough time to write tests. After all, time is money. This is shortsighted. Installing a testing infrastructure and writing tests takes time, but this is a wise investment that pays dividends into the future. Tests help us write code that works well the first time. Tests let us continuously iterate without breaking old code. We may move forward at a slower pace than if we didn't use tests, but we won't waste countless development hours in the future

troubleshooting and refactoring bugs that were overlooked. In the long term, tests save money, prevent downtime, and inspire confidence.

When Do We Test?

I see many PHP developers write tests as an afterthought. These developers know testing is important, but they consider tests as something they *must* do instead of something they want to do. These developers often push testing to the very end of the application development process. They bang out a few passing tests to satisfy their management team and call it a day. This is wrong. Tests should be a foreground concern before development, during development, and after development.

Before

Install and configure your testing tools before you develop your application. It doesn't matter which testing tools you choose. Install them as if they are a vital application dependency. This makes it physically and mentally easier to test your application during development. This is also a good time to meet with your project manager to define higher-level application behavior.

During

Write and run tests as you build each piece of your application. Did you just add a new PHP class? Test it now, because you probably won't test it later. Testing while you develop helps you build confident and stable code, and it also helps you quickly find and refactor new code that breaks existing functionality.

After

You probably won't anticipate and test all of your application's behaviors during development. If you find a bug after you launch your application, write a new test to ensure that your bug fix works correctly. Tests are not a once-and-done thing. Tests are continuously modified and improved, just like the application itself. If you update your application's code, be sure you also update the affected tests.

What Do We Test?

We test the smallest pieces of our application. A PHP application, on a microcosmic scale, has PHP classes, methods, and functions. We should test each public class, method, and function to ensure it behaves as we expect in isolation. If we know each piece works well on its own, we can be confident it also works well when integrated into the whole application. These tests are called *unit tests*.

Unfortunately, testing each individual piece does not guarantee it works correctly with the whole application. This is why we also test our application at a macrocosmic scale with automated testing tools that verify our application's higher-level behaviors. These tests are called *functional* tests.

How Do We Test?

We know why, when, and what to test. More important, let's chat about *how* we test code. There are several popular ways PHP developers approach testing. Some developers prefer unit tests. Some developers prefer test-driven development (TDD). And other developers prefer behavior-driven development (BDD). *These are not mutually exclusive.*

Unit Tests

The most popular approach to PHP application testing is unit testing. As I described previously, unit tests certify individual classes, methods, and functions in isolation from the larger application. The de facto standard PHP unit testing framework is PHPUnit, written by Sebastian Bergmann. Sebastian's PHPUnit framework adheres to the xUnit test architecture.

There are alternative PHP unit testing frameworks, like PHPSpec, available for you to use, too. However, most popular PHP frameworks provide PHPUnit tests. It's vital that you know how to read, write, and run PHPUnit tests if you intend to contribute to or release PHP components. I'll show you how to install, write, and run PHP unit tests at the end of this chapter.

Test-Driven Development (TDD)

Test-driven development means you write tests *before* you write application code. These tests purposefully fail and describe how your application *should* behave. As you build application functionality, your tests will eventually run successfully. TDD helps you build with a purpose; you know ahead of time what you will build and how it should work.

This does not mean that you must write all of your application tests before you write any code. Instead, write a few tests and then build the related functionality. Write tests and build. Write tests and build. TDD is iterative. Move forward in small sprints until your application is complete.

Behavior-Driven Development (BDD)

Behavior-driven development means that you write stories that describe how your application behaves. There are two types of BDD: SpecBDD and StoryBDD.

SpecBDD is a type of unit test that uses a fluid and human-friendly language to describe your application's implementation. SpecBDD accomplishes the same goal as alternative unit testing tools like PHPUnit. Unlike PHPUnit's xUnit architecture, SpecBDD tests use *human-readable stories* to describe behavior. For example, a PHPUnit test might be named `testRenderTemplate()`. An equivalent SpecBDD test might be named `itDrawsTheHomePage()`. The same SpecBDD test might use helper methods named `$this->shouldReturn()`, `$this->shouldBe()`, and `$this->shouldThrow()`. SpecBDD tests use a language that is much easier to read and understand than alternative xUnit tools. The most popular SpecBDD testing tool is PHPSpec.

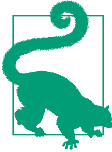
StoryBDD tools use the same human-friendly stories as SpecBDD tests. StoryBDD tools, however, are more concerned with higher-level behavior than with lower-level implementation. For example, a StoryBDD test confirms that your code creates and emails a PDF report. A SpecBDD test, on the other hand, confirms that a specific PDF generator class method correctly renders a PDF file for a given set of input parameters. The difference is scope. StoryBDD resembles something a project manager would write (e.g., "this should generate and email me a report"). A SpecBDD test resembles something a developer would write (e.g., "this class method should receive an array of data and write it to this PDF file"). StoryBDD and SpecBDD testing tools are not mutually exclusive. They are often used together to build a more comprehensive set of tests. You'll often sit with your project manager to write generic StoryBDD tests that define your application's generic behavior, and then you'll write SpecBDD tests when you design and build your application's implementation. The most popular StoryBDD testing tool is Behat.

Write StoryBDD tests that describe your business logic and not a specific implementation. A good StoryBDD test confirms "a shopping cart total increases when I add a product to the cart." A bad StoryBDD test confirms "a shopping cart total increases when I send an HTTP `PUT` request to the */cart* URL with the body `product_id=1&quantity=2`." The first test is generic and describes only the high-level business logic. The second test is too specific and describes a particular implementation.

PHPUnit

Let's talk about how to install, write, and run PHPUnit tests. It takes a bit of work to get the infrastructure in place, but it's dead simple to write and run your PHPUnit tests afterward. Before we dig too deep into PHPUnit, let's quickly review some vocabulary. Your PHPUnit tests are grouped into *test cases*, and your test cases are grouped into *test suites*. PHPUnit runs your test suites with a *test runner*.

A test case is a single PHP class that extends the `PHPUnit_Framework_TestCase` class. Each test case contains public methods whose names begin with `test`; these methods are individual tests that assert specific scenarios to be true. Each assertion can pass or fail. You want all assertions to pass.



A test case class name must end with `Test`, and its filename must end with *Test.php*. A hypothetical test case class name is `FooTest`, and that class lives in a file named *FooTest.php*.

A test suite is a collection of related test cases. If you are working on a single PHP component, oftentimes you'll only ever have a single test suite. If you are testing a larger PHP application with many different subsystems or components, you may find it best to organize tests into multiple test suites.

A test runner is exactly what it sounds like. It is a way for PHPUnit to run your test suites and output the result. The default PHPUnit test runner is the *command-line runner* that is invoked with the `phpunit` command in your terminal application.

Directory Structure

Here's how I prefer to organize my PHP projects. The topmost project directory has a *src/* directory where I keep my source code. It also has a *tests/* directory where I keep my tests. Here's an example directory structure:

```
src/
tests/
    bootstrap.php
composer.json
phpunit.xml
.travis.yml
```

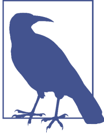
src/
: This directory contains my PHP project's source code (i.e., PHP classes).

tests/
: This directory contains my PHP project's PHPUnit tests. This directory contains a *bootstrap.php* file that is included by PHPUnit before the unit tests are run.

composer.json
: This file lists my PHP project's dependencies managed by Composer, including the PHPUnit test framework.

phpunit.xml
: This file provides configuration details for the PHPUnit test runner.

.travis.yml
: This file provides configuration details for the Travis CI continuous testing web service.



Look at your favorite PHP component or framework's source code on GitHub and you'll see it uses a similar organization.

Install PHPUnit

First we need to install PHPUnit and the Xdebug profiler. PHPUnit runs our tests. The Xdebug profiler generates helpful code coverage information. Composer is the easiest way to install the PHPUnit test framework. Open your terminal application, navigate to your project's topmost directory, and run this command:

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

This command downloads the PHPUnit test framework into your project's *vendor/* directory, and it updates your project's *composer.json* file so that the `phpunit/phpunit` package is listed as a project dependency. The `phpunit` binary is installed in your project's *vendor/bin/* directory. You can add this directory to your environment path, or you can reference `vendor/bin/phpunit` whenever you invoke the PHPUnit command line test runner. The PHPUnit framework classes are autoloaded into your PHP application with your project's other Composer-managed dependencies.

Install Xdebug

The Xdebug PHP extension is a bit trickier to install. If you installed PHP with your package manager, you can install Xdebug the same way (Example 5-1).

Example 5-1. How to install Xdebug

```
# Ubuntu
sudo apt-get install php7.0-xdebug

# CentOS
sudo yum install php70-php-pecl-xdebug
```

If you installed PHP from source, you'll need to install the Xdebug extension with the `pecl` command:

```
pecl install xdebug
```

Next, update your *php.ini* configuration file with the path to the compiled Xdebug extension.

You can find your PHP extensions directory with the `php-config --extension-dir` or `php -i | grep extension_dir` commands.

Append this line to your *php.ini* file using your own PHP extension path:

```
zend_extension="/PATH/TO/xdebug.so"
```

Restart PHP and you're good to go. We'll discuss the Xdebug profiler in Chapter 6.

Configure PHPUnit

Now let's configure PHPUnit in our project's *phpunit.xml* file.

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="tests/bootstrap.php">
    <testsuites>
        <testsuite name="whovian">
            <directory suffix="Test.php">tests</directory>
        </testsuite>
    </testsuites>

    <filter>
        <whitelist>
            <directory>src</directory>
        </whitelist>
    </filter>
</phpunit>
```

PHPUnit test runner settings are attributes on the `<phpunit>` XML root element. The most important setting, in my opinion, is the `bootstrap` setting; it specifies the path (relative to the *phpunit.xml* file) to a PHP file that is included before the PHPUnit test runner executes our tests. We'll autoload our application's Composer dependencies in the *bootstrap.php* file so they are available to our PHPUnit tests. The *phpunit.xml* file also specifies the path to our test suite (i.e., a directory that contains related test cases); PHPUnit runs all PHP files in this directory whose file names end with *Test.php*. Finally, this configuration file lists the directories included in our code coverage analysis with the `<filter>` element. In the previous example XML, the `<whitelist>` element tells PHPUnit to generate code coverage only for code in the *src/* directory.

The gist of this configuration file is to specify our PHPUnit settings in one location. This makes our lives easier locally because we don't have to specify these settings each time we use the `phpunit` command-line runner. This configuration file also lets us apply the same PHPUnit settings on remote continuous testing servers like Travis CI. After you update the *phpunit.xml* configuration file, update the *tests/bootstrap.php* file with this code:

```php
<?php
// Enable Composer autoloader
require dirname(__DIR__) . '/vendor/autoload.php';
```

Make sure you install your Composer dependencies before running PHPUnit tests.

The Whovian Class

Before we write unit tests, we need something to test. Here's a hypothetical PHP class named `Whovian` that has a pretty strong opinion about a particular BBC television show. Place this class definition into the *src/Whovian.php* file:

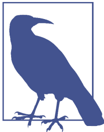
```php
<?php
class Whovian
{
    /**
     * @var string
     */
    protected $favoriteDoctor;

    /**
     * Constructor
     * @param  string $favoriteDoctor
     */
    public function __construct($favoriteDoctor)
    {
        $this->favoriteDoctor = (string)$favoriteDoctor;
    }

    /**
     * Say
     * @return string
     */
    public function say()
    {
        return 'The best doctor is ' . $this->favoriteDoctor;
    }

    /**
     * Respond to
     * @param  string $input
     * @return string
     * @throws \Exception
     */
    public function respondTo($input)
    {
        $input = strtolower($input);
```

```
            $myDoctor = strtolower($this->favoriteDoctor);

            if (strpos($input, $myDoctor) === false) {
                throw new Exception(
                    sprintf(
                        'No way! %s is the best doctor ever!',
                        $this->favoriteDoctor
                    )
                );
            }

            return 'I agree!';
        }
    }
```

The `Whovian` class constructor sets the instance's favorite doctor. The `say()` method returns a string with the instance's favorite doctor. And its `respondTo()` method receives a statement from another `Whovian` instance and responds accordingly.

The WhovianTest Test Case

The unit tests for our `Whovian` class live in the *test/WhovianTest.php* file. We call a group of related tests a *test suite*. In our example, all tests beneath the *test/* directory belong to the same test suite. Each class file beneath the *test/* directory is called a *test case*, and its class methods that begin with `test` (e.g., `testThis` or `testThat`) are individual tests. Each individual test uses assertions to verify a given condition. An assertion can pass or fail.



Find a list of PHPUnit assertions on the PHPUnit website. Some assertions are undocumented; you can find all available assertions in the source code on GitHub.

Each PHPUnit test case is a class that extends the `PHPUnit_Framework_TestCase` class. Let's declare a test case named `WhovianTest` in the *test/WhovianTest.php* file:

```
<?php
require dirname(__DIR__) . '/src/Whovian.php';

class WhovianTest extends PHPUnit_Framework_TestCase
{
    // Individual tests go here
}
```

Remember, unit tests verify a public interface's expected behavior. We'll test the three public methods in the `Whovian` class. We'll write a unit test to ensure that the `__construct()` method argument becomes the instance's preferred doctor. Next, we'll write

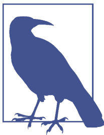
a unit test to ensure that the `say()` method's return value mentions the instance's preferred doctor. Finally, we'll write two tests for the `respondTo()` method. One test ensures that the method's return value is the string `"I agree!"` if the input matches its preferred doctor. The second test ensures the method throws an exception if the input does not match its preferred doctor.

Test 1: __construct()

Our first test confirms that the constructor sets the `Whovian` instance's favorite doctor:

```
public function testSetsDoctorWithConstructor()
{
    $whovian = new Whovian('Peter Capaldi');
    $this->assertAttributeEquals('Peter Capaldi', 'favoriteDoctor', $whovian);
}
```

This test instantiates a new `Whovian` instance with one string argument: `"Peter Capaldi"`. We use the PHPUnit assertion method `assertAttributeEquals()` to assert the `favoriteDoctor` property on the `$whovian` instance equals the string `"Peter Capaldi"`.



The PHPUnit assertion `assertAttributeEquals()` receives three arguments. The first argument is the expected value; the second argument is the property name; and the final argument is the object to inspect. What's neat is that the `assertAttributeEquals()` method can inspect and verify protected properties using PHP's reflection capabilities.

Why do we inspect the favorite doctor value with the `assertAttributeEquals()` assertion instead of a getter method (e.g., `getFavoriteDoctor()`)? When we write a test, we test *only one specific method in isolation*. Ideally, our test does not rely on other methods. In this particular example, we test the `__construct()` method and verify that it assigns its argument value to the object's `$favoriteDoctor` property. The `assertAttributeEquals()` assertion lets us inspect the object's internal state without relying on a separate, untested getter method.

Test 2: say()

Our next test confirms that the `Whovian` instance's `say()` method returns a string value that contains its favorite doctor's name:

```
public function testSaysDoctorName()
{
    $whovian = new Whovian('David Tennant');
    $this->assertEquals('The best doctor is David Tennant', $whovian->say());
}
```

We use the PHPUnit assertion `assertEquals()` to compare two values. The assertion's first argument is the expected value. Its second argument is the value to inspect.

Test 3: respondTo() in agreement

Now let's test how a `Whovian` instance responds in agreement with another Whovian:

```
public function testRespondToInAgreement()
{
    $whovian = new Whovian('David Tennant');

    $opinion = 'David Tennant is the best doctor, period';
    $this->assertEquals('I agree!', $whovian->respondTo($opinion));
}
```

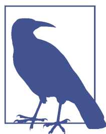
This test is successful because the `Whovian` instance's `respondTo()` method receives a string argument that includes the name of its favorite doctor.

Test 4: respondTo() in disagreement

But what if a `Whovian` *disagrees*? Get out of the area as quickly as possible, because s#!t is going to hit the fan. Well, actually, it'll just throw an exception. Let's test that:

```
/**
 * @expectedException Exception
 */
public function testRespondToInDisagreement()
{
    $whovian = new Whovian('David Tennant');

    $opinion = 'No way. Matt Smith was awesome!';
    $whovian->respondTo($opinion);
}
```

If this test throws an exception, the test passes. Otherwise, the test fails. We can test this condition with the `@expectedException` annotation.



PHPUnit provides several annotations that can control a given test. Read more about PHPUnit annotations in the PHPUnit documentation.

Run Tests

After you write each test, you should run your test suite to ensure that it passes. This is really simple to do. Open your terminal application and navigate to your project's topmost directory (the same directory as your *phpunit.xml* configuration file). We'll use the PHPUnit binary installed with Composer. Use this command to start the PHPUnit test runner:

```
vendor/bin/phpunit -c phpunit.xml
```

The `-c` option specifies the path to the PHPUnit configuration file. The terminal shows the results from the PHPUnit command-line test runner, and they look like Figure 5-1.

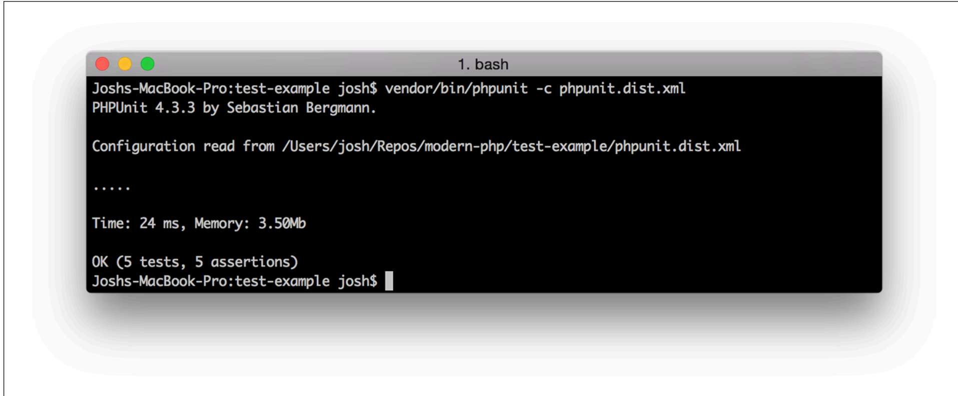


Figure 5-1. PHPUnit test results

These results tell us:

1. PHPUnit read our configuration file.
2. PHPUnit took 24 ms to complete.
3. PHPUnit used 3.5 MB of memory.
4. PHPUnit successfully ran five tests and five assertions.

Code Coverage

We know our PHPUnit tests pass. However, are we sure we tested as much of our code as possible? Perhaps we forgot to test something. We can see exactly which code is tested (and untested) with PHPUnit's code coverage report (Figure 5-2). We already specify the path(s) to our source code files in the PHPUnit configuration file. All PHP files in the whitelisted directories are included in PHPUnit's code coverage report. We can generate code coverage each time we run the PHPUnit test runner:

```
vendor/bin/phpunit -c phpunit.xml --coverage-html coverage
```

This is the same command we used earlier, except we append the new `--coverage-html` option whose value is the path to a the code coverage report directory. After you run this command, open the newly generated *coverage/index.html* file in a web browser to see the code coverage results. Ideally, you want to see 100% coverage across the board. However, 100% coverage is *not realistic* and definitely should not be

a requirement. How much coverage is good is subjective and varies from project to project.

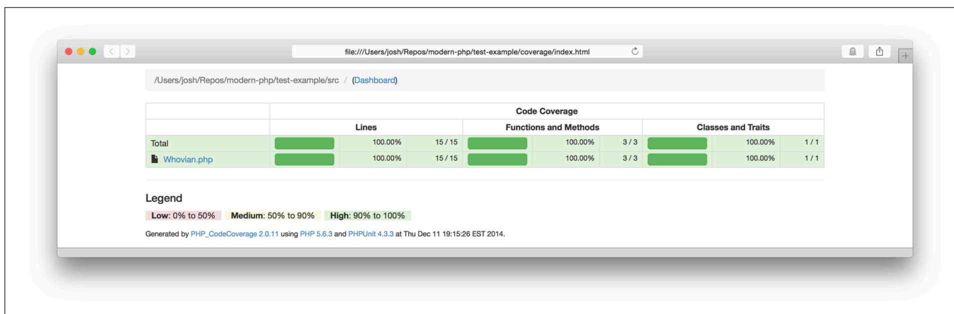
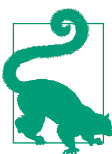


Figure 5-2. PHPUnit code coverage report



Use PHPUnit's code coverage report as a *guideline* to improve your code. Don't use code coverage percentages as requirements.

Continuous Testing with Travis CI

Sometimes even the best PHP developers forget to write tests. This is why it is important to automate your tests. The best tests are like a good backup strategy—out of sight and out of mind. *Tests should run automatically*. My favorite continuous testing service is Travis CI because it has native hooks into GitHub repositories. I can run my application tests within Travis CI every time I push code to GitHub. Travis CI runs my tests against multiple PHP versions, too.

Setup

If you have not used Travis CI before, go to *https://travis-ci.org* (for public repositories) or *https://travis-ci.com* (for private repositories). Log in with your GitHub account. Follow the on-screen instructions to choose which repository to test with Travis CI.

Next, create the *.travis.yml* Travis CI configuration file in your application's topmost directory. Don't forget the leading . character! Save, commit, and push the Travis CI configuration file to your GitHub repository. Here's an example Travis CI configuration:

```
language: php
php:
  - 5.6
  - 7.0
```

```
  - hhvm
install:
  - composer install --no-dev --quiet
script: phpunit -c phpunit.xml --coverage-text
```

The Travis CI configuration is written in YAML format and includes these settings:

`language`
: This is the language used for our application. We set this to `php`. This value is case-sensitive!

`php`
: Travis CI runs our application tests against these PHP versions. It is important that you test against all PHP versions supported by your application.

`install`
: This is a bash command executed by Travis CI before it runs application tests. This is where you instruct Travis CI to install your project's Composer dependencies. It is important that you use the `--no-dev` option to avoid installing unnecessary development dependencies.

`script`
: This is the bash command executed by Travis CI to run application tests. By default, this is `phpunit`. You can override Travis CI's default command with this setting. In this example, we tell Travis CI to use our custom PHPUnit configuration file and generate plain text coverage results.

Run

Travis CI automatically runs your application tests every time you push new commits to your GitHub repository and emails you the test results. How cool is that? There are, of course, many more Travis CI settings to further customize the Travis CI testing environment (e.g., install custom PHP extensions, use custom `ini` settings, and so on). Read more about Travis CI configuration for PHP at Travis CI.

Further Reading

Here are a few links to help you learn more about PHP application testing:

- *https://phpunit.de/*
- *https://www.phpspec.net/en/latest/manual/introduction.html*
- *http://behat.org/*
- *https://leanpub.com/grumpy-phpunit*
- *https://leanpub.com/grumpy-testing*

- *https://www.littlehart.net/atthekeyboard/*

What's Next

In this chapter we learned why, when, and how to write tests. Testing our applications builds confidence and creates more predictable code. However, tests do not let us analyze application *performance*. This is why we must also *profile* our applications. That's what I want to talk about next.

CHAPTER 6
Profiling

Profiling is how we analyze application performance. It is a great way to debug performance issues and pinpoint bottlenecks in your application code. In other words, if your application is slow, use a profiler to figure out why. Profilers let us traverse the entire PHP call stack, and they tell us which functions or methods are called, in what order, how many times, with what arguments, and for how long. We can also see how much memory and CPU are used throughout the application request lifecycle.

When to Use a Profiler

You don't need to profile your PHP applications immediately. You only profile PHP applications if there is a performance issue that is otherwise hard to diagnose. How do you know if you have a performance issue? Some issues are obvious (e.g., a database query takes too long). Other issues may not be as obvious.

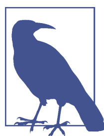
You can detect performance issues with benchmarking tools like Apache Bench and Siege. A benchmarking tool allows you to test your application performance *externally*, much like an application user would with a web browser. Benchmarking tools let you set the number of concurrent users and total number of requests that hit a specific application URL. When the benchmarking tool finishes, it tells you the number of requests per second that your application sustained (among other statistics). If you find a particular URL sustains only a small number of requests per second, you may have a performance issue. If the performance issue is not immediately obvious, you use a profiler.

Types of Profilers

There are two types of profilers. There are those that should run only during development, and there are those that can run during production.

Xdebug is a popular PHP profiling tool written by Derick Rethans, but it should only be used as a profiler during development because it consumes a lot of system resources to analyze your application. Xdebug profiler results are not human-readable, so you'll need an application to parse and display the results. KCacheGrind and WinCacheGrind are good applications for visualizing Xdebug profiler results.

XHProf is a popular PHP profiler originally written by Facebook, now owned and maintained by the open source community. It is intended to be run during development *and* production. XHProf's profiler results are also not human-readable, but there are companion web applications that will display results in a human-readable format.



Both Xdebug and XHProf are PHP extensions, and you can install them with your operating system's package manager. They can also be installed with `pecl`.

Xdebug

Xdebug is one of the most popular PHP profilers, and it makes it easy to analyze your application's call stack to find bottlenecks and performance issues. Refer to Example 5-1 in Chapter 5 for Xdebug installation instructions.

Configure

Xdebug configuration lives in your *php.ini* file. Here are the Xdebug profiler configuration settings I recommend. Make sure you specify your own profiler output directory. Restart your PHP process after saving these settings:

```
xdebug.profiler_enable = 0
xdebug.profiler_enable_trigger = 1
xdebug.profiler_output_dir = /path/to/profiler/results
```

`xdebug.profiler_enable = 0`
: This instructs Xdebug to not run automatically. We don't want Xdebug to run automatically on each request, because that would drastically decrease performance and impede development.

`xdebug.profiler_enable_trigger = 1`
: This instructs Xdebug to run on-demand. We can activate Xdebug profiling per-request by adding the XDEBUG_PROFILE=1 query parameter to any of our PHP application's URLs. When Xdebug detects this query parameter, it profiles the current request and generates a report in the output directory specified by the `xdebug.profiler_output_dir` setting.

`xdebug.profiler_output_dir = /path/to/profiler/results`
: This is the directory path that contains generated profiler results. Profiler reports can be massive (e.g., 500 MB or larger) for complex PHP applications. Make sure you change this value to the correct filesystem path for your application.

I recommend you keep profiler results beneath your PHP application's topmost directory. This makes it easy to find and review profiler results while developing your application.

Trigger

The Xdebug profiler does not run automatically because the `xdebug.profiler_enable` setting is `0`. We trigger the Xdebug profiler for a single request by adding the *XDEBUG_PROFILE=1* query parameter to any PHP application URL. An example HTTP request URL might be */users/show/1?XDEBUG_PROFILE=1*. When Xdebug detects the `XDEBUG_PROFILE` query parameter, it activates and runs the profiler for the current request. The profiler results are dumped into the directory specified by the `xdebug.profiler_output_dir` setting.

Analyze

The Xdebug profiler generates results in the CacheGrind format. You'll need a CacheGrind-compatible application to review the profiler results. Some good applications for reviewing CacheGrind files are:

- WinCacheGrind for Windows
- KCacheGrind for Linux
- WebGrind for web browsers

Mac OS X users can install KCacheGrind with Homebrew using this command:

```
brew install qcachegrind
```

Homebrew is a package manager for OS X. We discuss Homebrew in ???.

XHProf

XHProf is a newer PHP application profiler. It was originally created by Facebook and is intended to be run during both development and production. It does not col-

lect as much information as Xdebug's profiler, but it consumes fewer system resources, making it suitable for production environments.

Install

The easiest way to install XHProf is with your operating system's package manager (assuming you installed PHP the same way):

```
# Ubuntu
sudo apt-get install build-essential;
sudo pecl install mongo;
sudo pecl install xhprof-beta;

# CentOS
sudo yum groupinstall 'Development Tools';
sudo pecl install mongo;
sudo pecl install xhprof-beta;
```

Append these lines to your *php.ini* file, and restart your PHP process to load the new extensions:

```
extension=xhprof.so
extension=mongo.so
```

XHGUI

XHProf is most useful when paired with XHGUI, Facebook's companion web application used to review and compare XHProf profiler output. XHGUI is a PHP web application and requires:

- Composer
- Git
- MongoDB
- PHP 5.3+
- PHP `mongo` extension

I assume these system requirements are installed. I also assume the XHGUI web application lives in the */var/sites/xhgui/* directory. This directory path is probably different on your server, so keep that in mind:

```
cd /var/sites;
git clone https://github.com/perftools/xhgui.git;
cd xhgui;
php install.php;
```

The XHGUI web application has a *webroot/* directory. Update your web server virtual host's document root to this directory.

Configure

Open XHGUI's *config/config.default.php* file in a text editor. By default, XHProf collects data for only 1% of all HTTP requests. This is fine for production, but you may want to collect data more frequently during development. You can increase XHProf's data collection by editing these lines in the *config/config.default.php* file:

```
'profiler.enable' => function() {
    return rand(0, 100) === 42;
},
```

Change these lines to:

```
'profiler.enable' => function() {
    return true; // <-- Run on every request
},
```

XHProf assumes your PHP application runs on a single server. It also assumes your MongoDB database does not require authentication. If your MongoDB server does require authentication, update the Mongo database connection in the *config/config.default.php* file.

Trigger

You must include the XHGUI web application's *external/header.php* file at the very beginning of your PHP application. It's easiest to use PHP's `auto_prepend_file` INI configuration setting. You can set this in the *php.ini* configuration file:

```
auto_prepend_file = /var/sites/xhgui/external/header.php
```

Or you can set this in your nginx virtual host configuration:

```
fastcgi_param PHP_VALUE "auto_prepend_file=/var/sites/xhgui/external/header.php";
```

Or you can set this in your Apache virtual host configuration:

```
php_admin_value auto_prepend_file "/var/sites/xhgui/external/header.php"
```

Restart PHP, and XHProf will begin collecting and saving information into its MongoDB database. You can review and compare XHProf runs at the XHGUI virtual host's URL.

New Relic Profiler

Another popular PHP profiler is New Relic. This is actually a web service that uses a custom operating system daemon and PHP extension to hook into your PHP application and report data back to the web service. Unlike Xdebug and XHProf, New Relic's PHP profiler is not free. That being said, I adore New Relic and recommend it if your budget allows. Like XHProf, New Relic's PHP profiler is meant to be run during

production, and it gives you a near real-time view of your application's performance with a really nice online dashboard. Learn more on New Relic's website.

Blackfire Profiler

The makers of Symfony make a PHP profiler called Blackfire. It provides unique visualization tools to help discover application bottlenecks. Blackfire has a PHP extension and agent that runs on your application server. It reports data back to the Blackfire server for analysis and visualization in the Blackfire web application.

Further Reading

I hope I've introduced you to PHP profiling in this chapter so that you feel comfortable finding, installing, and using a PHP profiler most appropriate for your application. Here are a few links to help you learn more about PHP profiling:

- *http://www.sitepoint.com/the-need-for-speed-profiling-with-xhprof-and-xhgui/*
- *https://blog.engineyard.com/2014/profiling-with-xhprof-xhgui-part-1*
- *https://blog.engineyard.com/2014/profiling-with-xhprof-xhgui-part-2*
- *https://blog.engineyard.com/2014/profiling-with-xhprof-xhgui-part-3*

What's Next

At this point we've talked a lot about modern PHP, including new features, good practices, provisioning, tuning, deployment, testing, and profiling. I hope you have filled your brain with tons of fun ideas to implement in your next PHP applications.

Now I want to take a few minutes to chat about the future of PHP. A lot is happening in the PHP ecosystem. The future of PHP is unfolding as we speak thanks to the latest PHP 7 release with vastly improved performance, and thanks to HHVM, Hack, and the PHP-FIG. Let's explore HHVM and Hack, specifically, and figure out what they mean for PHP's future.

CHAPTER 7

HHVM and Hack

Think what you will about the Facebook application, but I have nothing but praise for the brilliant folks working at Facebook. Facebook Open Source has developed several important projects in the last few years, two of which have had significant impact in the PHP community.

The first initiative is HHVM, or the *Hip Hop Virtual Machine*. This alternative PHP engine was released in October 2013. Its just-in-time (JIT) compiler provides performance many times better than PHP-FPM. In fact, WP Engine recently migrated to HHVM and realized 3.9x faster custom Wordpress installations. MediaWiki also transitioned to HHVM, and it has realized drastic improvements in both response times and throughput.

The second initiative is Hack, a new server-side language that is a modification of the PHP language. Hack is mostly backward-compatible with PHP code, although it extends the PHP language with strict typing, new data structures, and a real-time type checking server. That being said, Hack's own developers prefer to call Hack a *dialect* of PHP and not a new language.

HHVM

Since 1994, if you said *PHP interpreter* you meant the Zend Engine. The Zend Engine *was* PHP. It was the one and only PHP interpreter. Then Mark Zuckerberg came along and created this little thing called *Thefacebook* on February 4, 2004. Mr. Zuckerberg and his growing company wrote the Facebook application predominantly with PHP because the language is easy to learn and simple to deploy. The PHP language lets Facebook quickly onboard new developers to grow, innovate, and iterate its platform.

Fast forward, and Facebook is a veritable empire. Its infrastructure is massive. Facebook is so huge that the traditional Zend Engine became a bottleneck for its developers. The Facebook team had a hugely growing user base (by 2007, its user base surpassed 1 in 10 people on the planet), and it had to figure out a way to improve performance without simply building more data centers and buying more servers.

PHP at Facebook

The PHP language is traditionally interpreted, not compiled. This means that your PHP code remains PHP code until it is sent through an interpreter when executed on the command line or requested by a web server. The PHP script is read by the PHP interpreter and converted into a set of existing Zend Opcodes (machine-code instructions), and the Zend Opcodes are executed with the Zend Engine. Unfortunately, interpreted languages execute slower than compiled languages because they must be converted to machine code during every execution. This taxes system resources. Facebook realized this performance bottleneck and, in 2010, began working on a PHP-to-C++ compiler called HPHPc.

The HPHPc compiler converts PHP code into C++ code. It then compiles the C++ code into an executable that is deployed to production servers. HPHPc was largely successful; it improved Facebook's performance and reduced the strain on its servers. However, HPHPc's potential performance approached a ceiling, it was not 100% compatible with the complete PHP language, and it required a time-consuming compile process that created a lengthy feedback loop for developers. Facebook needed a hybrid solution that delivered superior performance but also allowed for faster development without expensive compile time.

Facebook began working on the next iteration of HPHPc, called HHVM. HHVM converts and caches PHP code into an intermediary bytecode format, and it uses a JIT compiler to translate and optimize its bytecode cache into x86_64 machine code. HHVM's JIT compiler enables many low-level performance optimizations that are simply not possible by compiling PHP directly to C++ with HPHPc. HHVM also enables a fast feedback loop for developers because it compiles bytecode into machine code only when PHP scripts are requested by a web server—just in time, you might say—much like a traditional interpreted language. What's more amazing is that HHVM's performance eclipsed HPHPc's performance in November 2012, and it continues to improve (Figure 7-1).

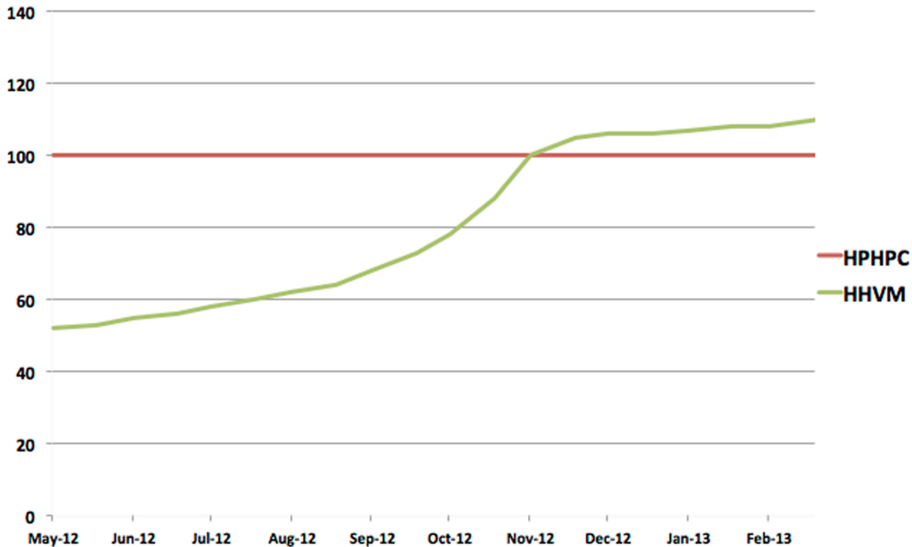


Figure 7-1. HHVM vs. HPHPc Performance

HPHPc was deprecated soon after HHVM's peformance exceeded its own, and HHVM is currently Facebook's preferred PHP interpreter.

Don't let HHVM intimidate you! Its implementation may be complex, but at the end of the day HHVM is just a replacement for the more familiar `php` and `php-fpm` binaries:

- You execute PHP scripts with the `hhvm` binary on the command line, just like the `php` binary.
- You use the `hhvm` binary to create a FastCGI server, just like the `php-fpm` binary.
- HHVM uses a *php.ini* configuration file, just like the traditional Zend Engine. It even uses the same INI directives.
- HHVM has native support for many common PHP extensions.

HHVM and Zend Engine Parity

Facebook's original HPHPc compiler was not compatible with the complete PHP language (i.e., the Zend Engine). Complete parity is an aspiration for Facebook because it lets HHVM become a drop-in replacement for the Zend Engine.

Facebook tested HHVM against the most popular PHP frameworks to ensure compatibility with real-world PHP 5 code. Facebook is close to 100% compatibility. However, Facebook has shifted its focus to user-reported issues on the HHVM issue tracker to tackle remaining edge-case issues. HHVM is not yet 100% compatible with the traditional Zend Engine, but it's getting closer every day. Facebook, Baidu, and Wikipedia already use HHVM in production. HHVM can also run Wordpress, Drupal, and many popular PHP frameworks.

Is HHVM Right for Me?

HHVM isn't the right choice for everyone. There are far easier ways to improve application performance. Reducing HTTP requests and optimizing database queries are low-hanging fruit that noticeably improve application performance and response time. If you have not made these optimizations, do them first before you consider HHVM. Facebook's HHVM is for developers who have already made these optimizations and *still* need faster applications. If you believe you need HHVM, here are some resources to help you make the best decision:

Extensions
: View a list of PHP extensions compatible with HHVM.

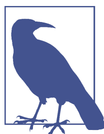
Issue Tracker
: Track open HHVM issues.

FAQs
: Read HHVM frequently asked questions.

Blog
: Follow the latest HHVM news.

Install

HHVM is easy to install on the most popular Linux distributions. It was originally developed for Ubuntu (my preferred Linux distibution), so I use Ubuntu in the following examples.



Facebook provides prebuilt packages for other Linux distributions, including Debian and Fedora. You can build HHVM from source on even more Linux distributions.

Per Facebook's instructions, you can install HHVM on the latest version of Ubuntu with the Aptitude package manager like this:

```
sudo apt-get install software-properties-common
sudo apt-key adv --recv-keys --keyserver hkp://keyserver.ubuntu.com:80 \
    0x5a16e7281be7a449
sudo add-apt-repository "deb http://dl.hhvm.com/ubuntu $(lsb_release -sc) main"
sudo apt-get update
sudo apt-get install hhvm
```

The preceding code adds HHVM's GNU Privacy Guard (GPG) public key for package verification. It adds the HHVM package repository to our local list of repositories. Finally, it installs HHVM with Aptitude like any other software package. The HHVM binary is installed at */usr/bin/hhvm*.

Configure

HHVM uses a *php.ini* configuration file just as the Zend Engine does. This file exists at */etc/hhvm/php.ini* by default, and it contains many of the same INI settings used by the Zend Engine. You can find a complete list of HHVM *php.ini* directives at *https://docs.hhvm.com/hhvm/configuration/INI-settings*.

If you run HHVM as a FastCGI server, add server-related INI directives into the */etc/hhvm/server.ini* file. You can find a complete list of HHVM server directives at *https://docs.hhvm.com/hhvm/configuration/introduction*. The HHVM wiki page is weak on details, so you may want to peruse these HHVM support communities, too:

- StackOverflow
- IRC Channel
- Facebook Page

The default */etc/hhvm/server.ini* file should be sufficient to get you started. It looks like this:

```
; php options

pid = /var/run/hhvm/pid

; hhvm specific

hhvm.server.port = 9000
hhvm.server.type = fastcgi
hhvm.server.default_document = index.php
hhvm.log.use_log_file = true
hhvm.log.file = /var/log/hhvm/error.log
hhvm.repo.central.path = /var/run/hhvm/hhvm.hhbc
```

The most notable settings are `hhvm.server.port = 9000` and `hhvm.server.type = fastcgi`; they tell HHVM to run as a FastCGI server on local port 9000.

When you execute the `hhvm` binary, you specify the path to your configuration files with the `-c` option. If you use `hhvm` to execute command-line scripts, you only need the */etc/hhvm/php.ini* configuration file:

```
hhvm -c /etc/hhvm/php.ini my-script.php
```

If you use the `hhvm` binary to start a FastCGI server, you need both the */etc/hhvm/php.ini* and */etc/hhvm/server.ini* files:

```
hhvm -m server -c /etc/hhvm/php.ini -c /etc/hhvm/server.ini
```

Extensions

HHVM cannot use PHP extensions that are compiled for the Zend Engine unless the extensions use Facebook's Zend Extension Source Compatibility Layer. Fortunately, most of the PHP extensions we take for granted are supported by HHVM out of the box. Other third-party PHP extensions (e.g., the GeoIP extension) can be compiled separately and loaded into HHVM as a dynamic extension. You can find a list of PHP extensions compatible with HHVM on GitHub.

Monitor HHVM with Supervisord

HHVM is just fine for your production server, but it's not infallible. I recommend you keep tabs on HHVM's master process with Supervisord, a process monitor that starts the HHVM process on boot and automatically restarts the HHVM process if HHVM fails.

If you are unfamiliar with Supervisord, Chris Fidao has an excellent tutorial.

Install Supervisord with this command if you haven't already:

```
sudo apt-get install supervisor
```

Next, make sure the */etc/supervisor/supervisord.conf* configuration file has these two lines:

```
[include]
files = /etc/supervisor/conf.d/*.conf
```

These two lines let us create a configuration file in the */etc/supervisor/conf.d/* directory for each supervised application. Next, create the */etc/supervisor/conf.d/hhvm.conf* file with this content:

```
[program:hhvm]
command=/usr/bin/hhvm -m server -c /etc/hhvm/php.ini -c /etc/hhvm/server.ini
```

```
directory=/home/deploy
autostart=true
autorestart=true
startretries=3
stderr_logfile=/home/deploy/logs/hhvm.err.log
stdout_logfile=/home/deploy/logs/hhvm.out.log
user=deploy
```

The most important settings are:

`command`
: Supervisord runs this command to kick off the HHVM process. We use the `-m` option to run HHVM in server mode. We also use the `-c` option to provide the path to HHVM's *php.ini* and *server.ini* configuration files.

`autostart`
: This causes the HHVM process to start when the Supervisord process starts (e.g., on system boot).

`autorestart`
: This prompts Supervisord to restart the HHVM process if it fails.

`startretries`
: This is the number of times Supervisord should try to start the HHVM process before Supervisord considers this process a failure.

`user`
: This is the user that owns the HHVM process. I recommend you use an unprivileged user for security purposes. In this example, I use the same unprivileged `deploy` user we created in Example 2-1.

Make sure you manually create the */home/deploy/logs* directory, because Supervisord does not create it for you.

After you finish editing the Supervisord configuration files, run these two commands to reload and apply your changes:

```
sudo supervisorctl reread;
sudo supervisorctl update;
```

You can review all processes managed by Supervisord with this command:

```
sudo supervisorctl
```

You can start, stop, or restart a single Supervisord program as shown in the example below. In this example, `hhvm` is the program name specified at the top of the */etc/supervisor/conf.d/hhvm.conf* file:

```
sudo supervisorctl start hhvm;
sudo supervisorctl stop hhvm;
sudo supervisorctl restart hhvm;
```

So far we've installed HHVM, and we monitor the HHVM process with Supervisord. We still need a web server to proxy requests to HHVM. Remember, HHVM runs a FastCGI server exactly as we do in Chapter 2 with PHP-FPM. We'll use the HHVM FastCGI server to handle PHP requests sent from Nginx.

HHVM, FastCGI, and Nginx

HHVM communicates with a web server (e.g., Nginx) with the FastCGI protocol. We need to create an Nginx virtual host that proxies PHP requests to the HHVM FastCGI server. Here's an example Nginx virtual host definition that does that:

```
server {
    listen 80;
    server_name example.com;
    index index.php;
    client_max_body_size 50M;
    error_log /home/deploy/apps/logs/example.error.log;
    access_log /home/deploy/apps/logs/example.access.log;
    root /home/deploy/apps/example.com/current/public;

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    location ~ \.php {
        include fastcgi_params;
        fastcgi_index  index.php;
        fastcgi_param  SCRIPT_FILENAME ument_root$fastcgi_script_name;
        fastcgi_pass 127.0.0.1:9000;
    }
}
```

From this point forward, I assume Nginx is installed and running on your server. Refer to Chapter 2 for Nginx installation instructions.

Assuming you followed the Nginx installation instructions in Chapter 2, create a file at */home/deploy/apps/example.com/current/public/index.php* with this content:

```
<?php
phpinfo();
```

Make sure the `example.com` domain points to your server's IP address and visit *http://example.com/index.php* in a web browser. You should see the word "HipHop" appear in your browser window.

You can force your computer to point any domain name to any IP address by updating your local */etc/hosts* file. For example, this line points the domain name `example.com` to IP address `192.168.33.10`:

```
192.168.33.10 example.com
```

Congratulations! You've installed HHVM as a FastCGI server that can run your PHP application. But a FastCGI server isn't cool. You know what's cool? Hack. HHVM can run that, too.

The Hack Language

Hack is a server-side language that is similar to and seamless with PHP. Hack's developers even call Hack a dialect of PHP. Why did Facebook create something so similar to PHP? Facebook created the Hack language for several reasons. The Hack language adds new time-saving data structures and interfaces that are unavailable in PHP. More important, Hack introduces *static typing* to help us write more predictable and stable code. Static typing surfaces errors earlier in the development process using a near-realtime type checking server.

Are new data structures, interfaces, and static typing worth the time required to learn a new(ish) language and toolchain? Maybe. You have to remember that Facebook *is Facebook*. It has thousands of developers all working on a gargantuan codebase. If Facebook can optimize even the smallest part of its development process, it reaps a large reward in both developer efficiency and a more stable, well-performing codebase.

I do not recommend you drop what you're doing and immediately port your existing applications from PHP to Hack. However, if you are starting a new project and have time to install and learn Hack, then—by all means—go wild. You'll certainly benefit from Hack's data structures and static typing.

Convert PHP to Hack

To convert code from PHP to Hack, change `<?php` to `<?hh`. That's it. This is PHP code:

```
<?php
echo "I'm PHP";
```

And this is equivalent Hack code:

```
<?hh
echo "I'm Hack";
```

Facebook makes it super-easy to go from PHP to Hack because it understands that converting a large, existing codebase is not a quick task. Start your codebase migration by only changing `<?php` to `<?hh`. Next, introduce a few static types. Later on, explore some Hack data structures. The transition to Hack is gradual and painless, and it happens on your schedule; this is by design.

What is a Type?

Before we compare dynamic and static typing, it's probably helpful to define *type*. Most PHP programmers think a type is the form of data assigned to a variable. For example, the expression `$foo = "bar"` implies the `$foo` variable's value is a string. The expression `$bar = 14` implies the `$bar` variable's value is an integer. These examples demonstrate types, yes, but they betray the full definition of a type.

A *type* is a nebulous label that we assign to properties of an application to prove that certain behaviors exist and, to our own expectations, are fundamentally correct. I'm paraphrasing Chris Smith's excellent explanation of programming types.

We can expand our definition of a type to a syntactical annotation that clarifies the identity of program variables, arguments, or return values. Type annotations (or *hints*) are used in both PHP and Hack. You've probably seen code like this:

```
<?php
class WidgetContainer
{
    protected $widgets;

    public function __construct($widgets = array())
    {
        $this->widgets = array_values($widgets);
    }

    public function addWidget(Widget $widget)
    {
        $this->widgets[] = $widget;

        return this;
    }

    public function getWidget($index)
    {
        if (isset($this->widgets[$index]) === false) {
```

```
            throw new OutOfRangeException();
        }

        return $this->widgets[$index];
    }
}
```

This is an arbitrary example, but it uses syntax hints to enforce specific application properties. For example, in the `addWidget()` method signature we use a `Widget` hint before the `$widget` argument to tell PHP we expect the method argument to be an instance of class `Widget`. The PHP interpreter enforces this expectation. If an argument is provided that is not an instance of class `Widget`, the code fails. In this example, the type is our annotated expectation that the `addWidget()` method accepts arguments only of class `Widget`.

Our earlier naive examples (e.g., `$foo = "bar"`) and this `WidgetContainer` example both demonstrate types. The first example demonstrates a type that proves a variable is a string, even though we don't explicitly annotate the expectation. The PHP interpreter is smart enough to *infer* the string type in this example based on the code syntax. The second example creates a type with an annotation that explicitly defines the expected behavior of the `addWidget()` method, and the PHP interpreter enforces this behavior based on our explicit hint rather than making an inference.

Types are more than inferred identities and annotations. However, these are the two manifestations you'll see and use most often when writing PHP and Hack code. You can learn more about programming types in Benjamin C. Pierce's book "Types and Programming Languages."

If you thought that PHP type hints *are* static types, you're probably scratching your head right about now because I just burst your bubble. Both static and dynamic typing help us write code that behaves correctly according to our expectations, and both employ their own type systems. The main differences between static and dynamic typing are *when program types are checked* and *how a program is tested for correctness*.

Static Typing

The correct behavior of a statically typed program is *implied by the code*, via inferences, annotations, or other language-specific types. If a statically typed program compiles successfully, we can be confident the program is proven to behave as written. The program's types become our tests, and they ensure that the program satisfies our basic expectations.

Did you notice I used the word *compiles*? Statically typed languages are often compiled. Type checking and error reporting are delegated to the language compiler. This

is nice, because the compiler surfaces type-related program errors at compile time before the application is deployed into production. Unfortunately, compiled languages imply a lengthy feedback loop. A program must be compiled to reveal errors, and complicated programs take a long time to compile. This decelerates development.

The upside to statically typed programs is that they are usually more stable because their behavior is proven by the compiler's type checker. However, we should still write separate tests to verify that the program behavior is *correct*. If a program compiles, that only means the program does what the code says it should do. That does not mean the program does what we intend it to do. That being said, static typing saves us from writing type-related unit tests as we do for dynamically typed programs.

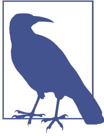
Dynamic Typing

Unlike static typing, dynamic typing cannot enforce code behavior at compile time, because the program types are not checked until runtime. Dynamically typed programs are often interpreted, too. PHP is a dynamically typed and interpreted language. This means that *every time* you execute a PHP script—either directly on the command line or indirectly via a web server—the PHP code is read by an interpreter, converted into a set of preexisting opcodes codes, and executed.

So how do you find errors if PHP is not compiled? Errors are surfaced during runtime. This is both a blessing and a curse. It's good because we can iterate quickly. We write code and run it. Feedback is near-instantaneous. Unfortunately, we lose the inherent accuracy and tests provided by static type checking. Separate unit tests become far more important to validate types and intended behavior. Our tests must cover all possible behaviors. This works for the behavior we anticipate, but it fails miserably for the behavior we do not anticipate. Unanticipated behaviors gnash their teeth during runtime as PHP errors, and we must handle them gracefully with friendly messages and appropriate logging.

Hack Goes Both Ways

Static typing is Hack's biggest selling point. Even more interesting is that Hack does static *and* dynamic typing. Remember, Hack is mostly backward-compatible with regular PHP. This means Hack supports all of PHP's dynamic typing features that you expect. This is possible because Hack is run with HHVM's JIT compiler. The Hack code is type checked as it is written with a standalone type checker. The Hack code is read, optimized, and cached into an intermediary bytecode by HHVM. A Hack file is only converted into x86_64 machine code and executed on demand. It's really the best of both worlds. We get the accuracy and safety of static typing with Hack's type checker (more on this next) and the flexibility and quick iteration of dynamic typing thanks to HHVM's JIT compiler.



There are a few PHP features *not* supported by Hack. They are listed at *https://docs.hhvm.com/hack/unsupported/introduction*. These features are supported by HHVM when executing normal PHP code.

Hack Type Checking

Hack comes with a standalone type-checking server that runs in the background and type-checks your code *in realtime*. This is huge. This is also the main reason why Facebook created the Hack language. Hack's instantaneous type checking provides the accuracy and safety of static typing without the lengthy feedback loop. If you are using Hack without its type checker, you're holding it wrong.

Here's how to set up Hack's type checker for your application. First, I assume HHVM is installed and running. If not, refer to the HHVM section for installation instructions. Next, create an empty file named *.hhconfig* in your project's topmost directory. This tells the Hack type checker which directory to analyze. The type checker watches files beneath this directory and type-checks the appropriate files whenever it detects filesystem changes. Start the Hack type checker by executing the `hh_client` command in or beneath your project's topmost directory.

Hack's type checker does have a few limitations. Per Hack's online documentation:

> The type checker assumes that there is a global autoloader that can load any class on demand. This means that it insists that all class and function names are unique, and has no notion of checking imports or anything of that nature. Futhermore, it does not support conditional definitions of functions or classes — it must be able to statically know what is and what is not defined. It is of course perfectly possible to have a project that meets these requirements without a global autoloader, and the type checker will work fine on such a project, but a project using an autoloader was the intended use case.
>
> Mixing HTML and Hack code are not supported by the type checker. Following and statically analyzing these complicated mode switches is unsupported, particularly since much modern code doesn't make use of this functionality. Hack code can output markup to the browser in a simple way via echo, or using a templating engine or XHP for more complex scenarios.

Hack Modes

Hack code can be written in three modes: `strict`, `partial`, or `decl`. If you are starting a project with Hack, I recommend you use `strict` mode. If you are migrating existing PHP code to Hack, or if your project uses both PHP and Hack code, you may want to use `partial` mode. The `decl` mode lets you integrate legacy, untyped PHP code into an otherwise `strict` Hack codebase. You declare the mode at the very top

of the file, after and adjacent to the opening Hack or PHP tag (see the following examples). Mode names are case-sensitive:

`<?hh // strict`
: Strict mode requires all code to be appropriately annotated. The Hack type checker will catch all possible type-related errors. This mode also prevents your Hack code from using non-Hack code (e.g., legacy PHP code). Be sure you read up on Hack type annotations before you commit to `strict` mode. Among other requirements, all Hack arrays *must* be typed; you cannot use an untyped array in Hack. You must also annotate return types for functions and methods.

`<?hh // partial`
: Partial mode (the default) allows Hack code to use PHP code that has not been converted to Hack. Partial mode also does not require you to annotate *all* of a function or method's arguments. You can annotate a subset of the arguments without angering the Hack type checker. If you are just getting started with Hack, or if you are converting an existing PHP codebase, this is probably the best mode for you.

`<?php // decl`
: `decl` mode lets `strict` Hack code call untyped code. This is often the case when newer Hack code depends on a legacy, untyped PHP class. In this scenario, the legacy PHP code should declare itself in `decl` mode before the newer Hack code can use it.

Hack Syntax

Hack supports type annotations for class properties, method arguments, and return types. These annotations are checked with Hack's standalone type checker in accordance with each file's mode.

Read a complete list of available type annotations.

Let's revisit our earlier `WidgetContainer` example and introduce type annotations. The updated Hack code looks like this:

```
01. <?hh // strict
02. class WidgetContainer
03. {
04.     protected Vector<Widget> $widgets;
05.
06.     public function __construct(array<Widget> $widgets = array())
```

```
07.     {
08.         foreach ($widgets as $widget) {
09.             $this->addWidget($widget);
10.         }
11.     }
12.
13.     public function addWidget(Widget $widget) : this
14.     {
15.         $this->widgets[] = $widget;
16.
17.         return this;
18.     }
19.
20.     public function getWidget(int $index) : Widget
21.     {
22.         if ($this->widgets->containsKey($index) === false) {
23.             throw new OutOfRangeException();
24.         }
25.
26.         return $this->widgets[$index];
27.     }
28. }
```

Property annotations

On line 4, we declare the `$widgets` class property with the `Vector<Widget>` annotation. This annotation tells us two things:

- This property is a `Vector` (similar to a numerically indexed array).
- This property must contain only `Widget` instances.

Argument annotations

This is probably familiar to those of you who already use PHP type hints. On line 6, we annotate the `__construct()` method's argument with the `array<Widget>` annotation. This annotation tells us two things:

- The argument must be an array.
- The argument must contain only `Widget` instances.

Unlike the property annotation on line 4, this argument can be either a numeric or an associative array. We iterate the array argument's values and add them to the `Vector` data structure. If you did want the argument to be either a numeric or an associative array, you could use the `array<int, Widget>` or `array<string, Widget>` annotations respectively.

Return-type annotations

On lines 13 and 20, we annotate the methods' return types. The `addWidget()` method returns itself (more on this soon). The `getWidget()` method returns a `Widget` instance. Return-type annotations are declared *after* the method signature's closing parenthesis and *before* the method body's opening bracket.

The exception to this rule is the `__construct()` method. One might think the constructor's return value is `void`; it's not. You should not annotate the constructor method's return type.

Some developers like to enable *method chaining*. This means that a class method returns itself so that multiple method calls can be chained together like this:

```
$object->methodOne()->methodTwo();
```

Hack lets you annotate this behavior with the `this` return type. We use the `this` annotation with the `addWidget()` method on line 13.

Hack Data Structures

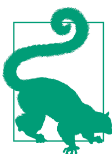
The Hack language's headline feature is static typing. However, Hack also provides new data structures and interfacs that are not found in PHP. These can potentially save you development time versus implementing similar workarounds in vanilla PHP. Some of Hack's new data structures and interfaces are:

- Collections (vectors, maps, sets, and pairs)
- Generics
- Enums
- Shapes
- Tuples

Many of these data structures complement, clarify, or supplement PHP's functionality. For example, Hack's Collection interfaces clarify PHP's array ambiguity. Generics let you create data structures to handle homogenous values of a given type that is inferred only when an instance of the generic class is created; this alleviates the need to manually enforce type checking inside a class with PHP's `instanceof` method. Enums are helpful for creating a set of named constants without resorting to abstract classes. Shapes help you type-check data structures that should have a fixed set of keys. And tuples let you use arrays of an immutable length.

Please don't feel like you need to rush out and implement all of these data structures. I admit, some of them are of limited and niche utility. Some data structures duplicate

(and extend) functionality found in other data structures. I suggest you read up on which data structures are available and only use them if and when you need them.



I believe the most useful Hack data structures are the various Collection interfaces. These provide more appropriate and predictable behavior than PHP's `array` data structure. It's best to use a Collection instead of a PHP array.

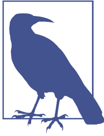
HHVM/Hack vs. PHP

If HHVM and Hack are so awesome, why should you use PHP? I'm asked this question a lot. I'm also asked if and when PHP will meet its demise. The answer is not black-and-white. It's more a muddy neutral gray.

HHVM is the first true competitor to the traditional Zend Engine PHP runtime. HHVM is proven to perform better and be more memory-efficient than the PHP 5.6 Zend Engine on many real-world benchmarks. I think this caught the PHP core development team by surprise. In fact, HHVM's mere existence is probably responsible for PHP 7's increased performance and reduced memory usage. PHP 7 performance is competitive with, if not better than, HHVM. The point is that HHVM creates competition, and competition helps everyone. Both HHVM and the Zend Engine will improve, and PHP developers will reap the benefits. Neither HHVM nor the Zend Engine is going to win or lose. I believe they will coexist and feed off of their competitive energies.

The Hack language, in my opinion, is better than PHP. There are several reasons for this. First, the Hack language was built by Facebook to answer specific needs. It is focused. It has purpose. And it is not developed by committee. The PHP language, in contrast, has evolved piecemeal over a longer period of time. PHP answers many different needs, and it is controlled by a committee that is not known for its cordial agreements. As of PHP 5.x, the Hack language is the better option for its strict type checking and support for legacy PHP code. I believe a lot of Hack's best features will eventually find their way into PHP. And vice versa. In fact, the Hack language team has said it intends to maintain future compatibility with the Zend Engine. Again, I believe competition will improve both languages and they'll enjoy a symbiotic relationship.

An example of this symbiosis is the official PHP specification. Until recently, the PHP language *was* the Zend Engine for lack of alternative implementations. The introduction of HHVM prompted several developers at Facebook to announce a PHP language specification. This specification is an amazing development in the PHP community, and it ensures that current and future PHP implementations (Zend Engine, HHVM, and so on) all support the same fundamental language.



You can read the official PHP implementation on GitHub at *https://github.com/php/php-langspec*.

Further Reading

We've touched on a lot of HHVM and the Hack language in a very short period of time. There are simply not enough pages to cover everything these two initiatives have to offer. Instead, I'll point you to these helpful resources:

- *http://hhvm.com*
- *http://hacklang.org*
- @ptarjan on Twitter
- @SaraMG on Twitter
- @HipHopVM on Twitter
- @HackLang on Twitter

CHAPTER 8
Community

The PHP community is your most valuable resource. It is diverse, vibrant, and global. I encourage you to participate in the PHP community to learn from and share with other PHP developers. There's *always* more to learn, and your PHP community is the best way to continue learning. It's also a great way to meet and help other developers.

Local PUG

My first advice is to find and join your local PHP User Group (PUG). Many cities have them. You can find your local PUG at *https://php.ug/*. Your local PUG is the best opportunity to meet and network with fellow PHP developers in your local community.

If there isn't a nearby PUG, you have several options. You can start your own PUG. Unless you live in the middle of a jungle, I bet there are like-minded nearby PHP developers who would love to join a PUG. Otherwise, you can join NomadPHP—an online user group with monthly speakers and lightning talks that cover all sorts of PHP features and practices.

Conferences

There are numerous PHP conferences every year. Conferences are an excellent opportunity to meet and mingle with the greatest minds in the PHP community. You can listen to and talk with PHP speakers and thought leaders. And you can stay up-to-date with emerging features and modern practices. Conferences are also an excuse to take a minivacation. You can find a list of upcoming PHP conferences at *https://php.net/conferences/*.

Mentoring

If you are a beginner PHP developer and need advice or assistance, you can find a mentor at *https://phpmentoring.org/*. Many expert PHP developers donate their time to help new PHP developers become better. If you are already an expert PHP developer, consider signing up as a PHP mentor. There are many beginner PHP developers who don't know how or where to start, and your mentorship will be invaluable.

Stay Up-to-Date

The PHP language changes frequently. Here are a few resources to help you stay up-to-date with newer PHP features and modern practices.

Websites

- *https://php.net*
- *https://php.net/docs.php*
- *http://www.php-fig.org*
- *http://www.phptherightway.com*

Mailing Lists

- *https://php.net/mailing-lists.php*

Twitter

- @official_php
- @phpc

Podcasts

- *https://voicesoftheelephpant.com/*
- *http://looselycoupled.info*
- *http://elephantintheroom.io*
- *http://phptownhall.com*
- *http://devhell.info*
- *http://www.phpclasses.org/blog/category/podcast/*

- *http://threedevsandamaybe.com/*

Humor

- @phpbard
- @phpdrama

About the Author

Josh Lockhart created the Slim Framework, a popular PHP micro framework that enables rapid Web application and API development. Josh also started and currently curates PHP The Right Way, a popular initiative in the PHP community that encourages good practices and disseminates quality information for PHP developers around the world.

Josh is a developer at New Media Campaigns, a full-service web design, development, and marketing agency in Carrboro, North Carolina. He enjoys building custom applications with HTML, CSS, PHP, JavaScript, Bash, and various content management frameworks.

He graduated from the Information and Library Science program at the University of North Carolina at Chapel Hill in 2008. He currently resides in Chapel Hill, North Carolina with his wonderful wife, Laurel, and their two dogs.

You can follow Josh on Twitter, read his blog at *https://joshlockhart.com*, and track his open source projects on GitHub.