



Современный PHP

Джош Локхарт



O'REILLY®

Modern PHP

New Features and Good Practices

Josh Lockhart

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Современный РНР

Новые возможности и передовой опыт

Джош Локхарт



Москва, 2016

УДК 004.738.5:004.438РНР

ББК 32.973.4

Л73

Л73 Джош Локхарт

Современный PHP. Новые возможности и передовой опыт / пер. с англ. Рагимов Р. Н – М: ДМК Пресс, 2016. – 304 с.: ил.

ISBN 978-5-97060-184-6

Из книги вы узнаете, как PHP превратился в зрелый полнофункциональный объектно-ориентированный язык, с пространствами имен и постоянно растущей коллекцией библиотек компонентов. Автор демонстрирует новые возможности языка на практике. Вы узнаете о передовых методах проектирования и конструирования приложений, работы с базами данных, обеспечения безопасности, тестирования, отладки и развертывания.

Если вы уже знакомы с языком PHP и желаете расширить свои знания о нем, то эта книга для вас!

УДК 004.738.5:004.438РНР

ББК 32.973.4

Original English language edition published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. Copyright © 2015 O'Reilly Media, Inc. Russian-language edition copyright © 2015 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

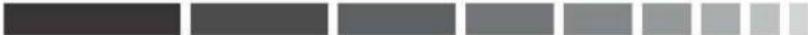
ISBN 978-1-49190-501-2 (англ.)

ISBN 978-5-97060-184-6 (рус.)

Copyright © 2015 Josh Lockhart

**© Оформление, перевод на русский язык,
ДМК Пресс, 2016**

Лорел посвящается



ОГЛАВЛЕНИЕ

Об авторе	13
Предисловие	14
Что нужно знать об этой книге	14
Структура книги	15
Соглашения, принятые в этой книге	16
Использование примеров кода	16
Как связаться с нами.....	17
Благодарности.....	17
ЧАСТЬ I.	
Особенности языка	19
Глава 1. Новый PHP	20
Прошлое	20
Настоящее	21
Будущее	23
Глава 2. Особенности	25
Пространства имен	25
Зачем нужны пространства имен	28
Объявление	28
Импорт и псевдонимы	30
Полезные советы.....	32
Интерфейсы.....	34
Трейты	39
Зачем нужны трейты	39
Как создать трейт	41
Как использовать трейт.....	43
Генераторы	44
Создание генератора.....	45
Использование генератора.....	46
Замыкания	48
Создание	49
Прикрепление состояния.....	50
Расширение Zend OPcache	53



Включение расширения Zend OPcache	54
Настройка расширения Zend OPcache	55
Использование расширения Zend OPcache.....	56
Встроенный HTTP-сервер	56
Запуск сервера.....	57
Настройка сервера	57
Сценарии маршрутизации	58
Обнаружение встроенного сервера	58
Недостатки	59
Что дальше.....	59
ЧАСТЬ II.	
Передовые технологии	61
Глава 3. Стандарты	62
PHP-FIГ приходит на помощь	63
Совместимость фреймворков	64
Интерфейсы	64
Автозагрузка.....	64
Стиль	65
Что такое PSR?.....	65
PSR-1: Базовый стиль оформления кода	66
PSR-2: Строгий стиль оформления кода.....	68
PSR-3: Интерфейс журналирования	73
Создание компонента журналирования PSR-3	73
Использование компонента журналирования PSR-3	74
PSR-4: Автозагрузка.....	75
Почему автозагрузка так важна	76
Модель автозагрузки PSR-4	77
Как написать автозагрузчик PSR-4 (и почему этого делать не нужно)	78
Глава 4. Компоненты	80
Почему надо использовать компоненты?.....	80
Что представляют собой компоненты?	81
Компоненты и фреймворки	83
Не все фреймворки плохи.....	84
Использование инструмента, соответствующего задаче	85
Поиск компонентов	85
Магазин	86
Выбор	87
Оставьте отзыв	88
Использование PHP-компонентов	88
Установка Composer	89

Как использовать Composer	90
Пример проекта	92
Composer и закрытые хранилища	96
Создание PHP-компонентов	98
Имена производителя и пакета	98
Пространства имен	99
Организация файловой системы	99
Файл composer.json	100
Файл README	103
Реализация компонента	103
Управление версиями	106
Размещение на сайте Packagist	106
Использование компонента	107
Глава 5. Передовой опыт.....	109
Санирование и проверка ввода, и экранирование вывода	110
Санирование ввода	110
Проверка данных	114
Экранирование вывода	115
Пароли	116
Не храните пароли в открытом виде	116
Не ограничивайте пароли ваших пользователей	116
Не отправляйте пароли пользователей по электронной почте	117
Хеширование паролей пользователей с помощью bcrypt	117
Программный интерфейс хеширования паролей	119
Программный интерфейс хеширования паролей для PHP < 5.5.0	124
Даты, время и часовые пояса	125
Установка часовового пояса по умолчанию	125
Класс DateTime	125
Класс DateInterval	127
Класс DateTimeZone	128
Класс DatePeriod	129
Компонент nesbot/carbon	130
Базы данных	131
Расширение PDO	131
Подключение базы данных и DSN	131
Параметризованные запросы	134
Результаты запроса	137
Транзакции	139
Многобайтовые строки	143
Кодировка символов	144
Отображение данных в кодировке UTF-8	145
Потоки данных	145
Обертки потоков	146
Контекст потока	150
Фильтры потоков	150

Пользовательские фильтры потоков	153
Ошибки и исключения	156
Исключения	157
Обработчики исключений	161
Ошибки.....	162
Обработчики ошибок	164
Ошибки и исключения в ходе разработки.....	166
Эксплуатация.....	168
ЧАСТЬ III.	
Развертывание, тестирование и настройка	171
Глава 6. Хостинг	172
Разделяемые серверы	172
Виртуальный выделенный сервер	173
Выделенный сервер	174
PaaS.....	175
Выбор тарифного плана хостинга.....	176
Глава 7. Комплектование	177
Наша цель	178
Настройка сервера.....	178
Первый вход	178
Обновление программного обеспечения	179
Непривилегированный пользователь	180
SSH-аутентификация с помощью парных ключей	181
Отключение парольной аутентификации и запрет входа пользователя root	183
PHP-FPM	184
Установка	184
Глобальная конфигурация	185
Настройка пулов	186
nginx.....	189
Установка	190
Виртуальный хост	190
Автоматизация комплектования	193
Делегирование комплектования	194
Дополнительные материалы	194
Что дальше.....	195
Глава 8. Настройка.....	196
Файл php.ini	196
Память	197

Zend OPcache	198
Выгрузка файлов	201
Максимальное время выполнения	202
Обслуживание сеансов	203
Буферизация вывода	204
Кэш Realpath	204
Что дальше	205
Глава 9. Развёртывание	206
Управление версиями	206
Автоматизация развертывания	207
Сделайте развертывание простым	207
Сделайте развертывание предсказуемым	207
Сделайте развертывание обратимым	207
Capistrano	207
Как это работает	208
Установка	208
Настройка	209
Аутентификация	211
Подготовка удаленного сервера	211
Обработчики Capistrano	212
Развертывание приложения	213
Откат к предыдущей версии приложения	213
Дополнительные материалы	213
Что дальше	213
Глава 10. Тестирование	214
Почему мы тестируем?	214
Когда мы тестируем?	215
Перед	215
В процессе	215
После	216
Что мы тестируем?	216
Как мы тестируем?	216
Модульное тестирование	216
Разработка через тестирование (TDD)	217
Разработка, основанная на функционировании (BDD)	217
PHPUnit	219
Структура каталогов	219
Установка PHPUnit	220
Установка Xdebug	221
Настройка PHPUnit	222
Класс Whovian	223
Класс теста WhovianTest	224



Запуск тестов	227
Охват кода	228
Непрерывное тестирование с помощью Travis CI	229
Установка	229
Запуск	230
Дополнительные материалы	231
Что дальше.....	231
Глава 11. Профилирование	232
Когда следует использовать профилировщик.....	232
Типы профилировщиков.....	233
Xdebug	233
Настройка.....	234
Включение	235
Анализ	235
XHProf	235
Установка	236
XHGUI.....	236
Настройка.....	237
Включение	237
Профилировщик New Relic	238
Профилировщик Blackfire	238
Дополнительные материалы	238
Что дальше.....	239
Глава 12. HHVM и Hack	240
HHVM	240
PHP в Facebook	241
Совместимость HHVM с Zend Engine	243
Будет ли HHVM правильным выбором для меня?	243
Установка	244
Настройка.....	245
Расширения.....	246
Мониторинг HHVM с помощью Supervisord.....	246
HHVM, FastCGI и Nginx	248
Язык Hack.....	250
Перевод с PHP на Hack	250
Что такое типы?	251
Статическая типизация	252
Динамическая типизация.....	253
Двойной подход языка Hack	254
Контроль типов в Hack	254
Режимы Hack	255
Синтаксис Hack	256

Структуры данных Hack.....	258
HHVM и Hack против PHP	259
Дополнительные материалы	261
Глава 13. Сообщество	262
Местная группа PHP-разработчиков	262
Конференции	262
Наставничество.....	263
Будьте в курсе	263
Сайты	263
Списки рассылок	263
Твиттер	263
Подкасты	263
Юмор	264
Приложение А. Установка PHP	265
Linux.....	265
Менеджеры пакетов.....	265
Ubuntu 14.04 LTS.....	266
CentOS 7	268
MAMP	270
Homebrew	273
Сборка из исходных текстов.....	277
Получение исходного кода.....	278
Windows.....	284
Скомпилированные файлы	285
WAMP	285
Zend Server	286
Приложение Б. Локальная среда разработки.....	287
VirtualBox.....	288
Vagrant	289
Команды	289
Боксы	290
Инициализация.....	290
Комплектование	291
Синхронизация каталогов	292
Быстрый старт	293
Предметный указатель	295
Об обложке.....	303



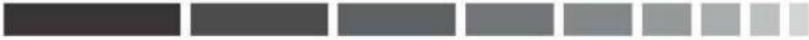
ОБ АВТОРЕ

Джош Локхарт (Josh Lockhart) – создатель Slim Framework (<http://slimframework.com/>), популярного микрофреймворка для PHP, обеспечивающего быструю разработку веб-приложений и программных интерфейсов. Джош также основал и до сих пор курирует «PHP The Right Way» (<http://www.phptherightway.com/>) – популярную в PHP-сообществе инициативу, поощряющую современные методики разработки и распространяющую качественную информацию для PHP-разработчиков по всему миру.

Джош работает программистом в New Media Campaigns (<http://www.newmediacampaigns.com/>), агентстве по предоставлению полного комплекса услуг веб-дизайн, разработки и маркетинга в Каррборо, Северная Каролина. Получает удовольствие от создания приложений с использованием HTML, CSS, PHP, JavaScript, Bash и различных фреймворков управления контентом.

Окончил курс Information and Library Science (<http://sils.unc.edu/>) в Университете Северной Каролины, в Чапел-Хилл, в 2008. В настоящее время проживает в Чапел-Хилл, штат Северная Каролина, вместе со своей замечательной женой Лорел и двумя их собаками.

Джоша можно найти в Твиттере (<https://twitter.com/codeguy>). Его блог доступен по адресу <https://joshlockhart.com>, а его проекты с открытым исходным кодом – на GitHub (<https://github.com/codeguy>).



ПРЕДИСЛОВИЕ

В Интернете можно найти миллионы электронных пособий по PHP. Большинство из них устарели и содержат описание отработавших свое технологий. Но Google продолжает выдавать ссылки на эти пособия, обеспечивая им бессмертие. Устаревшая информация не приносит пользы начинающим PHP-программистам, которые, основываясь на ней, создают медленные и ненадежные PHP-приложения. Я осознал эту проблему в 2013 году, и это явилось основной причиной появления инициативы «PHP The Right Way» (<http://www.phptherightway.com/>) по обеспечению доступа к качественной актуальной информации от авторитетных членов PHP-сообщества.

Книга «Современный PHP» служит той же цели. Эта книга не справочное руководство. Нет. Эта книга представляет собой дружеский и живой разговор между нами. Я познакомлю вас с современным языком программирования PHP. Расскажу о новейших PHP-технологиях, которыми ежедневно пользуюсь в работе и в своих проектах с открытым исходным кодом. И помогу вам начать использовать новейшие стандарты программирования, освоив которые, вы сможете распространять свои PHP-компоненты и библиотеки среди членов PHP-сообщества.

Я часто буду упоминать слово «сообщество», повторяя его снова и снова. PHP-сообщество дружелюбно, приветливо и всегда готово прийти на помощь, хотя, иногда встречаются досадные исключения. Если вам захочется узнать больше об определенной особенности, упомянутой в этой книге, обратитесь в местную группу пользователей PHP с возникшими вопросами. Я уверен, что вокруг вас найдутся PHP-разработчики, которые захотят помочь вам лучше узнать PHP. Ваша местная группа PHP-пользователей это бесценный источник, который позволит вам совершенствоваться в PHP и после завершения чтения этой книги.

Что нужно знать об этой книге

Прежде чем начать, я хочу, остановиться на том, что вы найдете в этой книге. Во-первых, я не смогу описать *все* подходы, используемые

в PHP. На это просто не хватит времени. Вместо этого, я расскажу, как я пользуюсь PHP. Да, это мой личный взгляд на вещи, но я использую те же методы и стандарты, что и многие другие PHP-разработчики. Все, что вы вынесете из нашей краткой беседы, вы сможете сразу же применить в своих проектах.

Во-вторых, я предполагаю, что вы знакомы с переменными, условными операторами, циклами и так далее. От вас не требуется знание языка PHP, но вы должны, по крайней мере, иметь базовое представление об этих фундаментальных понятиях программирования. Я не буду против, если вы захватите с собой кофе (я очень люблю кофе). А я принесу с собой все, что там полагается к кофе.

И в-третьих, я не настаиваю на использовании какой-либо конкретной операционной системы. Однако примеры кода написаны мной для Linux. Bash-команды, которые вы увидите в книге, я использую в Ubuntu и CentOS, но они также будут работать в OS X. Если вы пользуетесь Windows, настоятельно рекомендую установить и настроить виртуальную машину с Linux для опробования примеров кода, прилагаемых к этой книге.

Структура книги

Часть I посвящена описанию новых возможностей языка PHP, таких как пространства имен, генераторы и трейты (traits). Она познакомит вас с современным языком PHP и его особенностями, с которыми вы, возможно, до сих пор не сталкивались.

Часть II рассматривает передовые технологии, которые обязательно следует применять в своих PHP-приложениях. Вы слышали о PSR, но не совсем понимаете, что это такое и как им пользоваться? Вы хотите узнать, как безопасно обработать пользовательский ввод и выполнять защищенные запросы к базе данных? Тогда вам стоит ее прочесть.

Часть III содержит больше технических подробностей, чем первые две части. Она посвящена развертыванию, настройке, тестированию и профилированию PHP-приложений. Мы углубимся в методику развертывания с помощью Capistrano. Поговорим об инструментах тестирования, таких как PHPUnit и Travis CI. Обсудим настройку PHP и ее влияние на работу вашего приложения.

Приложение А содержит пошаговые инструкции по установке и настройке PHP-FPM.

Приложение В описывает процедуру создания локальной среды разработки максимально приближенной к среде действующего сервера. Мы познакомимся с Vagrant, Puppet, Chef и альтернативными им инструментами для быстрого начала работы.

Соглашения, принятые в этой книге

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов, адресов электронной почты, имен файлов и расширений имен файлов .

Моноширинный

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, типов данных, переменных окружения, инструкций и ключевых слов .

Моноширинный жирный

Обозначает команды или другой текст, который должен вводиться пользователем.

Моноширинный курсив

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста .



Так обозначается совет или рекомендация.



Так будут выделены общие примечания.



Так будут выделены предупреждения и предостережения.

Использование примеров кода

Сопроводительные материалы (примеры кода, упражнения и т. д.) можно загрузить на странице <https://github.com/codeguy/modern-php>.

Данная книга призвана оказать вам помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода

из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию необходимо получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издаельства и ISBN. Например: «*Modern PHP* by Josh Lockhart (O'Reilly). Copyright 2015 Josh Lockhart, 978-1-491-90501-2.».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

Как связаться с нами

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.

1005 Gravenstein Highway North Sebastopol, CA 95472

800-998-9938 (США или Канада)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на странице книги http://bit.ly/modern_php.

Свои пожелания и вопросы технического характера отправляйте по адресу bookquestions@oreilly.com.

Мы в Facebook: <http://facebook.com/oreilly>

Мы в Twitter: <http://twitter.com/oreillymedia>

Мы в YouTube: <http://www.youtube.com/oreillymedia>

Благодарности

Это моя первая книга. Когда О'Reilly (O'Reilly) предложил мне написать книгу «Современный PHP», это сильно взволновало меня и перепугало до смерти. Первое, что я сделал, исполнил танец Уолтера Хьюстона (Walter Huston). O'Reilly предложил мне написать книгу.

Это круто!? И тогда я спросил себя, а *смогу ли я написать так много страниц?* Написание книги долгий и трудоемкий процесс.

Конечно же, я сразу ответил «да». Я знал, что смогу написать книгу «Современный PHP», потому что у меня есть семья, друзья, со-служивцы, редакторы, рецензенты, которые и помогли мне во всем. Я хочу выразить признательность и поблагодарить всех своих сторонников за неоценимую поддержку. Без них эта книга никогда бы не была написана.

Прежде всего, я хочу поблагодарить моего редактора Эллисон Макдональд (Allyson MacDonald, *@allyatoreilly*) из O'Reilly Media. Элли мила, требовательна, благосклонна и умна. Она точно знает, как и когда мягко подтолкнуть меня в правильном направлении, когда я терял нить повествования. Я не могу себе представить лучшего редактора.

Я также хочу поблагодарить моих технических рецензентов Адама Феирхолма (Adam Fairholm, *@adamfairholm*) и Эда Финклера (Ed Finkler, *@funkatron*). Адам – блестящий веб-разработчик из Newfangled (<https://www.newfangled.com/>) и, пожалуй, самой известной его работой является IMVDb (<http://imvdb.com/>) – популярная база данных музыкальных клипов. Эд хорошо известен в PHP-сообществе за его невероятные навыки PHP-разработчика, его персональный подкаст */dev/hell* (<http://devhell.info/>) и его заслуживающую всяческих похвал компанию Open Sourcing Mental Illness (<http://funkatron.com/osmi>). Адам и Эд указали мне на все неясные, нелогичные и ошибочные моменты в моих черновиках. Эта книга стала намного лучше благодаря их замечаниям. Я навсегда в долгу перед ними за их советы и здравый смысл. Если ошибки или неточности все же просочились в окончательный вариант рукописи, это, безусловно, моя оплошность.

Мои коллеги из New Media Campaigns (<http://www.newmediacampaigns.com/>) были для меня постоянным источником вдохновения. Джоэл, Клей, Крис, Алекс, Патрик, Эшли, Ленни, Клэр, Тодд, Паскаль, Генри и Натан – снимаю шляпу перед вами за добрые ободряющие слова.

И, самое главное, я хочу поблагодарить мою семью: Лорел, Итан, Тесса, Чарли, Лайза, Гленн и Лиз. Спасибо вам за поддержку, без которой я бы никогда не закончил эту книгу. Моеей любимой жене Лорел отдельное спасибо за терпение. Спасибо тебе за кофе Caribou, когда я засиживался за рукописью допоздна. Спасибо за понимание, когда я писал в выходные дни. Спасибо за сохранение моего творческого настроя. Я люблю тебя сейчас и навсегда.



ЧАСТЬ I

**Особенности
языка**



ГЛАВА 1.

Новый PHP

Язык PHP переживает ренессанс. PHP трансформируется в современный язык сценариев с такими полезными особенностями, как пространства имен, трейты (traits), замыкания и встроенное кэширование байт-кода. Экосистема современного языка PHP также развивается. PHP-разработчики все реже используют монолитные фреймворки и все чаще небольшие специализированные компоненты. Менеджер зависимостей Composer внес революционные изменения в построение PHP-приложений. Он вывел нас из обнесенного неприметными стенами рассадника фреймворков и позволил смешивать и сочетать взаимодействующие между собой PHP-компоненты, лучше всего соответствующие нуждам PHP-приложений. Совместимость компонентов была бы невозможной без предложенных сообществом стандартов и их курирования со стороны PHP Framework Interop Group.

Книга «Современный PHP» является вашим путеводителем по новому PHP, она расскажет, как создавать и развертывать поразительные PHP-приложения, используя стандарты сообщества, лучшие методики и совместимые компоненты.

Прошлое

Прежде, чем приступать к знакомству с современным языком PHP, важно больше узнать о его происхождении. PHP – язык сценариев, интерпретируемый на стороне сервера. Это значит, что вы должны написать PHP-код, выгрузить его на веб-сервер и запустить с помощью интерпретатора. Язык PHP, как правило, используется в паре с веб-серверами Apache или nginx для поддержки динамического контента. Кроме того, язык PHP можно использовать для создания мощных приложений, запускаемых из командной строки (по аналогии с программами на bash, Ruby, Python и т. д.). Многие PHP-разработчики

не знают об этом и упускают из виду эту действительно интересную особенность. Но вы не должны входить в их число.

Вы можете прочитать официальную историю PHP на странице <http://php.net/manual/history.php.php>. Я не буду повторять, что так хорошо сказано создателем PHP Расмусом Лердорфом (Rasmus Lerdorf). Я просто скажу, что PHP имеет противоречивое прошлое. Язык PHP берет начало с коллекции CGI-сценариев, написанных Расмусом Лердорфом для отслеживания посещений его электронного резюме. Лердорф назвал свой набор CGI-сценариев «Инструментами личной домашней страницы» («Personal Home Page Tools»). Это раннее воплощение полностью отличается от сегодняшнего языка PHP, который мы знаем. Изначально инструменты PHP Лердорфа не были языком сценариев; они были инструментами поддержки элементарных переменных и автоматической интерпретации форм, основанных на встроенным синтаксисе HTML.

Между 1994 и 1998 годами, PHP претерпел множество изменений и даже несколько раз был переписан с нуля. Энди Гутманс (Andi Gutmans) и Зеев Сураски (Zeev Suraski), два программиста из Тель-Авива, объединили свои усилия с Расмусом Лердорфом, чтобы превратить PHP из небольшого набора CGI-инструментов в полноценный язык программирования с последовательным синтаксисом и базовой поддержкой объектно-ориентированного программирования. Они назвали свой конечный продукт PHP 3 и выпустили его в конце 1998 года. Новое название «PHP» не совпадало с предыдущим и представляло собой акроним от Hypertext Preprocessor (гипертекстовый процессор). PHP 3 был первой версией, которая уже походила на сегодняшний PHP. Она обеспечивала превосходную расширяемость для поддержки различных баз данных, протоколов и программных интерфейсов. Именно расширяемость PHP 3 привлекла к проекту внимание многих новых разработчиков. К концу 1998 года PHP 3 уже был установлен на ошеломляющих 10% всех веб-серверов.

Настоящее

Сегодня язык PHP быстро развивается и поддерживается десятками программистов со всего мира, входящих в состав основной команды разработчиков. Методики разработки на PHP также меняются. В прошлом программист обычно писал сценарий на PHP и выгружал его на действующий сервер по FTP, в надежде, что он заработает. Такая дикая по современным понятиям стратегия разработки использовалась из-за отсутствия средств локальной разработки.

В настоящее время, мы отказались от FTP, заменив его инструментами управления версиями. Программное обеспечение управления версиями, например Git, позволяет управлять историей развития программного кода, его ветвлением и слиянием ветвей. Локальные среды разработки стали идентичны действующим серверам, благодаря инструментам виртуализации, таким как Vagrant, и инструментам удаленного комплектования, таким как Ansible, Chef и Puppet. Мы используем специализированные PHP-компоненты с помощью менеджера зависимостей Composer. Наш PHP-код соответствует общественным стандартам PSR, курируемых PHP Framework Interop Group. Мы тщательно тестируем свой код с помощью инструментов, таких как PHPUnit. Разворачиваем свои приложения с помощью менеджера PHP-процессов FastCGI на веб-серверах, таких как nginx. И увеличиваем производительность приложений с помощью кэширования байт-кода.

Современный язык PHP поддерживает множество новых методов, которые могут быть вам незнакомы, они либо являются новыми для PHP либо уже имелись в старых версиях PHP и претерпели существенные изменения. Но это не должно вас волновать. Я подробно остановлюсь на каждом таком методе в этой книге.

Я очень рад, что PHP теперь имеет официальную проектную спецификацию, которой так не хватало до 2014 года.



Большинство зрелых языков программирования имеют спецификацию. Говоря простыми словами, спецификация PHP – это набор правил, определяющих, что именно значит быть языком PHP. Спецификация предназначена для разработчиков, создающих программы для анализа, интерпретации и выполнения кода PHP. Она не предназначена для разработчиков приложений и веб-сайтов на PHP.

Сара Гоулман (Sara Golemon) и Facebook анонсировали первую проектную спецификацию PHP на конференции OSCON O'Reilly в 2014 году. Официальный анонс можно найти во внутреннем списке рассылки PHP (<http://bit.ly/php-internals>), а спецификацию PHP – на GitHub (<http://bit.ly/php-langsdoc>).

Официальная спецификация языка PHP становится еще более важной, с учетом появления нескольких конкурирующих между собой движков PHP. Оригинальным движком PHP является Zend Engine (<http://www zend.com/en/company/community/php/>) – PHP-

интерпретатор, написанный на С и введенный в PHP 4. Двигок Zend Engine был создан Расмусом Лердорфом, Энди Гутманом и Зеев Су-раски. Сегодня Zend Engine является основным вкладом компании Zend в PHP-сообщество. Однако в настоящее время появился еще один крупный движок – HipHop Virtual Machine от Facebook. Спецификация языка гарантирует, что оба движка будут обладать базовой совместимостью.



PHP-движок – это программа для анализа, интерпретации и выполнения PHP-кода (например, Zend Engine или HipHop Virtual Machine). Его не следует путать с PHP – названием языка PHP в общем смысле.

Будущее

Двигок Zend Engine развивается быстрыми темпами, обрастая новыми функциями и улучшая производительность. Я объясняю ускоренное прогрессирование движка Zend Engine появлением новых конкурентов, а именно движка *HipHop Virtual Machine* от компании Facebook и языка программирования Hack.

Hack является новым языком программирования, надстройкой над PHP. Он вводит статическую типизацию, новые структуры данных и дополнительные интерфейсы, сохраняя при этом, обратную совместимость с существующей динамической типизацией в PHP. Язык Hack ориентирован на разработчиков, которые ценят быстроту разработки, присущую PHP, но нуждаются в предсказуемости и стабильности статической типизации.

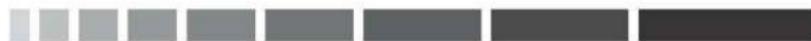


Мы обсудим преимущества и недостатки динамической и статической типизации в этой книге ниже. Разница между ними состоит в моменте проверки типов интерпретатором PHP. Динамические типы проверяются во время выполнения, а статические – во время компиляции. Дополнительную информацию вы найдете в главе 12.

Двигок HipHop Virtual Machine (HHVM) – это интерпретатор языков PHP и Hack, выполняющий *динамическую компиляцию* (just-in-time, сокращенно JIT) для улучшения производительности приложений и экономии памяти.

Я не думаю, что Hack и HHVM полностью заменят Zend Engine, но новые продукты Facebook оказывают огромное влияние на PHP-сообщество. Растущая конкуренция побудила команду Zend Engine анонсировать выход PHP 7 (<https://wiki.php.net/rfc/php7timeline>) и оптимизацию движка Zend Engine, чтобы быть на одном уровне с HHVM. Мы обсудим эти нововведения в главе 12.

Это захватывающее время для PHP-программиста. PHP-сообщество никогда не было таким энергичным и целеустремленным. Я надеюсь, что эта книга поможет вам освоить современные приемы программирования на PHP. Вам предстоит узнать много нового и еще больше нововведений маячит на горизонте. Будем считать эту книгу вашей дорожной картой. Итак, начнем.



ГЛАВА 2. Особенности

Современный язык PHP имеет много новых и интересных особенностей. Многие из этих особенностей являются новыми для программистов, использовавших ранние версии PHP, и станут приятным сюрпризом для программистов, перешедших на PHP с другого языка. Эти новые особенности делают язык PHP мощной платформой и обеспечивают эффективную разработку веб-приложений и инструментов командной строки.

Использование некоторых из этих особенностей не является обязательным, но они облегчают жизнь разработчикам. Некоторые особенности, напротив, имеют существенное значение. Пространства имен, например, являются краеугольным камнем современных стандартов PHP, и на них основывается методология разработки, применение которой современные разработчики на PHP считают само собой разумеющимся (например, автозагрузка). Я познакомлю вас с каждой из новых особенностей, объясню ее полезность и расскажу, как применять ее в ваших проектах.



Я предлагаю вам опробовать все приведенные в книге примеры на своем компьютере. Исходный код всех примеров можно найти в репозитории GitHub (<https://github.com/codeguy/modern-php>) .

Пространства имен

Если бы нужно было выбрать одну единственную особенность современного языка PHP, с которой я мог бы вас познакомить, этой особенностью стало бы *пространство имен*. Введенные в PHP 5.3.0, пространства имен являются важным инструментом организации кода в виртуальную иерархию, сопоставимую со структурой ката-

логов операционной системы. Каждый современный компонент или фреймворк организует свой код в соответствии со своим собственным глобально-уникальным пространством имен производителя, чтобы исключить конфликты с именами классов, используемых другими производителями.



Едва ли кому-то понравится войдя в кафе увидеть, что некий малоприятный субъект разложил книги, провода или еще что-то на нескольких столах. Не говоря уже о том, что он занял единственную доступную электрическую розетку. Он бесполезно занимает пространство, которое могли бы использовать другие люди. Образно говоря, этот человек не использует пространство имен. Не будьте таким человеком.

Давайте посмотрим, как пространства имен используются в настоящем PHP-компоненте. Входящий в фреймворк Symfony компонент `Symfony/httpfoundation` (<https://github.com/symfony/HttpFoundation>) является популярным PHP-компонентом, который управляет запросами и ответами HTTP. Самое важное здесь, что компонент `Symfony/httpfoundation` использует распространенные имена классов, такие как `Request`, `Response` и `Cookie`. Я гарантирую, что существует много других компонентов, которые используют эти же имена классов. Как же использовать компонент `Symfony/httpfoundation`, если другой код использует те же имена классов? Все просто – код компонента `Symfony/httpfoundation` заключен в уникальное пространство имен производителя `Symfony`. Посетите страницу компонента `Symfony/httpfoundation` (<https://github.com/symfony/HttpFoundation>) на сайте GitHub и перейдите к файлу `Response.php` (<http://bit.ly/response.php>). Он выглядит, как показано на рис. 2.1.

Посмотрите внимательно на строку 12:

```
namespace Symfony\Component\HttpFoundation;
```

Это объявление пространства имен и оно всегда помещается с новой строки сразу после открывающего тега `<?php`. Данное объявление пространства имен информирует о нескольких вещах. Во-первых, класс `Response` располагается в пространстве имен `Symfony` производителя (пространство имен производителя является корневым пространством имен). Во-вторых, класс `Response` располагается в подпространстве имен `Component`. Наконец, класс `Response` располагается в

еще одном подпространстве имен `HttpFoundation`. Можете заглянуть в другие файлы, находящиеся рядом с `Response.php`, и вы увидите, что они используют те же объявления пространства имен. Пространство имен (или подпространство имен) содержит соответствующие классы, точно также как каталог файловой системы содержит соответствующие файлы.

```
git clone https://github.com/symfony/HttpFoundation.git
cd HttpFoundation
git checkout master
git pull origin master
git log --oneline
```

```
1277 lines (1142 std:: - 36.00 kB)
Raw Blame History
1 //file
2
3 // ...
4
5 // This file is part of the Symfony package.
6 // ...
7 // (c) Fabien Potencier <fabien@symfony.com>
8 // ...
9 // For the full copyright and license information, please refer to the LICENSE
10 // file that was distributed with this package.
11 // ...
12
13 namespace Symfony\Component\HttpFoundation;
14
15 /**
16 * Response represents an HTTP response.
17 */
18 class Response
19 {
20     const HTTP_CONTINUE = 100;
21     const HTTP_SWITCHING_PROTOCOLS = 101;
22     const HTTP_PROCESSING = 102;
23     const HTTP_OK = 200;
24     const HTTP_CREATED = 201;
25     const HTTP_ACCEPTED = 202;
26     const HTTP_NON_AUTHORITATIVE_INFORMATION = 203;
```

Рис. 2.1. Содержимое файла Response.php из компонента symfony/httpfoundation



Подпространства разделяются символом \.

В отличие от файловой системы, являющейся физическим понятием операционной системы, пространство имен являются *виртуальным* понятием, и не обязано в точности соответствовать каталогам файловой системы. Можно сказать, что структура подпространств имен большинства PHP-компонентов действительно полностью повторяет структуру каталогов файловой системы для совместимости с популярным стандартом автозагрузки PSR-4 (о нем мы поговорим подробнее в главе 3).



С технической точки зрения пространства имен являются лишь системой обозначений языка PHP для получения общих префиксов имен классов, интерфейсов, функций и констант.

Зачем нужны пространства имен

Пространства имен важны, потому что позволяют создавать изолированный код, способный совместно работать с кодом других разработчиков. Они являются краеугольным камнем концепции компонентной экосистемы современного языка PHP. Авторы компонентов и фреймворков создают и распространяют код для множества PHP-разработчиков, и не могут знать или контролировать какие классы, интерфейсы, функции и константы будут использованы вместе с их собственным кодом. Эта же проблема имеет отношение к вашим собственным проектам. Если вы используете в проекте свои компоненты или классы, они должны продолжать работать после включения в проект любых внешних зависимостей.

Как я уже упоминал в связи с компонентом `symfony/httpfoundation`, ваш код и код других разработчиков может использовать одинаковые имена классов, интерфейсов, функций или констант. Без использования пространств имен совпадение имен в PHP вызовет ошибку. С пространствами имен, ваш код и код других разработчиков может использовать одинаковые имена классов, интерфейсов, функций или констант, так как они находятся в уникальных пространствах имен производителей.

Если вы разрабатываете крошечный личный проект с небольшим количеством зависимостей, совпадение имен классов, вероятно, не станет проблемой. Но когда вы работаете в команде, разрабатывающей большой проект с многочисленными внешними зависимостями, предотвращение конфликтов имен становится очень важной задачей. Вы не можете контролировать, какие классы, интерфейсы, функции и константы вводятся в глобальное пространство имен в зависимостях вашего проекта. Вот почему использование пространств имен имеет большое значение.

Объявление

Каждый класс, интерфейс, функция или константа принадлежит пространству (или подпространству) имен. Пространства имен объ-

являются в начале файла, сразу после открывающего тега `<?php`. Объявление пространства имен начинается с ключевого слова `namespace`, за которым следуют пробел, имя пространства имен, а затем – завершающий символ точки с запятой `:`.

Запомните, что пространства имен часто используются для объявления имени производителя. Следующий пример объявления пространства имен определяет имя производителя `Oreilly`:

```
<?php  
namespace Oreilly;
```

Все классы, интерфейсы, функции или константы, объявленные далее, будут заключены в пространство имен `Oreilly` и, в некотором роде, связаны с издательством O'Reilly Media. А если понадобится организовать код, связанный с этой книгой? Мы используем подпространство имен.

Подпространства имен объявляются точно так же, как в предыдущем примере. Разница лишь в том, чтобы добавить подпространство после пространства имен, разделив их символом `\`. В следующем примере объявляется подпространство с именем `ModernPHP`, находящееся внутри корневого пространства имен производителя `Oreilly`:

```
<?php  
namespace Oreilly\ModernPHP;
```

Все классы, интерфейсы, функции и константы, объявленные далее, будут заключены в пространство имен `Oreilly\ModernPHP` и, в некотором роде, связаны с этой книгой.

Все классы одного и того же пространства или подпространства имен не обязательно должны быть объявлены в одном файле. Вы можете указать пространство или подпространство имен в верхней части любого файла, и код этого файла становится частью этого пространства или подпространства имен. Это дает возможность распределить по отдельным файлам классы, которые принадлежат одному общему пространству имен.



Наиболее важным пространством имен является *пространство имен производителя*. Это – корневое пространство имен, идентифицирующее бренд или организацию, и должно быть глобально уникальным. Подпространства менее важны, но и они полезны для упорядочивания кода вашего проекта.

Импорт и псевдонимы

До появления пространств имен, разработчики PHP решали проблему совпадения имен с помощью назначения имен классам в стиле Zend. Эта схема именования классов была применена в фреймворке Zend, где имена классов использовали подчеркивание вместо разделителей каталогов файловой системы. Это соглашение преследовало две цели: гарантировать уникальность имен классов и упростить работу автозагрузчика путем замены подчеркиваний в именах PHP-классов разделителями каталогов файловой системы для определения путей к файлам классов.

Например, класс `Zend_Cloud_DocumentService_Adapter_WindowsAzure_Query` соответствовал файлу `Zend/Cloud/DocumentService/Adapter/WindowsAzure/Query.php`. Побочным эффектом от применения имен в стиле Zend, как можно видеть, являются абсурдно длинные имена класса. Можете считать меня ленивым, но ничто не заставит меня ввести такое имя класса более одного раза.

Пространства имен в современном языке PHP не лишены той же проблемы. Например, класс `Response` в компоненте `symfony\httpfoundation` имеет полное имя `\Symfony\Component\HttpFoundation\Response`. К счастью, PHP предоставляет возможность *импортировать* и использовать *псевдонимы* пространств имен.

Под импортированием имеется в виду определение пространств имен, классов, интерфейсов, функций и констант, которые будут использованы в каждом файле, после чего их можно использовать *без ввода полных имен*.

Под использованием псевдонимов имеется в виду возможность определения более кратких имен для ссылки на импортированные классы, интерфейсы, функции и константы.

Код, приведенный в примере 2.1, создает и отсылает HTTP-ответ `400 Bad Request` без импортирования и определения псевдонимов.

Пример 2.1. Пространство имен без псевдонима

```
<?php  
$response = new \Symfony\Component\HttpFoundation\Response ('Oops', 400);  
$response->send();
```

Вроде бы ничего страшного, но представьте, что вам нужно будет несколько раз создать экземпляр `Response` в одном файле. Ваши пальцы устанут от ввода. Теперь взгляните на пример 2.2. Он делает то же самое, но *использует импорт*.



Импортирование и определение псевдонимов классов, интерфейсов и прочих пространств имен можно осуществлять так же, как в PHP 5.3, а импортирование и определение псевдонимов функций и констант – как в PHP 5.6.

Пример 2.2. Пространство имен с псевдонимом по умолчанию

```
<?php
use Symfony\Component\HttpFoundation\Response;

$response = new Response('Oops', 400);
$response->send();
```

С помощью ключевого слова `use` мы указываем интерпретатору, что намерены использовать класс `Symfony\Component\HttpFoundation\Response`. Вводим длинное полное имя класса один раз и теперь можем создавать экземпляры класса `Response` без использования его полного имени. Выглядит круче?

Иногда я чувствую себя особенно ленивым, и тогда я использую псевдонимы. Давайте расширим пример 2.2. Допустим, что вместо `Response` мне захотелось вводить имя `Res`. Пример 2.3 показывает, как это сделать.



Импортировать имена с помощью ключевого слова `use` следует в начале каждого файла, сразу после открывающего тега `<?php` или объявления пространства имен.

При импортировании с использованием ключевого слова `use` не нужно использовать ведущий символ `\`, так как PHP предполагает, что имена пространств указываются полностью.

Ключевое слово `use` должно находиться в глобальной области (т. е. не внутри класса или функции), потому что оно используется во время компиляции. Оно может, однако, располагаться ниже объявления пространства имен и импортировать имена из другого пространства имен.

Пример 2.3. Пространство имен с пользовательским псевдонимом

```
<?php
use Symfony\Component\HttpFoundation\Response as Res;

$sr = new Res('Oops', 400);
$sr->send();
```

В этом примере, я изменил строку импорта класса Response. Я добавил к ней текст `as Res`, определив, что `Res` надо рассматривать как псевдоним класса `Response`. В отсутствие `as Res` в строке импорта PHP будет использовать псевдоним по умолчанию – само имя импортированного класса.

В PHP 5.6 имеется возможность импорта функций и констант. Для этого требуется подправить синтаксис ключевого слова `use`. Чтобы импортировать функцию, замените `use` на `use func`:

```
<?php  
use func Namespace\functionName;  
  
functionName();
```

Чтобы импортировать константу, замените `use` на `use constant`:

```
<?php  
use constant Namespace\CONST_NAME;  
  
echo CONST_NAME;
```

Псевдонимы функций и констант работают точно также как псевдонимы классов.

Полезные советы

Множественный импорт

Чтобы импортировать несколько классов, интерфейсов, функций или констант в один файл, нужно поместить несколько операторов `use` в начало файла. PHP воспринимает сокращенный синтаксис импорта, который позволяет объединить несколько операторов `use` в одну строку, например:

```
<?php  
use Symfony\Component\HttpFoundation\Request,  
      Symfony\Component\HttpFoundation\Response,  
      Symfony\Component\HttpFoundation\Cookie;
```

Не делайте этого. Это сбивает с толку и вносит путаницу. Я рекомендую помешать каждый оператор `use` в отдельную строку, как показано ниже:

```
<?php  
use Symfony\Component\HttpFoundation\Request;  
use Symfony\Component\HttpFoundation\Response;  
use Symfony\Component\HttpFoundation\Cookie;
```

Ваш код станет длиннее на несколько символов, но будет легче читаться и корректироваться.

Несколько пространств имен в одном файле

PHP позволяет определить несколько пространств имен в единственном PHP-файле, например:

```
<?php
namespace Foo {
    // Объявления классов, интерфейсов, функций и констант
}

namespace Bar {
    // Объявления классов, интерфейсов, функций и констант
}
```

Однако такой подход противоречит рекомендуемой методике *один класс в одном файле*. Используйте в каждом файле только одно пространство имен, чтобы сделать код более простым для чтения и корректировки.

Глобальное пространство имен

Встретив ссылку на класс, интерфейс, функцию или константу *без указания пространства имен*, PHP предполагает, что класс, интерфейс, функция или константа находится в текущем пространстве имен. Если это предположение оказывается неверным, PHP попытается найти класс, интерфейс, функцию или константу. Если нужно сослаться на класс, интерфейс, функцию или константу *внутри другого пространства имен*, необходимо использовать полное имя класса (имя пространства имен + имя класса). Вы можете ввести полное имя класса или импортировать код в текущее пространство имен с помощью ключевого слова `use`.

Код, не имеющий своего пространства имен, существует в *глобальном пространстве имен*. Встроенный класс `Exception` лучший тому пример. Чтобы сослаться на имя в глобальном пространстве имен из кода внутри другого пространства имен, можно предварить его символом `\`. Например, метод `\My\App\Foo::doSomething()` в примере 2.4 вызовет ошибку, потому что PHP попытается найти класс `\My\App\Exception`, которого не существует.

Пример 2.4. Неквалифицированное имя класса внутри чужого пространства имен

```
<?php
```

```
namespace My\App;

class Foo
{
    public function doSomething()
    {
        $exception = new Exception();
    }
}
```

Добавьте префикс \ к имени класса `Exception`, как показано в примере 2.5. Это подскажет интерпретатору PHP, что класс `Exception` следует искать не в текущем, а в глобальном пространстве имен.

Пример 2.5. Квалифицированное имя класса внутри чужого пространства имен

```
<?php
namespace My\App;

class Foo
{
    public function doSomething()
    {
        throw new \Exception();
    }
}
```

Автозагрузка

Пространства имен также представляют основу для работы автозагрузчика, определяемого стандартом PSR4, созданным PHP Framework Interop Group (PHP-FIG). Этот автозагрузчик используется в большинстве современных PHP-компонентов и позволяет выполнять автозагрузку зависимостей проекта с помощью менеджера зависимостей Composer. Поближе с Composer и PHP-FIG мы познакомимся в главе 4. А сейчас просто запомните, что современная экосистема языка PHP и ее компонентная архитектура была бы невозможна без пространств имен.

Интерфейсы

Освоение интерфейсов изменило мою жизнь, как программиста на PHP, и позволило легко интегрировать PHP-компоненты сторонних разработчиков в мои приложения. Интерфейсы не являются новой возможностью, но являются важной функцией, которую вы должны знать и использовать ежедневно.

Так что же такое интерфейс в языке PHP? Интерфейс – это соглашение между двумя PHP-объектами, которое позволяет одному объекту вне зависимости от того, чем является другой объект, пользоваться возможностями этого другого объекта. Интерфейсы помогают отделить код от его зависимостей и использовать код сторонних разработчиков, реализующий ожидаемый интерфейс. Нас не должно волновать, как сторонний код реализует интерфейс, нас заботит только, что он *выполняет*, реализуя интерфейс. Приведу обыденный пример.

Представим, что я только что прибыл в Майами, штат Флорида, на конференцию разработчиков Sunshine PHP. Мне нужно средство для передвижения по городу, поэтому я направляюсь к местной площадке проката автомобилей. У них имеются крошечный компактный Hyundai, универсал Subaru и (к моему большому удивлению) Bugatti Veyron. Мне нужно средство для передвижения по городу и все три транспортных подходят для этого. Но каждый автомобиль делает это по-разному. Hyundai Accent неплох, но хотелось бы чуть больше шарма. У меня нет детей, поэтому универсал для меня слишком просторен. Пожалуй, я возьму Bugatti.

В действительности я могу управлять любым из этих трех автомобилей, потому что все они обеспечивают одинаковый ожидаемый интерфейс. Каждый автомобиль имеет руль, педаль газа, педаль тормоза и указатели поворотов, и каждый использует бензин в качестве топлива. Bugatti, вероятно, обладает большей мощностью, чем та с которой я смогу справиться, но интерфейс управления у него такой же, как у Hyundai. Так как все три машины имеют один и тот же ожидаемый интерфейс, у меня есть возможность выбора предпочтаемого лично мной автомобиля (и если честно, то я, наверное, остановлюсь на Hyundai).

Это соответствует концепции объектно-ориентированного языка PHP. Если я напишу код, который будет полагаться на объект определенного класса (и, следовательно, на конкретную реализацию), мой код станет по своей сути ограниченным, потому что сможет использовать объекты только одного класса. Однако если я напишу код, который *полагается на интерфейс*, он сможет использовать любой объект, реализующий этот интерфейс. Мой код не будет волновать, как реализован интерфейс, его будет заботить только то, для чего реализован интерфейс. Давайте подкрепим это примером.

У меня есть некий PHP-класс с именем `DocumentStore`, который собирает тексты из разных источников: извлекает HTML-тексты из

Интернета, читает потоки ресурсов и осуществляет сбор выходных данных команд в консоли. Каждый документ, сохраняется в экземпляре `DocumentStore` и имеет уникальный идентификатор. Определение класса `DocumentStore` приводится в примере 2.6.

Пример 2.6. Определение класса DocumentStore

```
class DocumentStore
{
    protected $data = [];

    public function addDocument(Documentable $document)
    {
        $key = $document->getId();
        $value = $document->getContent();
        $this->data[$key] = $value;
    }

    public function getDocuments()
    {
        return $this->data;
    }
}
```

Как же это сработает, если метод `addDocument()` принимает только экземпляры `Documentable`? Это хорошее наблюдение. Однако, `Documentable` не является классом. Это – интерфейс, определение которого приводится и в примере 2.7.

Пример 2.7. Определение интерфейса Documentable

```
interface Documentable
{
    public function getId();

    public function getContent();
}
```

Это определение указывает, что любой объект, реализующий интерфейс `Documentable`, должен иметь общедоступные методы `getId()` и `getContent()`.

И какая от этого польза? Интерфейс полезен тем, что позволяет создавать классы для выборки документов с совершенно разной реализацией. В примере 2.8 приводится реализация, извлекающая HTML-тексты по удаленным адресам с помощью утилиты `curl`.

Пример 2.8. Определение класса HtmlDocument

```
class HtmlDocument implements Documentable
{
    protected $url;

    public function __construct($url)
    {
        $this->url = $url;
    }

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

    public function getContent()
    {
        $ch = curl_init();
        curl_setopt($ch, CURLOPT_URL, $this->url);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
        curl_setopt($ch, CURLOPT_CONNECTTIMEOUT, 3);
        curl_setopt($ch, CURLOPT_FOLLOWLOCATION, 1);
        curl_setopt($ch, CURLOPT_MAXREDIRS, 3);
        $html = curl_exec($ch); curl_close($ch);

        return $html;
    }
}
```

Другая реализация (пример 2.9) может читать поток ресурсов.

Пример 2.9. Определение класса StreamDocument

```
class StreamDocument implements Documentable
{
    protected $resource;
    protected $buffer;

    public function __construct($resource, $buffer = 4096)
    {
        $this->resource = $resource;
        $this->buffer = $buffer;
    }

    public function getId()
    {
        return 'resource-' . (int)$this->resource;
    }

    public function getContent()
```

```

    }

    $streamContent = '';
    rewind($this->resource);
    while (feof($this->resource) === false) {
        $streamContent .= fread($this->resource, $this->buffer);
    }

    return $streamContent;
}

}

```

И еще одна реализация (пример 2.10), извлекающая результат консольной команды.

Пример 2.10. Определение класса CommandOutputDocument

```

class CommandOutputDocument implements Documentable
{
    protected $command;

    public function __construct($command)
    {
        $this->command = $command;
    }

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

    public function getContent()
    {
        return shell_exec($this->command);
    }
}

```

Пример 2.11 демонстрирует использование класса DocumentStore с тремя реализациями сбора документов.

Пример 2.11. Класс DocumentStore

```

<?php
$documentStore = new DocumentStore();

// Добавление HTML-документа
$htmlDoc = new HtmlDocument('https://php.net');
$documentStore->addDocument($htmlDoc);

// Добавление документа из потока

```

```
$streamDoc = new StreamDocument(fopen('stream.txt', 'rb'));
$documentStore->addDocument($streamDoc);

// Добавление вывода консольной команды
$cmdDoc = new CommandOutputDocument('cat/etc/hosts');
$documentStore->addDocument($cmdDoc);

print_r($documentStore->getDocuments());
```

Это выглядит необычно, потому что классы `HTMLDocument`, `StreamDocument` и `CommandOutputDocument` не имеют ничего общего кроме интерфейса.

В конце концов, использование интерфейсов ведет к созданию гибкого кода, реализация которого может быть поручена другим разработчикам. Множество людей (например, сотрудники вашего отдела, пользователи вашего проекта с открытым кодом или разработчики, с которыми вы никогда не пересекались) смогут написать код, который будет взаимодействовать с вашим кодом, ничего не зная о нем, кроме интерфейса.

Трейты

Многих из моих друзей PHP-разработчиков смущают *трейты* (traits), новое понятие, введенное в PHP 5.4.0. Трейты ведут себя подобно классам, но выглядят как интерфейсы. Ну и что же это такое? Ни то ни другое или то и другое одновременно.

Трейты – это классы с частичной реализацией (то есть константами, свойствами и методами), которая может включаться в существующие классы. Трейты исполняют две обязанности: определяют, что класс может делать (как интерфейсы) и обеспечивают модульность реализации (как классы).



Вы можете быть знакомы с аналогами трейтов в других языках программирования. Например, трейты PHP похожи на компонуемые модули или *примеси* (*mixins*) в Ruby.

Зачем нужны трейты

Язык PHP использует классическую модель наследования. Это означает, что любая иерархия всегда начинается с одного общего кор-

невого класса, обеспечивающего базовую реализацию. Далее корневой класс *расширяется* для создания специализированных классов, которые *наследуют* реализации своих непосредственных родителей. Это называется иерархией наследования, и это – общая модель, используемая многими языками программирования.

Классическая модель наследования работает хорошо почти всегда. Но, как быть, если двум несвязанным наследованием классам нужно придать схожее поведение? Например, класс магазина `RetailStore` и класс автомобиля `Car` являются очень разными классами, и не имеют общего предка в иерархии наследования. Тем не менее, оба класса должны выдавать свои географические координаты, широту и долготу для отображения на карте.



Если это поможет, обратимся к школьному курсу биологии. Помните, как вы узнали о системе биологической классификации? Есть шесть царств. Каждое царство делится на типы. Каждый тип делится на биологические классы. Классы делятся на отряды, отряды на семейства, семейства на роды, роды на виды. Каждое иерархическое расширение представляет собой дальнейшую специализацию.

Трейты были созданы именно для этой цели. Они поддерживают модульную реализацию, которая может *внедряться* в ничем не связанные классы. Кроме того, трейты упрощают повторное использование кода.

Первая идея (не самая лучшая), что приходит в голову: создать общий родительский класс `Geocodable` и унаследовать его в классах `RetailStore` и `Car`. Это *плохое* решение, потому что заставляет два несвязанных класса иметь общего предка, что нарушит естественность иерархии наследования.

Вторая идея (более удачная) – определить интерфейс `Geocodable` с методами определения географического положения. Классы `RetailStore` и `Car` смогут реализовать интерфейс `Geocodable`. Это хорошее решение, позволяющее сохранить классам естественную иерархию наследования, но оно требует дублирования кода определяющего географическое положение, что противоречит принципу DRY.

Моей третьей (и самой лучшей) идеей будет создание трейта `Geocodable`, объявляющего и реализующего методы определения географического положения. Затем я могу включить трейт `Geocodable`

в классы `RetailStore` и `Car`, не загрязняя их естественную иерархию наследования.



DRY является аббревиатурой от *Do not repeat yourself* (Не повторяйся). Считается хорошим правилом никогда не дублировать один и тот же код в нескольких местах. При этом вы не должны будете изменять код в одном месте, только потому, что изменили такой же код в другом месте. Более подробную информацию можно найти в Википедии ([https://ru.wikipedia.org/wiki/Don't_repeat_yourself](https://ru.wikipedia.org/wiki/Don%27t_repeat_yourself)).

Как создать трейт

Вот как определяется трейт в PHP:

```
<?php
trait MyTrait {
    // Здесь находится реализация трейта
}
```

Давайте вернемся к нашему примеру `Geocodable` и покажем применение трейтов на практике. Мы сошлись на том, что классам `RetailStore` и `Car` следует дать возможность определения географического положения и решили, что наследование и интерфейсы не являются лучшим решением. Вместо этого, мы создадим трейт `Geocodable`, который возвращает координаты широты и долготы для нанесения на карту. Полное определение трейта `Geocodable` приводится в примере 2.12.



Рекомендуется размещать только один трейт в файле, так же как один класс или интерфейс.

Пример 2.12. Определения трейта `Geocodable`

```
<?php
trait Geocodable {
    /** @var string */
    protected $address;
    /**
     * @var \Geocoder\Geocoder */
}
```

```
protected $geocoder;

/** @var \Geocoder\Result\Geocoded */
protected $geocoderResult;

public function setGeocoder(\Geocoder\GeocoderInterface $geocoder)
{
    $this->geocoder = $geocoder;
}

public function setAddress($address)
{
    $this->address = $address;
}

public function getLatitude()
{
    if (isset($this->geocoderResult) === false) {
        $this->geocodeAddress();
    }

    return $this->geocoderResult->getLatitude();
}

public function getLongitude()
{
    if (isset($this->geocoderResult) === false) {
        $this->geocodeAddress();
    }

    return $this->geocoderResult->getLongitude();
}

protected function geocodeAddress()
{
    $this->geocoderResult = $this->geocoder->geocode($this->
address);

    return true;
}
```

Трейт `Geocodable` включает только свойства и методы, необходимые для определения географического положения. Он не делает ничего больше.

Наш трейт `Geocodable` определяет три свойства класса: адрес (строка), объект геокодера (экземпляр класса `\Geocoder\Geocoder` из пре-
восходного компонента `willdurand/geocoder` (<http://geocoder.php>).

`org()` Уильяма Дюрана (William Durand)), и объект для сохранения результата, возвращаемого геокодером (экземпляр класса `\Geocoder\Result\Geocoded`). Мы также определили четыре общедоступных и один защищенный метод. Метод `setGeocoder()` используется для внедрения объекта геокодера. Метод `setAddress()` используется для установки адреса. Методы `getLatitude()` и `getLongitude()` возвращают широту и долготу, соответственно. А метод `geocodeAddress()` передает адресную строку в экземпляр класса `Geocoder` для получения результата геокодирования.

Как использовать трейт

Использовать трейты в PHP очень просто. Добавьте код `use MyTrait;` в определение PHP-класса, как показано ниже. Естественно, замените `MyTrait` на соответствующее имя трейта:

```
<?php
class MyClass
{
    use MyTrait;

    // Здесь находится реализация класса
}
```

Вернемся к нашему примеру `Geocodable`. Мы определили трейт `Geocodable` в примере 2.12. Теперь обновим наш класс `RetailStore` так, чтобы он использовал трейт `Geocodable` (пример 2.13). Для краткости я не привожу полную реализацию класса `RetailStore`.



Пространства имен и трейты импортируются с помощью ключевого слова `use`. В чем же состоит разница при импорте. Пространства имен, классы, интерфейсы, функции и константы импортируются *вне определения* класса. Трейты импортируются *внутри определения* класса. Различие тонкое, но важное.

Пример 2.13. Определение класса `RetailStore`

```
<?php
class RetailStore
{
    use Geocodable;

    // Здесь находится реализация класса
}
```

Это все, что требуется сделать. Теперь каждый экземпляр класса `RetailStore` сможет использовать свойства и методы, предоставляемые трейтом `Geocodable`, как показано в примере 2.14.

Пример 2.14. Трейты

```
<?php
$geocoderAdapter = new \Geocoder\HttpAdapter\CurlHttpAdapter();
$geocoderProvider = new \Geocoder\Provider\GoogleMapsProvider
($geocoderAdapter);
$geocoder = new \Geocoder\Geocoder($geocoderProvider);

$store = new RetailStore();
$store->setAddress('420 9th Avenue, New York, NY 10001 USA');
$store->setGeocoder($geocoder);

$latitude = $store->getLatitude();
$longitude = $store->getLongitude();
echo $latitude, ':', $longitude;
```



Интерпретатор PHP копирует и вставляет трейты в определения классов во время компиляции, но не обеспечивает защиту от несовместимости, вызванной этим действием. Если трейт предполагает наличие в классе свойств или методов (не определенных в самом трейте), убедитесь, что эти свойства и методы существуют в соответствующих классах.

Генераторы

Генераторы PHP, на удивление полезные функции, введенные в PHP 5.5.0, все еще недостаточно эффективно используются. Я полагаю, что многие PHP-разработчики не пользуются генераторами, потому что цель их применения не очевидна. Генераторы являются просто итераторами. Вот и все.



Генераторы PHP не являются панацеей для всех вариантов использования итераций, потому что не позволяют заранее узнать следующее значение итерации, в них отсутствует возможность перемотки назад или вперед. Вы можете перемещаться только в одном направлении, только вперед. Генераторы имеют одноразовый характер. Нельзя повторить выполнение генератора еще раз. Однако можно повторно создать или клонировать генератор при необходимости.

В отличие от стандартного итератора, генераторы не требуют реализации тяжеловесного интерфейса `Iterator`. Вместо этого, генераторы *вычисляют и выдают значения по требованию*. Это оказывает огромное влияние на производительность приложений. Подумайте об этом. Стандартный итератор обычно перебирает предварительно подготовленные и помещенные в память наборы данных. Это неэффективно, особенно для больших наборов формализуемых данных, которые могут быть непосредственно вычислены. Гораздо привлекательнее на этом фоне выглядят генераторы, вычисляющие последовательности значений на лету, не расходуя драгоценную память для их хранения.

Создание генератора

Генераторы легко создавать, потому что они являются функциями PHP, содержащими одно или несколько ключевых слов `yield`. В отличие от обычных функций PHP, генераторы никогда не возвращают значение. Они только *выдают (yield)* значения. Пример 2.15 демонстрирует простой генератор.

Пример 2.15. Простой генератор

```
<?php
function myGenerator() {
    yield 'value1';
    yield 'value2';
    yield 'value3';
}
```

Довольно просто, да? Когда вы вызываете функцию генератора, PHP возвращает объект, принадлежащий классу `Generator`. Этот объект можно использовать в операторе `foreach()`. В каждой итерации PHP будет опрашивать экземпляр `Generator` для вычисления и выдачи следующего значения итерации. Что характерно, генератор сохраняет свое внутреннее состояние на момент выдачи значения и восстанавливает его при запросе следующего значения. Генератор продолжает приостанавливать и возобновлять выполнение до тех пор, пока не достигнет конца своего определения как функции или пустого оператора `return;`. В примере 2.15 демонстрируется использование генератора в цикле:

```
<?php
foreach (myGenerator() as $yieldedValue) {
```

```

    echo $yieldedValue, PHP_EOL;
}

```

Результат выполнения:

```

value1
value2
value3

```

Использование генератора

Мне нравится демонстрировать, как генераторы экономят память, на примере реализации простой функции `range()`. Для начала давайте пойдем неправильным путем (пример 2.16).

Пример 2.16. Генерация диапазона чисел (плохая)

```

<?php
function makeRange($length) {
    $dataset = [];
    for ($i = 0; $i < $length; $i++) {
        $dataset[] = $i;
        .
    }

    return $dataset;
}

$customRange = makeRange(1000000);
foreach ($customRange as $i) {
    echo $i, PHP_EOL;
}

```

Пример 2.16 нерачительно использует память. Метод `makeRange()` в примере 2.16 помещает миллион целых чисел в предварительно подготовленный массив. Генератор PHP может сделать то же самое, храня в памяти в каждый момент времени только одно целое число, как показано в примере 2.17.

Пример 2.17. Генерация диапазона чисел (хорошая)

```

<?php
function makeRange($length) {
    for ($i = 0; $i < $length; $i++) {
        yield $i;
    }

    foreach (makeRange(1000000) as $i) {
        echo $i, PHP_EOL;
    }
}

```

Это надуманный пример. Однако, представьте себе всевозможные наборы данных, которые могут быть вычислены. Числовые последовательности (например, числа Фибоначчи) являются очевидными кандидатами. Таким же способом можно перебрать в цикле потоковый ресурс. Представьте, что нужно проанализировать файл размером 4 Гб, содержащий значения, разделенные запятыми (CSV), а на выделенном виртуальном сервере PHP-процессу доступен только 1 Гб памяти. То есть нет возможности разместить весь файл в памяти. Пример 2.18 показывает, как эту задачу решить с помощью генератора!

Пример 2.18. Генератор CSV

```
<?php
function getRows($file) {
    $handle = fopen($file, 'rb');
    if ($handle === false) {
        throw new Exception();
    }
    while (feof($handle) === false) {
        yield fgetcsv($handle);
    }
    fclose($handle);
}

foreach (getRows('data.csv') as $row) {
    print_r($row);
}
```

В этом примере память выделяется только для одной CSV-строки, а не для всего CSV-файла. Кроме того реализация итераций инкапсулирована в аккуратный пакет, что позволяет при необходимости быстро изменить вид получаемых данных (например, CSV, XML, JSON), не изменяя кода, перебирающего данные.

Генераторы являются компромиссом между универсальностью и простотой. Генераторы – это итераторы, поддерживающие перемещение *только вперед* на один шаг. Это значит, что нельзя использовать генератор для перемотки вперед или поиска в наборе данных, можно только запросить у генератора расчет и выдачу следующего значения. Генераторы особенно полезны для перебора больших или выражаемых числами наборов данных, требующего очень малого количества оперативной памяти. Они позволяют решать простые итеративные задачи с помощью меньшего объема кода, чем более сложные итераторы.

Генераторы не добавляют функциональности в язык PHP. Все, что делают генераторы, можно реализовать без них. Тем не менее, генераторы существенно упрощают определенные задачи, используя при этом меньше памяти. Если потребуется больше гибкости, например, будет необходимо быстрая перемотка вперед или поиск в наборе данных, лучше написать собственный класс, реализующий интерфейс `Iterator` (<http://php.net/manual/class.iterator.php>) или воспользоваться одним из встроенных итераторов библиотеки Standard PHP Library (SPL, <http://php.net/manual/spl iterators.php>).



Для знакомства с другими примерами использования генераторов, прочтайте статью «What Generators Can Do For You» (<http://bit.ly/ircmaxwell>) Энтони Феррара (Anthony Ferrara, @ircmaxell в Twitter).

Замыкания

Замыкания (closure) и *анонимные функции* (anonymous function) появились в PHP 5.3.0, и являются двумя моими любимыми и наиболее часто используемыми особенностями PHP. Несмотря на их пугающие названия (по крайней мере, это было моим первым впечатлением от знакомства с ними), они довольно просты для понимания. Это очень полезные инструменты и должны иметься в арсенале каждого разработчика.

Замыкание – это функция, которая инкапсулирует состояние, окружавшее ее на момент создания. Инкапсулированное состояние продолжает существовать внутри замыкания, даже когда окружение замыкания перестанет существовать. Эта концепция трудна для понимания, но как только вы освоите ее, все изменится.

Анонимные функции – это всего лишь функции без имени. Анонимные функции могут присваиваться в качестве значений переменным и передаваться куда угодно, как и любые другие объекты PHP. Но это еще и функции, которые можно вызвать и передать им аргументы. Анонимные функции особенно полезны в роли функций или методов обратного вызова.

Замыкания и анонимные функции в языке PHP используют тот же синтаксис что и функции, но это не должно вас обманывать. На самом деле это объекты, которые *только выглядят* как функции. Если рас-

смотреть внимательнее замыкание или анонимную функцию можно увидеть, что они являются экземплярами класса `Closure`. Замыкания считаются значениями базового класса, подобно строкам или целым числам.



Теоретически замыкания и анонимные функции – разные понятия. Но в PHP они считаются одним и тем же. Поэтому когда я говорю замыкание, то имею в виду также и анонимную функцию и наоборот.

Создание

Итак, мы уже знаем, что замыкания в PHP похожи на функции. Поэтому вас не удивит пример 2.19 создания замыканий.

Пример 2.19. Простое замыкание

```
<?php
$closure = function ($name) {
    return sprintf('Hello %s', $name);
};
echo $closure("Josh");
// Выведет --> "Hello Josh"
```

Это все. В примере 2.19 создается объект замыкания и присваивается переменной `$closure`. Замыкание выглядит, так же как стандартная функция: оно использует тот же синтаксис, принимает аргументы и возвращает значение, но у него нет имени.



Мы можем ссылаться на переменную `$closure`, потому что значением переменной является замыкание и объект замыкания при реализации использует магический метод `_invoke()`. PHP находит и вызывает метод `_invoke()` всякий раз, когда пара скобок () следует за именем переменной.

Я обычно использую объекты замыканий в качестве функций и методов обратного вызова. Многие функции в PHP принимают в качестве аргументов функции обратного вызова, например, такие как `array_map()` и `preg_replace_callback()`. Это прекрасная возможность задействовать анонимные функции! Помните, что замыкания могут

передаваться в другие функции в качестве аргументов, подобно любым другим значениям. В примере 2.20, я использую объект замыкания в качестве аргумента функции `array_map()`.

Пример 2.20. Замыкание для функции `array_map`

```
<?php
$numbersPlusOne = array_map(function ($number) {
    return $number + 1;
}, [1, 2, 3]);
print_r($numbersPlusOne);
// Выведет --> [2, 3, 4]
```

Впечатляет! До появления замыканий у PHP-разработчиков не было выбора, кроме создания отдельной именованной функции и обращения к ней по имени. При этом выполнение происходило несколько медленнее, и реализация обратного вызова была отделена от его использования. Разработчики, принадлежащие к старой школе PHP, использовали бы такой код:

```
<?php
// Реализация именованной функции обратного вызова
function incrementNumber ($number) {
    return $number + 1;
}

// Использование функции обратного вызова
$numbersPlusOne = array_map('incrementNumber', [1, 2, 3]);
print_r($numbersPlusOne);
```

Этот код работает, но он менее лаконичен и не так аккуратен, как код в примере 2.20. Если функция используется в роли функции обратного вызова только один раз, нет никакой необходимости создавать отдельную именованную функцию `incrementNumber()`. Использование замыканий в этом случае позволяет получить более краткий и понятный код.

Прикрепление состояния

К настоящему моменту я продемонстрировал использование безымянных (или *анонимных*) функций в качестве функций обратного вызова. Давайте посмотрим теперь, как прикрепить состояние и «закрепить» его в замыкании. Разработчиков на JavaScript могут смутить замыкания в PHP, потому что они автоматически не закрепляют состояние, как это делают замыкания в JavaScript. Вместо этого, в PHP

нужно вручную прикрепить состояние с помощью метода `bindTo()` объекта замыкания или ключевого слова `use`.

Чаще прикрепление состояния к замыканию выполняется с использованием ключевого слова `use`, так что давайте сначала рассмотрим его (пример 2.21). При подключении переменной к замыканию с помощью ключевого слова `use`, она сохраняет свое значение, *пока остается прикрепленной к замыканию*.

Пример 2.21. Прикрепление состояния к замыканию с помощью ключевого слова `use`

```
<?php
function enclosePerson($name) {
    return function ($doCommand) use ($name) {
        return sprintf('%s, %s', $name, $doCommand);
    };
}

// Прикрепление строки "Clay" к замыканию
$clay = enclosePerson('Clay');

// Вызов замыкания
echo $clay('get me sweet tea!');
// Выведет --> "Clay, get me sweet tea!"
```

В примере 2.21 именованная функция `enclosePerson()` принимает аргумент `$name` и возвращает объект замыкания, который *замыкает* аргумент `$name`. Возвращаемый объект замыкания сохраняет значение аргумента `$name` и вне функции `enclosePerson()`. Переменная `$name` продолжает существовать в замыкании!



С помощью ключевого слова `use` можно передать в замыкание несколько аргументов. Разделите аргументы запятыми, как это делается с аргументами любой функции или метода в языке PHP.

Помните, что замыкания в PHP являются объектами. Каждый экземпляр замыкания имеет свое внутреннее состояние, доступное через ключевое слово `this`, так же как внутреннее состояние любого другого объекта в PHP. Состояние по умолчанию объекта замыкания достаточно прозаично: оно включает необычный метод `__invoke()` и метод `bindTo()`. Это все.

Однако, метод `bindTo()` открывает доступ к некоторым интересным возможностям. Он позволяет *связать* внутреннее состояние объекта `closure` с *другим объектом*. Метод `bindTo()` принимает важный второй аргумент, который определяет PHP-класс объекта для связывания с замыканием. Это позволяет получить доступ к защищенным и приватным переменным связанного объекта.

Метод `bindTo()` часто используется PHP-фреймворками для отображения маршрутов в адреса URL в анонимных функциях обратного вызова. Фреймворки принимают анонимную функцию и привязывают ее к объекту приложения. Это позволяет ссылаться на объект основного приложения внутри анонимной функции с помощью ключевого слова `$this`, как показано в примере 2.22.

Пример 2.22. Прикрепление состояния замыкания с помощью метода `bindTo`

```
01. <?php
02. class App
03. {
04.     protected $routes = array();
05.     protected $responseStatus = '200 OK';
06.     protected $responseContentType = 'text/html';
07.     protected $responseBody = 'Hello world';
08.
09.     public function addRoute($routePath, $routeCallback)
10.     {
11.         $this->routes[$routePath] = $routeCallback->bindTo($this,
CLASS__);
12.     }
13.
14.     public function dispatch($currentPath)
15.     {
16.         foreach ($this->routes as $routePath => $callback) {
17.             if ($routePath === $currentPath) {
18.                 $callback();
19.             }
20.         }
21.
22.         header('HTTP/1.1 ' . $this->responseStatus);
23.         header('Content-type: ' . $this->responseContentType);
24.         header('Content-length: ' . mb_strlen($this->responseBody));
25.         echo $this->responseBody;
26.     }
27. }
```

Обратите особое внимание на метод `addRoute()`. Он принимает в аргументах путь маршрута (например, `/users/josh`) и функцию об-

ратного вызова для обработки маршрута. Метод `dispatch()` принимает текущий путь HTTP-запроса и вызывает функцию обратного вызова для соответствующего маршрута. Магия происходит в строке 11, когда мы связываем функцию обратного вызова обработки маршрута с текущим экземпляром класса `App`. Это позволяет создать функцию обратного вызова, которая может манипулировать состоянием экземпляра класса `App`:

```
<?php
$app = new App();
$app->addRoute('/users/josh', function () {
    $this->responseContentType = 'application/json; charset=utf8';
    $this->responseBody = '{"name": "Josh"}';
});
$app->dispatch('/users/josh');
```

Расширение Zend OPcache

Кэширование байт-кода не является чем-то новым для языка PHP. И раньше для этого использовались дополнительные автономные расширения, такие как Alternative PHP Cache (APC), eAccelerator, ionCube и XCache. Но до сих пор ни одно из них *не было включено* в основной дистрибутив PHP. Начиная с версии PHP 5.5.0, в PHP включено собственное встроенное расширение *кэширования байт-кода* Zend OPcache.

Во-первых, позвольте мне объяснить, что такое кэширование байт-кода, и в чем его полезность. PHP является *интерпретируемым* языком. Когда интерпретатор PHP выполняет сценарий, он анализирует код сценария, компилирует его в набор команд Zend Opcodes (инструкции на промежуточном языке, <http://bit.ly/zend-opcode>) и выполняет полученный байт-код. Все это проделывается для каждого PHP-файла при каждом запросе. При этом возникает много накладных расходов, особенно если один и тот же PHP-сценарий анализируется, компилируется и выполняется снова и снова. Если бы имелся способ *кэшировать* скомпилированный ранее байт-код, это уменьшило бы время отклика приложения и снизило нагрузку на системные ресурсы. И вам повезло, такой способ существует.

Кэш байт-кода хранит скомпилированный ранее байт-код. Это означает, что интерпретатору PHP не нужно читать, анализировать и скомпилировать PHP-код при каждом запросе. Вместо этого, интер-

претатор может прочитать скомпилированный ранее байт-код из памяти и сразу выполнить его. Это дает огромную экономию времени и может существенно улучшить производительность приложений.

Включение расширения Zend OPcache

По умолчанию расширение Zend OPcache отключено, поэтому его нужно явно включить при компиляции PHP.



Если вы выбираете веб-хостинг, убедитесь, что выбрали хорошего поставщика услуг, который предоставляет PHP версии 5.5.0 или выше с включенным кэшированием байт-кода Zend OPcache.

Если вы компилируете PHP самостоятельно, например на виртуальном или на выделенном сервере, включите сначала эту поддержку с помощью команды `./configure`:

```
--enable-opcache
```

После компиляции PHP, вы должны указать путь к модулю расширения Zend OPcache в файле `php.ini`, добавив в него строку:

```
zend_extension=/path/to/opcache.so
```

Путь к файлу расширения Zend OPcache выводится сразу же после успешной компиляции PHP. Если вы не запомнили его, что часто случается и со мной, узнать путь к каталогу расширения можно, выполнив команду:

```
php-config --extension-dir
```



Если вы используете популярное расширение Xdebug (<http://xdebug.org/>) несравненного Дерика Ретанса (Derick Rethans), загрузка расширения Zend OPcache в файле `php.ini` должна выполняться до расширения Xdebug.

После обновления файла `php.ini` перезапустите PHP-процесс и можно двигаться дальше. Убедитесь, что Zend OPcache работает правильно, можно, создав PHP-файл с таким содержимым:

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

Откройте этот PHP-файл в веб-браузере и прокручивайте вниз, пока не увидите раздел расширения Zend OPcache, как показано на рис. 2.2. Если вы не нашли этого раздела, значит расширение Zend OPcache не запущено.



Рис. 2.2. Настройки расширения Zend OPcache

Настройка расширения Zend OPcache

Если расширение Zend OPcache уже включено, следует настроить его параметры в конфигурационном файле `php.ini`. Ниже показаны настройки, которые нравятся мне:

```
opcache.validate_timestamps = 1 // "0" для действующего сервера
opcache.revalidate_freq = 0
opcache.memory_consumption = 64
opcache.interned_strings_buffer = 16
opcache.max_accelerated_files = 4000
opcache.fast_shutdown = 1
```



Более подробно настройки Zend OPcache будут описаны в главе 8. Полный список настроек Zend OPcache можно найти на сайте PHP.net (<http://bit.ly/php-config>).

Использование расширения Zend OPcache

Использовать расширение Zend OPcache легко, потому что оно после включения работает автоматически. Zend OPcache автоматически кэширует скомпилированный байт-код PHP в память и выполняет подготовленный байт-код, если он имеется.

Будьте осторожны, настраивая параметр `opcache.validate_timestamps`. Когда он имеет значение 0, Zend OPcache не проверяет факт изменения сценариев PHP и поэтому вы должны вручную очистить кэш байт-кода Zend OPcache для применения изменений, внесенных в PHP-файлы. Эта настройка хороша при промышленной эксплуатации, но неудобна в процессе разработки. Вы можете включить автоматическую проверку кэша с помощью следующих параметров настроек в `php.ini`:

```
opcache.validate_timestamps=1  
opcache.revalidate_freq=0
```

Встроенный HTTP-сервер

Знаете ли вы что, начиная с версии PHP 5.4.0, в PHP имеется встроенный веб-сервер? Это еще одна жемчужина, малоизвестная PHP-разработчикам, которые считают что им необходим Apache или nginx для предварительного просмотра PHP-приложений. Встроенный веб-сервер нельзя использовать для промышленной эксплуатации, но он является идеальным инструментом для локальной разработки.



Запомните, что встроенный сервер является полноценным веб-сервером. Он общается по протоколу HTTP и может обрабатывать любые статические ресурсы, а не только PHP-файлы. Его использование для локальной разработки является удобной альтернативой установке MAMP, WAMP или других супертяжелых веб-серверов.

Я пользуюсь встроенным веб-сервером PHP ежедневно, независимо от того пишу ли я на языке PHP или нет. Я применяю его для предварительного просмотра приложений, использующих фреймворки Laravel (<http://laravel.com/>) и Slim (<http://slimframework.com/>).

Я пользуюсь им при работе над сайтами, использующими фреймворк управления контентом Drupal. Я также использую его для предварительного просмотра статической разметки HTML и CSS, когда работаю над разметкой.

Запуск сервера

Начать работу с веб-сервером PHP просто. Откройте терминал, перейдите в корневой каталог проекта и выполните следующую команду:

```
php -S localhost:4000
```

Эта команда запустит веб-сервер PHP, доступный по адресу *localhost*. Он прослушивает порт 4000, а текущий рабочий каталог окажется корневым каталогом веб-сервера.

Теперь можно открыть веб-браузер и перейти по адресу <http://localhost:4000> для предварительного просмотра вашего приложения. При просмотре приложения в веб-браузере, в терминале выводится каждый HTTP-запрос, благодаря этому можно заметить момент, когда приложение вернет ответ 400 или 500.

Иногда полезно иметь доступ к веб-серверу PHP с других машин в локальной сети (например, для предварительного просмотра на iPad или на локальной машине Windows). Для этого, укажите веб-серверу PHP прослушивать все интерфейсы, указав IP-адрес 0.0.0.0 вместо *localhost*:

```
php -S 0.0.0.0:4000
```

Чтобы остановить веб-сервер PHP, закройте терминал или нажмите **Ctrl+C**.

Настройка сервера

Часто для приложения требуется определить собственный INI-файл с настройками PHP, особенно если приложение предъявляет нестандартные требования к использованию памяти, загрузке файлов, профилированию или кэшированию байт-кода. Вы можете указать встроенному серверу PHP какой INI-файл ему использовать с помощью ключа *-c*:

```
php -S localhost:8000 -c app/config/php.ini
```



Хорошой идеей является хранение пользовательского INI-файла в корневом каталоге приложения и, возможно, отдать его под контроль инструмента управления версиями, если он используется совместно несколькими разработчиками.

Сценарии маршрутизации

Встроенный сервер PHP обладает одним вопиющим упущением. В отличие от Apache или nginx, он не поддерживает файлы *.htaccess*. Это затрудняет использование *фронт-контроллеров* (front controllers), которые применяются во многих популярных фреймворках.



Фронт-контроллер – это единственный PHP-файл, которому передаются все HTTP-запросы (через файлы *.htaccess* или правила перезаписи). Фронт-контроллер отвечает за маршрутизацию запросов и вызов соответствующего PHP-кода. Такой подход, например, используется фреймворком Symfony и многими другими популярными фреймворками.

Встроенный сервер PHP несколько смягчает это упущение с помощью *сценариев маршрутизации* (router scripts). Сценарий маршрутизации выполняется перед каждым HTTP-запросом. Если он возвращает ложное значение, возвращается статический ресурс, на который ссылается URI текущего HTTP-запроса. В противном случае вывод сценария маршрутизации возвращается как HTTP-тело ответа. Другими словами, с помощью сценария маршрутизации можно добиться того же эффекта, который обеспечивает файл *.htaccess*.

Чтобы задействовать сценарий маршрутизации, при запуске встроенного сервера PHP передайте ему в путь к соответствующему PHP-файлу:

```
php -S localhost:8000 router.php
```

Обнаружение встроенного сервера

Иногда полезно знать, обслуживается ли сценарий встроенным веб-сервером PHP или традиционным веб-сервером, таким как Apache или nginx. Возможно, вам понадобится установить конкретные заголовки для nginx (например, `Status:`), которые не должны

устанавливаться для веб-сервера PHP. Веб-сервер PHP можно обнаружить с помощью функции `php_sapi_name()`. Эта функция возвращает строку `cli-server`, если текущий сценарий обслуживается встроенным сервером PHP:

```
<?php
if (php_sapi_name() === 'cli-server') {
    // Веб-сервер PHP
} else {
    // Другой веб-сервер
}
```

Недостатки

Встроенный веб-сервер PHP не должен использоваться для промышленной эксплуатации. Он предназначен только для локальной разработки. Если вы собираетесь использовать встроенный веб-сервер PHP в качестве рабочего, приготовьтесь к жалобам разочарованных пользователей и массе уведомлений о простое от Pingdom (<https://www.pingdom.com/>).

- Встроенный сервер работает неэффективно, потому что обрабатывает запросы по одному за раз, блокируя обработку следующего HTTP-запроса до завершения обслуживания предыдущего. Ваше веб-приложение будет простаивать, если файл PHP ожидает завершения медленного запроса к базе данных или ответа от удаленного программного интерфейса.
- Встроенный сервер поддерживает лишь ограниченное число MIME-типов (<http://bit.ly/built-in-ws>).
- Встроенный сервер ограничивает перезапись URL-адреса с помощью сценариев маршрутизации. Вам понадобится Apache или nginx для реализации более продвинутых возможностей перезаписи URL-адресов.

Что дальше

Современный язык PHP имеет много мощных возможностей, которые помогут сделать ваши приложения лучше. В этой главе я рассказал о своих любимых функциях. Вы можете узнать больше о новых особенностях языка PHP, посетив сайт PHP (<http://php.net/manual/features.php>).

Я уверен, что вам понравится использовать в своих приложениях особенности, рассмотренные выше. Однако важно пользоваться этими функциями правильно, в соответствии со стандартами PHP-сообщества. И именно об этом мы поговорим в следующей главе.



ЧАСТЬ II

Передовые технологии

ГЛАВА 3.

Стандарты

Существует ошеломляющее количество PHP-компонентов и PHP-фреймворков. Есть макрофреймворки, такие как Symfony (<http://symfony.com/>) и Laravel (<http://laravel.com/>). Есть микрофреймворки, такие как Silex (<http://silex.sensiolabs.org/>) и Slim (<http://laravel.com/>). И еще есть доставшиеся в наследство фреймворки, такие как CodeIgniter (<http://www.codeigniter.com/>), которые были разработаны задолго до появления современных PHP-компонентов. Современная экосистема языка PHP является настоящей плавильной печью кода, помогающей разрабатывать удивительные приложения.

К сожалению, старые PHP-фреймворки разрабатывались по-отдельности и не поддерживают разделение кода с другими PHP-фреймворками. Если ваш проект использует один из этих старых фреймворков, вы увязнете в нем и не сможете выйти за пределы экосистемы фреймворка. Это нормально, если вас полностью устраивают инструменты, предоставляемые фреймворком. Однако, что если вы используете фреймворк CodeIgniter, но при этом хотите подключить вспомогательную библиотеку из фреймворка Symfony? Вам очень повезет, если не придется писать одноразовый адаптер специально для вашего проекта.

«В данном случае мы имеем отсутствие взаимопонимания»

Хладнокровный Люк

Видите в чем проблема? Фреймворки, разрабатывавшиеся в изоляции, не были предназначены для взаимодействия с другими фреймворками. Это крайне неудобно как для разработчиков (их творческие возможности ограничены рамками выбранного фреймворка), так и для самих фреймворков (в них повторяется уже существующий в других проектах код). Однако у меня есть хорошая новость. PHP-сообщество отказалось от централизованной модели фреймворков в

пользу распределенной экосистемы эффективных совместимых специализированных компонентов.

PHP-FIG приходит на помощь

Несколько разработчиков PHP-фреймворков признали существование этой проблемы и обсудили ее в 2009 году на *php|tek* (популярная конференция по PHP, <http://tek.phparch.com/>). Они обговорили способы взаимодействий между фреймворками для повышения их эффективности. Что если, например, не писать новый накрепко привязанный к фреймворку класс журналирования, а воспользоваться общим классом журналирования *monolog* (<https://github.com/Seldaek/monolog>)? Или, вместо создания собственных классов HTTP-запросов и HTTP-откликов воспользоваться готовыми классами компонента *symfony/httpfoundation* (<http://bit.ly/symf-docs>) из фреймворка Symfony? Чтобы это сработало, PHP-фреймворки должны разговаривать на одном языке, что позволит им общаться и делиться кодом с другими фреймворками. Им нужны *стандарты*.

Разработчики PHP-фреймворков, по счастливой случайности, встретившиеся на конференции *php|tek*, в конечном итоге создали PHP Framework Interop Group (PHP-FIG, <http://www.php-fig.org/>). Группа PHP-FIG является группой представителей от PHP-фреймворков, которые, согласно информации на веб-сайте PHP-FIG, «договариваются о взаимосвязях между нашими проектами и определяют пути сотрудничества». Группа PHP-FIG готовит рекомендации, которых разработчики фреймворков PHP добровольно придерживаются для улучшения взаимосвязей между фреймворками и возможности использования общего кода.

PHP-FIG – это самопровозглашенная группа представителей фреймворков. Ее члены не избираются и они ничем не выделяются, кроме готовности помочь улучшению экосистемы PHP. Любой желающий может подать просьбу о вступлении. И кто угодно может отреагировать на рекомендации группы PHP-FIG, находящиеся в процессе утверждения. Окончательные рекомендации PHP-FIG, как правило, принимаются к исполнению и реализуются разработчиками многих популярных фреймворков. Я настоятельно рекомендую вам поучаствовать в работе группы PHP-FIG, отправив отзыв, и помочь формированию будущих особенностей ваших любимых PHP-фреймворков.



Важно понимать, что группа PHP-FIG выдает рекомендации. Это не правила и не требования. Эти рекомендации являются тщательно продуманными предложениями, которые облегчают жизнь PHP-разработчиков (и авторов PHP-фреймворков).

Совместимость фреймворков

Целью PHP-FIG является поддержка совместимости фреймворков. А совместимость фреймворков обеспечивает совместную работу с помощью интерфейсов, автозагрузки и стилей.

Интерфейсы

Фреймворки взаимодействуют посредством общих интерфейсов. Интерфейсы несут информацию о методах, предлагаемых сторонними производителями, и избавляют фреймворки от необходимости беспокоиться, как именно сторонние производители реализуют интерфейсы.



Интерфейсы, поддерживаемые в языке PHP, подробно описаны в главе 2.

Например, фреймворки готовы совместно использовать общий объект журналирования, предполагающий реализацию методов `emergency()`, `alert()`, `critical()`, `error()`, `warning()`, `notice()`, `info()` и `debug()`. Как именно эти методы будут реализованы, не имеет значения. Фреймворки заботят только наличие реализаций этих методов в сторонних зависимостях.

Интерфейсы позволяют PHP-разработчикам создавать, встраивать и использовать специализированные компоненты вместо монолитных фреймворков.

Автозагрузка

Фреймворки могут взаимодействовать с помощью механизма автозагрузки, который автоматически отыскивает PHP-класс и загружает его по требованию интерпретатора PHP во время выполнения.

До появления стандартов PHP компоненты и фреймворки использовали свои собственные уникальные автозагрузчики с помощью магического метода `_autoload()` или появившегося позднее метода `spl_autoload_register()`. Это требовало знания особенностей использования уникального автозагрузчика для каждого компонента и фреймворка. В настоящее время, большинство современных компонентов и фреймворков совместимы с общим стандартом автозагрузки. Это означает возможность объединять и сочетать несколько PHP-компонентов с помощью единственного автозагрузчика.

Стиль

Фреймворки могут взаимодействовать с помощью общего стиля оформления кода. Стиль определяет отступы, использование заглавных букв и размещение скобок (среди прочего). Если фреймворки используют согласованный стандартный стиль оформления кода, разработчикам не нужно изучать новый стиль при использовании каждого нового фреймворка. Код фреймворка будет уже знаком им. Стандартный стиль также облегчает процесс адаптации новых участников проекта, которые потратят больше времени на отладку за счет уменьшения времени на изучение незнакомого стиля оформления кода.

Использование стандартного стиля улучшает также наши собственные проекты. Если каждый разработчик использует свой уникальный стиль написания кода, значительно отличающийся от других, это превратится в проблему при совместной работе нескольких разработчиков над одним кодом. Стандартный стиль помогает членам команды быстро разобраться в коде, независимо от его автора.

Что такое PSR?

PSR – это аббревиатура *PHP Standards Recommendation* (стандартные рекомендации PHP). Если вы недавно читали какой-нибудь блог, связанный с PHP, то, наверное, заметили термины PSR-1, PSR-2, PSR-3 и так далее. Это – рекомендации группы PHP-FIG. Их имена начинаются с PSR- и заканчиваются номером. Каждая рекомендация группы PHP-FIG посвящена решению конкретной проблемы, которая касается большинства PHP-фреймворков. Вместо множества решений одной и той же проблемы разработчики фреймворков могут, придерживаясь рекомендаций группы PHP-FIG, создать общее ее решение.

Группа PHP-FIG опубликовала пять рекомендаций, которые приведены в этой книге:

- PSR-1: базовый стиль оформления кода (<http://www.php-fig.org/psr/psr-1/>);
- PSR-2: строгий стиль оформления кода (<http://www.php-fig.org/psr/psr-2/>);
- PSR-3: интерфейс инструмента журналирования (<http://www.php-fig.org/psr/psr-3/>);
- PSR-4: автозагрузка (<http://www.php-fig.org/psr/psr-4/>).



Если вы насчитали только четыре рекомендации, то вы правы. Группа PHP-FIG объявила свою первую рекомендацию PSR-0 (<http://www.php-fig.org/psr/psr-0/>) устаревшей. Первая рекомендация была заменена новой рекомендацией PSR-4 (<http://www.php-fig.org/psr/psr-4/>).

Обратите внимание, что рекомендации PHP-FIG соответствуют трем способам достижения совместимости, о которых я упоминал выше: интерфейсы, автозагрузка и стиль кода. Это не просто совпадение.

Меня действительно волнует судьба рекомендаций PHP-FIG. Они являются основой современной экосистемы языка PHP. Они определяют средства взаимодействия компонентов и фреймворков. Я признаю, что стандарты PHP не являются самой животрепещущей темой, но их изучение (на мой взгляд) создает предпосылки для понимания современного языка PHP.

PSR-1: Базовый стиль оформления кода

Если вы хотите научиться писать код на языке PHP, совместимый со стандартами сообщества, то начните со знакомства с рекомендацией PSR-1. Это самый простой стандарт. Он настолько прост, что вы, вероятно, уже используете его, не прилагая для этого никаких усилий. PSR-1 – это набор простых правил, которым легко следовать. Целью PSR-1 является определение базового стиля оформления программного кода для PHP-фреймворков, придерживающихся рекомендаций. Для совместимости с PSR-1 код должен удовлетворять следующим правилам:

Теги PHP

Код на языке PHP должен обрамляться тегами `<?php ?>` или `<?= ?>`. В самом коде не должны использоваться никакие другие теги PHP.

Кодировка

Все PHP-файлы должны содержать текст в кодировке UTF-8 без маркеров порядка следования байтов (Byte Order Mark, BOM). Звучит сложно, но текстовые редакторы и интегрированные среды разработки, как правило, делают это автоматически.

Задача

Файл может либо определять символы (классы, трейты, функции, константы и так далее), либо выполнять действие, имеющее побочные эффекты (например, производить вывод или манипулировать данными). Один и тот же PHP-файл не должен делать и то и другое. Добиться этого просто, от вас потребуется лишь немного предусмотрительности и последовательности.

Автозагрузка

Пространства имен и классы должны поддерживать стандарт автозагрузки PSR-4. Для этого нужно лишь выбрать соответствующие имена символов и убедиться, что файлы с их определениями находятся в нужных местах. Мы обсудим PSR-4 чуть ниже.

Имена классов

Имена PHP-классов должны записываться в верблюжьей нотации (`CamelCase`). Этот формат также называют нотацией первых заглавных букв (`TitleCase`). Примеры таких имен: `CoffeeGrinder`, `CoffeeBean` и `PourOver`.

Имена констант

Имена PHP-констант должны записываться только символами верхнего регистра. Для разделения слов, при необходимости, следует использовать подчеркивания. Примеры: `woot`, `LET_OUR_POWERS_COMBINE` и `GREAT_SCOTT`.

Имена методов

Имена PHP-методов должны записываться в верблюжий нотации. Причем, первый символ в имени метода должен быть буквой нижнего регистра, а каждое последующее слово в име-

ни метода должно начинаться с буквы верхнего регистра. Примеры: `phpIsAwesome`, `iLoveBacon` и `tenantIsMyFavoriteDoctor`.

PSR-2: Строгий стиль оформления кода

Следующий шаг после реализации рекомендации PSR-1 – реализация PSR-2. Стандарт PSR-2 определяет еще более строгие правила оформления PHP-кода.

Строгое оформление PSR-2 можно считать божьим даром для фреймворков PHP, в развитии которых участвует много разработчиков со всего мира, каждый из которых привносит в код свой собственный уникальный стиль и свои предпочтения. Общий строгий стиль код помогает разработчикам писать код, простой и понятный для других разработчиков.

Рекомендация PSR-2 определяет более строгие правила, чем рекомендация PSR-1. Некоторые из правил PSR-2 могут не совпадать с вашими предпочтениями. Тем не менее, стилю PSR-2 следуют многие популярные фреймворки. Вы не обязаны использовать стиль PSR-2, но если вы ему следите, другим разработчикам будет легче прочесть, использовать или вносить свой вклад в ваш код.



Используйте строгий стиль PSR-2. Даже при том, что я называю его *строгим*, он достаточно прост в применении. Рано или поздно он станет для вас привычным. Кроме того, имеются инструменты, осуществляющие автоматическое форматирование программного кода в соответствии со стилем PSR-2.

Реализация PSR-1

Применение стиля PSR-2 требует применения стиля кода PSR-1.

Отступы

Это спорный вопрос, который обычно делит PHP-разработчиков на два лагеря. Одни предпочитают оформлять отступы с помощью символов табуляции. Другие (более сдержанные) предпочитают использовать пробелы. Рекомендация PSR-2 указывает, что каждый отступ должен оформляться четырьмя пробелами.



Исходя из личного опыта, я считаю, что пробелы лучше подходят для оформления отступов, потому что пробел является постоянной мерой ширины и обычно одинаково отображается в разных текстовых редакторах. Табуляция же может иметь разную ширину в разных редакторах. Используйте четыре пробела для отступов, чтобы обеспечить лучшую визуальную преемственность вашего кода.

Файлы и строки

Строки в PHP-файлах должны завершаться символом перевода строки (linefeed, LF), в стиле Unix (LF), а сам файл должен заканчиваться одной пустой строкой и не должен включать завершающий PHP-тег `?>`. Строки *не должны* превышать в ширину 80 символов. В крайнем случае никакие строки *не должны* содержать более 120 символов. Строки не должны заканчиваться пробелами. На первый взгляд кажется, что следование этим правилам привносит массу дополнительной работы, но это нет так. Большинство текстовых редакторов автоматически переносят строки по достижении заданной длины, удаляют конечные пробелы и завершают строки в стиле Unix. Все это они делают автоматически, без всякого вашего участия.



Требование опускать завершающий тег `?>`, показалось мне сначала странным. Тем не менее, этот прием помогает избежать непредвиденных ошибок вывода. Если добавить закрывающий тег `?>`, а после него – пустую строку, интерпретатор будет выводить эту пустую строку, что может привести к ошибке (например, при установке HTTP заголовков).

Ключевые слова

Я знаю, что многие PHP-разработчики вводят TRUE, FALSE и NULL буквами верхнего регистра. Если вы поступаете так же, попробуйте отказаться от этой привычки и используйте только буквы нижнего регистра. Рекомендация PSR-2 требует вводить все ключевые слова языка PHP только буквами в нижнем регистре

Пространства имен

Каждое объявление пространства имен должно сопровождаться одной пустой строкой. Точно так же, при импорте или опре-

делении псевдонима пространства имен с помощью ключевого слова `use`, необходимо после блока объявлений с ключевыми словами `use` добавить одну пустую строку. Например:

```
<?php  
namespace My\Component;  
  
use Symfony\Components\HttpFoundation\Request;  
use Symfony\Components\HttpFoundation\Response;  
  
class App  
{  
    // Определение класса  
}
```

Классы

Как и отступы, размещение скобок, ограничивающих определение класса, является темой бурных дискуссий. Некоторые предпочитают размещать открывающую скобку в одной строке с именем класса. Другие – в отдельной строке после имени класса. Рекомендация PSR-2 требует, чтобы открывающая скобка в определении класса находилась в отдельной строке, сразу после имени класса, как это показано в следующем примере. Закрывающая скобка определения класса должна находиться в отдельной строке после тела определения класса. Вероятно, вы так и делали раньше, в этом требовании нет ничего страшного. Если ваш класс наследует другой класс или реализует интерфейс, ключевые слова `extends` и `implements` должны находиться в одной строке с именем класса:

```
<?php  
namespace My\App;  
  
class Administrator extends User  
{  
    // Определение класса  
}
```

Методы

Скобки, ограничивающие определение методов, должны размещаться так же, как скобки, ограничивающие определение классов. Открывающая скобка открывающая скобка должна находиться в отдельной строке, сразу после имени метода, а закрывающая скобка – в отдельной строке, сразу после определения метода. Обратите пристальное внимание на аргументы

метода. Круглые скобки не должны отделяться от аргументов пробелами. За каждым аргументом (кроме последнего) должна следовать запятая и один пробел:

```
<?php
namespace Animals;

class StrawNeckedIbis
{
    public function flapWings($numberOfTimes = 3, $speed = 'fast')
    {
        // Определение метода
    }
}
```

Область видимости

Для каждого свойства и метода класса необходимо явно объявлять *область видимости*. Под объявлением видимости понимается наличие одного из ключевых слов `public`, `protected` или `private`. Область видимости определяет доступность свойства или метода внутри класса и за его пределами. Разработчики PHP старой школы привыкли предварять свойства класса ключевым словом `var` и начинать имена приватных методов с символа подчеркивания `_`. *Не делайте этого*. Используйте одно из перечисленных выше объявлений видимости. Если объявляется абстрактное или финальное свойство или метод класса, ключевые слова `abstract` и `final` должны находиться *перед* объявлением области видимости. Если объявляется статическое свойство или метод, квалификатор `static` должен находиться *после* объявления видимости:

```
<?php
namespace Animals;

class StrawNeckedIbis
{
    // Статическое свойство с общедоступной видимостью
    public static $numberOfBirds = 0;

    // Метод с общедоступной видимостью
    public function construct()
    {
        static::$numberOfBirds++;
    }
}
```

Управляющие конструкции

Эту рекомендацию лично я воспринял как подножку. Все *ключевые слова управляющих конструкций* должны сопровождаться одним пробелом. К ключевым словам управляющих конструкций относятся: `if`, `elseif`, `else`, `switch`, `case`, `while`, `do while`, `for`, `foreach`, `try` и `catch`. Если управляющая конструкция требует пары круглых скобок, они не должны отделяться пробелами от параметров внутри. В отличие от определений классов и методов, открывающая фигурная скобка должна помещаться в одной строке с ключевым словом, а закрывающая – в отдельной строке, после тела управляющей конструкции. Следующий короткий пример демонстрирует эти требования:

```
<?php
$gorilla = new \Animals\Gorilla;
$ibis = new \Animals\StrawNeckedIbis;

if ($gorilla->isAwake() === true) {
    do {
        $gorilla->beatChest();
    } while ($ibis->isAsleep() === true);

    $ibis->flyAway();
}
```



Есть возможность автоматизировать приведение кода в соответствие со стилями PSR-1 и PSR-2. Многие редакторы автоматически форматируют код в соответствии с правилами PSR-1 и PSR-2. Существуют также инструменты, которые помогут проверить и отформатировать код по стандартам PHP. Один из таких инструментов: PHP Code Sniffer (<http://bit.ly/phpsniffer>), известный так же как phpcs. Этот инструмент (используется непосредственно из командной строки или из среды разработки) сообщает о несоответствиях между вашим кодом и заданным стандартом. Установить phpcs можно с помощью большинства диспетчеров пакетов (таких как PEAR, Homebrew, Aptitude или Yum).

Для автоматического исправления большинства несоставимостей можно также использовать PHP-CS-Fixer (<http://cs.sensiolabs.org/>) Фабьена Потенсьера (Fabien Potencier). Этот инструмент несовершенен, но он выполнит большую часть работы по достижению совместимости с PSR, практически без всяких усилий с вашей стороны.

PSR-3:

Интерфейс журналирования

Третья рекомендация PHP-FIG не является набором предписаний, как ее предшественницы. PSR-3 описывает интерфейс и определяет методы, которые могут быть реализованы PHP-компонентами журналирования.



Компонент журналирования – это объект, записывающий сообщения различной важности в указанное устройство вывода. Сообщения используются для диагностики, выявления и устранения проблем в работе приложения, его стабильности и производительности. Примерами могут служить запись отладочной информации в текстовый файл в процессе разработки, фиксирование статистики посещаемости сайта в базе данных или отсылка по электронной почте диагностики серьезных ошибок администратору сайта. Самым популярным PHP-компонентом журналирования является `monolog/monolog` (<https://packagist.org/packages/monolog/monolog>), созданный Хорди Боггiano (Jordi Boggiano).

Многие PHP-фреймворки реализуют журналирование с определенными возможностями. До появления рекомендаций PHP-FIG, каждый фреймворк решал задачу журналирования по-своему. Созданный группой PHP-FIG интерфейс журналирования PSR-3 соответствует преобладающему в современном языке PHP духу взаимодействия и специализации. Фреймворки, принимающие совместимую с PSR-3 концепцию журналирования, поддерживают две важные идеи: реализация задачи журналирования передается сторонним производителям и конечные пользователи могут выбрать наиболее подходящий им компонент. Это беспрогрышный вариант для всех.

Создание компонента журналирования PSR-3

Компонент журналирования, совместимый с рекомендацией PSR-3, должен включать класс, реализующий интерфейс `Psr\Log\LoggerInterface`. Интерфейс PSR-3 повторяет протокол системного журнала RFC 5424 (<http://tools.ietf.org/html/rfc5424>) и требует реализации девяти методов:

```
<?php
namespace Psr\Log;

interface LoggerInterface
{
    public function emergency($message, array $context = array());
    public function alert($message, array $context = array());
    public function critical($message, array $context = array());
    public function error($message, array $context = array());
    public function warning($message, array $context = array());
    public function notice($message, array $context = array());
    public function info($message, array $context = array());
    public function debug($message, array $context = array());
    public function log($level, $message, array $context = array());
}
```

Каждый метод интерфейса отображает соответствующий уровень протокола RFC 5424 и принимает два аргумента. Первый аргумент – `$message` – должен быть строкой или объектом с методом `__toString()`. Второй аргумент – `$context` – является необязательным и предоставляет массив значений для подстановки взамен тегов в первом аргументе.

Чтобы создать компонент журналирования PSR-3, нужно определить новый PHP-класс, реализующий интерфейс `Psr\Log\LoggerInterface` и обеспечить конкретную реализацию для каждого из методов интерфейса.



Аргумент `$context` позволяет формировать сложные сообщения, за счет подстановки значений взамен тегов в тексте сообщения. Тег имеет вид: `{placeholder-name}`. Он содержит открывающую фигурную скобку `{`, имя тега и закрывающую фигурную скобку `}`. Имя тега не должно содержать пробелов. Аргумент `$context` – это ассоциативный массив с именами тегов в качестве ключей (без фигурных скобок) и фактическими значениями, которые должны замещать соответствующие им теги в тексте сообщения.

Использование компонента журналирования PSR-3

Если вы решили создать собственный компонент журналирования PSR-3, бросьте эту затею и лучше потратьте свое время на что-то более полезное. Я настоятельно не рекомендую писать свой собствен-

ный компонент журналирования. Зачем? Ведь уже существуют отличные реализации!

Если вам нужен компонент журналирования, соответствующий стандарту PSR-3, возьмите компонент `monolog/monolog` (<https://packagist.org/packages/monolog/monolog>). Не тратьте время на поиск. Компонент Monolog полностью реализует интерфейс PSR-3 и позволяет легко добавлять пользовательские форматы сообщений и их обработчики. Обработчики компонента Monolog отсылают журналируемые сообщения в текстовые файлы, журналы событий, на электронную почту, HipChat, Slack, серверы сети, удаленные программные интерфейсы, базы данных и вообще куда угодно. В очень редких случаях компонент Monolog не содержит нужного обработчика, но и тогда легко создать и интегрировать в него свой собственный обработчик сообщений. Пример 3.1 демонстрирует, как легко настроить Monolog и выполнить вывод журналируемых сообщений в текстовый файл.

Пример 3.1. Использование компонента Monolog

```
<?php
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

// Подготовка сообщения
$log = new Logger('myApp');
$log->pushHandler(new StreamHandler('logs/development.log',
Logger::DEBUG));
$log->pushHandler(new StreamHandler('logs/production.log',
Logger::WARNING));

// Использование компонентов журналирования
$log->debug('This is a debug message');
$log->warning('This is a warning message');
```

PSR-4: Автозагрузка

Четвертая рекомендация PHP-FIG описывает стандартную модель *автозагрузки*. Автозагрузка предназначена для поиска классов, интерфейсов или трейтов PHP и загрузки их по требованию в интерпретатор PHP во время выполнения. Компоненты и фреймворки PHP, поддерживающие стандарт автозагрузки PSR-4, могут размещаться и загружаться с помощью только *одного* автозагрузчика. Это большое преимущество, учитывая наличие в экосистеме современного PHP огромного количества взаимодействующих компонентов.

Почему автозагрузка так важна

Как часто вы видели вот такой код в начале PHP-файлов?

```
<?php  
include 'path/to/file1.php';  
include 'path/to/file2.php';  
include 'path/to/file3.php';
```

Достаточно часто, не так ли? Вам, наверное, знакомы функции `require()`, `require_once()`, `include()` и `include_once()`. Они загружают внешние PHP-файлы в текущий сценарий и прекрасно подходят для случаев, когда программа состоит из небольшого числа файлов. А если необходимо подключить сто PHP-сценариев или даже тысячу? Функции `require()` и `include()` плохо масштабируются, именно поэтому автозагрузке отводится такая важная роль в PHP. Автозагрузчик отыскивает классы, интерфейсы или трейты и загружает их по мере необходимости во время выполнения, без явного включения файлов, как в предыдущем примере.

До того, как PHP-FIG представила рекомендацию PSR-4, авторы компонентов и фреймворков PHP использовали функции `__autoload()` и `spl_autoload_register()` для регистрации собственных автозагрузчиков. К сожалению, каждый компонент и фреймворк PHP использовал свой уникальный автозагрузчик, и каждый из них имел свой механизм поиска и загрузки классов, интерфейсов и трейтов. Разработчики, использующие такие компоненты и фреймворки, обязаны были вызывать автозагрузчики каждого из компонентов при развертывании приложения PHP. Я постоянно пользуюсь компонентом обработки шаблонов Twig (<http://twig.sensiolabs.org/>) от Sensio Labs. Он потрясающий. Но, без PSR-4, я должен был бы прочитать документацию компонента Twig и выяснить, как зарегистрировать его автозагрузчик в файле начальной загрузки моего приложения, что можно это сделать, например, так:

```
<?php  
require_once '/path/to/lib/Twig/Autoloader.php';  
Twig_Autoloader::register();
```

Представьте, что вы должны разобраться и зарегистрировать уникальные автозагрузчики для каждого PHP-компонента, используемого в вашем приложении. Группа PHP-FIG осознала эту проблему и предложила рекомендацию PSR-4, определяющую организацию автозагрузки для облегчения взаимодействия компонентов. Благо-

дая PSR-4, мы можем автоматически загрузить все необходимые PHP-компоненты с помощью единственного автозагрузчика. Это потрясающе! Большинство современных компонентов и фреймворков совместимы с PSR-4. Если вы создаете и распространяете свои собственные компоненты, обеспечьте их совместимость с PSR-4! В числе совместимых компонентов можно назвать: Symfony, Doctrine, Monolog, Twig, Guzzle, SwiftMailer, PHPUnit, Carbon и многие другие.

Модель автозагрузки PSR-4

Рекомендация PSR-4 определяет стратегию поиска и загрузки классов, интерфейсов и трейтов во время выполнения. Она не требует внесения изменений в ваш код. Вместо этого PSR-4 предполагает определенный порядок распределения кода по каталогам в файловой системе и пространствам имен. Автозагрузка PSR-4 опирается на пространства имен и каталоги файловой системы для поиска и загрузки классов, интерфейсов и трейтов.

Основой PSR-4 является отображение префикса верхнего уровня пространства имен в конкретный каталог файловой системы. Например, я могу определить, что классы, интерфейсы и трейты из пространства имен \Oreilly\ModernPHP находятся в каталоге *src*/ файловой системы. Интерпретатор PHP теперь знает, что любые классы, интерфейсы и трейты с префиксом пространства имен \Oreilly\ModernPHP находятся в подкаталогах и файлах в каталоге *src*. Например, пространство имен \Oreilly\ModernPHP\Chapter1 соответствует каталогу *src/Chapter1*, а класс \Oreilly\ModernPHP\Chapter1\Example определен в файле *src/Chapter1/Example.php*.



PSR-4 позволяет отобразить префикс пространства имен в каталог файловой системы. Префиксом может быть одно из пространств имен верхнего уровня. Префикс пространства имен также может быть пространством имен верхнего уровня и любым количеством подпространств. Это достаточно гибкая модель.

Вспомните, мы уже говорили о пространствах имен производителей главе 2. Стратегия автозагрузки PSR-4 имеет самое непосредственное отношение к авторам компонентов и фреймворков, код которых используют другие разработчики. Код PHP-компонента существует в уникальном пространстве имен производителя, и автор компонента определяет, какому каталогу файловой системы соответ-

ствует пространство имен производителя, именно так, как я и описал ранее. Мы рассмотрим эту идею подробнее в главе 4.

Как написать автозагрузчик PSR-4 (и почему этого делать не нужно)

Мы знаем, что совместимый с рекомендацией PSR-4 код имеет префикс пространства имен, который отображается в базовый каталог файловой системы. Мы также знаем, что вложенные подпространства имен отображаются в подкаталоги базового каталога файловой системы. Пример 3.2 демонстрирует реализацию автозагрузчика, заимствованную с сайта PHP-FIG (<http://bit.ly/php-fig>), который находит и загружает классы, интерфейсы и трейты, основываясь на стратегии автозагрузчика PSR-4.

Пример 3.2. Автозагрузчик PSR-4

```
<?php
/**
 * Пример реализации автозагрузчика для проекта.
 *
 * После регистрации этого автозагрузчика с помощью SPL, следующая
 * строка должна загрузить класс \Foo\Bar\Baz\Qux
 * из файла /path/to/project/src/Baz/Qux.php:
 *
 *     new \Foo\Bar\Baz\Qux;
 *
 * @param string $class      Полное имя класса.
 * @return void
 */
spl_autoload_register(function ($class) {
    // Базовый префикс пространства имен для проекта
    $prefix = 'Foo\\Bar\\';

    // базовый каталог для префикса пространства имен
    $base_dir = DIR . '/src/';

    // класс использует префикс пространства имен?
    $len = strlen($prefix);
    if (strcmp($prefix, $class, $len) !== 0) {
        // нет, переходим к следующему зарегистрированному
        // автозагрузчику
        return;
    }
    // получить относительное имя класса
```

```
$relative_class = substr($class, $len);

// заменить префикс пространства имен базовым каталогом,
// заменить разделители пространств имен разделителями
// каталогов в относительном имени класса,
// добавить расширение .php
$file = $base_dir . str_replace('\\', '/', $relative_class) . '.php';

// если файл существует, загрузить его
if (file_exists($file)) {
    require $file;
}
});
```

Скопируйте его и вставьте в свое приложение, измените, значения переменных \$prefix и \$base_dir, и вы получите свой рабочий автозагрузчик PSR-4. Однако если вы вознамеритесь писать собственный автозагрузчик PSR-4, остановитесь и спросите себя, а действительно ли необходимо то, что вы делаете. Почему? Потому что можно использовать автозагрузчики PSR-4, которые автоматически генерируются менеджером зависимостей Composer. Это удобно и это именно то, о чём мы поговорим в следующей главе 4.

ГЛАВА 4.

Компоненты

В современном языке PHP предпочтение отдается не монолитным фреймворкам, а решениям, целиком состоящим из специализированных и взаимодействующих между собой компонентов. Когда я собираюсь разрабатывать новое PHP-приложение, я, как правило, не планирую использовать только фреймворк Laravel или только фреймворк Symfony. Вместо этого, я обдумываю, как объединить в единое целое существующие компоненты, подходящие для решения моей задачи.

Почему надо использовать компоненты?

Современные PHP-компоненты сейчас используют многие PHP-программисты. Несколько лет назад я и понятия не имел о PHP-компонентах. Прежде я не знал лучших способов разработки PHP-приложений, чем применение больших фреймворков, таких как Symfony или CodeIgniter. Я ограничивался закрытой экосистемой одного фреймворка и предлагаемой ей инструментами. Если фреймворк не имел нужного мне функционала, я считал, что мне не повезло, и создавал дополнительный функционал сам. Было довольно трудно интегрировать свои библиотеки или библиотеки сторонних разработчиков в огромные фреймворки, так как они не предусматривали общих интерфейсов. Я с облегчением могу сообщить, что времена изменились, и мы больше не являемся пленниками монолитных фреймворков и не ограничены их рамками.

Сегодня у нас есть выбор из обширной и постоянно растущей коллекции специализированных компонентов. Зачем тратить время на написание библиотеки HTTP-запросов и HTTP-откликов, если уже существует компонент `guzzle/http` (<https://packagist.org/packages/guzzle/http>)

`guzzle/http`)? Зачем создавать новый маршрутизатор, если уже имеются замечательные компоненты `aura/router` (<https://packagist.org/packages/aura/router>) и `ogno/route` (<https://packagist.org/packages/ogno/route>)? Зачем тратить время на написание адаптера к службе хранения S3 Amazon, если можно использовать компоненты `aws/aws-sdk-php` (<https://packagist.org/packages/aws/aws-sdk-php>) и `league/flysystem` (<https://packagist.org/packages/league/flysystem>)? Вы уже поняли мою мысль. Другие разработчики потратили много часов на создание, совершенствование и тестирование специализированных компонентов, каждый из которых делает что-то определенное очень хорошо. Было бы глупо не воспользоваться этими компонентами для быстрого создания хорошего приложения, а вместо этого тратить время на изобретение велосипеда.

Что представляют собой компоненты?

Компонент – это комплект кода, позволяющий решить конкретные задачи. Например, если приложение должно отправлять и получать HTTP-запросы, можно воспользоваться компонентом, который реализует эти функции. Если потребуется анализировать разделенные запятыми данные, можно воспользоваться компонентом, который сделает это. Если приложению понадобится способ журналирования сообщений, можно воспользоваться компонентом и для этого. Чтобы не воспроизводить заново уже реализованную функциональность, мы пользуемся PHP-компонентами и высвобождаем больше времени на решение задач именно нашего проекта.



Технически PHP-компонент является совокупностью связанных между собой классов, интерфейсов и трейтов, решающих единую задачу. Классы, интерфейсы и трейты компонента, как правило, существуют в одном пространстве имён.

На любом продуктовом рынке есть хорошие и плохие продукты. То же относится и к PHP-компонентам. Точно так же, как вы осматриваете со всех сторон яблоки в магазине, существуют несколько приемов определения качественности PHP-компонентов. Далее перечислены некоторые признаки хороших PHP-компонентов:

Строгая специализация

Хороший PHP-компонент предназначен для решения только одной конкретной задачи. Он не должен походить на мастера на все руки, ничего не делающего на отлично. Он должен мастерски решать только одну проблему. Он одержим решением одной задачи и скрывает свою гениальность под простым пользовательским интерфейсом.

Небольшой размер

PHP-компонент не должен быть больше, чем это требуется. Он содержит наименьшее возможное количество кода, необходимого для решения одной задачи. Объем кода может варьироваться. PHP-компонент может состоять из одного класса. Он также может иметь и несколько классов, организованных в пространства имен. Не существует единственно правильного количества классов для PHP-компонента. Однако компонент все свои классы использует для решения одной задачи.

Взаимодействие

PHP-компонент хорошо играет в команде. В конце концов, изюминкой PHP-компонентов является возможность их сотрудничества с другими компонентами для построения больших решений. PHP-компонент не загрязняет глобальное пространство имен своим кодом – он существует в собственном пространстве имен, избегая конфликтов с другими компонентами.

Хорошее тестирование

PHP-компонент хорошо протестирован. Достичь этого позволяют его небольшие размеры. Если компонент невелик и строго специализирован, его легко тестировать. Действия его сосредоточены на одной задаче и его зависимости могут быть легко идентифицированы и сымитированы. Лучшие PHP-компоненты имеют свои собственные тесты и достаточно полно охвачены ими.

Хорошая документированность

PHP-компонент хорошо документирован. Разработчики не должны испытывать трудностей при его установке и использовании. Хорошая документация позволяет достичь этого. PHP-компонент должен быть снабжен файлом *README*, который описывает, что делает компонент, как его установить и использовать. Компонент может также иметь собственный

веб-сайт с более подробной информацией. Хорошо документированным должен быть и исходный код PHP-компоненты. Классы, методы и свойства должны иметь встроенные аннотации, описывающие код, параметры, возвращаемые значения и потенциальные исключения.

Компоненты и фреймворки

Недостаток фреймворков (особенно старых) в том, что они ограничивают свободу выбора. Выбирая фреймворк, мы получаем возможность использовать инструменты фреймворка. Фреймворки, как правило, предлагают шведский стол инструментов. Но иногда требуется что-то специфическое, чего фреймворк не предусматривает, и при этом возникают большие сложности, связанные с поиском и интегрированием внешних PHP-библиотек. Интеграция стороннего кода в фреймворк затруднена, потому что внешний код и PHP-фреймворк обычно не имеют общих интерфейсов.

Выбирая фреймворк, мы накрепко связываем себя с ним. Мы полагаемся на команду разработчиков фреймворка. Мы предполагаем, что разработчики фреймворка будет продолжать тратить свое время на его доработку и обеспечивать его соответствие современным стандартам. Но очень часто этого не происходит. Фреймворки огромны, и на их поддержание требуется масса времени и сил. У разработчиков, сопровождающих проект, своя собственная жизнь, работа и интересы. А жизнь, место работы и интересы иногда меняются.



Справедливости ради следует отметить, что крупные PHP-компоненты также могут остаться без поддержки, особенно, если компонент сопровождается только один основной разработчик.

Кроме того, кто может утверждать, что именно этот конкретный фреймворк будет оставаться лучшим инструментом для работы? Крупные проекты, существующие на протяжении многих лет, должны хорошо работать и настраиваться, как сейчас, так и в будущем. Неправильный выбор PHP-фреймворка может стать преградой этому. Старые PHP-фреймворки, выходят из моды, в настоящее время они выглядят медленными и устаревшими, теряют былую популярность. В старых фреймворках часто использовался процедурный подход, а

не современный объектно-ориентированный. Новые члены вашей команды могут быть незнакомы с особенностями старого фреймворка. Есть много других причин использовать или не использовать определенный PHP-фреймворк.

Не все фреймворки плохи

Пока я рассмотрел только минусы фреймворков. Но не все фреймворки так уж плохи. Фреймворк Symfony (<http://symfony.com/>) – прекрасный пример современного PHP-фреймворка. Фабьен Потенсьер (Fabien Potencier) и Sensio Labs (<http://sensiolabs.com/>) разработали фреймворк Symfony в виде системы небольших отдельных компонентов Symfony (<http://symfony.com/components>). Эти компоненты могут использоваться в приложениях как вместе, в качестве основы, так и по отдельности.

Другие, более старые фреймворки также обновляются, приближаясь к современным PHP-компонентам. Примером может служить система управления контентом Drupal (<https://www.drupal.org/>). Версия Drupal 7 написана с в процедурном стиле, то есть весь код существует в глобальном пространстве имен. Она игнорирует современные методы организации наследования. Однако версия Drupal 8 является огромным и заслуживающим всяких похвал шагом к современному уровню. Она использует преимущества схемы отдельных PHP-компонентов для построения современной платформы управления контентом.



Самыми популярными современными PHP-фреймворками являются:

- Aura (<http://auraphp.com/framework>)
- Laravel (<http://laravel.com/>)
- Symfony (<http://symfony.com/>)
- Yii (<http://www.yiiframework.com/>)
- Zend (<http://framework.zend.com/>)

Фреймворк Laravel (<http://laravel.com/>), написанный Тейлором Аутвеллом (Taylor Otwell), – еще один популярный PHP-фреймворк. Как и Symfony, фреймворк Laravel опирается на собственную библиотеку компонентов Illuminate (<https://github.com/illuminate>). Однако (на момент публикации) компоненты Laravel нелегко выделить для использования по отдельности. Фреймворк Laravel не придерживает-

ся стандарта PSR-2 и схемы Semantic Versioning (<http://semver.org/>). Но пусть это вас не пугает. Laravel остается замечательным фреймворком, помогающим создавать очень мощные приложения.

Использование инструмента, соответствующего задаче

Обязательно ли использовать компоненты или фреймворки? *Используйте инструмент, соответствующий задаче*. Большинство современных PHP-фреймворков построено на основе соглашений об использовании небольших компонентов. При работе над небольшим проектом, для реализации которого достаточно небольшой коллекции PHP-компонентов, используйте компоненты. Компоненты значительно облегчают приобретение и использование существующих инструментов, давая возможность сосредоточиться не на рутине, а на решении высокуюровневой задачи. Компоненты также помогают сделать код легким и быстрым. Используя только необходимый код, очень просто будет заменить один компонент другим, лучше подходящим для проекта.

При работе над большим проектом в команде разработчиков больше подойдут соглашения, дисциплинированность и структурность, предоставляемые фреймворками, поэтому используйте фреймворки. Но фреймворки очень часто принимают решения за нас и требуют строгого соблюдения своих наборов правил. Фреймворки являются менее гибкими, но мы получаем гораздо больше функций «из коробки», чем при использовании наборов PHP-компонентов. Если такой компромисс приемлем, воспользуйтесь всеми средствами фреймворка для управления проектом и ускорения его разработки.

Поиск компонентов

Найти современные PHP-компоненты можно на сайте Packagist (<https://packagist.org/>, рис. 4.1), являющимся де факто каталогом PHP-компонентов. Этот сайт объединяет PHP-компоненты в одном месте и обеспечивает их поиск по ключевым словам. На Packagist перечислены лучшие PHP-компоненты. Я снимаю шляпу перед Хорди Боггиано (Jordi Boggiano, <http://seld.be/>) и Игорем Видлером (Igor Wiedler, <https://igor.io/archive.html>) за создание такого бесценного для сообщества ресурса.

Packagist The PHP Package Repository

Search packages...

Packagist is the main Composer repository. It aggregates public PHP packages installable with Composer.

Getting Started

Define Your Dependencies

Put a file named `composer.json` at the root of your project, containing your project dependencies:

```
{
  "require": {
    "vendor/package": "1.1.2",
    "another/vendor/package": "1.0.0"
  }
}
```

For more information about packages versions usage, see the composer documentation.

Install Composer In Your Project

Run this in your commandline:

```
curl -s https://getcomposer.org/installer | php
```

Or download `composer.phar` into your project root.

See the Composer documentation for complete installation instructions on various platforms.

Install Dependencies

Execute this in your project root:

```
php composer.phar install
```

Autoload Dependencies

Publishing Packages

Define Your Package

Put a file named `composer.json` at the root of your package, containing this information:

```
{
  "name": "your-username/package-name",
  "description": "A short description of what your package does",
  "require": {
    "1.1.2 || 1.0.0"
  },
  "another-vendor/package": "1.0.0"
}
```

This is the strictly minimal information you have to give.

For more details about package naming and the fields you can use to document your package better, see the [aboutPage](#).

Commit The File

You surely don't need help with that.

Publish It

Login or register on this site, then hit the submit button in the menu.

Once you entered your public repository URL in there, your package will be automatically crawled periodically. You just have to make sure you keep the `composer.json` file up-to date.

Рис.4.1. Сайт Packagist



Меня часто спрашивают, какие компоненты я считаю лучшими. Этот выбор достаточно субъективен. Тем не менее, я в основном согласен со списком PHP-компонентов, перечисленных в Awesome PHP (<https://github.com/ziadoz/awesome-php>). Этот список лучших компонентов курирует Джейми Йорк (Jamie York, <https://github.com/ziadoz>).

Магазин

Не тратьте время на решение проблем, которые уже решены. Вам нужно отправить или получить HTTP-сообщения? Перейти на сайт Packagist и выполните поиск по слову «`http`», первым в списке результатов окажется компонент `Guzzle`. Используй его. Вам нужно реализовать парсинг CSV-файла? Перейти на Packagist и выполните поиск по слову «`csv`», выберите компонент `CSV` и используйте его. Читайте сайт Packagist магазином PHP-компонентов, где можно купить лучшие ингредиенты для ваших блюд. Наверняка на Packagist имеется PHP-компонент, который решит вашу проблему.

Выбор

Что делать, если на сайте Packagist нашлось несколько нужных PHP-компонентов? Как выбрать лучший? Сайт Packagist ведет статистику для каждого PHP-компонента. Сайт Packagist предоставляет информацию о том, сколько раз каждый PHP-компонент был загружен и сколько голосов «за» он получил (в виде числа звездочек, рис. 4.2). Часто загружаемый и имеющий больше звездочек компонент должен быть хорошим вариантом выбора (что не всегда верно). Не стоит сбрасывать со счетов и новые компоненты с небольшим количеством загрузок. Сайт ежедневно пополняется множеством новых компонентов.

Достаточно трудно выбрать лучший PHP-компонент, если поиск на сайте Packagist по ключевым словам возвращает большое количество результатов. Не всегда можно положиться на статистику загрузок, потому что общее мнение не всегда является верным. Это проблема, которую сайту Packagist еще предстоит решить, тем более что он становится все популярнее. Я советую при выборе PHP-компонента полагаться на рекомендации.

The screenshot shows a browser window with the URL <https://packagist.org/search?q=CSV>. The page title is "Packagist - The PHP Package Repository". Below the title, there's a search bar with the text "CSV" and a note: "Packagist is the main Composer repository. It aggregates public PHP packages installable with Composer." The main content area displays a list of packages:

Package	Description	Type	Downloads	Rating
league/csv	Csvdata manipulation made easy inPHP	PHP	↓ 1 475992	★ 1 804
goodby/csv	CSVimport/export library	PHP	↓ 478968	★ 678
keboola/csv	Keboola CSV reader and writer	PHP	↓ 250258	★ 108
mnnshankar/csv	Easily generate CSV files	PHP	↓ 30679	★ 44
xp-framework/csv	Contains the XP Framework's CSV API	PHP	↓ 3510	★ 0
Wilgucki/csv	Laravel package for writing and reading CSV files	PHP	↓ 3 054	★ 10

Рис.4.2. Результаты поиска на сайте Packagist

Оставьте отзыв

Если вы нашли PHP-компонент, который вам понравился, отметьте его звездочкой на Packagist и поделитесь своим мнением с другими PHP-разработчиками в Twitter, Facebook, IRC, Slack или где-то еще. Это поможет лучшему PHP-компоненту всплыть вверх, чтобы его заметили и другие разработчики.

Использование PHP-компонентов

Сайт Packagist – это каталог для поиска PHP-компонентов. А устанавливать их следует с помощью Composer (<https://getcomposer.org/>) – менеджера зависимостей PHP-компонентов, запускаемого в командной строке. Менеджеру зависимостей Composer нужно указать требуемые PHP-компоненты, а он загрузит и настроит автозагрузку этих компонентов в проект. Всё это делается достаточно просто. Так как Composer является менеджером зависимостей, он разрешает и загружает зависимости выбранных компонентов (затем их зависимости и так далее).

Менеджер зависимостей Composer способен взаимодействовать с сайтом Packagist. Если указать менеджеру зависимостей Composer, что требуется задействовать компонент `guzzlehttp/guzzle`, Composer выберет компонент `guzzlehttp/guzzle` на сайте Packagist, найдет адрес URL репозитория компонента, определит подходящую версию и все ее зависимости. Затем Composer загрузит компонент `guzzlehttp/guzzle` со всеми его зависимостями в ваш проект.

Достоинство менеджера Composer заключается в том, что он берет на себя решение сложных проблем управления зависимостями и автозагрузкой. Автозагрузка – это процесс автоматической загрузки PHP-классов по требованию без явной их загрузки с помощью функций `require()`, `require_once()`, `include()` или `include_once()`. Ранние версии PHP позволяли создавать пользовательские автозагрузчики, основанные на функции `_autoload()`. Эта функция автоматически вызывается интерпретатором PHP при создании класса, который не был загружен ранее. Позже в PHP была введена более гибкая функция `spl_autoload_register()` из библиотеки SPL. Она обеспечивает автоматическую загрузку PHP-классов по желанию разработчика. К сожалению, отсутствие единого стандарта требовало создания

специальных автозагрузчиков для каждого проекта. Это осложняло использование кода, созданного и распространяемого другими разработчиками, потому что каждый разработчик предоставлял свой собственный автозагрузчик.

Группа PHP Framework Interop Group осознала эту проблему и создала стандарт PSR-0 (позднее замененный стандартом PSR-4). Стандарты PSR-0 и PSR-4 определяют структуру распределения кода по пространствам имен и каталогам файловой системы, совместимую с единой стандартной реализацией автозагрузки. Как упоминалось в главе 3, нет необходимости писать свой автозагрузчик, соответствующий стандарту PSR-4. Менеджер зависимостей Composer автоматически генерирует PSR-совместимый автозагрузчик для *всех* PHP-компонентов в проекте. Composer обеспечивает эффективное абстрагирование управления зависимостями и автозагрузки.



Я считаю, что менеджер зависимостей Composer является наиболее важным приобретением PHP-сообщества. Он полностью изменил мой подход к созданию PHP-приложений. Я использую Composer в каждом PHP-проекте, потому что он существенно упрощает интеграцию и использование PHP-компонентов сторонних производителей в моих приложениях. Если вы еще не пользуетесь менеджером зависимостей Composer, начните его изучение прямо сейчас.

Установка Composer

Composer прост в установке. Откройте терминал и выполните команду:

```
curl -sS https://getcomposer.org/installer | php
```

Эта команда загрузит установочный сценарий менеджера зависимостей Composer, основанный на curl, выполнит его с помощью php и создаст файл *composer.phar* в текущем рабочем каталоге. Файл *composer.phar* является двоичным выполняемым файлом Composer.



Никогда не выполняйте код, который не глядя загрузили из Интернета. Просмотрите его, чтобы точно знать, что именно он будет делать. Также убедитесь, что загрузили код с помощью протокола HTTPS.

Я предлагаю перемещать и переименовывать загруженный двоичный файл Composer в каталог `/usr/local/bin/composer` командой:

```
sudo mv composer.phar /usr/local/bin/composer
```

Не забудьте выполнить следующую команду, чтобы сделать файл `composer` выполняемым:

```
sudo chmod +x /usr/local/bin/composer
```

И, наконец, добавьте каталог `/usr/local/bin` в переменную окружения `PATH`, вставив следующую строку в файл `~/.bash_profile`:

```
PATH=/usr/local/bin:$PATH
```

Теперь можно выполнить команду `composer` в терминале, чтобы увидеть список возможных параметров (рис. 4.3).

```
Composer version 1.0.0 2016-07-19 01:28:52
Usage: composer [options] [arguments]
Options:
  -h, --help           Display this help message
  -q, --quiet          Do not output any message
  -V, --version         Display this application version
  --ansi               Force ANSI output
  --no-ansi             Disable ANSI output
  -n, --no-interaction Do not ask any interactive question
  --profile            Display timing and memory usage information
  --no-plugins          Whether to disable plugins
  -d, --working-dir    If specified, use the given directory as working directory.
  -vvv, --verbose       Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug
Available commands:
  archive      Short information about Composer
  archive     Create an archive of this composer package
  browse      Opens the package's repository URL or homepage in your browser.
  clear-cache  Clears composer's internal package cache.
  clear-profiles  Clears composer's internal profile cache.
  create-project Create new project from a package into given directory.
  depends      Shows which packages cause the given package to be installed.
  find         Finds packages matching criteria to identify common errors.
  dump-autoloader  Dumps the autoloader
  dumpautoload  Dumps the autoloader
  exec        Execute a vendored binary/script
  global      Set global composer settings in the global composer dir ($COMPOSER_HOME).
  help        Displays help for a command
  home       Opens the package's repository URL or homepage in your browser.
  info        Show information about packages
  install     Install the packages declared in current composer.json file in current directory.
  update     Update the declared dependencies from the composer.json file if present, or fall back to the composer.lock
```

Рис.4.3. Параметры командной строки Composer

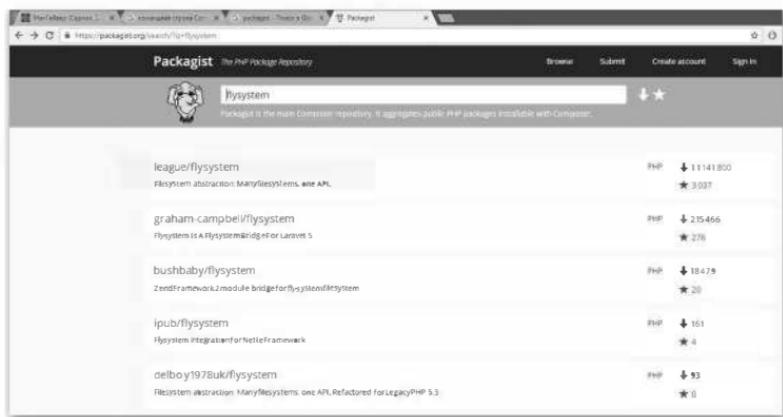
Как использовать Composer

Теперь, после установки Composer, загрузим несколько PHP-компонентов. Composer обычно используется для загрузки PHP-компонентов в проект.

Имена компонентов

Во-первых, нужно составить список компонентов, необходимых в проекте. В частности, обратите внимание на имена производителей и пакетов каждого компонента. Каждый PHP-компонент имеет имя

производителя и имя пакета. Например, именем производителя популярного компонента `league/flysystem` является `league`, а именем пакета – `flysystem`. Имена производителей и пакетов разделяются символом `/`. Имя производителя вместе с именем пакета формируют полное имя компонента `league/flysystem`. Название производителя является глобально уникальным и обеспечивает глобальную идентификацию принадлежности пакета. Имя пакета уникально идентифицирует пакет с данным именем производителя. Composer и Packagist используют соглашение об именовании `vendor/package`, чтобы предотвратить конфликты имен компонентов от разных производителей. Вы можете найти имена производителей и пакетов PHP-компонентов в каталоге компонентов на сайте Packagist (рис. 4.4).



Установка компонента

Каждый PHP-компонент может иметь несколько доступных версий (например, 1.0.0, 1.5.0 или 2.15.0). Все доступные версии перечислены в каталоге компонентов на сайте Packagist.

К счастью, нет необходимости выяснять номер последней стабильной версии каждого из компонентов. Менеджер зависимостей Composer сделает это за нас. Перейдите в корневой каталог проекта в терминале и выполните следующую команду, отдельно для каждого PHP-компонента:

```
composer require vendor/package
```

Замените `vendor/package` именами производителей и пакетов компонентов. Например, чтобы установить компонент Flysystem выполните команду:

```
composer require league/flysystem
```

Эта команда укажет менеджеру зависимостей Composer, что тот должен найти и установить последнюю стабильную версию PHP-компоненты. Команда также укажет Composer обновить компонент, до следующей главной версии компонента, но не включая ее. В предыдущем примере, в октябре 2014 года, при установленном компоненте Flysystem версии 0.5.9, команда обновит его до версии 0.*.* , ближайшей к версии 1.*.

Результат выполнения этой команды можно увидеть во вновь созданном или обновленном файле *composer.json* в корневом каталоге проекта. Эта команда также создаст файл *composer.lock*. Сохраните оба файла в системе управления версиями.



Схема Semantic Versioning

Современные компоненты PHP используют схему **Semantic Versioning** (<http://semver.org/>), где номер версии состоит из трех чисел, разделенных точками (.) (например, 1.13.2). Первое число называется *главным номером версии*. Главный номер версии увеличивается, только когда внесенные в PHP-компонент изменения нарушают обратную совместимость. Второе число называется *младшим номером версии*. Младший номер версии увеличивается после внесения в PHP-компонент незначительных изменений, не нарушающих обратной совместимости. Третье и последнее число называется *номером исправлений версии*. Номер исправлений увеличивается, когда в PHP-компоненте исправляются ошибки без нарушения обратной совместимости.

Пример проекта

Давайте закрепим навыки работы с Composer на примере создания PHP-приложения, которое сканирует адреса URL в CSV-файле и выводит список всех недоступных адресов. Наш проект будет отправлять HTTP-запрос по каждому из адресов. Если в ответ будет получен HTTP-отклик с кодом состояния большим или равным 400, приложение отправит недоступный адрес на стандартное устройство вывода. Наш проект будет приложением, запускаемым из командной строки, принимающим путь к CSV-файлу в первом и единственном аргументе командной строки. То есть, передав сценарию путь к CSV-

файлу, мы увидим список недоступных адресов на устройстве стандартного вывода:

```
php scan.php /path/to/urls.csv
```

Каталог проекта показан на рис. 4.5.

```
1. bash
Josh-MacBook-Pro:url-scanner-app josh$ ls -lah
total 0
drwxr-xr-x  4 josh  staff  1368 Oct 25 12:01 .
drwxr-xr-x  9 josh  staff  3068 Oct 25 12:00 ..
-rw-r--r--  1 josh  staff   08 Oct 25 12:01 scan.php
-rw-r--r--  1 josh  staff   08 Oct 25 12:01 urls.csv
Josh-MacBook-Pro:url-scanner-app josh$
```

Рис. 4.5. Структура каталогов проекта

Первое, что я делаю, приступая к разработке нового проекта, – определяю, какие задачи могут быть решены с помощью существующих PHP-компонентов. Сценарий *scan.php* открывает и перебирает содержимое CSV-файла, поэтому нам понадобиться PHP-компонент, который умеет читать и в цикле выбирать CSV-данные. Сценарий *scan.php* также посылает HTTP-запрос по каждому адресу из CSV-файла, поэтому нам понадобится PHP-компонент, способный отправлять HTTP-запросы и проверять HTTP-ответы. Конечно, можно написать собственный код для перебора CSV-файла или отправки HTTP-запросов, но зачем тратить на это наше время, если эти задачи уже были решены раньше? Помните, что нашей целью является сканирование адресов из списка, а не создание библиотеки HTTP и библиотеки парсера CSV.

Поискав на сайте Packagist, я нашел PHP-компоненты *guzzlehttp/guzzle* и *league/csv*. Первый управляет HTTP-сообщениями, а последний анализирует CSV-данные и выполняет их обход. Давайте установим эти компоненты с помощью менеджера зависимостей Composer, выполнив следующие команды в корневом каталоге проекта:

```
composer require guzzlehttp/guzzle;
composer require league/csv;
```

Эти команды загрузят компоненты в новый каталог *vendor/*, в корневом каталоге проекта, и создадут файлы *composer.json* и *composer.lock*.

Файл `composer.lock`

После установки зависимостей проекта с помощью менеджера Composer можно заметить, что Composer создал файл `composer.lock`. Этот файл содержит список всех PHP-компонентов, используемых в проекте, с указанием полных номеров их версий (включая главный номер версии, младший номер версии и номер исправлений). Тем самым для проекта будут зафиксированы конкретные версии PHP-компонентов.

Почему это так важно? При наличии файла `composer.lock`, Composer загружает определенные версии PHP-компонентов, перечисленные в нем, независимо от последней версии компонента доступной на сайте Packagist. Сохраните файл `composer.lock` в системе управления версиями и передайте его всем членам своей команды, чтобы они могли использовать те же версии PHP-компонентов, что и вы. Если члены команды, сервер для разработки и рабочий сервер будут использовать одни и те же версии PHP-компонентов, это уменьшит риск ошибок, вызванных расхождениями версий компонентов.

Но имейте в виду, что команда `composer install` не будет устанавливать версии, более новые, чем перечисленные в файле `composer.lock`. Если понадобиться загрузить новые версии компонентов и обновить файл `composer.lock`, воспользуйтесь командой `composer update`. Команда `composer update` обновляет компоненты до их последних стабильных версий, а также обновляет файл `composer.lock` новыми номерами версий PHP-компонентов.

Автозагрузка PHP-компонентов

Теперь, после установки PHP-компонентов в проект, возникает следующий вопрос: как их использовать? К счастью, в процессе загрузки PHP-компонентов Composer создает единый PSR-совместимый автозагрузчик для всех зависимостей проекта. Все, что нам остается сделать, это добавить в начало файла `scan.php` оператор `require` для подключения автозагрузчика Composer:

```
<?php  
require 'vendor/autoload.php';
```

Автозагрузчик Composer – это обычный PHP-файл `autoload.php`, находящийся в каталоге `vendor/`. При загрузке PHP-компонентов, Composer просматривает файл `composer.json` каждого компонента чтобы определить, как компонент предпочитает выполнять автома-

тическую загрузку, и, получив эту информацию, создает для него локальный PSR-совместимый автозагрузчик. То есть, в проекте можно создать экземпляр любого PHP-компонента и он будет автоматически загружен по требованию! Довольно просто, не так ли?

Реализация `scan.php`

Давайте завершим сценарий `scan.php`, используя компоненты Guzzle и CSV. Напомню, что путь к файлу CSV передается в первом аргументе командной строки (которые доступны в виде массива `$argv`). Сценарий `scan.php` приводится в примере 4.1.

Пример 4.1. Приложение сканирования адресов

```
<?php
// 1. Подключить автозагрузчика Composer
require 'vendor/autoload.php';

// 2. Создать экземпляр http-клиента Guzzle
$client = new \GuzzleHttp\Client();

// 3. Открыть файл CSV и обойти его содержимое
$csv = new \League\Csv\Reader($argv[1]);
foreach ($csv as $csvRow) {
    try {
        // 4. Послать запрос HTTP OPTIONS
        $httpResponse = $client->options($csvRow[0]);

        // 5. Проверить полученный код состояния HTTP
        if ($httpResponse->getStatusCode() >= 400) {
            throw new \Exception();
        }
    } catch (\Exception $e) {
        // 6. Вывести недоступный URL-адрес в устройство
        // стандартного вывода
        echo $csvRow[0] . PHP_EOL;
    }
}
```



Обратите внимание, как используются пространства имен `\League\Csv` и `\GuzzleHttp` при создании экземпляров компонентов `league/csv` и `guzzlehttp/guzzle`. Откуда я узнал, что надо использовать именно эти пространства имен? Я прочел документацию компонентов `guzzlehttp/guzzle` и `league/csv`. Напомню, что все хорошие PHP-компоненты имеют документацию.

Добавьте несколько адресов в файл *urls.csv*, по одному в строке. Удостоверьтесь, что, по крайней мере, один адрес является недопустимым. Затем откройте терминал и выполните сценарий *scan.php*:

```
php scan.php urls.csv
```

Здесь вызывается интерпретатор `php`, которому передается два аргумента. Первый аргумент – путь к сценарию *scan.php*. Второй аргумент – путь к CSV-файлу со списком адресов. Если какой-либо из адресов вернет HTTP-ответ с кодом ошибки, будет выведен в окно терминала.



PHP-сценарии командной строки

Знаете ли вы, что на PHP можно писать сценарии для вызова из командной строки? Это отличный способ автоматизации задач технического обслуживания веб-приложения. Для получения более подробной информации о PHP-сценариях командной строки посетите страницы:

- <http://php.net/manual/wrappers.php.php>
- <https://php.net/manual/reserved.variables.argv.php>
- <https://php.net/manual/reserved.variablesargc.php>

Composer и закрытые хранилища

До сих пор предполагалось использование общедоступных PHP-компонентов с открытым исходным кодом. При том, что я многократно создавал и использовал программное обеспечение с открытым исходным кодом, я должен признать, что, применять только PHP-компоненты с открытым исходным кодом получается не всегда. Иногда в одном приложении необходимо совмещать открытые и проприetaryные компоненты. Это особенно относится к компаниям, которые используют PHP-компоненты собственной разработки, код которых не может быть открыт по лицензионным причинам или вопросам безопасности. Менеджер зависимостей Composer легко справляется и с этим.

Composer может также управлять защищенными компонентами, чьи хранилища требуют аутентификации. Когда выполняется команда `composer install` или `composer update`, Composer сообщает, что хранилище компонента требует аутентификации. Менеджер зависимостей также спросит, не хотите ли вы сохранить учетные данные

хранилища в локальном файле *auth.json* (создается в одном каталоге с файлом *composer.json*). Вот как выглядит пример файла *auth.json*:

```
{  
    "http-basic": [  
        "example.org": [  
            "username": "your-username",  
            "password": "your-password"  
        ]  
    ]  
}
```

В большинстве случаев нежелательно сохранять файл *auth.json* в системе управления версиями. Вместо этого, разработчики проекта обычно создают собственные файлы *auth.json* со своими учетными данными для аутентификации.

Если вы не хотите дожидаться, пока Composer запросит учетные данные, можете вручную определить учетные данные для удаленного компьютера, выполнив следующую команду:

```
composer config http-basic.example.org your-username your-password
```

В этом примере, команда `http-basic` сообщает менеджеру зависимостей Composer, что мы добавляем данные аутентификации для данного домена. Имя хоста `example.org` идентифицирует удаленную машину с закрытым хранилищем компонентов. Последние два аргумента – это имя пользователя и пароль. По умолчанию команда сохранит учетные данные в файле *auth.json* текущего проекта.

Учетные данные можно также сохранить глобально, добавив в команду флаг `--global`. Этот флаг позволит менеджеру зависимостей Composer использовать ваши учетные данные для всех проектов на вашем локальном компьютере:

```
composer config --global http-basic.example.org your-username your-password
```

Глобальные учетные данные хранятся в файле `~/.composer/auth.json`. Если вы используете Windows, глобальные учетные данные хранятся в каталоге `%APPDATA%\\Composer`.



Более подробную информацию о Composer и закрытых хранилищах можно получить на странице <http://bit.ly/auth-manage>, посвященной управлению аутентификацией в Composer.

Создание PHP-компонентов

К этому моменту вы должны уметь находить и использовать PHP-компоненты. Теперь поговорим о *создании* PHP-компонентов. В частности, преобразуем приложение сканера адресов в PHP-компонент и добавим его в каталог компонентов на сайте Packagist.

Создание PHP-компонентов – это отличный способ поделиться своей работой с членами PHP-сообщества. Сообщество построено на принципах обмена и взаимопомощи. Если вы пользуетесь компонентами с открытым исходным кодом, всегда приятно в ответ внести и свой вклад в виде нового открытого и инновационного компонента.



Удостоверьтесь, что вы не пишете уже существующий компонент. Если вы предлагаете улучшения к уже существующему компоненту, подумайте об оформлении улучшений исходного компонента, передав их автору компонента в виде запроса. В противном случае, вы рискуете поспособствовать возникновению путаницы и увеличению фрагментированности экосистемы PHP-компонентов.

Имена производителя и пакета

Перед тем, как разработать PHP-компонент, я выбираю имена производителя и пакета компонента. Напомню, что каждый компонент использует глобально уникальную комбинацию имени производителя и имени пакета, чтобы избежать конфликта имен с другими компонентами. Я рекомендую использовать в именах только строчные буквы.

Имя производителя является брендом, или идентификатором производителя компонента. Многие из моих собственных компонентов PHP имеют имя производителя `codegu`, потому что это мой идентификатор. Выберите имя производителя, которое лучше всего соответствует вам или бренду вашего компонента.



Выполните поиск на сайте Packagist, прежде чем выбрать имя производителя, чтобы убедиться, что оно уже не занято другим разработчиком.

Имя пакета идентифицирует PHP-компонент выбранного производителя. Несколько компонентов могут иметь одно и то же имя производителя. В этом примере я буду использовать имя производителя `modernphp` и имя пакета `scanner`.

Пространства имен

Как рассказывалось в главе 2, каждый компонент существует в собственном пространстве имен, поэтому он не загрязняет глобальное пространство имен или не конфликтует с другими компонентами, которые используют те же имена PHP-классов.

Существует распространенное заблуждение, что пространство имен PHP-компонента должно совпадать с именами производителя и пакета компонента. Это неверно. Пространство имен PHP-компонента никак не связано с именами производителя и пакета компонента. Имена производителя и пакета используются только сайтом Packagist и менеджером зависимостей Composer для идентификации компонента. А пространство имен компонента используется при обращении к компоненту из PHP-кода.

Для этого урока мы создадим компонент, существующий в пространстве имен `Oreilly\ModernPHP`. Этого пространства имен еще не существует. Я просто взял его с потолка для данного конкретного компонента.

Организация файловой системы

Обычно PHP-компоненты имеют следующую организацию каталогов в файловой системе:

`src/`

Каталог с исходным кодом (например, с файлами PHP-классов) компонента.

`tests/`

Каталог с тестами для компонента. Мы не будем использовать этот каталог в данном примере.

`composer.json`

Конфигурационный файл Composer. Этот файл описывает компонент и указывает автозагрузчику Composer порядок отображения пространства имен компонента в каталог `src/`.

README.md

Файл с разметкой Markdown. Содержит полезную информацию о компоненте, в том числе имя, описание, сведения об авторе, порядок использования, рекомендации сподвижникам, лицензионное соглашение и благодарности.

CONTRIBUTING.md

Файл с разметкой Markdown. Описывает, как другие разработчики могут внести свой вклад в этот компонент.

LICENSE

Обычный текстовый файл с лицензионным соглашением компонента.

CHANGELOG.md

Файл с разметкой Markdown. Перечисляет изменения в каждой из версий компонента.



Если у вас возникли проблемы с организацией собственного PHP-компонентта, Взгляните на шаблон организации PHP-компонентов на сайте PHP League: <https://github.com/thephpleague/skeleton>.

Файл *composer.json*

Файл *composer.json* является *обязательным* и должен содержать допустимую разметку JSON с информацией, используемой менеджером Composer для поиска, установки и автозагрузки PHP-компонента. Он также содержит информацию для каталога компонентов Packagist.

Пример 4.2 демонстрирует содержимое файла *composer.json* для нашего компонента сканера адресов. Оно включает все свойства, которые я часто использую в своих собственных PHP-компонентах.

Пример 4.2. Файл *composer.json* компонента сканирования адресов

```
{  
    "name": "modernphp/scanner",  
    "description": "Scan URLs from a CSV file and report inaccessible URLs",  
    "keywords": ["url", "scanner", "csv"],  
    "homepage": "http://example.com",  
    "license": "MIT",  
}
```

```
"authors": [
    {
        "name": "Josh Lockhart",
        "homepage": "https://github.com/codeguy",
        "role": "Developer"
    }
],
"support": [
    "email": "help@example.com"
],
"require": {
    "php": ">=5.4.0",
    "guzzlehttp/guzzle": "~5.0"
},
"require-dev": {
    "phpunit/phpunit": "~4.3"
},
"suggest": {
    "league/csv": "~6.0"
},
"autoload": {
    "psr-4": {
        "Oreilly\\ModernPHP\\": "src/"
    }
}
}
```

Здесь много всего, чтобы сразу все охватить, поэтому пройдемся по всем свойствам в *composer.json* подробно:

name

Имена производителя и пакета компонента, разделенные символом /. Это значение отображается на сайте Packagist.

description

Несколько предложений, кратко описывающих компонент. Это описание также отображается на Packagist.

keywords

Несколько ключевых слов, описывающих компонент. Эти ключевые слова помогут другим разработчикам найти компонент на Packagist.

homepage

Адрес сайта компонента.

license

Лицензия, на основе которой распространяется PHP-компонент. Я предпочитаю использовать MIT Public License.

Более подробную информацию о лицензиях на программное обеспечение можно найти на сайте <http://choosealicense.com>. Не забывайте всегда снабжать свои компоненты ссылками на лицензии.

`authors`

Сведения об авторах проекта. Вы должны указать, по крайней мере, имя и ссылку на сайт для каждого из авторов.

`support`

Сведения о том, где пользователи компонента смогут найти техническую поддержку. Я предпожитаю указывать здесь адрес электронной почты и адрес форума поддержки. Также можно указать, например, IRC-канал.

`require`

Список PHP-компонентов, от которых зависит данный компонент. Для каждой зависимости указывается имя производителя/имя пакета и минимальный номер подходящей версии. Я также указываю минимальную версию PHP для данного компонента. Все перечисленные в этом свойстве зависимости предназначены и для разработки, и для рабочего проекта.

`require-dev`

Подобно свойству `require`, но перечисляет только зависимости, необходимые для разработки. Например, я часто включаю в список `phpunit`, как зависимость для разработки, что позволяет другим разработчикам писать и запускать тесты для компонента. Эти зависимости устанавливаются только в процессе разработки. Они отсутствуют в рабочих проектах.

`suggest`

Подобно свойству `require`, но содержит другие компоненты, которые могут быть полезны при использовании вместе с данным компонентом. В отличие свойства `require`, значениями этого объекта являются текстовые поля свободного формата, описывающие каждый из предлагаемых компонентов. Composer не устанавливает перечисленные здесь компоненты.

`autoload`

Сообщает автозагрузчику Composer, как осуществлять автозагрузку компонента. Я рекомендую использовать авто-

загрузчик PSR-4, как это показано в примере 4.2. Под свойством `psr-4` помещается отображение префикса пространства имен компонента в относительный путь к корневому каталогу компонента. Это делает компонент совместимым со стандартом PSR-4 автозагрузки. В примере 4.2 пространство имен `Oreilly\ModernPHP` отображается в каталог `src/`. Пространство имен в отображении должно заканчиваться двумя обратными слешами (`\\"`), чтобы избежать конфликта имен с другими компонентами, которые используют пространство имен с аналогичной последовательностью символов. В данном случае, если предположить, что создается гипотетический класс `Oreilly\ModernPHP\Url\Scanner`, Composer загрузит файл PHP-класса `src\Url\Scanner.php`.



Более подробную информацию об организации файла `composer.json` можно найти на сайте getcomposer.org.

Файл *README*

С файла *README* пользователи обычно начинают знакомство с компонентом. Это особенно характерно для компонентов, размещенных на сайтах GitHub и Bitbucket. Поэтому, важно, чтобы файл *README* компонента содержал, по крайней мере, следующую информацию:

- имя компонента и его описание;
- инструкцию по установке;
- инструкцию по использованию;
- инструкцию по тестированию;
- инструкцию другим разработчикам;
- сведения о поддержке;
- сведения об авторах;
- лицензию программного обеспечения.

Реализация компонента

А теперь, после подготовки всех ингредиентов компонента, приступим к его реализации. Под реализацией понимается создание

классов, интерфейсов и трейтов, образующих PHP-компонент. Какие именно классы вы напишете, и сколько их будет, полностью зависит от предназначения PHP-компонента. Однако, все классы, интерфейсы и трейты компонента должны находиться в каталоге `src/` и существовать под префиксом пространства имен компонента, указанным в файле `composer.json`.



Сайты GitHub и BitBucket могут отображать файлы `README` в формате Markdown. Это означает, что есть возможность подготовить богато форматированные файлы `README`, содержащие заголовки, списки, ссылки и изображения. Используйте это для своих нужд! Для этого достаточно просто добавить расширение `.md` или `.markdown` к имени файла `README`. То же можно сделать с файлами `CONTRIBUTING` и `CHANGELOG`. Узнать больше о формате Markdown можно на сайте <http://bit.ly/markdown-doc>.

Для иллюстрации я создам один PHP-класс `Scanner` в подпространстве имен `Url` пространства имен `Oreilly\ModernPHP`, которые перечислены в файле `composer.json`. Класс `Scanner` хранится в файле `src/Url/Scanner.php`. Он реализует ту же логику, что и пример приложения сканера адресов, с той лишь разницей, что инкапсулирует функционал сканирования адресов в PHP-классе (пример 4.3).

Пример 4.3. Класс компонента сканирования адресов

```
<?php
namespace Oreilly\ModernPHP\Url;
class Scanner
{
    /**
     * @var array Массив URL-адресов
     */
    protected $urls;

    /**
     * @var \GuzzleHttp\Client
     */
    protected $httpClient;

    /**
     * Конструктор
     * @param array $urls Массив URL-адресов для сканирования
     */
    public function __construct(array $urls)
```

```
    }

    $this->urls = $urls;
    $this->httpClient = new \GuzzleHttp\Client();
}

/**
 * Возвращает недопустимые URL-адреса
 * @return array
 */
public function getInvalidUrls()
{
    $invalidUrls = [];
    foreach ($this->urls as $url) {
        try {
            $statusCode = $this->getStatusCodeForUrl($url);
        } catch (\Exception $e) {
            $statusCode = 500;
        }
        if ($statusCode >= 400) {
            array_push($invalidUrls, [
                'url' => $url,
                'status' => $statusCode
            ]);
        }
    }
    return $invalidUrls;
}

/**
 * Возвращает код состояния HTTP для URL-адреса
 * @param string $url Удаленный URL-адрес
 * @return int Код состояния HTTP
 */
protected function getStatusCodeForUrl($url)
{
    $httpResponse = $this->httpClient->options($url);

    return $httpResponse->getStatusCode();
}
```

Вместо парсинга и обхода CSV-файла, мы внедрили массив адресов в конструктор класса `Scanner`, чтобы класс сканирования адресов был как можно более универсальным. Требование обязательного использования CSV-файла ограничивает область применения компонента. Потребовав передачу массива адресов, мы даем возможность конечному пользователю самому решить, как именно получить массив адресов (из PHP-массива, CSV-файла, итератора и т. д.). При этом мы по-прежнему рекомендуем компонент `league/csv`, потому

что он может быть полезен разработчикам, использующим наш компонент. Мы включили компонент `league/csv` в свойство `suggest` файла `composer.json`.

Класс `Scanner` имеет жесткую зависимость от компонента `guzzlehttp/guzzle`. Однако, мы изолировали HTTP-запрос каждого адреса в методе `getStatusCodeForUrl()`. Это позволяет применить заглушки (или *переопределения*) при реализации этого метода, для модульного тестирования компонента, чтобы тесты не были завязаны на соединение с Интернетом.

Управление версиями

Мы почти закончили. Прежде, чем мы добавить компонент в каталог на сайте Packagist, мы должны опубликовать его в общедоступном хранилище кода. Я предполагаю опубликовать свои PHP-компоненты с открытым кодом на сайте GitHub. Однако с тем же успехом можно использовать любое другое общедоступное хранилище (я опубликовал этот компонент на GitHub и его можно найти по ссылке <https://github.com/modern-php/scanner>).

Неплохо также пометить каждый релиз компонента, используя схему Semantic Versioning. Это позволит пользователям компонента запрашивать конкретные версии (например, `~1.2`). Я создал тег `1.0.0` для компонента сканера адресов.

Размещение на сайте Packagist

Теперь все готово к размещению компонента на сайт Packagist. Если вы никогда ранее не пользовались GitHub, создайте учетную запись на сайте Packagist. Войти на сайт Packagist можно так же с помощью учетной записи GitHub.

После входа щелкните на большой зеленой кнопке **Submit Package** (Отправить пакет) в правом верхнем углу (см. рис. 4.6). Введите полный адрес репозитория Git в текстовое поле **Repository URL** (URL репозитория) и щелкните на кнопке **Check** (Проверить). Packagist проверит адрес хранилища и запросит подтверждение на размещение. Нажмите кнопку **Submit** (Отправить), чтобы завершить размещение компонента. В ответ Packagist добавит компонент в каталог и перенаправит вас на страницу с описанием компонента, как показано на рис. 4.6.

Как видите, имя компонента, описание, ключевые слова, зависимости и предложения получены из файла `composer.json` компонента.

Также обратите внимание, что здесь отображены ветви хранилища и теги. Packagist устанавливает прямую связь между тегами хранилища и семантикой номеров версий. Вот почему я рекомендую отмечать хранилища фактическими номерами версий, такими как 1.0.0, 1.1.0 и так далее. Однако также имеется большое, выделенное красным, сообщение оповещения с текстом:

This package is not auto-updated. Please set up the GitHub Service Hook for Packagist so that it gets updated whenever you push!

(Этот пакет не обновляется автоматически. Настройте службу GitHub Service Hook для Packagist, чтобы каталог обновлялся синхронно с вашими изменениями!)

Мы можем активировать ловушку сайтов GitHub или Bitbucket, которая уведомит сайт Packagist об обновлении хранилища компонента. Более подробную информацию о настройках ловушек хранилища можно найти на странице <https://packagist.org/profile/>.

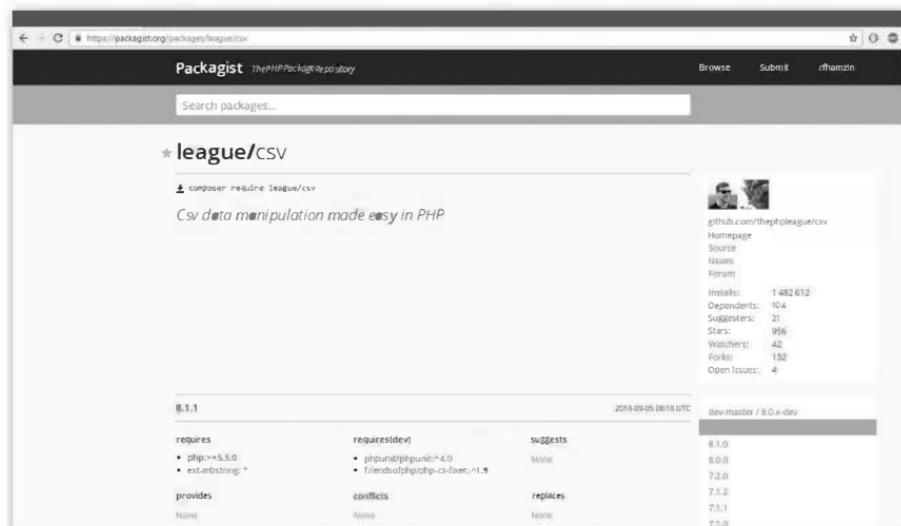


Рис.4.6. Описание компонента на сайте Packagist

Использование компонента

Мы закончили! Теперь любой желающий сможет установить компонент сканера адресов с помощью менеджера зависимостей Composer и использовать его в своих PHP-приложениях. Выполним

те следующую команду в окне терминала, чтобы установить компонент:

```
composer require modernnphp/scanner
```

После этого можно использовать компонент, как это показано в примере 4.4.

Пример 4.4. Использование компонента сканирования адресов

```
<?php
require 'vendor/autoload.php';
$urls = [
    'http://www.apple.com',
    'http://php.net',
    'http://sdfssdwerw.org'
];
$scanner = new \Oreilly\ModernPHP\Url\Scanner($urls);
print_r($scanner->getInvalidUrls());
```

ГЛАВА 5.

Передовой опыт

В этой главе собран передовой опыт разработки PHP-приложений. Следуя описываемым здесь методикам, вы сделаете свои приложения более быстрыми, безопасными и стабильными. Язык PHP – это конгломерат инструментов, созданных за длительный период времени, и мы рассмотрим наиболее удачные приемы их применения. Со временем, при появлении новых версий языка PHP, инструменты заменялись новыми и более эффективными. Но, к сожалению, язык PHP по-прежнему содержит устаревшие инструменты и их неосторожное использование приводит к разработке медленных и ненадежных приложений. Весь фокус в том, чтобы знать, какие инструменты использовать, а какие игнорировать. Эта глава содержит всю необходимую для этого информацию.

Я не занимаюсь проповедованием «передового опыта» с вершины академической башни, сделанной из слоновой кости. Эта глава содержит практические советы, которым я сам следую ежедневно при работе над всеми своими проектами. Почерпнутые здесь знания вы сможете сразу же применить в собственных проектах.



Передовой опыт, описанный в этой главе, применим не только к старым, но и к современным версиям языка PHP. Однако, как использовать эти методики, зависит от версии языка PHP. В более новых версиях языка PHP имеются инструменты, которые упрощают применение этих методик. В данной главе описано, как применять передовой опыт, опираясь на новейшие инструменты языка PHP, включенные в версии 5.3 и выше.

Санирование и проверка ввода, и экранирование вывода

Фокс Малдер (Fox Mulder) прав, утверждая: «Не доверяй никому». Никогда не доверяйте *никаким* данным, полученным из источника, не находящегося под вашим непосредственным контролем. Вот некоторые, из таких внешних источников:

- `$_GET;`
- `$_POST;`
- `$_REQUEST;`
- `$_COOKIE;`
- `$argv;`
- `php://stdin;`
- `php://input;`
- `file_get_contents();`
- удаленные базы данных;
- удаленные службы;
- данные от ваших клиентов.

Все эти внешние источники данных являются потенциальными источниками хакерских атак, связанных с инъекцией вредоносных данных в ваши сценарии (преднамеренной или случайной). Писать PHP-схемарии, принимающие вводимые пользователем данные и отображающие их, достаточно просто. Но написание их с соблюдением условий *безопасности* потребует некоторого дополнительного осмысления. Простейший совет, который я могу вам дать: *санировать* ввод, *проверять* данные и *контролировать* вывод.

Санирование ввода

При *санировании* ввода (т. е., очистка данных, поступивших из любых источников, перечисленных выше), необходимо экранировать или удалить небезопасные символы. Санацию входных данных следует производить *прежде*, чем они достигнут хранилища данных в вашем приложении (например, Redis или MySQL). Это – первая линия обороны. Например, предположим, что ваш сайт принимает комментарии в формате HTML. По умолчанию, ничто не помешает посетителю добавить следующий «хитрый» тег `<script>` в текст комментария:

```
<p>
    Это была полезная статья!
</p>
<script>window.location.href='http://example.com';</script>
```

Если не выполнить санирование этого комментария, в базу данных запишется злонамеренный код, что приведет к его включению в разметку веб-сайта при отображении. Когда посетитель сайта перейдет на страницу с таким неочищенным комментарием, он будет перенаправлен на вредоносный сайт. Это лишь один из примеров, почему необходимо санировать входные данные, которые вы не можете контролировать. Исходя из моего опыта, существуют несколько типов входных данных, с которыми вы будете сталкивать наиболее часто: тексты в формате HTML, SQL-запросы и сведения из профилей пользователей (например, адреса электронной почты и номера телефонов).

Тексты в формате HTML

Выполнить санирование текста в формате HTML можно с помощью функции `htmlentities()` (пример 5.1), заменяющей специальные символы (например, `&`, `>`, `″`) на эквивалентные им HTML-сущности. Эта функция экранирует все HTML-символы в строке и делает ее безопасной при помещении ее в хранилище данных вашего приложения.

Однако, функция `htmlentities()` не очень «умна». Она не проверяет ввод в формате HTML. По умолчанию она не экранирует одиночные кавычки, и не определяет кодировку входной строки. Ниже приводится пример правильного использования функции `htmlentities()`. В первом аргументе ей передается входная строка. Во втором – константа `ENT_QUOTES`, побуждающая функцию преобразовывать одинарные кавычки. В третьем – кодировка входной строки.

Пример 5.1. Санирование ввода с помощью функции `htmlentities()`

```
<?php
$input = '<p><script>alert ("Вы выиграли в нигерийскую лотерею!") ;
</script></p>';
echo htmlentities($input, ENT_QUOTES, 'UTF-8');
```

Если понадобится более тщательно санировать входные данные в формате HTML, воспользуйтесь библиотекой HTML Purifier – очень надежной и безопасной PHP-библиотекой, которая санирует данные в формате HTML в соответствии с определяемыми вами правилами.

Недостатками библиотеки HTML Purifier является низкая скорость работы и сложность настройки.



Не санируйте текст в формате HTML с помощью функций регулярных выражений, таких как `preg_replace()`, `preg_replace_all()` и `preg_replace_callback()`. Регулярные выражения сложны, входной текст может содержать недопустимую разметку HTML, из-за чего увеличивается риск допустить ошибку.

SQL-запросы

Иногда бывает нужно создать на основе входных данных SQL-запрос. Такие данные могут поступать в строке запроса HTTP (например, `?user=1`) или в виде URI-сегмента (например, `/users/1`). Если не проявить осторожности, злоумышленники смогут целенаправленно трансформировать ваши SQL-запросы и посеять хаос в вашей базе данных. Например, я встречал множество начинающих PHP-программистов, формирующих SQL-запросы путем конкатенации необработанных входных данных из `$_GET` и `$_POST`, как показано в примере 5.2.

Пример 5.2. Плохой SQL-запрос

```
$sql = sprintf(
    'UPDATE users SET password = "%s" WHERE id = %s',
    $_POST['password'],
    $_GET['id']
);
```

Это очень плохо! Представьте, что произойдет, если некто отправит следующий HTTP-запрос вашему PHP-сценарию?

```
POST /user?id=1 HTTP/1.1
Content-Length: 17
Content-Type: application/x-www-form-urlencoded

password=abc"--
```

Этот HTTP-запрос сменит пароли *всех* пользователей на abc, потому что многие базы данных рассматривают символы -- в SQL-запросе как начало комментария и проигнорируют весь следующий за ними текст. *Никогда не используйте не прошедшие санитаризации входные данные в SQL-запросе.* Если понадобится встроить входные

данные в SQL-запрос, используйте *подготовленный заранее оператор PDO*. PDO – это встроенная в язык PHP абстракция базы данных, обеспечивающая единый интерфейс для работы с разными базами данных. Подготовленные заранее операторы PDO являются инструментом для санации и безопасного включения внешних данных в SQL-запросы, позволяя избегать проблем, проиллюстрированных в примере 5.2. Я считаю PDO и операторы PDO настолько важными инструментами, что посвятил им отдельный раздел, далее в этой главе.

Сведения из профилей пользователей

Если приложение работает с учетными записями пользователей, вы, наверняка, столкнетесь с адресами электронной почты, номерами телефонов, почтовыми индексами и прочими сведениями, связанными с пользователями. Язык PHP предлагает сценарий их санирования, основанный на функциях `filter_var()` и `filter_input()`. Эти две функции принимают в качестве аргументов различные флаги для санации разных видов входных данных: электронных писем, URL-кодированных строк, целых чисел, чисел с десятичной точкой, символов HTML, URL-адресов и заданного диапазона символов ASCII.

Пример 5.3 демонстрирует санацию адреса электронной почты с помощью удаления из него всех символов, за исключением букв, цифр и символов !#\$%&!*+-/=?^_`{|}~@.{[].}

Пример 5.3. Санация адреса электронной почты из профиля пользователя

```
<?php
$email = 'john@example.com';
$emailSafe = filter_var($email, FILTER_SANITIZE_EMAIL);
```

Пример 5.4 демонстрирует, как обезопасить сведения о пользователе, удалив из них символы с кодами ASCII меньше 32 и больше 127.

Пример 5.4. Санирование интернациональных символов из профиля пользователя

```
<?php
$string = "\nIñtërnàtiònàlizætiòn\t";
$safeString = filter_var(
    $string,
    FILTER_SANITIZE_STRING,
    FILTER_FLAG_STRIP_LOW|FILTER_FLAG_ENCODE_HIGH
);
```



Более подробную информацию о флагах и параметрах функции `filter_var()` можно найти на странице <http://php.net/manual/function.filter-var.php>.

Проверка данных

Важное место занимает также *проверка* данных. В отличие от санации, при проверке не происходит удаление информации. Проверка лишь отвечает на вопрос: соответствуют ли входные данные ожидаемым значениям. Если вы планируете получить адрес электронной почты, убедитесь, что входные данные представляют собой именно адрес электронной почты. Если вы планируете получить номер телефона, убедитесь, что входные данные являются именно номером телефона. Это все, что делает проверка. Проверка должна гарантировать, что в хранилище приложения попадает только правильная и надлежащим образом отформатированная информация. Столкнувшись с неправильными данными, вы сможете прервать операцию сохранения данных и вывести сообщение об ошибке пользователю вашего приложения. Кроме того проверка предотвращает возможные ошибки базы данных. Например, если база данных MySQL предполагает ввод значения типа `DATETIME`, а получает вместо него строку `next year`, она либо выдаст ошибку, либо использует значение по умолчанию (естественно неправильное). И в том и в другом случае, целостность данных приложения будет нарушена записью недостоверных данных.

Проверить данные, введенные пользователем, можно с помощью функции `filter_var()`, используя один из флагов `FILTER_VALIDATE_*`. Язык PHP предоставляет флаги для проверки булевых значений, адресов электронной почты, десятичных чисел с точкой, целых чисел, IP-адресов, регулярных выражений и URL-адресов. Пример 5.5 демонстрирует проверку адреса электронной почты.

Пример 5.5. Проверка адреса электронной почты

```
<?php
$input = 'john@example.com';
$isEmail = filter_var($input, FILTER_VALIDATE_EMAIL);
if ($isEmail !== false) {
    echo "Success";
} else {
    echo "Fail";
}
```

Обратите внимание на значение, возвращаемое функцией `filter_var()`. Если проверка завершилась успешно, функция вернет оригинальное значение. В случае неудачи она вернет значение `false`.

Хотя функция `filter_var()` предоставляет множество флагов типов проверок, она не предназначена для проверки абсолютно всего. Я рекомендую применять для проверок следующие дополнительные компоненты:

- `aura/filter`
- `respect/validation`
- `symfony/validator`



Проверяйте и санируйте входные данные, чтобы убедиться в их безопасности и соответствии предъявляемым требованиям.

Экранирование вывода

Когда дело доходит до вывода веб-страницы или формирования ответа службы, важно не забывать об **экранировании** вывода. Это еще один уровень защиты, предотвращающий попадание вредоносного кода в формируемую страницу и исключение возможности их случайного выполнения пользователями вашего приложения.

Экранирование вывода осуществляется с помощью функции PHP `htmlentities()`, уже упоминавшуюся выше. Не забудьте передать ей константу `ENT_QUOTES` во втором аргументе, чтобы она преобразовывала как одинарные, так и двойные кавычки. Укажите в третьем аргументе нужную кодировку (обычно `UTF-8`). Пример 5.6 демонстрирует экранирование вывода HTML-текста перед отображением.

Пример 5.6. Экранирование вывода с помощью функции `htmlentities`

```
<?php  
$output = '<p><script>alert("NSA backdoor installed");</script>';  
echo htmlentities($output, ENT_QUOTES, 'UTF-8');
```

Некоторые механизмы шаблонов в PHP, такие например как `twig/twig` (мой любимый) или `smarty/smarty` экранируют вывод автоматически. Механизм шаблонов Twig от Sensio Labs, к примеру, экранирует по умолчанию весь вывод, если специально не отключить его. Он делает это блестяще и обеспечит хорошую защиту вашим веб-приложениям.

Пароли

Безопасности паролей придается огромное значение, учитывая рост количества атак. Как часто вы аннулировали операции с кредитной картой из-за взлома крупной розничной сети? Многие розничные торговцы стали (и будут становиться) жертвами хакеров, потому что не защищают свои системы с помощью современных методик обеспечения безопасности. Приложения на PHP ничем не отличаются от других, они также являются уязвимыми для атак, если не принять соответствующих мер предосторожности.

Одной из важнейших мер предосторожности является поддержание безопасности паролей. В ваши обязанности входит обеспечение безопасного управления, кодирования и хранения паролей пользователей. И не имеет значения, является ли приложение тривиальной игрой или сводом сверхсекретных деловых документов. Ваши пользователи доверили вам свою информацию и ждут от вас организации ее охраны, основанной на передовых методиках сохранения безопасности.

Я знаю многих PHP-разработчиков не имеющих понятия о безопасном управлении паролями. Реализовать такое управление достаточно сложно. К счастью, в языке PHP имеются встроенные инструменты, значительно облегчающие обеспечение безопасности паролей. В этом разделе мы рассмотрим, как использовать эти инструменты в соответствии с современными методиками обеспечения безопасности.

Не храните пароли в открытом виде

Вы не должны хранить пароли в открытом виде. У вас не должно быть *даже возможности* узнать пароли пользователей. Всегда существует риск взлома базы данных приложения, поэтому пароли не должны храниться в ней в открытом виде и должна быть исключена возможность их дешифровки. Утечка паролей является серьезным нарушением доверия и приведет вас или вашу компанию к необходимости нести тяжелое бремя юридической ответственности. Чем меньше вы знаете, тем в большей вы безопасности.

Не ограничивайте пароли ваших пользователей

Меня всегда расстраивает, когда веб-сайт требует, чтобы пароль моей учетной записи соответствовал определенному формату. Я сердусь, когда пароль моей учетной записи не может включать более чем

{N} символов. *Почему!*? Я понимаю, что ввод ограничений на формат паролей связан с совместимостью с устаревшими приложениями или базами данных, но это не является оправданием плохой политики безопасности.

Никогда не ограничивайте пароли ваших пользователей. Требуя соответствия паролей определенному шаблону, вы фактически оказываете услугу злоумышленникам, пытающимся взломать ваше приложение. Если и необходимо чем-то ограничить пароли пользователей, то это только минимальная длина. Ничем также не обосновано применение черного списка часто используемых паролей или паролей, имеющихся в словаре.

Не отправляйте пароли пользователей по электронной почте

Никогда не отправляйте пароли по электронной почте. Если вы вышлете мне мой пароль по электронной почте, я сделаю относительно вас три важных заключения: вы знаете мой пароль, вы храните мой пароль в открытом или легко дешифруемом виде и у вас не возникает никаких сомнений при отправке моего пароля через Интернет в виде обычного текста.

Вместо этого отправьте по электронной почте адрес, перейдя по которому, я смогу ввести или изменить свой пароль. Обычно веб-приложение генерируют уникальный ключ, который может быть использован только один раз для ввода или изменения пароля. Предположим, что я забыл свой пароль. Я щелкаю на кнопке «Забыл пароль» в форме входа и приложение перенаправляет меня в форму, где я смогу ввести мой адрес электронной почты для запроса нового пароля. Приложение генерирует уникальный ключ, связывает его с введенным адресом электронной почты и отправляет письмо с адресом, который включает в себя уникальный ключ в виде части адреса или строкового параметра запроса. Открыв страницу по полученному адресу, приложение проверит ключ и в случае успеха позволит ввести новый пароль. После того как я выберу новый пароль, приложение аннулирует ключ.

Хеширование паролей пользователей с помощью *bcrypt*

Хешируйте пароли пользователей. Не *шифруйте*, а именно хешируйте. Шифрование и хеширование не синонимы. Шифрование –

это двусторонний алгоритм, то есть то, что зашифровано, может быть впоследствии расшифровано. Хеширование – алгоритм односторонний. Хешированные данные нельзя вернуть в первоначальное представление и идентичные данные не всегда имеют одинаковые хеши.

Сохраняя пароль пользователя в базе данных, сначала хешируйте его, а затем сохраняйте полученный *хеш пароля* в базе данных. Если хакеры взломают базу данных, они увидят только бесполезные хеши, взлом которых потребует огромного количества времени и ресурсы агентства национальной безопасности Соединённых Штатов.

Существует множество алгоритмов хеширования (например, MD5, SHA1, *bcrypt*, *scrypt*). Некоторые из них работают быстро, но предназначены только для проверки целостности данных. Другие работают медленно, но более надежны и безопасны. Когда дело касается генерации и хранения паролей нас больше устроят пусты и медленные, но надежные и безопасные алгоритмы.

На сегодняшний день наиболее безопасным аттестованным алгоритмом хеширования является *bcrypt*. По сравнению с MD5 и SHA1, алгоритм *bcrypt* работает значительно медленнее. Для защиты от атак с помощью радужных таблиц алгоритм *bcrypt* автоматически использует соль (*salt*). Алгоритм *bcrypt* требует большого количества времени (измеряемого секундами) для итеративного хеширования данных, чтобы сгенерировать супер-защищенное окончательное значение хеш-функции. Количество итераций хеширования называется *коэффициентом производительности* (*work factor*). Увеличение коэффициента производительности вызывает экспоненциальное увеличение времени, которое должны будут затратить злоумышленники на взлом хешей паролей. Алгоритм *bcrypt* можно легко адаптировать к будущему увеличению быстродействия компьютеров с помощью увеличения значения коэффициента производительности.

Алгоритм *bcrypt* был тщательно исследован экспертами. Специалисты, гораздо более компетентные, чем я, исследовали алгоритм *bcrypt* на предмет возможности взлома, но до сих пор ничего не нашли. Очень важно полагаться только на аттестованные алгоритмы хеширования. Никогда не пытайтесь создать свой собственный алгоритм хеширования. Один в поле не воин и вы наверняка не являетесь экспертом в области криптографии (если это все же не так, мне остается только сказать: «Привет вам, Брюс Шнайер (Bruce Schneier)»).

Программный интерфейс хеширования паролей

Как видите, при работе с паролями пользователей необходимо учитывать множество нюансов. Однако, Энтони Феррара (Anthony Ferrara) был достаточно любезен, чтобы взять на себя труд по созданию для PHP 5.5.0 программного интерфейса хеширования паролей. Встроенный программный интерфейс хеширования паролей в языке PHP предоставляет несложные в использовании функции, значительно упрощающие хеширование и проверку паролей. К тому же, они по умолчанию используют алгоритм хеширования bcrypt.



Энтони Феррара (также известный в Twitter как @ircmaxell) является разработчиком для разработчиков (Developer Advocate) в Google и известным авторитетом в областях, связанных с производительностью и безопасностью кода на языке PHP. Энтони также является автором программного интерфейса хеширования паролей для PHP. Я рекомендую вам прочитать его блог в Twitter. От себя лично хочу поблагодарить Энтони. Он внес большой вклад в улучшение безопасности приложений на PHP и сделал более доступным использование передовых методик безопасности.

При создании веб-приложений вы наверняка столкнетесь с необходимостью реализации двух следующих задач: регистрация пользователя и вход пользователя. Рассмотрим, как программный интерфейс хеширования паролей упрощает их решение.

Регистрация пользователя

Веб-приложение не может существовать без пользователей, а пользователям необходим способ регистрации своей учетной записи. Предположим, что наше гипотетическое приложение имеет в своем составе PHP-файл */register.php*. Этот PHP-файл получает URL-кодированный HTTP-запрос POST, содержащий адрес электронной почты и пароль. Если адрес электронной почты является правильным и пароль содержит не менее восьми символов, приложение должно создать учетную запись пользователя. Ниже приводится пример HTTP-запроса POST:

```
POST /register.php HTTP/1.1
Content-Length: 43
```

```
Content-Type: application/x-www-form-urlencoded
email=john@example.com&password=sekritshhh!
```

В примере 5.7 представлен файл *register.php*, получающий HTTP-запрос *POST*.

Пример 5.7. Сценарий регистрации пользователя

```
01 <?php
02 try {
03     // Проверить адрес электронной почты
04     $email = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
05     if (! $email) {
06         throw new Exception('Неправильный адрес электронной почты');
07     }
08
09     // Проверить пароль
10    $password = filter_input(INPUT_POST, 'password');
11    if (! $password || mb_strlen($password) < 8) {
12        throw new Exception('Пароль должен содержать не менее 8 символов');
13    }
14
15     // Создать хеш пароля
16    $passwordHash = password_hash(
17        $password,
18        PASSWORD_DEFAULT,
19        ['cost' => 12]
20    );
21    if ($passwordHash === false) {
22        throw new Exception('Ошибка при хешировании пароля');
23    }
24
25     // Создать учетную запись пользователя (ЭТО ПСЕВДОКОД)
26    $user = new User();
27    $user->email = $email;
28    $user->password_hash = $passwordHash;
29    $user->save();
30
31     // Перенаправить на страницу входа
32    header('HTTP/1.1 302 Redirect');
33    header('Location: /login.php');
34 } catch (Exception $e) {
35     // Отчет об ошибке
36     header('HTTP/1.1 400 Bad request');
37     echo $e->getMessage();
38 }
```

В примере 5.7:

- Строки 4–7 проверяют адрес электронной почты пользователя. Если адрес неправильный, возбуждается исключение.

- Строки 10–13 проверяют пароль, извлеченный из тела HTTP-запроса. Если пароль содержит менее восьми символов, возбуждается исключение.
- Строки 16–23 создают хеш пароля вызовом функции `password_hash()` программного интерфейса хеширования паролей PHP. В первом аргументе функции `password_hash()` передается пароль в незашифрованном виде. Во втором аргументе – константа `PASSWORD_DEFAULT`, требующая использовать алгоритм хеширования `bcrypt`. В последнем аргументе – массив с параметрами хеширования. Ключ `cost` массива определяет коэффициент производительности алгоритма `bcrypt`. По умолчанию коэффициент производительности равен 10, но вы должны увеличить коэффициент производительности для вашей конфигурации оборудования до значения, при котором на хеширование пароля будет затрачиваться от 0,1 до 0,5 секунды. Если хеширование пароля заканчивается неудачей, возбуждается исключение.
- Строки 26–29 сохраняют гипотетическую учетную запись. Эти строки содержат псевдокод и вы должны заменить их реальным кодом, подходящим вашему приложению. Обратите внимание, что сохраняется хеш пароля, а не сам пароль, выделенный из тела HTTP-запроса. Здесь также сохраняется адрес электронной почты, который используется для поиска и входа пользователя в свою учетную запись.



Храните хеши паролей в столбце базы данных с типом `VARCHAR(255)`. Это позволит в будущем без проблем перейти к хранению более длинных хешей паролей, полученных с помощью алгоритмов, которые придут на смену `bcrypt`.

Вход пользователя

Наше гипотетическое приложение также имеет в своем составе PHP-файл `/login.php`. Этот файл принимает HTTP-запрос `POST` с адресом электронной почты и пароль, используемый для идентификации, аутентификации и входа пользователя. Вот как мог бы выглядеть такой HTTP-запрос `POST`:

```
POST /login.php HTTP/1.1
Content-Length: 43
```

```
Content-Type: application/x-www-form-urlencoded
email=john@example.com&password=sekritshhh!
```

Файл *login.php* отыскивает учетную запись пользователя по адресу электронной почты, проверяет переданный пароль, используя хеш пароля из учетной записи, и выполняет вход пользователя в учетную запись. Содержимое файл *login.php* приводится в примере 5.8.

Пример 5.8. Сценарий входа пользователя

```
01 <?php
02 session_start();
03 try {
04     // Извлечь адрес электронной почты из тела запроса
05     $email = filter_input(INPUT_POST, 'email');
06
07     // Извлечь пароль из тела запроса
08     $password = filter_input(INPUT_POST, 'password');
09
10    // Найти учетную запись по адресу электронной почты (ЭТО ПСЕВДОКОД)
11    $user = User::findByEmail($email);
12
13    // Проверить пароль с помощью хеша пароля из учетной записи
14    if (password_verify($password, $user->password_hash) == false) {
15        throw new Exception('Invalid password');
16    }
17
18    // Повторно хешировать пароль при необходимости (см. примечание ниже)
19    $currentHashAlgorithm = PASSWORD_DEFAULT;
20    $currentHashOptions = array('cost' => 15);
21    $passwordNeedsRehash = password_needs_rehash(
22        $user->password_hash,
23        $currentHashAlgorithm,
24        $currentHashOptions
25    );
26    if ($passwordNeedsRehash === true) {
27        // Сохранить новый хеш пароля (ЭТО ПСЕВДОКОД)
28        $user->password_hash = password_hash(
29            $password,
30            $currentHashAlgorithm,
31            $currentHashOptions
32        );
33        $user->save();
34    }
35
36    // Сохранить состояния входа пользователя в сеансе
37    $_SESSION['user_logged_in'] = 'yes';
38    $_SESSION['user_email'] = $email;
39
40    // Перенаправить на страницу профиля пользователя
```

```
41     header('HTTP/1.1 302 Redirect');
42     header('Location: /user-profile.php');
43 } catch (Exception $e) {
44     header('HTTP/1.1 401 Unauthorized');
45     echo $e->getMessage();
46 }
```

В примере 5.8:

- Строки 5 и 8 извлекают адрес электронной почты и пароль из тела HTTP-запроса.
- Стока 11 отыскивает учетную запись пользователя по адресу электронной почты, полученному из тела HTTP-запроса. Здесь я использую псевдокод и вы должны заменить эту строку кодом, приемлемым для вашего приложения.
- Строки 14–16 сравнивают пароль, извлеченный из тела HTTP-запроса, с хешем пароля в учетной записи пользователя. Сравнение выполняется с помощью функции `password_verify()`. Если проверка заканчивается неудачей, возбуждается исключение.
- Строки 19–34 проверяют соответствие хеша пароля пользователя в учетной записи текущим параметрам алгоритма хеширования, заданным в аргументах функции `password_needs_rehash()`. Если хеш пароля устарел, создается новое значение хеша с применением текущих параметров алгоритма и хеш в учетной записи пользователя обновляется.

Проверка пароля

Функция `password_verify()` сравнивает незашифрованный пароль из тела HTTP-запроса с хешем пароля, хранящимся в учетной записи пользователя. Эта функция принимает два аргумента. В первом передается незашифрованный пароль. Во втором – хеш пароля из учетной записи пользователя. Если функция `password_verify()` вернет `true`, пароль правильный и выполняется вход пользователя. В противном случае пароль неправильный и процесс входа прерывается.

Повторное хеширование пароля

Ниже строки 17 в примере 5.8, после успешной аутентификации, можно выполнить вход пользователя. Но перед тем как сделать это, необходимо проверить, не устарел ли хеш пароля, хранимый в учетной записи пользователя. Если он устарел, создается новый хеш. За-

чем обновлять хеши паролей? Предположим, что приложение было создано два года назад, и тогда использовался коэффициент производительности алгоритма bcrypt равный 10. На сегодняшний день хакеры стали умнее, а компьютеры быстрее, и мы решили использовать коэффициент производительности равный 20. Но, к сожалению, все еще существуют учетные записи, хранящие хеши паролей, созданные при значении коэффициента производительности алгоритма bcrypt 10. После проверки правильности пароля, проверяется необходимость обновления хеша пароля пользователя с помощью функции `password_needs_rehash()`. Эта функция определяет, создан ли текущий хеш пароля с помощью тех же параметров алгоритма хеширования. Если хеш пароля нуждается в обновлении, выполняется хеширование незашифрованного пароля, полученного из тела HTTP-запроса, с использованием текущих параметров алгоритма, и обновляется значение хеша в учетной записи пользователя.



Проще всего использовать функцию `password_needs_rehash()` в сценарии входа пользователя, потому в нем есть доступ к старому хешу пароля и к незашифрованному паролю.

Программный интерфейс хеширования паролей для PHP < 5.5.0

Пусть вас не пугает отсутствие возможности использовать версию PHP 5.5.0 или выше. Вы можете использовать компонент Энтони Феррары (Anthony Ferrara) `ircmaxell/password-compat`. В нем реализованы все функции программного интерфейса хеширования паролей PHP, перечисленные ниже:

- `password_hash()`
- `password_get_info()`
- `password_needs_rehash()`
- `password_verify()`

Компонент `ircmaxell/password-compat` похож как две капли воды на программный интерфейс хеширования паролей последних версий PHP. Подключите компонент к вашему приложению с помощью Composer и работайте дальше.

Даты, время и часовые пояса

Манипуляции с датами и временем достаточно сложны. Практически каждый PHP-разработчик хотя бы раз ошибался при работе с датами и временем. Поэтому я рекомендую при обработке дат и времени не полагаться только на собственные силы. В этой области присутствует достаточно много тонких моментов, в их числе форматы дат, часовые пояса, переход на летнее время, високосные годы, секунды координации и месяцы с разными количествами дней. И все это способно внести искажения в расчеты. Воспользуйтесь появившимися в PHP 5.2.0 классами `DateTime`, `DateInterval` и `DateTimeZone`. Это полезные классы, предоставляющие простой объектно-ориентированный интерфейс для правильного создания и манипулирования датами, временем и часовыми поясами.

Установка часового пояса по умолчанию

Первое, что нужно сделать, – определить часовой пояс по умолчанию для функций даты и времени PHP. Если не установить часовой пояс по умолчанию, PHP выведет сообщение `E_WARNING`. Существует два способа установки часового пояса по умолчанию. Можно определить часовой пояс по умолчанию в файле `php.ini`:

```
date.timezone = 'America/New_York';
```

Или установить его во время выполнения с помощью функции `date_default_timezone_set()` (пример 5.9).

Пример 5.9. Установка часового пояса по умолчанию

```
<?php  
date_default_timezone_set('America/New_York');
```

Любое из решений требует указать идентификатор часового пояса. Полный список идентификаторов часовых поясов можно найти на странице <http://php.net/manual/timezones.php>.

Класс `DateTime`

Класс `DateTime` обеспечивает объектно-ориентированный интерфейс для управления значениями даты и времени. Отдельный экземпляр класса `DateTime` представляет определенную дату и время. Использование конструктора класса `DateTime` (пример 5.10) – самый простой способ создания нового экземпляра класса `DateTime`.

Пример 5.10. Класс DateTime

```
<?php  
$datetime = new DateTime();
```

При вызове без аргументов, конструктор класса `DateTime` создает экземпляр, соответствующий текущей дате и времени. Конструктору класса `DateTime` можно передать строковый аргумент с определенной датой и временем (пример 5.11). Строковый аргумент должен использовать один из допустимых форматов даты и времени, перечисленных на странице <http://php.net/manual/datetime.formats.php>.

Пример 5.11. Вызов конструктора класса DateTime с аргументом

```
<?php  
$datetime = new DateTime('2014-04-27 5:03 AM');
```

В идеальном мире вы могли бы получать данные о дате и времени в понятном для PHP формате. К сожалению, так бывает далеко не всегда. Иногда приходится работать со значениями даты и времени, заданными в разных и часто в самых неожиданных форматах. Я сталкиваюсь с этой проблемой ежедневно. Многие из пользователей моих приложений предоставляют электронные таблицы Excel, содержащие данные для импорта в приложение, и каждая из них определяет значения даты и времени в самых разных форматах. Класс `DateTime` справляется со всем этим без вопросов.

Воспользуйтесь статическим методом `DateTime::createFromFormat()`, чтобы создать экземпляр `DateTime` из строки с датой и временем в произвольном формате. В первом аргументе этому методу передается строка с форматом даты и времени. Во втором – строка с датой и временем, использующая заданный в первом аргументе (пример 5.12).

Пример 5.12. Статический конструктор класса DateTime

```
<?php  
$datetime = DateTime::createFromFormat('M j, Y H:i:s', 'Jan 2, 2014 23:04:12');
```



Статический метод `DateTime::createFromFormat()` принимает те же форматы даты и времени, что и функция `date()`. Допустимые форматы даты и времени можно найти на странице <http://php.net/manual/datetime.createfromformat.php>.

Класс DateInterval

Класс `DateInterval` обычно применяется для манипуляций над экземплярами класса `DateTime`. Экземпляр класса `DateInterval` представляет либо абсолютный период времени (например, «два дня»), либо относительный (например, «вчера»). Экземпляры класса `DateInterval` можно использовать для изменения экземпляров `DateTime`. Например, класс `DateTime` имеет методы `add()` и `sub()` для манипуляции значениями экземпляров `DateTime`. Оба метода принимают экземпляр класса `DateInterval`, определяющий период времени, прибавляемый или вычитаемый из экземпляра класса `DateTime`.

Создается экземпляр класса `DateInterval` с помощью своего конструктора. Конструктор класса `DateInterval` принимает строковый аргумент со *спецификацией периода*. Спецификации периода кажутся на первый взгляд сложными, но их не так уж много. Во-первых, спецификация интервала – это строка, которая начинается с буквы `r`. За ней следует целое число. И, наконец, добавляется *обозначение периода*, которое квалифицирует предыдущее целое значение. Допустимые обозначения периода:

- `y` (годы);
- `m` (месяцы);
- `d` (дни);
- `w` (недели);
- `h` (часы);
- `m` (минуты);
- `s` (секунды).

Спецификация интервала может включать значения даты и времени. Если присутствует значение времени, оно отделяется от значения даты буквой `t`. Например, спецификация интервала `r2D` означает *два дня*. Спецификация интервала `r2DT5H2M` означает *два дня пять часов и две минуты*.

Пример 5.13 демонстрирует изменение экземпляра класса `DateTime` при помощи метода `add()` на заданный период времени.

Пример 5.13. Класс DateInterval

```
<?php
// Создать экземпляр класса DateTime
$datetime = new DateTime('2014-01-01 14:00:00');

// Создать период в две недели
```

```
$interval = new DateInterval('P2W');

// Изменить значение экземпляра класса DateTime
$datetime->add($interval);
echo $datetime->format('Y-m-d H:i:s');
```

Можно также создать экземпляр класса `DateInterval` с отрицательным значением (пример 5.14). Это позволит сместить экземпляр класса `DatePeriod` в обратном направлении!

Пример 5.14. Отрицательный класс DateInterval

```
$dateStart = new \DateTime();
$dateInterval = \DateInterval::createFromString('-1 day');
$datePeriod = new \DatePeriod($dateStart, $dateInterval, 3);
foreach ($datePeriod as $date) {
    echo $date->format('Y-m-d'), PHP_EOL;
}
```

Результат выполнения:

```
2014-12-08
2014-12-07
2014-12-06
2014-12-05
```

Класс `DateTimeZone`

Если ваше приложение обслуживает клиентов по всему свету, вы почти наверняка столкнетесь с часовыми поясами. Работа с часовыми поясами достаточно сложна и многие PHP-разработчики в них путаются.

PHP предлагает для работы с часовыми поясами класс `DateTimeZone`. Чтобы создать экземпляр этого класса, достаточно лишь передать правильный идентификатор часового пояса в конструктор:

```
<?php
$timezone = new DateTimeZone('America/New_York');
```



Полный список допустимых идентификаторов часовых поясов можно найти на странице <http://php.net/manual/timezones.php>.

Обычно экземпляры класса `DateTimeZone` используются при создании экземпляров класса `DateTime`. Во втором необязательном аргу-

менте конструктору класса `DateTime` можно передать экземпляр класса `DateTimeZone`. Значение экземпляра `DateTime` и все его изменения будут привязаны к определенному часовому поясу. Если опустить второй аргумент, часовым поясом будет считаться часовой пояс, установленный по умолчанию:

```
<?php  
$timezone = new DateTimeZone('America/New_York');  
$datetime = new DateTime('2014-08-20', $timezone);
```

После создания экземпляра `DateTime` его часовой пояс можно изменить вызовом метода `setTimezone()` (пример 5.15).

Пример 5.15. Использование класса `DateTimeZone`

```
<?php  
$timezone = new DateTimeZone('America/New_York');  
$datetime = new \DateTime('2014-08-20', $timezone);  
$datetime->setTimezone(new DateTimeZone('Asia/Hong_Kong'));
```

Я считаю, что проще всего всегда работать в часовом поясе UTC. Часовым поясом моего сервера является UTC и часовым поясом для моих программ на PHP по умолчанию также является UTC. При сохранении значения даты и времени в базе данных я также использую часовой пояс UTC. При выводе данных пользователям я преобразовываю значения даты и времени часового пояса UTC в соответствующий часовой пояс.

Класс `DatePeriod`

Иногда требуется получить последовательность значений дат и времени, повторяющихся через определенный интервал времени. Хорошим примером могут послужить повторяющиеся события календаря. Для решения такой задачи используется класс `DatePeriod`, конструктор которого принимает три обязательных аргумента:

- экземпляр класса `DateTime`, представляющий начальные значения даты и времени;
- экземпляр класса `DateInterval`, представляющий интервал времени между двумя следующими друг за другом значениями датами и времени;
- целое число, задающее общее количество значений даты и времени.

Экземпляр класса `DatePeriod` является итератором, и каждая его итерация возвращает экземпляр класса `DateTime`. В примере 5.16 демонстрируется получение трех значений даты и времени, разделенных двухнедельными интервалами.

Пример 5.16. Использование класса `DatePeriod`

```
<?php
$start = new DateTime();
$interval = new DateInterval('P2W');
$period = new DatePeriod($start, $interval, 3);

foreach ($period as $nextDateTime) {
    echo $nextDateTime->format('Y-m-d H:i:s'), PHP_EOL;
}
```

Конструктор класса `DatePeriod` принимает также четвертый необязательный аргумент, который явно определяет дату и время окончания итераций. Если нужно исключить начальную дату из итераций в заданном периоде, передайте константу `DatePeriod::EXCLUDE_START_DATE` в последнем аргументе конструктора (пример 5.17).

Пример 5.17. Использование класса `DatePeriod`

```
<?php
$start = new DateTime();
$interval = new DateInterval('P2W');
$period = new DatePeriod(
    $start,
    $interval,
    3,
    DatePeriod::EXCLUDE_START_DATE
);

foreach ($period as $nextDateTime) {
    echo $nextDateTime->format('Y-m-d H:i:s'), PHP_EOL;
}
```

Компонент *nesbot/carbon*

Если вы часто работаете с датами и временем, воспользуйтесь PHP-компонентом `nesbot/carbon` Брайана Несбитта (Brian Nesbitt). Компонент `nesbot/carbon` предоставляет простой пользовательский интерфейс с множеством полезных методов для работы со значениями даты и времени.

Базы данных

Многие PHP-приложения хранят информацию в различных базах данных, таких как MySQL, PostgreSQL, SQLite, MSSQL или Oracle. Для каждой базы данных имеется свое расширение для поддержки взаимодействий программ на PHP с базой данных. Для связи с MySQL, например, используется расширение `mysql_i`, которое добавляет различные функции `mysql_i_*()` в язык PHP. SQLite3 использует расширение `sqlite3`, которое добавляет классы `sqlite3`, `sqlite3_stmt` и `sqlite3_result`. Если вы работаете с разными базами данных в одном или нескольких проектах, вам придется установить и изучить соответствующие PHP-расширения баз данных и их интерфейсы, что увеличит ваши когнитивные и технические накладные расходы.

Расширение PDO

Вот почему в PHP добавлено встроенное расширение PDO. Расширение PDO (*PHP data objects – объекты данных PHP*) – это набор классов, обеспечивающих взаимодействие с различными базами данных через единый пользовательский интерфейс. Оно обеспечивает абстрагирование от реализации баз данных. С помощью этого расширения можно выполнять запросы к базе данных, используя единый интерфейс, независящий от конкретной базы данных.



При том, что расширение PDO предоставляет единый интерфейс для работы с различными базами данных, мы по-прежнему должны сами писать инструкции на языке SQL. В этом и состоит недостаток PDO. Каждая база данных имеет свои особенности и, как правило, учет этих особенностей приводит к уникальному синтаксису SQL. Я рекомендую вам использовать стандарт ANSI/ISO SQL при работе с PDO, чтобы избежать ошибок при смене систем управления базой данных. Если вы не можете обойтись без специфичных функций конкретной базы данных, имейте в виду, что вам нужно будет изменить SQL-операторы при смене системы управления базой данных.

Подключение базы данных и DSN

Для начала нужно выбрать систему управления базой данных, лучше всего подходящую вашему приложению. Установите базу данных,

создайте схему и, если необходимо, загрузите начальный набор данных. Затем создайте экземпляр класса PDO. Экземпляр PDO установит связь между PHP и базой данных.

Конструктор класса PDO принимает в качестве аргумента строку DSN (Data Source Name – имя источника данных), которая определяет детали соединения с базой данных. Стока DSN начинается с имени драйвера базы данных (например, mysql или sqlite), затем идет двоеточие и остальная часть строки соединения.

Строки соединения DSN различны для каждой базы данных, но они, как правило, включают в себя:

- имя хоста или IP-адрес;
- номер порта;
- имя базы данных;
- кодировку.



Более подробную информацию о формате DSN для вашей базы данных можно найти на странице <http://php.net/manual/pdo.drivers.php>.

Во втором и третьем аргументах конструктору класса PDO передаются имя пользователя базы данных и пароль. Задайте эти аргументы, если база данных требует аутентификации.

В примере 5.18 PDO устанавливает соединение с базой данных MySQL с именем acme. База данных доступна по IP-адресу 127.0.0.1 и она прослушивает стандартный порт MySQL 3306. Имя пользователя базы данных josh и пароль sekrit. Кодировка соединения utf8.

Пример 5.18. Конструктор PDO

```
<?php
try {
    $pdo = new PDO(
        'mysql:host=127.0.0.1;dbname=books;port=3306;charset=utf8',
        'USERNAME',
        'PASSWORD'
    );
} catch (PDOException $e) {
    // Ошибка подключения к базе данных
    echo "Database connection failed";
    exit;
}
```

Первый аргумент конструктора класса PDO – строка DSN. Она начинается с текста `mysql:`, определяющего драйвер PDO MySQL для подключения к базе данных MySQL. За символом двоеточия идет список ключей и значений, разделенных точкой с запятой. Здесь, в частности, задаются такие параметры, как `host`, `dbname`, `port` и `charset`.



Конструктор PDO возбуждает исключение `PDOException`, если подключение к базе данных заканчивается неудачей. Это исключение обязательно должно быть перехвачено и обработано.

Храните учетные данные базы данных в секрете

Пример 5.18 отлично подходит для демонстрационных целей, но он не безопасен. *Никогда не помещайте учетные данные базы данных в PHP-файлы*, особенно в общедоступные PHP-файлы. Если исходный код PHP передается HTTP-клиентам, из-за ошибки или неправильной настройки сервера, учетные данные базы данных могут стать общедоступными. Чтобы избежать этого, размещайте учетные данные базы данных в конфигурационной файле, расположенному в иерархии выше корневого каталога документов, и подключайтесь к вашим PHP-файлам.



Не включайте учетные данные в управление версиями. Защитите учетные данные с помощью файла `.gitignore`. В противном случае, вы опубликуете данные, которые должны содержаться в тайне, в своем репозитории. Это особенно плохо, если вы пользуетесь общедоступным репозиторием.

В следующем примере демонстрируется содержимое файла `settings.php` с учетными данными подключения к базе данных. Он находится в корневом каталоге проекта, но выше корневого каталога документов. Файл `index.php`, доступный клиентам веб-сервера, помещен в корневой каталог документов. Этот файл использует учетные данные из файла `settings.php`:

```
[project_root]
    settings.php
    public_html/ <-- корневой каталог документов
        index.php
```

Содержимое файла *settings.php*:

```
<?php
$settings = [
    'host' => '127.0.0.1',
    'port' => '3306',
    'name' => 'acme',
    'username' => 'USERNAME',
    'password' => 'PASSWORD',
    'charset' => 'utf8'
];

```

В примере 5.19 приводится содержимое файла *index.php*, подключающего файл *settings.php* и устанавливающего соединение с базой данных.

Пример 5.19. Конструктор PDO с внешними настройками

```
<?php
include('../settings.php');
$pdo = new PDO(
    sprintf(
        'mysql:host=%s;dbname=%s;port=%s;charset=%s',
        $settings['host'],
        $settings['name'],
        $settings['port'],
        $settings['charset']
    ),
    $settings['username'],
    $settings['password']
);

```

Этот способ намного безопаснее. Если и произойдет утечка файла *index.php*, учетные данные базы данных останутся неизвестными.

Параметризованные запросы

Теперь у нас есть соединение PDO с базой данных, и мы можем читать и писать данные в базу данных с помощью SQL-операторов. Но это еще не все. При создании PHP-приложений часто требуется динамически вставлять в SQL-операторы сведения из текущих HTTP-запросов. Например, нужно по URL-адресу */user?email=john@example.com* вывести информацию о профиле для конкретной учетной записи пользователя. Для этого URL-адреса SQL-выражение будет примерно таким:

```
SELECT id FROM users WHERE email = "john@example.com";
```

Начинающий PHP-разработчик построит SQL-выражение следующим образом:

```
$sql = sprintf(  
    'SELECT id FROM users WHERE email = "%s",  
    filter_input(INPUT_GET, 'email')  
,
```

Это плохой подход, потому что SQL-выражение включает необработанную строку из HTTP-запроса. Применение такого подхода равносильно расстиланию ковровой дорожки с надписью «Добро пожаловать» перед хакерами, намеревающимися навредить вашему PHP-приложению. Разве вы не слышали о таблицах маленько-го Бобби (*little Bobby Tables*, <http://xkcd.com/327/>)? Чрезвычайно важно санировать ввод пользователя при его использовании в SQL-выражениях. К счастью, расширение PDO делает санирование ввода весьма простой задачей, предлагая поддержку *параметризованных запросов* (*prepared statements*) и *связанных параметров* (*bound parameters*).

Поддержка параметризованных запросов реализована в виде класса `PDOStatement`. Однако, я редко создаю экземпляры `PDOStatement` напрямую. Вместо этого, я получаю объект параметризованного запроса с помощью метода `prepare()` экземпляра PDO. Этот метод принимает строку с SQL-выражением в первом аргументе и возвращает экземпляр `PDOStatement`:

```
<?php  
$sql = 'SELECT id FROM users WHERE email = :email';  
$statement = $pdo->prepare($sql);
```

Обратите внимание на SQL-выражение. Текст `:email` – это *именованный заполнитель* (*named placeholder*) с которым безопасно можно связать любое значение. В примере 5.20 я связал строку из HTTP-запроса с заполнителем `:email`, используя метод `bindValue()` экземпляра `$statement`.

Пример 5.20. Параметризованный запрос с адресом электронной почты

```
<?php  
$sql = 'SELECT id FROM users WHERE email = :email';  
$statement = $pdo->prepare($sql);  
  
$email = filter_input(INPUT_GET, 'email');  
$statement->bindValue(':email', $email);
```

Параметризованный запрос автоматически санирует значение `$email` и защищает базу данных от атак посредством SQL-инъекций. В строку SQL-выражения можно включить несколько именованных заполнителей и вызвать метод `bindValue()` для каждого из них.

В примере 5.20 именованный заполнитель `:email` связывается со строковым значением. А что если мы изменим наше SQL-выражение, чтобы осуществлять поиск пользователей по числовым значениям ID? В этом случае, мы должны будем передать третий аргумент методу `bindValue()` параметризованного запроса, чтобы указать тип данных значения, связываемого с заполнителем. Если третий аргумент опущен, подготовленное выражение считает, что связываемым значением является строка.

Пример 5.21 является модификацией примера 5.20 и выполняет поиска пользователя по числовому идентификатору, а не по адресу электронной почты. Числовое значение ID извлекается из строкового параметра HTTP-запроса с именем `id`.

Пример 5.21. Параметризованный запрос с ID

```
<?php
$sql = 'SELECT email FROM users WHERE id = :id';
$statement = $pdo->prepare($sql);

$userId = filter_input(INPUT_GET, 'id');
$statement->bindValue(':id', $userId, PDO::PARAM_INT);
```

В третьем аргументе здесь передается константа `PDO::PARAM_INT`. Она указывает, что связываемое значение является целым числом. Имеется еще несколько констант PDO, определяющих различные типы данных:

```
PDO::PARAM_BOOL
PDO::PARAM_NULL
PDO::PARAM_INT
PDO::PARAM_STR (подразумевается по умолчанию)
```



Список всех констант PDO можно найти на странице
<http://php.net/manual/pdo.constants.php>.

Результаты запроса

Теперь у нас есть параметризованный запрос, и мы готовы обратиться к базе данных. Метод параметризованного запроса `execute()` выполняет SQL-выражение, содержащее любые связанные данные. Если вы собираетесь выполнить запрос `INSERT`, `UPDATE` или `DELETE`, вызовите метод `execute()` и это все, что вам потребуется. Если же вы собираетесь выполнить запрос `SELECT`, то, наверное, предполагаете получить соответствующие записи из базы данных. Результаты запроса можно получить с помощью методов `fetch()`, `fetchAll()`, `fetchColumn()` и `fetchObject()` параметризованного запроса.

Метод `fetch()` класса `PDOStatement` возвращает следующую запись из результирующего набора. Я использую этот метод для итераций по большим наборам, особенно если весь набор результатов не может целиком поместиться в доступной памяти (пример 5.22).

Пример 5.22. Результаты параметризованного запроса в виде ассоциативного массива

```
<?php
// Подготовка и выполнение SQL-запроса
$sql = 'SELECT id, email FROM users WHERE email = :email';
$statement = $pdo->prepare($sql);
$email = filter_input(INPUT_GET, 'email');
$statement->bindValue(':email', $email, PDO::PARAM_STR);
$statement->execute();

// Обход результатов
while (($result = $statement->fetch(PDO::FETCH_ASSOC)) !== false) {
    echo $result['email'];
}
```

Здесь в первом аргументе методу `fetch()` передается константа `PDO::FETCH_ASSOC`. Этот аргумент определяет, в каком виде методы `fetch()` и `fetchAll()` должны вернуть результаты запроса. Можно использовать любую из следующих констант:

`PDO::FETCH_ASSOC`

Методы `fetch()` и `fetchAll()` вернут ассоциативный массив. Ключами массива являются имена столбцов таблицы базы данных.

`PDO::FETCH_NUM`

Методы `fetch()` и `fetchAll()` вернут массив с числовыми индексами. Ключами массива являются числовые индексы столбцов таблицы базы данных в результате запроса.

PDO::FETCH_BOTH

Методы `fetch()` и `fetchAll()` вернут массив, который имеет как ассоциативные, так и числовые ключи. Является сочетанием констант `PDO::FETCH_ASSOC` и `PDO::FETCH_NUM`.

PDO::FETCH_OBJ

Методы `fetch()` и `fetchAll()` вернут объект, свойствами которого являются имена столбцов таблицы базы данных.



Более подробную информацию о выборке результатов в PDO можно найти на странице <http://php.net/manual/pdostatement.fetch.php>.

При работе с небольшим набором результатов можно получить сразу все результаты, вызвав метод `fetchAll()` параметризованного запроса (пример 5.23). Я рекомендую избегать этого метода, если вы не абсолютно уверены, что все результаты запроса поместятся в доступной памяти.

Пример 5.23. Выборка всех результатов в виде ассоциативного массива

```
<?php
// Подготовка и выполнение SQL-запроса
$sql = 'SELECT id, email FROM users WHERE email = :email';
$stmt = $pdo->prepare($sql);
$email = filter_input(INPUT_GET, 'email');
$stmt->bindValue(':email', $email, PDO::PARAM_STR);
$stmt->execute();

// Обход результатов
$results = $stmt->fetchAll(PDO::FETCH_ASSOC);
foreach ($results as $result) {
    echo $result['email'];
}
```

Если вас интересует лишь один столбец в результатах запроса, можно использовать метод `fetchColumn()`. Этот метод действует подобно методу `fetch()`, но возвращает значение единственного столбца из следующей записи результата запроса (пример 5.24). Единственным аргументом метода `fetchColumn()` является индекс нужного столбца.

Пример 5.24. Выборка одного столбца одной строки за раз

```
<?php
// Подготовка и выполнение SQL-запроса
```

```
$sql = 'SELECT id, email FROM users WHERE email = :email';
$stmt = $pdo->prepare($sql);
$email = filter_input(INPUT_GET, 'email');
$stmt->bindValue(':email', $email, PDO::PARAM_STR);
$stmt->execute();

// Обход результатов
while (($email = $stmt->fetchColumn(1)) !== false) {
    echo $email;
}
```

В примере 5.24 столбец `email` является вторым в списке столбцов в SQL-запросе. Из этого следует, что он будет вторым столбцом и в результатах запроса, поэтому я передаю методу `fetchColumn()` число 1 (нумерация столбцов начинается с нуля).



Порядок столбцов в результатах запроса соответствует порядку столбцов в SQL-запросе.

Также можно воспользоваться методом `fetchObject()` параметризованного запроса для выборки следующей записи из результатов в виде объекта, имена свойств которого совпадают с именами столбцов в SQL-запросе (пример 5.25).

Пример 5.25. Выборка записей в виде объекта

```
<?php
// Подготовка и выполнение SQL-запроса
$sql = 'SELECT id, email FROM users WHERE email = :email';
$stmt = $pdo->prepare($sql);
$email = filter_input(INPUT_GET, 'email');
$stmt->bindValue(':email', $email, PDO::PARAM_STR);
$stmt->execute();

// Обход результатов
while (($result = $stmt->fetchObject()) !== false) {
    echo $result->name;
}
```

Транзакции

Расширение PDO также поддерживает *транзакции*. Транзакция – это набор операторов базы данных, которые выполняются *атомарно*.

но. Другими словами, транзакция представляет совокупность SQL-запросов, которые либо будут все успешно выполнены либо не будет выполнен ни один из них. Атомарность транзакций обеспечивает согласованность данных, надежность и живучесть. Приятным побочным эффектом транзакции является улучшение производительности, потому что очередь из нескольких запросов выполняется одновременно.



Не все базы данных поддерживают транзакции. Ознакомьтесь с документацией вашей базы данных и связанного с ней драйвера PDO для получения дополнительной информации.

С помощью расширения PDO использовать транзакции очень просто. Вы готовите и выполняете SQL-запросы в точности, как в примере 5.25. Есть только одно отличие. Выполняемые SQL-инструкции следует окружить методами `beginTransaction()` и `commit()`. Метод `beginTransaction()` заставляет PDO ставить SQL-запросы в очередь, а не выполнять их немедленно. Метод `commit()` выполняет стоящие в очереди запросы в рамках атомарной транзакции. Если выполнение хотя бы одного запроса в транзакции будет неудачным, ни один из запросов транзакции не будет применен. Запомните, что транзакция обеспечивает выполнение либо всех запросов, либо ни одного.

Атомарность важна там, где первостепенное значение имеет целостность данных. Давайте рассмотрим пример кода, обрабатывающего транзакции банковских счетов. Наш код может пополнять счет. Он также может снимать средства со счета, предполагая, что на нем достаточно средств. Сценарий в примере 5.26 переводит 50 долларов с одного счета на другой. Он не использует транзакцию базы данных.

Пример 5.26. Запрос к базе данных без транзакции

```
<?php
require 'settings.php';

// Соединение PDO
try {
    $pdo = new PDO(
        sprintf(
            'mysql:host=%s;dbname=%s;port=%s;charset=%s',
            ...
```

```

        $settings['host'],
        $settings['name'],
        $settings['port'],
        $settings['charset']
    ),
    $settings['username'],
    $settings['password']
);
} catch (PDOException $e) {
    // Ошибка соединения с базой данных
    echo "Ошибка соединения с базой данных";
    exit;
}

// Выражения
$stmtSubtract = $pdo->prepare('
    UPDATE accounts
    SET amount = amount - :amount
    WHERE name = :name
');
$stmtAdd = $pdo->prepare('
    UPDATE accounts
    SET amount = amount + :amount
    WHERE name = :name
');

// Списание средств со счета 1
$fromAccount = 'Checking';
$withdrawal = 50;
$stmtSubtract->bindParam(':name', $fromAccount);
$stmtSubtract->bindParam(':amount', $withdrawal, PDO::PARAM_INT);
$stmtSubtract->execute();

// Зачисление средств на счет 2
$toAccount = 'Savings';
$deposit = 50;
$stmtAdd->bindParam(':name', $toAccount);
$stmtAdd->bindParam(':amount', $deposit, PDO::PARAM_INT);
$stmtAdd->execute();

```

Выглядит прекрасно, не так ли? Но это совсем не так. Представьте, что сервер вдруг отключится после списания 50 долларов со счета 1 и до того как будет пополнен счет 2. Возможно же отключение электроэнергии, пожар, наводнение или еще какая-то катастрофа, от которой пострадает компания, предоставляющая услуги хостинга. Что произойдет с 50 долларами, снятыми со счета 1? Деньги не будут зачислены на счет 2. Они исчезнут. Мы можем защитить целостность данных с помощью транзакции базы данных (пример 5.27).

Пример 5.27. Запрос к базе данных с транзакцией

```
<?php
require 'settings.php';

// Соединение PDO
try {
    $pdo = new PDO(
        sprintf(
            'mysql:host=%s;dbname=%s;port=%s;charset=%s',
            $settings['host'],
            $settings['name'],
            $settings['port'],
            $settings['charset']
        ),
        $settings['username'],
        $settings['password']
    );
} catch (PDOException $e) {
    // Ошибка соединения с базой данных
    echo "Ошибка соединения с базой данных";
    exit;
}

// Выражение
$stmtSubtract = $pdo->prepare('
    UPDATE accounts
    SET amount = amount - :amount
    WHERE name = :name
');
$stmtAdd = $pdo->prepare('
    UPDATE accounts
    SET amount = amount + :amount
    WHERE name = :name
');

// Начало транзакции
$pdo->beginTransaction();

// Списание средств со счета 1
$fromAccount = 'Checking';
$withdrawal = 50;
$stmtSubtract->bindParam(':name', $fromAccount);
$stmtSubtract->bindParam(':amount', $withdrawal, PDO::PARAM_INT);
$stmtSubtract->execute();

// Зачисление средств на счет 2
$toAccount = 'Savings';
$deposit = 50;
$stmtAdd->bindParam(':name', $toAccount);
```

```
$stmtAdd->bindParam(':amount', $deposit, PDO::PARAM_INT);
$stmtAdd->execute();

// Подтверждение транзакции
$pdo->commit();
```

В примере 5.27 списание и зачисление обернуты единой транзакцией. Это гарантирует, что либо обе операции будут успешно выполнены, либо не будет выполнена ни одна из них. Наши данные останутся согласованными.

Многобайтовые строки

PHP считает, что строки состоят из 8-битных символов, каждый из которых занимает один байт памяти. К сожалению, это наивное предположение, которое перестает быть правильным, как только вы начинаете работать с неанглийскими символами. Вам может понадобиться локализация вашего PHP-приложения для пользователей из других стран. В блог могут добавляться комментарии на испанском, немецком или норвежском языках. Имена пользователей могут содержать символы с умлautами. Я полагаю, вы достаточно часто будете сталкиваться с *многобайтовыми* символами и должны научиться правильно их интерпретировать.

Под *многобайтовым* символом я подразумеваю любой символ, не являющийся одним из 128 символов традиционного набора символов ASCII, например: л, ё, а, ö, å, æ и ø. Существует огромное число других символов. Функции в языке PHP для работы со строками по умолчанию предполагают, что все строки состоят только из 8-битных символов. Если при обработке строк Юникода с многобайтовыми символами воспользоваться встроенными функциями PHP, вы получите неожиданные результаты.



Юникод (Unicode) является международным стандартом, в котором каждому символу из множества языков ставится в соответствие уникальное число. Этот стандарт поддерживается некоммерческой организацией Unicode Consortium (<http://www.unicode.org/>).

Избежать ошибок при работе с многобайтовыми строками поможет расширение `mbstring`. Это расширение предоставляет функции для работы со строками многобайтовых символов, которые заменя-

ют большинство встроенных строковых функций в языке PHP. Например, вместо встроенной функции `strlen()` можно использовать многобайтовый аналог `mb_strlen()`.

По сей день я все еще приучаю себя использовать функции для работы с многобайтовыми строками из расширения `mbstring` вместо встроенных функций. К этому достаточно сложно привыкнуть, но вам необходимо пользоваться только функциями для работы с многобайтовыми строками, если вы работаете со строками Юникода. В противном случае данные в Юникоде будут испорчены.



Я использую строку «Íftérgnátiónaлизætioн» при тестировании моих PHP-приложений с поддержкой многобайтовых символов.

Кодировка символов

Используйте кодировку UTF-8. Если бы этот раздел состоял из единственного совета, это был бы именно он. Кодировку UTF-8 правильно интерпретируют все современные браузеры. Кодировка определяет способ упаковки данных в Юникод для хранения в памяти или обмена по сети между сервером и клиентом. Кодировка UTF-8 лишь одна из многих доступных кодировок, но она является самой популярной и поддерживается всеми современными веббраузерами.



Разъяснения о Юникоде и UTF-8

Том Скотт (Tom Scott) приводит лучшее из всех виденных мной разъяснение о Юникоде и UTF-8 на странице <http://bit.ly/its-unicode>. Джоэл Спольски (Joel Spolsky) также написал хорошее разъяснение, которое можно найти на его сайте <http://bit.ly/jspolsky>.

Работа с кодировками достаточно сложна и смущает многих разработчиков. При работе с многобайтными строками, воспользуйтесь следующими советами:

1. Всегда выясняйте или определяйте кодировку ваших данных.
2. Храните данные в кодировке UTF-8.
3. Отображайте данные в кодировке UTF-8.

Расширение `mbstring` предназначено не только для работы со строками Юникода. Оно также преобразует многобайтовые строки из одной кодировки в другую. Это полезно, например, если нужно экспортировать данные из таблицы Excel со специфичной для Windows кодировкой символов в данные с кодировкой UTF-8. Используйте функции `mb_detect_encoding()` и `mb_convert_encoding()` для преобразования строк Юникода из одной кодировки в другую.

Отображение данных в кодировке UTF-8

При работе с многобайтовыми символами важно уведомить PHP, что вы собираетесь работать с символами в кодировке UTF-8. Проще всего это сделать в файле `php.ini`:

```
default_charset = "UTF-8";
```

Набор символов по умолчанию используется многими функциями PHP, в том числе `htmlentities()`, `html_entity_decode()`, `htmlspecialchars()` и функциями `mbstring`. Это значение также добавляется по умолчанию в заголовок `Content-Type`, если иное явно не задано с помощью функции `header()`, как показано ниже:

```
<?php  
header('Content-Type: application/json; charset=utf-8');
```



Функцию `header()` можно использовать только до того, как будет выполнен любой вывод, возвращаемый сценарием PHP.

Я также рекомендую включить следующий тег `meta` в заголовок HTML-документа:

```
<meta charset="UTF-8"/>
```

Потоки данных

Потоки данных (*streams*), наверное, самая поразительная и редко используемая особенность современного языка PHP. Хотя потоки были введены еще в PHP 4.3.0, многие разработчики до сих пор ничего не знают о них, потому что упоминания о потоках встречаются достаточно редко и они плохо документированы.

«Потоки были введены в PHP 4.3.0, как средство обобщения файлов, сетевых подключений, операций сжатия данных и других операций, которые имеют общий набор функций и порядок применения. Если говорить простым языком, поток – это объект ресурса, поведение которого соответствует потоковому стилю. То есть, он поддерживает операции чтения или записи в линейной последовательности, а также переход `fseek()` к произвольному местоположению внутри потока.»

– Руководство PHP

Нечто труднопроизносимое, верно? Давайте сократим это определение и преобразуем его во что-то более понятное. Поток – это средство передачи данных из пункта отправления в пункт назначения. Вот и все. Пунктами отправления и назначения могут быть файлы, процессы, сетевые подключения, архивы ZIP или TAR, временная память, стандартный ввод или вывод или любой другой ресурс, который может быть обернут потоком (<http://php.net/manual/wrappers.php>).

Читая из файла или записывая в файл, вы уже используете потоки. Читая из `php://stdin` или записывая в `php://stdout`, вы также используете потоки. На потоках основана реализация многих функций ввода/вывода в языке PHP, таких как `file_get_contents()`, `fopen()`, `fgets()` и `fwrite()`. Функции потоков языка PHP предназначены для работы с различными потоковыми ресурсами (в пунктах отправления и назначения) с помощью единого интерфейса.



Я представляю себе потоки как трубы, несущую воду из одного места в другое. Пока вода течет через трубу из пункта отправления в пункт назначения, можно фильтровать ее, преобразовать, добавлять или удалять. (Подсказка: вода служит метафорой данных).

Обертки потоков

Есть разные типы потоковых данных, для каждого из которых требуется уникальный протокол чтения и записи. Мы называем эти протоколы *обертками потоков* (<http://php.net/manual/wrappers.php>). Например, мы можем читать и писать данные в файловой системе, общаться с удаленными веб-серверами посредством протоколов HTTP, HTTPS или SSH (Secure Shell), открывать, читать и писать архивы

ZIP, RAR или PHAR. И все эти способы предполагают осуществление следующего общей последовательности действий:

1. Открыть соединение.
2. Прочитать данные.
3. Записать данные.
4. Закрыть соединение.

Несмотря на общность последовательности действий, сами эти действия, например, чтение и запись в файл или отправка и получение HTTP-сообщения, отличаются друг от друга. Но обертки потоков скрывают эти различия за общим интерфейсом.

Каждый поток имеет *схему* и *цель*. Схема и цель указываются в *идентификаторе* потока, используя знакомый нам формат:

```
<scheme>://<target>
```

Элемент `<scheme>` определяет обертку потока. Элемент `<target>` – источник данных. В примере 5.28 показано создание входящего/исходящего PHP-потока для программного интерфейса сайта Flickr. Здесь использована обертка HTTP-потока.

Пример 5.28. Обертка HTTP-потока для программного интерфейса сайта Flickr

```
<?php
$json = file_get_contents(
    'http://api.flickr.com/services/feeds/photos_public.gne?format=json'
);
```

Пусть вас не вводят в заблуждение сходство аргумента с традиционным адресом веб-сайта. Строковый аргумент функции `file_get_contents()` на самом деле является именно идентификатором потока. Схема `http` указывает на необходимость использования обертки HTTP-потока. Остальная часть аргумента является определением цели потока. Определение цели выглядит как традиционный адрес веб-сайта только потому, что задается обертка HTTP-потока. Такое сходство не характерно для других оберток.



Перечитайте этот пункт несколько раз, пока он не укоренится в вашей памяти. Многие PHP-разработчики не понимают, что традиционный веб-адрес на самом деле является замаскированным идентификатором обертки PHP-потока.

Обертка потока file://

Методы `file_get_contents()`, `fopen()`, `fwrite()` и `fclose()` используются для чтения и записи в файловой системе. Мы редко рассматриваем эти функции как средства для работы с PHP-потоками, потому что обертка `file://` является оберткой по умолчанию. Мы используем PHP-потоки и даже не догадываемся об этом! В примере 5.29 создается входящий/исходящий поток для файла `/etc/hosts` с помощью обертки ПОТОКОВ `file://`.

Пример 5.29. Неявное использование обертки file://

```
<?php
$handle = fopen('/etc/hosts', 'rb');
while (feof($handle) !== true) {
    echo fgets($handle);
}
fclose($handle);
```

Пример 5.30 решает ту же задачу. Но здесь в идентификаторе потока явно указывается обертка `file://`.

Пример 5.30. Явное использование обертки потока file://

```
<?php
$handle = fopen('file:///etc/hosts', 'rb');
while (feof($handle) !== true) {
    echo fgets($handle);
}
fclose($handle);
```

Мы обычно опускаем обертку потока `file://`, потому что она используется по умолчанию.

Обертка потока php://

PHP-разработчики, которые пишут сценарии командной строки, по достоинству оценят обертку потока `php://`. Она служит для связи со стандартным вводом, стандартным выводом и стандартным выводом ошибок. С помощью функций PHP для работы с файлами можно открывать, читать и записывать следующие четыре потока:

```
php://stdin
```

Доступен только для чтения. Получает данные из стандартного устройства ввода. Например, PHP-сценарий может использовать этот поток для получения информации, переданной через командную строку.

`php://stdout`

Позволяет записывать данные в текущий выходной буфер. Доступен только для записи, не может быть прочитан и не поддерживает операцию позиционирования.

`php://memory`

Позволяет читать и писать данные в системную память. Недостатком этого PHP-потока является ограниченность объема памяти. Поэтому вместо него безопаснее использовать поток `php://temp`.

`php://temp`

Действует так же, как `php://memory`, но по исчерпании доступной памяти переключается на запись во временный файл.

Другие обертки потоков

Язык PHP и PHP-расширения поддерживают множество других оберток потоков. Например, существуют обертки для работы с архивами ZIP и TAR, FTP-серверами, библиотеками сжатия данных, программным интерфейсом сайта Amazon и так далее. Популярным заблуждением является мнение, что функции `fopen()`, `fgets()`, `fputs()`, `feof()`, `fclose()` и другие предназначены только для работы с файлами в файловой системе. Это *не так*. Файловые функции работают со *всеми* обертками потоков, которые их поддерживают. Например, функции `fopen()`, `fgets()`, `fputs()`, `feof()` и `fclose()` можно использовать при работе с архивами ZIP, веб-службой Amazon S3 (с помощью специальной обертки S3, <http://bit.ly/streamwrap>) или даже с облачным хранилищем данных Dropbox (с помощью специальной обертки Dropbox, <http://www.dropbox-php.com/>).



Более подробные сведения об обертке потоков `php://` можно найти на странице <http://bit.ly/s-wrapper>.

Пользовательские обертки потоков

Имеется также возможность написать собственную обертку потоков. PHP предоставляет пример класса `streamWrapper`, который демонстрирует, как написать собственную обертку потока, поддерживающую все или только некоторые из функций для работы с файлами.

Более подробную информацию о пользовательских обертка можно найти на страницах:

- <http://php.net/manual/class.streamwrapper.php>
- <http://php.net/manual/stream.streamwrapper.example-1.php>

Контекст потока

Некоторые PHP-потоки принимают необязательный набор параметров, или *контекст потока* для настройки поведения потока. Различные обертки поддерживают разные параметры контекста. Создать контекст потока можно с помощью функции `stream_context_create()`. Возвращаемый ею объект контекста может быть передан в большинство функций для работы с файлами и потоками.

Например, знаете ли вы, что можно отправить HTTP-запрос `POST` с помощью функции `file_get_contents()`? Это можно сделать с помощью объекта контекста потока (пример 5.31).

Пример 5.31. Контекст потока

```
<?php
$requestBody = '{"username":"josh"}';
$context = stream_context_create(array(
    'http' => array(
        'method' => 'POST',
        'header' => "Content-Type: application/json; charset=utf-8;\r\n".
                     "Content-Length: " . mb_strlen($requestBody),
        'content' => $requestBody
    )
));
$response = file_get_contents('https://my-api.com/users', false, $context);
```

Контекст потока является ассоциативным массивом, в котором ключом массива верхнего уровня служит имя обертки потока. Значения массива контекста потока специфичны для каждой обертки. Список допустимых параметров можно найти в документации по соответствующей обертке.

Фильтры потоков

До сих пор мы говорили только об открытии, чтении и записи потоков. Но главное достоинство PHP-потоков состоит в возможности фильтрации, преобразования, добавления или удаления данных при передаче потока. Например, представьте, что можно открыть поток файла с разметкой Markdown и *автоматическое* преобразовать ее в формат HTML в процессе чтения в память.



PHP предоставляет несколько встроенных потоковых фильтров, в том числе `string.rot13`, `string.ToUpper`, `string.ToLower` и `string.strip_tags`. Они не особенно полезны на практике. Используйте вместо них свои фильтры потоков.

Фильтр к существующему потоку применяется с помощью функции `stream_filter_append()`. В примере 5.32 показано использование фильтра `string.ToUpper` для чтения текстового файла и преобразования его содержимого в верхний регистр. Я не рекомендую использовать именно этот конкретный фильтра потоков. Я просто хочу продемонстрировать применение фильтра к потоку.

Пример 5.32. Пример фильтра потоков `string.ToUpper`

```
<?php
$handle = fopen('data.txt', 'rb');
stream_filter_append($handle, 'string.ToUpper');
while(feof($handle) !== true) {
    echo fgets($handle); // <-- Выведет все символы в верхнем регистре
}
fclose($handle);
```

Применить фильтр к потоку можно также с помощью обертки `php://filter`. Это сработает, только если применить фильтр в момент открытия потока. Пример 5.33 демонстрирует выполнение той же задачи, что и в предыдущем примере, за исключением того, что фильтр применяется с помощью `php://filter`.

Пример 5.33. Пример применения фильтра `string.ToUpper` с помощью `php://filter`

```
<?php
$handle = fopen('php://filter/read=string.ToUpper/resource=data.txt', 'rb');
while(feof($handle) !== true) {
    echo fgets($handle); // <-- Выведет все символы в верхнем регистре
}
fclose($handle);
```

Обратите внимание на первый аргумент функции `fopen()`. Это – идентификатор потока с оберткой `php://`. Ниже приводится идентификатор целевого потока:

```
filter/read=<filter_name>/resource=<scheme>://<target>
```

Такой способ может показаться излишне сложным по сравнению с использованием функции `stream_filter_append()`. Однако, некоторые

файловые функции в PHP, такие как `file()` или `fpassthru()` не дают возможности применить фильтры после их вызова. Обертка `php://filter` оказывается единственным способом применить фильтры при работе с этими функциями.

Рассмотрим реальный пример фильтрации потоков. На сайте New Media Campaigns (<http://www.newmediacampaigns.com/>) внутренняя система управления контентом на веб-сервере nginx архивирует журналы доступа и помещает их на сайт rsync.net (<http://rsync.net/>). Мы сохраняем один файл журнала в день, сжимая его с помощью архиватора `bzip2`. Имена файлов с журналами имеют формат `YYYY-MM-DD.log.bz2`. Меня попросили извлечь данные доступа к конкретной области за последние тридцать дней. На первый взгляд кажется, что такое задание означает массу работы, не так ли? Мне нужно вычислить диапазон дат, определить имена журналов, подсоединиться по FTP к сайту rsync.net, загрузить файлы, распаковывать их, прочитать каждый из файлов построчно, извлечь соответствующие строки и записать данные о доступе в назначенное устройство. Хотите, верьте, хотите, нет, но PHP-потоки позволяют сделать все это, написав менее 20-и строк кода (пример 5.34).

Пример 5.34. Перебор сжатых архиватором `bzip` журналов с помощью `DateTime` и потоковых фильтров

```
01 <?php
02 $dateStart = new \DateTime();
03 $dateInterval = \DateInterval::createFromDateString('-1 day');
04 $datePeriod = new \DatePeriod($dateStart, $dateInterval, 30);
05 foreach ($datePeriod as $date) {
06     $file = 'sftp://USER:PASS@rsync.net/' . $date->format('Y-m-d') . '.log.bz2';
07     if (file_exists($file)) {
08         $handle = fopen($file, 'rb');
09         stream_filter_append($handle, 'bzip2.decompress');
10         while (feof($handle) !== true) {
11             $line = fgets($handle);
12             if (strpos($line, 'www.example.com') !== false) {
13                 fwrite(STDOUT, $line);
14             }
15         }
16         fclose($handle);
17     }
18 }
```

В примере 5.34:

- Строки 2–4 создают экземпляр `DatePeriod`, охватывающий последние 30 дней, с помощью отрицательного однодневного интервала.

- Стока 6 подготавливает имя файла журнала с помощью экземпляра `DateTime`, возвращаемого в каждой итерации по экземпляру `DatePeriod`.
- Строки 8–9 открывают поток файла журнала на `gsync.net` с помощью обертки SFTP. Распаковывание файла журнала `bzip2` выполняется «на лету», путем применения потокового фильтра `bzip2.decompress` к потоковому ресурсу файла журнала.
- Строки 10–15 выполняют обход распакованного содержимого файла журнала с помощью стандартных функций.
- Строки 12–14 проверяют каждую строку на наличие искомого домена. Если домен присутствует в строке, она выводится в устройство стандартного вывода.

Потоковый фильтр `bzip2.decompress` позволяет автоматически распаковывать файлы журналов при их чтении. Альтернативное решение – ручная распаковка файлов журналов во временный каталог с помощью `shell_exec()` или `bzdecompress()`, обход распакованных файлов и удаление всех распакованных файлов при завершении работы сценария. Использование PHP-потоков позволяет придать решению простоту и элегантность.



Вот и мы – бригада ведер!

PHP-поток делит данные на последовательность ведер, где каждое ведро содержит фиксированное количество данных (например, 4096 байт). Если использовать нашу метафору трубы, вода переносится из пункта отправления в пункт назначения в отдельных ведрах, которые плывут по трубе и проходят через фильтры потоков. Каждый фильтр принимает и манипулирует одним или несколькими ведрами за раз. Ведро или ведра, полученные с помощью фильтра в любой момент времени, называется бригадой ведер.

Пользовательские фильтры потоков

Имеется также возможность создавать пользовательские фильтры потоков. Основная причина, побуждающая использовать фильтры потоков, фактически заключается в возможности создания пользовательских фильтров. Пользовательские фильтры потоков – это PHP-классы, наследующие встроенный класс `php_user_filter`. Пользовательский класс должен реализовать методы `filter()`, `onCreate()` и

`onClose()`. Кроме того, пользовательский фильтр следует зарегистрировать вызовом функции `stream_filter_register()`.

Давайте создадим пользовательский фильтр потока, исключающий бранные слова из потока при чтении данных в память (пример 5.35). Сначала, нужно создать PHP-класс, наследующий класс `php_user_filter`. Этот класс должен реализовать метод `filter()`, действующий как сито, через которое проходит поток ведер. Он получает бригаду ведер из текущего выше фильтра потока, обрабатывает каждый объект ведра в бригаде и отсылает бригаду ведер вниз по течению к месту назначения потока. Это и будет класс пользовательского потока `DirtyWordsFilter`.



Каждый объект ведра в бригаде имеет два общедоступных свойства: `data` и `dataLen`. Это содержимое ведра и длина содержимого, соответственно.

Пример 5.35. Пользовательский фильтр потока DirtyWordsFilter

```
class DirtyWordsFilter extends php_user_filter
{
    /**
     * @param resource $in          Входящая бригада ведер
     * @param resource $out         Исходящая бригада ведер
     * @param int      $consumed   Количество обработанных байт
     * @param bool     $closing    Последняя бригада ведер в потоке?
     */
    public function filter($in, $out, &$consumed, $closing)
    {
        $words = array('grime', 'dirt', 'grease');
        $wordData = array();
        foreach ($words as $word) {
            $replacement = array_fill(0, mb_strlen($word), '*');
            $wordData[$word] = implode('', $replacement);
        }
        $bad = array_keys($wordData);
        $good = array_values($wordData);

        // Обойти все ведера во входящей бригаде
        while ($bucket = stream_bucket_make_writeable($in)) {
            // Заменить бранные слова в данных ведра
            $bucket->data = str_replace($bad, $good, $bucket->data);

            // Увеличить количество обработанных байт
        }
    }
}
```

```
    $consumed += $bucket->datalen;

    // Послать ведра в исходящую бригаду
    stream_bucket_append($out, $bucket);
}
return PSFS_PASS_ON;
}
```

Метод `filter()` получает, обрабатывает и отсылает дальше ведра с данными из потока. Внутри функции `filter()`, выполняется обход ведер в бригаде `$in` и бранные слова заменяются пометками цензуры. Этот метод возвращает константу `PSFS_PASS_ON`, сообщающую об успехе операции. Метод принимает четыре аргумента:

`$in`

Бригада из одного или нескольких входящих ведер с данными из входящего потока.

`$out`

Бригада из одного или более ведер, продолжающих движение вниз по течению в сторону пункта назначения.

`&$consumed`

Общее количество байтов потока, обработанное пользовательским фильтром

`$closing`

Признак последней бригады ведер.

Мы должны зарегистрировать пользовательский фильтр `DirtyWordsFilter` с помощью функции `stream_filter_register()` (пример 5.36).

Пример 5.36. Регистрация пользовательского фильтра потока `DirtyWordsFilter`

```
<?php
stream_filter_register('dirty_words_filter', 'DirtyWordsFilter');
```

В первом аргументе передается имя, идентифицирующее пользовательский фильтр. Во втором аргументе – имя класса пользовательского фильтра. После регистрации открывается возможность использовать наш пользовательский фильтр потока (пример 5.37).

Пример 5.37. Использование фильтра потока `DirtyWordsFilter`

```
<?php
$handle = fopen('data.txt', 'rb');
```

```
stream_filter_append($handle, 'dirty_words_filter');
while (feof($handle) !== true) {
    echo fgets($handle); // <-- Выведет очищенный текст
}
fclose($handle);
```



Если вы хотите узнать больше о PHP-потоках, посмотрите презентацию Элизабет Смит «Nomad PHP» на странице <http://bit.ly/nomad-php>. Она не бесплатна, но стоит заплаченных за нее денег. Вы также можете почитать о PHP-потоках в электронной документации PHP на странице <http://php.net/manual/ru/book.stream.php>.

Ошибки и исключения

Что-то пошло не так. Это часто случается. Независимо от того, насколько мы внимательны и сколько времени посвящаем работе над проектом, всегда есть вероятность допустить или пропустить ошибку. Случалось ли вам воспользоваться PHP-приложением, отображающим пустую белую страницу? А заходить на PHP-сайты, выбрасывающие трассировку стека? Такие неудачи указывает на ошибку программы или на необработанное исключение.

Ошибки и исключения – замечательные инструменты, позволяющие предвидеть неожиданные ситуации. Они помогают обнаруживать и решать проблемы. Ошибки и исключения похожи. Они сообщают, что что-то пошло не так, предоставляют сообщение с описанием и типом ошибки. Ошибки, однако, являются более древним механизмом. Они имеют процедурное устройство, останавливают выполнение сценария, и, если возможно, делегируют обработку ошибок глобальной функции-обработчику. Некоторые ошибки не допускают восстановления. В настоящее время программисты в основном полагаются на исключения, а не на ошибки, но все равно необходимо обеспечивать защиту и от ошибок, так как многие старые функции PHP (например, `fopen()`) по-прежнему вызывают ошибки, когда что-то идет не так.



В языке PHP имеется возможность обойти ошибки, добавив префикс `@` перед функцией, которая может вызвать ошибку (например, `@fopen()`). Но это антишаблон. Вместо этого я рекомендую изменять код, чтобы избежать подобных ситуаций.

Исключения – это объектно-ориентированное развитие системы обработки ошибок в языке PHP. Они являются экземплярами, которые возбуждаются и перехватываются. Исключения более гибко устроены и обрабатывают проблемы *на месте*, не останавливая выполнения сценария. Исключения являются средствами защиты и наступления. Мы должны предвидеть возможное возбуждение исключений в коде, полученным от сторонних поставщиков, заключив этот код в блок `try () catch {}`. Мы также можем вести наступление, возбудив исключение, при этом делегируя обработку исключения другим разработчикам, когда не знаем, как поступить в данной ситуации.

Исключения

Исключение – это объект класса `Exception`, *возбуждаемый*, когда возникает безвыходная ситуация, на которую невозможно повлиять (например, удаленный программный интерфейс не ответил, запрос к базе данных закончился неудачей или непременное условие не было удовлетворено). Я называю такие ситуации *исключительными ситуациями*. Исключения используются, чтобы принудительно делегировать ответственность за решение возникшей проблемы, а также для защиты от потенциальных проблем.

Создать экземпляр класса `Exception` можно с помощью ключевого слова `new`, как любой другой PHP-объект. Объект `Exception` содержит два основных свойства: сообщение и числовой код. Сообщение описывает, что пошло не так. Числовой код является необязательным и может использоваться для обеспечения контекста данного исключения. При создании экземпляра класса `Exception` нужно указать текст сообщения и, при желании, числовой код:

```
<?php  
$exception = new Exception('Опасность, Уилл Робинсон!', 100);
```

Исследовать содержимое объекта `Exception` можно с помощью общедоступных методов экземпляра `getCode()` и `getMessage()`:

```
<?php  
$code = $exception->getCode(); // 100  
$message = $exception->getMessage(); // 'Опасность...'
```

Возбуждение исключений

Экземпляр исключения можно присвоить переменной, но вообще предполагается, что исключения будут *возбуждаться*. Если вы пишете

те код для использования другими разработчиками, действуйте настурпательно только в исключительных ситуациях, то есть, возбуждайте исключения, только когда ваш код сталкивается с исключительной ситуацией и не может продолжать выполнение в сложившихся условиях. В частности, если авторы PHP-компонентов или фреймворков не могут определить, как обработать исключительную ситуацию, они возбуждают исключение и делегируют ответственность за его обработку разработчикам, использующим их код.

Когда возбуждается исключение, выполнение кода немедленно прекращается и код, следующий за оператором, возбудившим исключение, не выполняется. Для возбуждения исключения используется ключевое слово `throw`, за которым следует экземпляр `Exception`:

```
<?php  
throw new Exception('Что-то пошло не так. Время обедать!');
```

В каждый момент можно возбудить только экземпляр класса `Exception` (или его подкласса). Язык PHP предоставляет следующие встроенные подклассы `Exception`:

- `Exception` (<http://php.net/manual/class.exception.php>)
- `ErrorException` (<http://php.net/manual/class.errorexception.php>)

Стандартная библиотека PHP (Standard PHP Library или SPL, <http://php.net/manual/book.spl.php>) дополняет встроенные исключения языка PHP дополнительными подклассами `Exception`:

- `LogicException`
(<http://php.net/manual/class.logicexception.php>)
 - `BadFunctionCallException`
(<http://php.net/manual/class.badfunctioncallexception.php>)
 - * `BadMethodCallException`
(<http://php.net/manual/class.badmethodcallexception.php>)
 - `DomainException`
(<http://php.net/manual/class.domainexception.php>)
 - `InvalidArgumentException`
(<http://php.net/manual/class.invalidargumentexception.php>)
 - `LengthException`
(<http://php.net/manual/class.lengthexception.php>)

- ◆ `OutOfRangeException`
[\(http://php.net/manual/class.outofrangeexception.php\)](http://php.net/manual/class.outofrangeexception.php)
- `RuntimeException`
[\(http://php.net/manual/class.runtimeexception.php\)](http://php.net/manual/class.runtimeexception.php)
 - `OutOfBoundsException`
[\(http://php.net/manual/class.outofboundsexception.php\)](http://php.net/manual/class.outofboundsexception.php)
 - `OverflowException`
[\(http://php.net/manual/class.overflowexception.php\)](http://php.net/manual/class.overflowexception.php)
 - `RangeException`
[\(http://php.net/manual/class.rangeexception.php\)](http://php.net/manual/class.rangeexception.php)
 - `UnderflowException`
[\(http://php.net/manual/class.underflowexception.php\)](http://php.net/manual/class.underflowexception.php)
 - `UnexpectedValueException`
[\(http://php.net/manual/class.unexpectedvalueexception.php\)](http://php.net/manual/class.unexpectedvalueexception.php)

Каждый подкласс предназначен для определенной ситуации и определяет контекст *причины* исключения. Например, если метод PHP-компоненты ожидает строковый аргумент, содержащий не менее пяти символов, а передается строка длиной в два символа, возбуждается экземпляр `InvalidArgumentException`. Так как язык PHP реализует исключения в виде *классов*, вы можете расширять класс `Exception` и создавать собственные классы исключений, с их собственными свойствами и методами. Какие именно классы исключений понадобятся, решать вам. Выберите или создайте класс исключений, который лучше всего подойдет для ответа на вопрос: «Почему я возбудил это исключение?», – и задокументируйте свой выбор.

Перехват исключений

Возбужденные исключения должны быть *перехвачены* и правильно обработаны. Вы должны играть в защите, используя компоненты и фреймворки, написанные другими разработчиками. Хорошие компоненты и фреймворки снабжены документацией, объясняющей, когда и при каких обстоятельствах они возбуждают исключения. Соответственно вы должны предвидеть, перехватывать и обрабатывать эти исключения. Не перехваченные исключения прерывают выполнение PHP-приложения и, что еще хуже, могут вывести отладочную информацию перед пользователями вашего PHP-приложения. Мы не раз наблюдали такое. Очень важно перехватывать исключения и должным образом обрабатывать их.

Окружите код, который может вызвать исключение, блоком `try/catch`, чтобы перехватить и обработать потенциально возможные исключения. Пример 5.38 демонстрирует ошибку соединения PDO с базой данных, в результате которой возбуждается объект `PDOException`. Исключение перехватывается блоком `catch` и вместо вызывающей ужас трассировки стека выводится понятное сообщение об ошибке.

Пример 5.38. Перехват возбужденного исключения

```
<?php
try {
    $pdo = new PDO('mysql://host=wrong_host;dbname=wrong_name');
} catch (PDOException $e) {
    // Извлечь информацию для журналирования
    $code = $e->getCode();
    $message = $e->getMessage();

    // Вывести сообщение, понятное пользователю
    echo 'Что-то пошло не так. Зайдите попозже, пожалуйста.';
    exit;
}
```

Для перехвата нескольких типов исключений можно использовать несколько блоков `catch`. Это полезно, когда разные типы исключений должны обрабатываться по-разному. Также можно использовать блок `finally`, который *всегда* выполняется после перехвата *любого* исключения (пример 5.39).

Пример 5.39. Перехват нескольких исключений

```
<?php
try {
    throw new Exception('Не исключение PDO');
    $pdo = new PDO('mysql://host=wrong_host;dbname=wrong_name');
} catch (PDOException $e) {
    // Обработка исключения PDO
    echo "Перехвачено исключение PDO";
} catch (Exception $e) {
    // Обработка прочих исключений
    echo "Перехвачено общее исключение ";
} finally {
    // Выполняется всегда
    echo " Выполняется всегда";
}
```

В примере 5.39 первый блок `catch` перехватывает исключения `PDOException`. Все прочие исключения перехватываются вторым блоком `catch`. Для каждого перехваченного исключения выполняется

только один блок `catch`. Если интерпретатор PHP не найдет подходящего блока `catch`, исключение будет всплывать все выше, пока не вызовет прерывание сценария с фатальной ошибкой.

Обработчики исключений

У вас может возникнуть вопрос, а что необходимо сделать, чтобы *перехватить все исключения*? Это хороший вопрос. PHP позволяет зарегистрировать *глобальный обработчик исключений* для перехвата всех не перехваченных ранее исключений. Вы всегда должны устанавливать глобальный обработчик исключений. Обработчик исключений – это последняя страховка, позволяющая вывести соответствующее сообщение об ошибке пользователям PHP-приложения, если вы не смогли перехватить и обработать исключение раньше. В своих PHP-приложениях я использую обработчики исключений для вывода отладочной информации в процессе разработки и понятных сообщений пользователям в процессе эксплуатации.



Я настоятельно рекомендую журналировать исключения в обработчике. Механизм журналирования предупредит вас, когда что-то пойдет не так, и сохранит информацию об исключениях для последующего просмотра.

Обработчиком исключения может служить все то, что *может быть вызвано*. Я предпочитаю анонимные функции, но также можно использовать методы класса. Что бы вы ни выбрали, эта функция или метод должны принимать единственный аргумент класса `Exception`. Регистрирование обработчика исключений выполняется с помощью функции `set_exception_handler()`:

```
<?php
set_exception_handler(function (Exception $e) {
    // Обработка и журналирование исключения
});
```

Иногда может потребоваться заменить существующий обработчик исключений своим собственным. Этикет PHP предполагает восстановление существующего обработчика исключений после выполнения вашего кода. Восстановить предыдущий обработчик исключений можно с помощью функции `restore_exception_handler()` (пример 5.40).

Пример 5.40. Установка глобального обработчика исключений

```
<?php
// Зарегистрировать обработчик исключений
set_exception_handler(function (Exception $e) {
    // Обработка и журналирование исключения
});

// Здесь находится ваш код...

// Восстановить предыдущий обработчик исключений
restore_exception_handler();
```

Ошибки

В PHP, кроме исключений, имеются также функции регистрации ошибок. Это сбивает с толку многих PHP-разработчиков. Интерпретатор PHP может возбуждать различные типы ошибок, в том числе фатальные ошибки, ошибки времени выполнения, ошибки компиляции, ошибки запуска и (редко) возбуждаемые пользователем ошибки. PHP чаще всего возбуждает ошибки, связанные с синтаксическими ошибками или исключениями, которые не были перехвачены.

Разница между ошибками и исключениями не вполне очевидна. Ошибки обычно возникают, когда PHP-сценарий по некой причине не в состоянии продолжить выполнение (например, из-за синтаксической ошибки). Также имеется возможность возбуждать собственные ошибки с помощью функции `trigger_error()` и обрабатывать их в обработчике ошибок, но гораздо удобнее использовать для этого исключения. В отличие от ошибок, исключения могут возбуждаться и перехватываться на любом уровне PHP-приложения. Исключения предоставляют больше информации, чем контекст PHP-ошибок. И есть возможность создавать собственные классы исключений, наследуя базовый класс `Exception`. Исключения и надежный механизм журналирования, такой как Monolog, являются гораздо более универсальным решением, чем использование ошибок PHP. Тем не менее, современные PHP-разработчики должны отслеживать и обрабатывать как PHP-ошибки, так и PHP-исключения.

С помощью функции `error_reporting()` или директивы `error_reporting` в файле `php.ini` можно явно указать интерпретатору PHP, о каких ошибках сообщать, а какие игнорировать. Оба инструмента принимают константы с именами `E_*`, определяющие типы ошибок, о которых должны выдаваться сообщения, а какие должны игнорироваться.



Более подробную информацию об ошибках PHP можно найти на странице <http://php.net/manual/function.error-reporting.php>.

Кроме того можно указать интерпретатору PHP – должен ли он выводить сообщения об ошибках или оставаться немыми как рыба. При разработке, я предпочитаю, чтобы интерпретатор PHP выводил все сообщения об ошибках. При эксплуатации, я настраиваю журналирование большинства сообщений об ошибках, без их отображения. Над чем бы вы ни работали, я рекомендую всегда применять следующие четыре правила:

- не отключайте сообщения об ошибках;
- выводите сообщения об ошибках во время разработки;
- не выводите сообщения об ошибках в процессе эксплуатации;
- журналируйте ошибки и во время разработки, и во время эксплуатации.

Это мои настройки сообщений об ошибках в файле *php.ini* для этапа разработки:

```
; Выводить ошибки  
display_startup_errors = On  
display_errors = On  
  
; Сообщать обо всех ошибках  
error_reporting = -1  
  
; Включить журналирование ошибок  
log_errors = On
```

А это мои настройки сообщений об ошибках в файле *php.ini* для этапа эксплуатации:

```
; НЕ выводить ошибки  
display_startup_errors = Off  
display_errors = Off  
  
; Сообщать обо всех ошибках КРОМЕ маловажных  
error_reporting = E_ALL & ~E_NOTICE  
  
; Включить журналирование ошибок  
log_errors = On
```

Основное различие заключается в настройках вывода ошибок, возникающих в процессе разработки. Я не вывожу ошибки, возникающие в процессе эксплуатации. Однако я журналирую ошибки в обоих случаях. Если возникнет ошибка во время эксплуатации PHP-приложения (у меня такого никогда не происходит ... *закашлялся*), я смогу просмотреть файл журнала и узнать подробности.

Обработчики ошибок

Так же, как при работе с исключениями, можно установить глобальный обработчик ошибок с собственной логикой для перехвата и обработки ошибок. Обработчик ошибок позволит штатно завершить PHP-сценарий в случае ошибки.

Обработчиком ошибок, как и обработчиком исключений, может быть все, что может вызываться (например, функция или метод класса). Внутри обработчика вам нужно не забыть вызвать функцию `die()` или `exit()`. Если не прервать сценарий в обработчике ошибок, он продолжит выполнение с места, где произошла ошибка. Зарегистрировать глобальный обработчик ошибок можно с помощью функции `set_error_handler()`, передав ей функцию или метод для вызова:

```
<?php
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
    // Обработка ошибки
});
```

Обработчику ошибок передается пять аргументов:

`$errno`

Уровень серьезности ошибки (одна из констант `E_*`).

`$errstr`

Сообщение об ошибке.

`$errfile`

Имя файла, в котором произошла ошибка.

`$errline`

Номер строки в файле, где произошла ошибка.

`$errcontext`

Массив, указывающий на активную таблицу символов в момент возникновения ошибки. Этот аргумент является необязательным.

зательным и полезен только для расширенных возможностей отладки. Я обычно игнорирую его.

Имеется одна тонкость, которую обязательно нужно учитывать при использовании обработчика ошибок. Интерпретатор PHP будет передавать обработчику *все* ошибки, даже те, которые отключены в текущих настройках. Обработчик должен проверить код ошибки (первый аргумент) и поступить соответствующим образом. Обработчик ошибок можно настроить так, чтобы он вызывался только для выбранного подмножества типов ошибок, передав во втором аргументе функции `set_error_handler()` битовую маску констант `E_*` (например, `E_ALL | E_STRICT`).

Я предлагаю вам использовать подход, который я и многие другие PHP-разработчики используют в приложениях. Мне нравится преобразовывать ошибки в объекты класса `ErrorException`. Класс `ErrorException` наследует класс `Exception`. Он позволяет преобразовывать ошибки в исключения и направлять их в систему обработки исключений.



Не все ошибки можно преобразовать в исключения!
Нельзя преобразовать ошибки `E_ERROR`, `E_PARSE`,
`E_CORE_ERROR`, `E_CORE_WARNING`, `E_COMPILE_ERROR`,
`E_COMPILE_WARNING` и большинство ошибок `E_STRICT`.

При преобразовании ошибок следует учесть некоторые тонкости и преобразовывать только ошибки, удовлетворяющие параметру `error_reporting` в файле `php.ini`. Следующий пример демонстрирует функцию обработки ошибок, преобразующую ошибки в объекты `ErrorException`:

```
<?php
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
    if (!error_reporting() & $errno) {
        // Ошибка отсутствует в параметре error_reporting,
        // игнорировать ее.
        return;
    }
    .
    throw new \ErrorException($errstr, $errno, 0, $errfile, $errline);
});
```

Функция преобразует подходящие PHP-ошибки в объекты `ErrorException` и возбуждает исключение для его перехвата стандартной

системой обработки исключений. Считается хорошим тоном восстанавливать предыдущий обработчик ошибок (если таковой имеется), после завершения работы вашего кода. Восстановить предыдущий обработчик можно с помощью функции `restore_error_handler()` (пример 5.41).

Пример 5.41. Установка глобального обработчика ошибок

```
<?php
// Зарегистрировать обработчик ошибок
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
    if (!error_reporting() & $errno) {
        // Ошибка отсутствует в параметре error_reporting,
        // игнорировать ее.
        return;
    }
    throw new ErrorException($errstr, $errno, 0, $errfile, $errline);
});

// Здесь идет ваш код...

// Восстановить предыдущий обработчик ошибок
restore_error_handler();
```

Ошибки и исключения в ходе разработки

Мы уже знаем, что во время разработки следует выводить сообщения об ошибках. Но стандартные сообщения выглядят непривлекательно и часто разрывают нормальный вывод, что вносит хаос в отображение. Воспользуйтесь для вывода сообщений об ошибках компонентом Whoops (<https://github.com/filp/whoops>). Компонент Whoops – это современный PHP-компонент, обеспечивающий хорошо проработанную, удобную для чтения страницу диагностики ошибок PHP и исключений. Он создан и поддерживается Филипе Добрейра (Filipe Dobreira, <https://github.com/filp>) и Денисом Соколовым (Denis Sokolov, <https://github.com/denis-sokolov>), его внешний вид можно увидеть на рис. 5.1.

Вне всяких сомнений диагностический экран Whoops значительно лучше стандартного вывода сообщений об ошибках и исключениях интерпретатора PHP.

Кроме того компонент Whoops прост в использовании. Внесите изменения, приведенные ниже, в файл `composer.json` и запустите либо `composer install` либо `composer update`:

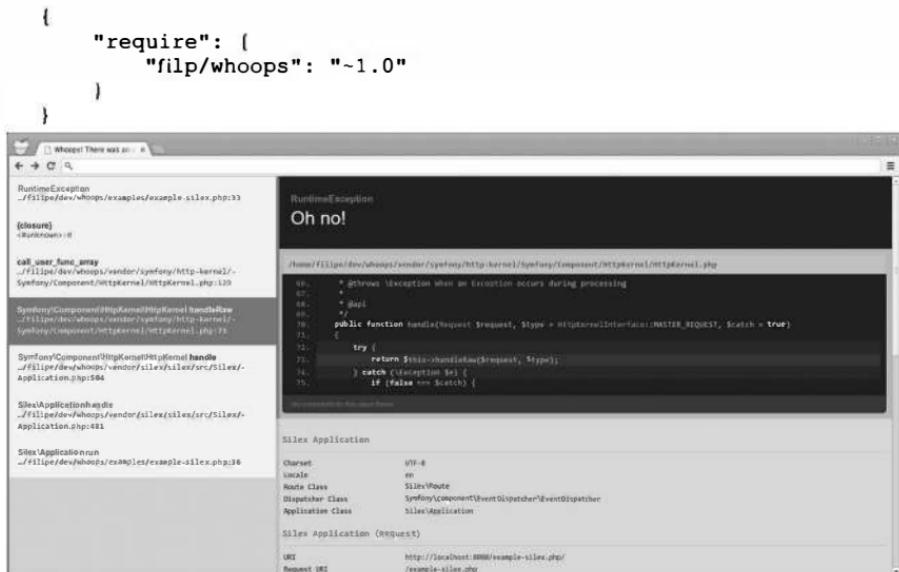


Рис. 5.1. Скриншот Whoops

Затем, зарегистрируйте обработчики ошибок и исключений Whoops в файле начальной загрузки PHP-приложения, как показано в примере 5.42.

Пример 5.42. Регистрация обработчиков Whoops

```

<?php
// Использовать автозагрузчик composer
require 'path/to/vendor/autoload.php';

// Установить обработчики ошибок и исключений Whoops
$whoops = new \Whoops\Run;
$whoops->pushHandler(new \Whoops\Handler\PrettyPageHandler);
$whoops->register();

```

Вот, собственно и все. Теперь, когда сценарий вызовет ошибку или не перехватит исключение, вы увидите диагностический экран Whoops.

В примере 5.42 применен обработчик `PrettyPageHandler` компонента Whoops, который создаст диагностический экран, приведенный на рис. 5.1. В Whoops также имеются другие обработчики, в том числе обработчики осуществляющие вывод в простом текстовом формате, в формате JSON, XML и (если ваш злобный босс не устает постоянно повторять слово *инициативность*) даже в формате SOAP, а также об-

работчик, способный вызывать функции обратного вызова. Я использую компонент Whoops во всех своих разработках.

Эксплуатация

Мы знаем, что во время эксплуатации необходимо журналировать ошибки. В языке PHP имеется функция `error_log()`, предназначенная для записи сообщений в локальный файл, системный журнал `syslog` или отправки их по электронной почте. Но существует более удобный вариант, он называется Monolog (<https://github.com/Seldaek/monolog>). Monolog – это очень удобный PHP-компонент, специализирующийся на журналировании. Он легко интегрируется в PHP-приложения с помощью Composer.

Сначала необходимо добавить пакет `monolog/monolog` в файл `composer.json`:

```
{  
    "require": {  
        "monolog/monolog": "~1.11"  
    }  
}
```

Затем установить компонент командой `composer install` или `composer update`, и добавить код из примера 5.43 в начало файла начальной загрузки PHP-приложения.

Пример 5.43. Использование Monolog для журналирования в процессе разработки

```
<?php  
// Использовать автозагрузчик Composer  
require 'path/to/vendor/autoload.php';  
  
// Импортировать пространства имен Monolog  
use Monolog\Logger;  
use Monolog\Handler\StreamHandler;  
  
// Зарегистрировать Monolog  
$log = new Logger('my-app-name');  
$log->pushHandler(new StreamHandler('path/to/your.log', Logger::WARNING));
```

Вот и все. Теперь у вас есть регистратор Monolog, который будет записывать все сообщения с уровнем серьезности `Logger::WARNING` и выше в файл `path/to/your.log`.

Компонент Monolog легко расширяется. Можно определить несколько обработчиков для журналирования сообщений различных

уровней. Например, можно установить второй обработчик Monolog для отправки по электронной почте администратору сообщений о критических, тревожных или требующих срочного вмешательства ошибках. Для этого понадобится PHP-компонент SwiftMailer, так что давайте добавим его в файл *composer.json* и выполним *composer update*:

```
[  
    "require": {  
        "monolog/monolog": "~1.11",  
        "swiftmailer/swiftmailer": "~5.3"  
    }  
}
```

Затем внесем изменения в код, добавив еще один обработчик Monolog, создающий экземпляр класса *SwiftMailer* для отправки сообщений по электронной почте (пример 5.44).

Пример 5.44. Использование Monolog во время эксплуатации

```
<?php  
// Использовать автозагрузчик Composer  
require 'vendor/autoload.php';  
  
// Импортировать пространства имен Monolog  
use Monolog\Logger;  
use Monolog\Handler\StreamHandler;  
use Monolog\Handler\SwiftMailerHandler;  
  
date_default_timezone_set('America/New_York');  
  
// Установить Monolog и основной обработчик  
$log = new Logger('my-app-name');  
$log->pushHandler(new StreamHandler('logs/production.log',  
Logger::WARNING));  
  
// Добавить обработчик SwiftMailer для критических ошибок  
$transport = \Swift_SmtpTransport::newInstance('smtp.example.com', 587)  
    ->setUsername('USERNAME')  
    ->setPassword('PASSWORD');  
$mailer = \Swift_Mailer::newInstance($transport);  
$message = \Swift_Message::newInstance()  
    ->setSubject('Website error!')  
    ->setFrom(array('daemon@example.com' => 'John Doe'))  
    ->setTo(array('admin@example.com'));  
$log->pushHandler(new SwiftMailerHandler($mailer, $message,  
Logger::CRITICAL));  
  
// Использовать регистратор  
$log->critical('Наш сервер горит!');
```

Теперь, при журналировании сообщений о критических, тревожных или требующих срочного вмешательства ошибках, Monolog будет отсылать электронное письмо с помощью объектов `$mailer` и `$message` компонента SwiftMailer, с текстом сообщения в теле письма.



ЧАСТЬ III

Развертывание, тестирование и настройка

ГЛАВА 6.

Хостинг

Итак, у вас есть PHP-приложение. Мои поздравления! Однако от него не будет никакой пользы, пока ваши пользователи не смогут его *использовать*. Необходимо поместить приложение на сервер и сделать его доступным для целевой аудитории. Вообще говоря, существует четыре способа размещения PHP-приложений: разделяемые (shared) серверы, виртуальные серверы, выделенные серверы и платформы как услуги. Каждый имеет присущие только ему преимущества и подходит для определенного типа приложений и планируемых размеров затрат.

Существует масса компаний, предоставляющих услуги вебхостинга, и это может ошеломить новичка в выборе площадки. Некоторые компании предоставляют только разделяемые серверы. Другие предоставляют на выбор разделяемые, виртуальные и выделенные серверы. В этой главе мы уделим внимание не столько самим компаниям, сколько параметрам предоставляемых ими серверов.

Разделяемые серверы

Разделяемый сервер является наиболее доступным вариантом хостинга со стоимостью от 1\$ до 10\$ в месяц. *Страйтесь не пользоваться услугами разделяемого хостинга.* Эта рекомендация никак не связана с качеством услуг и поддержкой пользователей компаниями, предоставляющими разделяемый хостинг. Есть много хороших компаний, предоставляющих разделяемый хостинг. Проще говоря, дело в параметрах разделяемого хостинга, не слишком благоприятных для разработчиков.

Пользование разделяемым сервером, как следует из его названия, означает использование серверных ресурсов совместно с другими клиентами. Если вы приобрели пакет услуг разделяемого хостинга, вам предоставят хостинг на той же физической машине, что и мно-

жеству других клиентов. Если машина имеет память объемом в 2 Гб, вашему PHP-приложению достанется лишь часть этой памяти, объем которой зависит от общего числа клиентов, зарегистрированных на этой машине. Если приложение хотя бы одного из клиентов выполняет на этом компьютере сценарий, содержащий ошибки, это может негативно сказаться и на работе вашего приложения. Некоторые из компаний, предоставляющие услуги разделяемого хостинга, переоценивают возможности своих серверов, и ваше приложение будет вынуждено постоянно бороться за системные ресурсы на перегруженной машине.

Настройка среды на разделяемом сервере также связана с трудностями. Например, вашему приложению может понадобиться Memcached (<http://memcached.org/>) или Redis (<http://redis.io/>) для быстрого кэширования памяти. Вы захотите установить Elasticsearch (<http://www.elasticsearch.org/>) для поддержки поиска в вашем приложении. К сожалению, для достижения этих целей настроить программное обеспечение разделяемого сервера будет сложно, если вообще возможно. В результате пострадает производительность и функциональность приложения.

Разделяемые серверы редко предоставляют удаленный доступ по SSH. Часто он заменяется лишь неполноценным доступом по (S)FTP. Это ограничение значительно урезает возможности автоматизации развертывания PHP-приложений.

Если бюджет вашего проекта невелик или ваши потребности весьма скромны, вам будет достаточно и разделяемого сервера. Однако если вы разрабатываете коммерческий сайт или PHP-приложение со средней популярностью, лучше использовать виртуальный выделенный сервер, выделенный сервер или PaaS (платформу как услугу).

Виртуальный выделенный сервер

Виртуальный выделенный сервер (Virtual Private Server, VPS), выглядит, ощущает себя и действует как физический сервер. Но это не физический сервер. VPS – это набор системных ресурсов, которые распределены по одной или нескольким физическим машинам. Это не мешает VPS обладать собственной файловой системой, иметь привилегированного пользователя, системные процессы и IP-адрес. Серверу VPS выделяется определенный объем памяти, доля вычисли-

тельной мощности процессора и доля пропускной способности сети. И все это – ваше.

Серверы VPS позволяют использовать больше системных ресурсов, чем разделяемые серверы. Сервер VPS обеспечивает доступ по SSH с правами суперпользователя root. И VPS ничем не ограничивает установку программного обеспечения. Но, большие возможности влекут за собой большую ответственность. Серверы VPS представляют свободный доступ ко всей операционной системе. Вашей заботой становится настройка и защита операционной системы, предназначеннной для вашего PHP-приложения. Серверы VPS идеально подходят большинству PHP-приложений. Они обеспечивают достаточные объемы системных ресурсов (таких как, процессор, память и пространство на диске), а также возможность их увеличения или уменьшения при необходимости. VPS стоит от 10\$ до 100\$ в месяц, в зависимости от объемов системных ресурсов, необходимых PHP-приложению. Если ваше PHP-приложение стало супер-популярным (сотни тысяч посетителей в месяц) и затраты на VPS стали слишком дорогими, имеет смысл задуматься о переходе на выделенный сервер.



Я почти всегда останавливаю свой выбор на серверах VPS из-за наилучшего соотношения цены, возможностей и гибкости. Моя любимая хостинговая компания Linode (<https://www.linode.com/>) предлагает тарифные планы как VPS, так и выделенного хостинга. Linode не является самым дешевым выбором, но мой опыт показывает что, хостинг Linode быстр и стабилен, кроме того Linode предоставляет отличные пособия.

Выделенный сервер

Выделенный сервер – это вмонтированный в стойку компьютер, который хостинговая компания устанавливает, эксплуатирует и поддерживает от вашего имени. Выделенные серверы могут быть скомпонованы, строго следя вашим спецификациям. Выделенные серверы являются реальными машинами, которые необходимо перевозить, устанавливать и обслуживать. Их невозможно установить и настроить так же быстро, как серверы VPS. Следует отметить, что выделенные серверы лучше подходят PHP-приложениям, требующим максимальной производительности.

Выделенные серверы работают так же, как серверы VPS. Вы имеете SSH-доступ с правами root к вновь установленной операционной системе и должны обеспечить ее защиту и настройку для работы вашего PHP-приложения. Преимуществом выделенного сервера является экономическая эффективность. В конце концов, аренда сервера VPS становится слишком дорогим удовольствием, так как вы потребляете все больше и больше системных ресурсов. Инвестировав в собственную инфраструктуру, вы сэкономите деньги.

Выделенный сервер стоит несколько сотен долларов в месяц, цена определяется спецификацией сервера. Он может быть необслуживаемым (вы сами занимаетесь обслуживанием сервера) или обслуживаемым (вы доплачиваете вашей хостинговой компании за обслуживание сервера).

PaaS

Платформы как услуги (Platforms as a Service, PaaS) позволяют быстро запустить в работу PHP-приложение и, в отличие от виртуального выделенного сервера или выделенного сервера, у вас не возникает необходимости настраивать PaaS. Все, что от вас потребуется, – войти в панель управления вашего провайдера PaaS и нажать несколько кнопок. Некоторые провайдеры PaaS поддерживают командную строку или программный интерфейс через HTTP, с помощью которого можно развертывать PHP-приложения и управлять ими. Ниже приводится список популярных провайдеров PaaS для PHP-приложений:

- AppFog (<https://appfog.com/>)
- AWS Elastic Beanstalk (<http://aws.amazon.com/elasticbeanstalk/>)
- Engine Yard (<https://www.engineyard.com/products/cloud>)
- Fortrabbit (<http://fortrabbit.com/>)
- Google App Engine (<http://bit.ly/g-app-engine>)
- Heroku (<https://devcenter.heroku.com/categories/php>)
- Microsoft Azure (<http://www.windowsazure.com/>)
- Pagoda Box (<https://pagodabox.com/>)
- Red Hat OpenShift (<http://openshift.com/>)
- Zend Developer Cloud (<http://bit.ly/z-dev-cloud>)

Цены за PaaS зависят от выбранного провайдера, но схожи с ценами за виртуальные выделенные серверы: от 10\$ до 100\$ в месяц. Вы пла-

тите за системные ресурсы, выделяемые вашему PHP-приложению. Системные ресурсы можно увеличивать и уменьшать при необходимости. Я рекомендую воспользоваться хостингом PaaS разработчикам, которые не желают сами управлять своими серверами.

Выбор тарифного плана хостинга

Выбирайте только то, что вам нужно, и только тогда, когда это вам нужно. Вы всегда сможете произвести масштабирование инфраструктуры хостинга в сторону увеличения или уменьшения. Для небольших PHP-приложений или прототипов больших приложений, поставщики PaaS, такие как Engine Yard или Heroku, является наилучшим и самым быстрым решением. Если вы предпочитаете иметь больший контроль над конфигурацией вашего сервера, выбирайте VPS. Если ваше приложение станет супер-популярным и ваш VPS начнет прогибаться под тяжестью миллионов посетителей (кстати, примите мои поздравления!), остановите выбор на выделенном сервере. Какой бы вариант хостинга вы ни выбрали, обязательно убедитесь, что он предоставляет самую последнюю стабильную версию PHP и все расширения, необходимые вашему PHP-приложению.

ГЛАВА 7.

Комплектование

После выбора хостинга можно заняться настройкой и комплектованием сервера. Буду честен, комплектование сервера – это скорее искусство, чем наука. Как вы будете комплектовать сервер, полностью зависит от потребностей вашего приложения.



Если вы используете PaaS, инфраструктура сервера управляется провайдером PaaS. Все, что вам нужно делать, это следовать инструкциям провайдера, чтобы переместить PHP-приложение на их платформу, и оно готово к работе.

Если вы не используете PaaS, вам придется сами укомплектовать либо сервер VPS, либо выделенный сервер к запуску PHP-приложения. Комплектование сервера не настолько сложный процесс, как может показаться (перестаньте смеяться), но потребует знакомства с командной строкой. Если вы не дружите с командной строкой, лучше воспользуйтесь PaaS от Engine Yard или Heroku.



В этой главе, я предполагаю, что вы знаете, как редактировать текстовый файл с помощью редактора командной строки, такого как nano (<http://www.nano-editor.org/>) или vim (<http://www.vim.org/>) (они встроены в большинство дистрибутивов Linux). В противном случае, вам понадобится альтернативный способ редактирования файлов на сервере.

Я не считаю себя системным администратором. Тем не менее, владение основами системного администрирования является невероятно ценным навыком для разработчиков приложений, который позволит достичь большей гибкости и надежности. В этой главе я поделюсь своими знаниями в области системного администрирования, чтобы

вы чувствовали себя комфортно, открыв терминал для комплектования сервера, на котором будет запущено ваше PHP-приложение. Затем, я приведу несколько дополнительных источников для дальнейшего совершенствования навыков системного администрирования.

Наша цель

Прежде всего необходимо обзавестись виртуальным выделенным сервером или выделенным сервером. Далее, нужно установить веб-сервер для приема HTTP-запросов. И, наконец, установить и настроить группу PHP-процессов для обработки PHP-запросов, эти процессы должны взаимодействовать с веб-сервером.

Несколько лет назад обычно устанавливали веб-сервер Apache и его модуль `mod_php`. Веб-сервер Apache создает уникальный дочерний процесс для обработки *каждого* HTTP-запроса. Модуль `mod_php` встраивает в каждый новый процесс отдельный интерпретатор PHP, даже в те дочерние процессы, которые предназначены для обработки только статических ресурсов, таких как сценарии на JavaScript, изображения или таблицы стилей. Это приводило к избыточным затратам системных ресурсов. В настоящее время все меньше и меньше PHP-разработчиков используют сервер Apache, потому что появились более эффективные решения.

Сегодня все чаще используется веб-сервер nginx (<http://nginx.org/>), который располагается перед коллекцией процессов PHP-FPM (и переправляет PHP-запросы). Его использование я продемонстрирую ниже в этой главе.

Настройка сервера

Сначала настроим виртуальный выделенный сервер (VPS). Я просто обожаю Linode (<http://linode.com/>). Это не самый дешевый провайдер VPS, зато один из самых надежных. Зайдем на сайт Linode (или на сайт провайдера, которому вы отдали предпочтение) и оплатим новый VPS. Ваш провайдер попросит выбрать дистрибутив Linux и пароль root для нового сервера.

Первый вход

Первое, что вы должны сделать, – войти на новый сервер. Давайте сделаем это сейчас. Откройте терминал на локальном компьютере и

выполните команду `ssh`, чтобы зайти на свой сервер. Не забудьте заменить IP-адрес в примере на IP-адрес вашего сервера:

```
ssh root@123.456.78.90
```



Многие провайдеры, предоставляющие VPS, например, Linode (<http://linode.com/>) и Digital Ocean (<https://www.digitalocean.com/>) насчитывают почасовую оплату. Это значит, что вы можете запустить и опробовать сервер VPS, практически ничего не платя.

У вас может быть запрошено подтверждение подлинности нового сервера. Введите `yes` и нажмите `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)? yes
```

Далее будет предложено ввести пароль суперпользователя `root`. Введите пароль и нажмите `Enter`:

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

Вы вошли в свой новый сервер!

Обновление программного обеспечения

Следующее, что необходимо сделать, – обновить программное обеспечение операционной системы с помощью следующих команд.

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

```
# CentOS  
yum update
```

В ответ на эти команды будет выведено много информации о загрузке и установке обновлений программного обеспечения операционной системы. Это очень важный первый шаг, потому что он гарантирует установку последних обновлений и исправлений программного обеспечения операционной системы, связанных с безопасностью.

Непривилегированный пользователь

Ваш новый сервер пока не защищен. Ниже приведено описание нескольких методов улучшения защищенности вашего сервера.

Создайте *непривилегированного* пользователя и в дальнейшем входите на сервер только как непривилегированный пользователь. Привилегированный пользователь *root* имеет неограниченную власть над вашим сервером. Он – Бог. Он может выполнить любую команду без лишних вопросов. *Вы должны, насколько это возможно, усложнить доступ в качестве привилегированного пользователя root к вашему серверу.*

Ubuntu

Создайте нового непривилегированного пользователя с именем *deploy*, выполнив команду из примера 7.1. В ответ на запрос введите пароль пользователя и следуйте выводимым на экран инструкциям.

Пример 7.1. Создание непривилегированного пользователя в Ubuntu

```
adduser deploy
```

Затем включите пользователя *deploy* в группу *sudo* следующей командой:

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

Это придаст пользователю *deploy* привилегии *sudo* (т. е. он сможет выполнять привилегированные задачи с помощью парольной аутентификации).

CentOS

Создайте нового непривилегированного пользователя с именем *deploy*, выполнив следующую команду:

```
adduser deploy
```

Задайте пароль для пользователя *deploy* следующей командой. При запросе введите и подтвердите новый пароль:

```
passwd deploy
```

Далее, включите пользователя *deploy* в группу *wheel* следующей командой:

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

Это придаст пользователю `deploy` привилегии `sudo` (т.е. он сможет выполнять привилегированные задачи с помощью парольной аутентификации).

SSH-аутентификация с помощью парных ключей

С локального компьютера вы сможете войти на свой новый сервер как непривилегированный пользователь `deploy` следующим образом:

```
ssh deploy@123.456.78.90
```

Вам будет предложено ввести пароль пользователя `deploy`, сделав это, вы будете авторизованы на сервере. Процесс входа можно сделать более безопасным, отключив парольную аутентификацию. Парольная аутентификация уязвима для атаки злоумышленников, основанной на полном переборе (`brute-force`), при которой непрерывно снова и снова повторяются попытки угадывания пароля. Вместо нее мы будем использовать для входа на сервер *SSH-аутентификацию с помощью парных ключей*.

Аутентификация с помощью парных ключей – достаточно сложная тема. Если не лезть в дебри, нужно создать пару «ключей» на локальном компьютере. Один из ключей является закрытым (он останется на вашем локальном компьютере), а второй – открытым (он копируется на удаленный сервер). Вместе они называются *парными ключами*, потому что сообщения, зашифрованные с помощью открытого ключа, могут быть расшифрованы только с помощью соответствующего ему закрытого ключа.

При попытке выполнить вход на удаленный компьютер с помощью SSH-аутентификации парными ключами, удаленный компьютер создает случайное сообщение, шифрует его с помощью открытого ключа и отправляет на локальный компьютер. Локальный компьютер расшифровывает сообщение с помощью закрытого ключа и возвращает расшифрованное сообщение на удаленный сервер. Далее удаленный сервер проверяет расшифрованное сообщение и предоставляет доступ к серверу. Это грубейшее упрощение, но оно точно передает смысл происходящего.

Если вход на удаленный сервер осуществляется с нескольких компьютеров, настройка SSH-аутентификации парными ключами может показаться утомительной, так как для этого потребуется создать пары открытых и закрытых ключей для каждого локального компьютера и

скопировать открытые ключи из каждой пары на удаленный сервер. В такой ситуации, вероятно, предпочтительнее продолжать использовать обычную аутентификацию с надежным паролем. Однако, если доступ к удаленному серверу осуществляется только с одного локального компьютера (что обычно и требуется многим разработчикам), имеет смысл использовать SSH-аутентификацию парными ключами. Создать парные ключи SSH-аутентификации на локальном компьютере можно с помощью следующей команды:

```
ssh-keygen
```

Следуйте выводимым на экран инструкциям и вводите необходимые данные в ответ на запросы. Эта команда создаст два файла на локальном компьютере: файл `~/.ssh/id_rsa.pub` (открытый ключ) и файл `~/.ssh/id_rsa` (закрытый ключ). Закрытый ключ должен оставаться на локальном компьютере и держаться в секрете. А открытый ключ следует скопировать на сервер. Скопировать открытый ключ можно с помощью команды безопасного копирования `scp` (`secure copy`):

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

Не забудьте добавить завершающее двоеточие! Эта команда выгрузит открытый ключ в домашний каталог пользователя `deploy` на удаленном сервере. Затем, войдите на удаленный сервер как пользователь `deploy`. После входа проверьте наличие каталога `~/.ssh` существует. Если такого каталога нет – создайте его командой:

```
mkdir ~/.ssh
```

Затем создайте файл `~/.ssh/authorized_keys` командой:

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

Этот файл будет содержать список открытых ключей, позволяющих вход на этот сервер. Выполните следующую команду, чтобы добавить недавно скопированный открытый ключ в файл `~/.ssh/authorized_keys`:

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

И, наконец, измените права доступа к нескольким каталогам и файлам, чтобы только пользователь `deploy` мог получить доступ к своему каталогу `~/.ssh` для чтения своего файла `~/.ssh/authorized_keys`. Назначить эти права можно следующими командами:

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

Мы сделали все что нужно! Теперь вы сможете входить на удаленный сервер с локального компьютера без ввода пароля.



Входить на удаленный сервер без ввода пароля можно только с компьютера, где хранится закрытый ключ!

Отключение парольной аутентификации и запрет входа пользователя root

Давайте сделаем удаленный сервер еще более безопасным. Мы отключим парольную аутентификацию всех пользователей и запретим пользователю `root` вход в систему, на время. Напомню, что пользователь `root` абсолютно свободен в своих действиях и поэтому нам нужно максимально затруднить доступ к нашему серверу с привилегиями этого пользователя.

Войдите на удаленный сервер как пользователь `deploy` и откройте файл `/etc/ssh/sshd_config` в текстовом редакторе. Это – конфигурационный файл сервера SSH. Найдите параметр `PasswordAuthentication` и измените его значение на `no`, при необходимости раскомментируйте этот параметр. Найдите параметр `PermitRootLogin` и измените его значение на `no`, при необходимости раскомментируйте этот параметр. Сохраните изменения и перезагрузите SSH-сервер следующей командой, чтобы применить изменения:

```
# Ubuntu
sudo service ssh restart

# CentOS
sudo systemctl restart sshd.service
```

Вы сделали все необходимое. Вы защитили сервер и теперь можно установить дополнительное программное обеспечение для запуска PHP-приложения. С этого момента все инструкции, должны выполняться от лица непривилегированного пользователя `deploy`.



Обеспечение безопасности сервера является непрерывной задачей, находящейся под постоянным контролем. Я рекомендую, в дополнение к предыдущим инструкциям, установить брандмауэр. Пользователи Ubuntu могут использовать UFW (<https://help.ubuntu.com/community/UFW>), а пользователи CentOS – iptables (<https://wiki.centos.org/HowTos/Network/iptables>).

PHP-FPM

Менеджер процессов PHP-FPM (PHP FastCGI Process Manager, <http://php.net/manual/en/install.fpm.php>) – это программное обеспечение, управляющее пулом связанных с ним PHP-процессов, которые получают и обрабатывают запросы от веб-сервера, например, nginx. Менеджер процессов PHP-FPM создает один мастер-процесс (как правило, запускаемый от имени привилегированного пользователя `root`), который управляет направлением HTTP-запросов одному или более дочерним процессам. Мастер-процесс PHP-FPM также управляет созданием дочерних PHP-процессов (в соответствии с увеличением трафика веб-приложений) и их уничтожением (если они слишком долго существуют или в них больше нет необходимости). Каждый процесс из пула PHP-FPM существует дольше одного HTTP-запроса, он может успеть обработать 10, 50, 100, 500 и более HTTP-запросов.

Установка

Простейший способ установить PHP-FPM основан на использовании встроенного менеджера пакетов операционной системы, этот способ требует выполнения следующих команд.



Подробное руководство по установке PHP-FPM можно найти в Приложении А.

```
# Ubuntu
sudo apt-get install python-software-properties;
sudo add-apt-repository ppa:ondrej/php5-5.6;
sudo apt-get update;
sudo apt-get install php5-fpm php5-cli php5-curl \
```

```
php5-gd php5-json php5-mcrypt php5-mysqlnd;  
  
# CentOS  
sudo rpm -Uvh \  
http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-5.  
noarch.rpm;  
sudo rpm -Uvh \  
http://rpms.famillecollet.com/enterprise/remi-release-7.rpm;  
sudo yum -y --enablerepo=epel,remi,remi-php56 install php-fpm  
php-cli php-gd \  
php-mbstring php-mcrypt php-mysqlnd php-opcache php-pdo php-devel;
```



Если попытка установить EPEL завершится неудачей, откройте веб-браузер и перейдите на страницу http://dl.fedoraproject.org/pub/epel/7/x86_64/e/. Просмотрите информацию об обновленной версии EPEL и ее использовании.

Глобальная конфигурация

В Ubuntu главным конфигурационным файлом PHP-FPM является файл */etc/php5/fpm/php-fpm.conf*. В CentOS – файл */etc/php-fpm.conf*. Откройте этот файл в текстовом редакторе.



Конфигурационные файлы PHP-FPM используют формат INI-файлов. Более подробную информацию о формате INI можно найти в Википедии на странице <https://ru.wikipedia.org/wiki/.ini>.

Ниже описываются наиболее важные *глобальные* параметры PHP-FPM, значения по умолчанию которых я рекомендую изменить. Эти два параметра могут быть закомментированы, при необходимости раскомментируйте их. Они обязывают мастер-процесс PHP FPM перезагружаться, если определенное количество дочерних процессов в течение определенного интервала времени будет завершено с ошибкой. Эти параметры обеспечивают основную подстраховку для процессов PHP-FPM, которая может решить несложные проблемы. Их использование не поможет справиться с более серьезными проблемами, связанными с плохим PHP-кодом.

```
emergency_restart_threshold = 10
```

Максимальное количество дочерних процессов PHP-FPM, которые могут завершаться с ошибкой в заданный интервал

времени, до того, как процесс мастер-FPM PHP будет штатно перезагружен.

```
emergency_restart_interval = 1m
```

Длина интервала времени, которая обуславливает применение параметра `emergency_restart_threshold`.



Более подробную информацию о глобальной конфигурации PHP-FPM можно найти на странице <http://php.net/manual/ru/install.fpm.configuration.php>.

Настройка пулов

В конфигурационном файле PHP-FPM имеется секция `Pool Definitions` с параметрами настройки каждого из пулов PHP-FPM. Пул PHP-FPM – это совокупность дочерних PHP-процессов. Каждому PHP-приложению обычно соответствует свой собственный пул PHP-FPM.

В Ubuntu секция `Pool Definitions` содержит единственную строку:

```
include=/etc/php5/fpm/pool.d/*.conf
```

CentOS подключает определения пулов в начале главного конфигурационного файла PHP-FPM:

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

Эта строка предлагает PHP-FPM загрузить файлы с определениями пулов, находящиеся в каталоге `/etc/php5/fpm/pool.d/` (для Ubuntu) или каталоге `/etc/php-fpm.d/` (для CentOS). Перейдите в этот каталог, где должен находиться единственный файл `www.conf`. Это – файл настройки пула PHP-FPM по умолчанию с именем `www`. Откройте этот файл в текстовом редакторе.



Определение каждого пула PHP-FPM начинается с имени пула, заключенного в квадратные скобки. Конфигурация пула PHP-FPM по умолчанию, например, начинается с `[www]`.

Каждый пул PHP-FPM выполняется с привилегиями пользователя и группы, которые вы указали. Я предпо читаю запускать каждый

пул PHP-FPM от имени непrivилегированного уникального пользователя, для упрощения идентификации процессов PHP-FPM каждого PHP-приложения при просмотре с помощью команд `top` или `ps aux`. Это хорошо еще и потому, что процессы каждого пула PHP-FPM выполняются в своей «песочнице» с разрешениями, соответствующими правам их пользователей и групп операционной системы.

Мы настроим пул PHP-FPM по умолчанию `www` для запуска с привилегиями пользователя `deploy` и его группы. Если вы еще этого не сделали, откройте файл конфигурации пула PHP-FPM `www` в текстовом редакторе. Я рекомендую заменить значения настроек по умолчанию на следующие:

```
user = deploy
```

Пользователь, который является владельцем дочерних процессов пула PHP-FPM. Укажите здесь имя непrivилегированного пользователя для вашего PHP-приложения.

```
group = deploy
```

Группа, владеющая дочерними процессами этого пула PHP-FPM. Укажите здесь имя непrivилегированной группы для вашего PHP-приложения.

```
listen = 127.0.0.1:9000
```

IP-адрес и номер порта, который этот пул PHP-FPM прослушивает и принимает запросы от nginx. Значение `127.0.0.1:9000` указывает, что этот конкретный пул PHP-FPM принимает входящие соединения на локальному порту 9000. Я использую порт 9000, но вы можете использовать любой другой номер непrivилегированного порта (номер порта больший, 1024), который не использует другой системный процесс. Мы вернемся к этой настройке, когда настроим виртуальный хост nginx.

```
listen.allowed_clients = 127.0.0.1
```

IP-адрес или IP-адреса, которые могут отправлять запросы этому пулу PHP-FPM. По соображениям безопасности, я задал `127.0.0.1`. Это значит, что только текущий компьютер может перенаправлять запросы на этот пул PHP-FPM. Этот параметр может быть по умолчанию закомментирован. Раскомментируйте его при необходимости.

```
pm.max_children = 51
```

Это значение определяет общее число процессов пула PHP-FPM, которые могут существовать одновременно. Для этого

параметра не существует единственного правильного значения. Вы должны протестировать PHP-приложение, чтобы опытным путем определить объем памяти, используемой каждым индивидуальным PHP-процессом, и задать значение этого параметра равным общему количеству PHP-процессов, которое позволяет иметь объем памяти вашего компьютера. Большинство малых и средних PHP-приложений обычно используют от 5 до 15 Мб памяти для каждого отдельного PHP-процесса (ваш диапазон может отличаться). Предположим, что у нашего компьютера 512 Мб памяти, доступной для этого пула PHP-FPM, мы можем установить это значение равным 512 Мбайт/10 Мбайт на процесс, или 51 процесс.

```
pm.start_servers = 3
```

Количество, доступных сразу после запуска пула PHP-FPM. Опять же, для этого параметра не существует единственного правильного значения. Для большинства малых и средних PHP-приложений, я рекомендую значение 2 или 3. Это гарантирует то, что начальные HTTP-запросы PHP-приложения не должны ждать инициализации процессов пулем PHP-FPM. Два или три процессы будут уже готовы, и находиться в ожидании.

```
pm.min_spare_servers = 2
```

Наименьшее количество процессов, которые существуют, когда PHP-приложение простояивает. Как правило, это значение устанавливается равным значению параметра `pm.start_servers`, и это гарантирует, что новым HTTP-запросам не придется ждать инициализации новых процессов пулем PHP-FPM.

```
pm.max_spare_servers = 4
```

Наибольшее количество процессов, которые существуют, когда PHP-приложение простояивает. Это значение, как правило, несколько больше, чем значение параметра `pm.start_servers`, и это гарантирует, что новым HTTP-запросам не придется ждать инициализации новых процессов пулем PHP-FPM.

```
pm.max_requests = 1000
```

Максимальное количество HTTP-запросов, которые каждый процесс из пула PHP-FPM обработает до своего уничтожения. Этот параметр помогает избежать накопления утечек памяти, вызванных ошибками в коде PHP-расширений или PHP-

библиотек. Я рекомендую установить это значение равным 1000, но вы должны уточнить его, основываясь на потребностях вашего приложения.

```
slowlog = /path/to/slowlog.log
```

Абсолютный путь к файлу журнала, куда записывается информация о HTTP-запросах, обработка которых занимает больше n секунд. Это полезно для выявления узких мест и отладки приложения. Имейте в виду, привилегии пула PHP-FPM должны давать право на запись в этот файл. Значение `/path/to/slowlog.log` приведено для примера, замените это значение вашим путем к файлу.

```
request_slowlog_timeout = 5s
```

Интервал времени, по истечении которого трассировка текущего HTTP-запроса выводится в файл журнала, указанный в настройке `slowlog`. Значение, которое вы выберете, зависит от того, какой запрос вы посчитаете медленным. Значение `5s` является разумным значением, с которого следует начать.

После редактирования и сохранения файла конфигурации PHP-FPM, перезагрузите мастер-процесс PHP-FPM следующей командой:

```
# Ubuntu  
sudo service php5-fpm restart  
  
# CentOS  
sudo systemctl restart php-fpm.service
```



Более подробную информацию о конфигурации пула PHP-FPM можно найти на странице <http://php.net/manual/install.fpm.configuration.php>.

nginx

nginx (произносится *ин джен екс* – *ин ген икс*) – это веб-сервер, аналог Apache, но гораздо проще в настройке и обычно использует меньше системной памяти. У меня нет возможности подробно описать nginx, но я хочу рассказать об установке nginx на сервере и настройке перенаправления запросов пулу PHP-FPM.

Установка

Проще всего установить nginx с помощью встроенного менеджера пакетов.

Ubuntu

В Ubuntu веб-сервер nginx устанавливается с помощью PPA. Под этим специфичным термином Ubuntu скрывается пакетный архив, поддерживаемый nginx-сообществом:

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

CentOS

В CentOS веб-сервер nginx также устанавливается с помощью дополнительного репозитория программного обеспечения сторонних производителей EPEL, который мы уже использовали ранее. Репозиторий программного обеспечения для CentOS по умолчанию не может содержать последней версии nginx:

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

Виртуальный хост

Далее, настроим *виртуальный хост* nginx для PHP-приложения. Виртуальный хост – это группа настроек, которые определяют доменное имя приложения, местонахождение PHP-приложения в файловой системе и порядок направления HTTP-запросов в пул PHP-FPM.

Прежде всего необходимо решить, где в файловой системе будет находиться приложение. Файлы PHP-приложений должны храниться в каталоге файловой системы, доступном для чтения и записи непривилегированному пользователю deploy. В нашем случае, я помещу файлы приложения в каталог */home/deploy/apps/example.com/current*. Нам также понадобится каталог для хранения файлов журналов приложения. Я помещу эти файлы в каталог */home/deploy/apps/logs*. Следующие команды создадут каталоги и присвоят им нужные разрешения:

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

Поместите PHP-приложение в каталог `/home/deploy/apps/example.com/current`. Конфигурация виртуальных хостов nginx предполагает, что в PHP-приложении имеется каталог `public/` – корневой каталог документов виртуального хоста.

Каждый виртуальный хост nginx имеет свой собственный файл конфигурации. Если вы используете Ubuntu, создайте файл конфигурации `/etc/nginx/sites-available/example.conf`. Если вы используете CentOS, создать файл конфигурации `/etc/nginx/conf.d/example.conf`. Откройте файл конфигурации `example.conf` в текстовом редакторе.

Настройки виртуального хоста nginx находятся в блоке `server {}`. Ниже приводится полный текст из файла конфигурации виртуального хоста:

```
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;
    }
}
```

Скопируйте этот текст и вставьте его в файл конфигурации виртуального хоста `example.conf`. Замените значения параметров `server_name`, `error_log`, `access_log` и `root` соответствующие вашему окружению. Вот краткое объяснение каждого из параметров виртуального хоста:

listen

Номер порта, который сервер nginx будет прослушивать и использовать для приема входящих HTTP-запросов. Обычно для HTTP-трафика используется порт 80 и для HTTPS-трафика – порт 443.

server_name

Доменное имя, идентифицирующее виртуальный хост. Смените это имя на имя домена вашего приложения и обеспечьте его соответствие IP-адресу сервера. Сервер nginx отправляет HTTP-запрос этому виртуальному хосту, если значение заголовка `Host`: совпадает с параметром `server_name` виртуального хоста.

index

Имя файла по умолчанию, используется, если в HTTP-запросе не указан URI-идентификатор ресурса.

client_max_body_size

Максимальный размер тела HTTP-запроса, принимаемого сервером nginx для этого виртуального хоста. Если размер тела запроса превысит это значение, сервер nginx вернет ответ HTTP 4xx.

error_log

Путь к файлу журнала ошибок для этого виртуального хоста.

access_log

Путь к файлу журнала доступа для этого виртуального хоста.

root

Корневой каталог документов.

В файле имеется также два блока `location`. Они определяют, как сервер nginx должен обрабатывать HTTP-запросы, соответствующие заданным URL-шаблонам. Первый блок `location / {}` использует директиву `try_files`, которая выполняет поиск фактических файлов соответствующих URI-идентификатору запроса. Если файл не найден, выполняется поиск в каталоге, который соответствует URI-идентификатору запроса. Если каталог не найден, URI-идентификатор HTTP-запроса заменяется на `/index.php` со строкой запроса, если она доступна. Замененный URL-адрес или любой за-

прос, чей URI-идентификатор заканчивается расширением `.php`, направляется в блок `location ~ \.php {}`.

Блок `location ~ \.php {}` перенаправляет HTTP-запросы в пул PHP-FPM. Помните, что мы создали пул PHP-FPM, принимающий запросы на порту 9000? Данный блок перенаправляет PHP-запросы в порт 9000 и пул PHP-FPM берет их обработку на себя.



Блок `location ~ \.php {}` содержит несколько дополнительных строк. Эти строки служат для предотвращения потенциальных атак удаленного выполнения кода (<http://bit.ly/remote-ex>).

В Ubuntu следует создать символическую ссылку на файл конфигурации виртуального хоста в каталоге `/etc/nginx/sites-enabled/`:

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

И, наконец, перезагрузите сервер nginx:

```
# Ubuntu
sudo service nginx restart

# CentOS
sudo systemctl restart nginx.service
```

Теперь PHP-приложение запущено и работает! Существует много способов настройки сервера nginx. В этой главе я описал только самые необходимые, потому что эта книга о языке PHP, а не о сервере nginx. Вы можете узнать больше о конфигурировании сервера nginx, посетив следующие ресурсы:

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

Автоматизация комплектования

Комплектование сервера является длительным процессом. Это не слишком увлекательный процесс, особенно если вы вручную комплектуете нескольких серверов. К счастью, существуют инструменты, автоматизирующие подготовку серверов. Наиболее популярными из них являются:

- Puppet (<http://puppetlabs.com/>)
- Chef (<https://www.getchef.com/chef/>)
- Ansible (<http://www.ansible.com/home>)
- SaltStack (<http://www.saltstack.com/>)

Инструменты отличаются друг от друга, но все они преследуют одну цель: автоматическое комплектование новых серверов на основе заданных спецификаций. Если вы управляете несколькими серверами, я настоятельно рекомендую научиться пользоваться инструментами комплектования, потому что это сэкономит вам массу времени.

Делегирование комплектования

Существуют, также, онлайн-службы, выполняющие комплектование серверов от вашего имени. Например, служба Forge (<https://forge.laravel.com/>) Тейлора Отуэлла (Taylor Otwell). Я был бета-тестером Forge – это действительно полезная служба. Она может комплектовать сервера Linode, Digital Ocean и других популярных провайдеров VPS.

Каждый сервер, укомплектованный с помощью Forge, автоматически защищается с применением тех же методов обеспечения безопасности, что были описаны выше. Служба Forge автоматически устанавливает связку nginx и PHP-FPM. Forge также упрощает развертывание PHP-приложений, установку SSL-сертификатов, создание заданий CRON и решает другие рутинные задачи системного администрирования. Я настоятельно рекомендую воспользоваться услугами службы Forge, если системное администрирование не является вашим любимым занятием.

Дополнительные материалы

Я считаю системное администрирование увлекательным занятием. Но, хотя мне нравиться возиться с командной строкой, я не хотел бы заниматься этим в течение всего рабочего дня. Самым лучшим учебным пособием по обучению разработчиков системному администрированию, на мой взгляд, является книга «Servers for Hackers» Криса Фидао (Chris Fidao), которую можно найти на сайте <https://book.serversforhackers.com/>.

Что дальше

В этой главе мы рассмотрели подготовку сервера к запуску PHP-приложений. Следующая глава посвящена настройке сервера для оптимизации производительности PHP-приложений.

ГЛАВА 8.

Настройка

К этому моменту, PHP-приложение должно быть запущено и снабжено собственным пулом процессов PHP-FPM сервера nginx. Но это еще не все. Необходимо настроить PHP в соответствии с особенностями вашего приложения и сервера. Конфигурация PHP по умолчанию напоминают стандартный костюм из местного универмага, он подходит всем, но не слишком хорошо сидит на вас. После настройки конфигурация PHP станет больше соответствовать костюму, сшитому на заказ по вашим меркам.

Но не слишком радуйтесь. Настройка PHP не станет лекарством от всех болезней. Плохой код останется плохим кодом. Например, настройка PHP не может решить проблемы, вызванные неоптимальными SQL-запросами или не отвечающим на запросы программным интерфейсом. Тем не менее, настройка PHP является средством повышения эффективности и производительности PHP-приложений, применение которого не потребует особых усилий.

Файл *php.ini*

Интерпретатор PHP конфигурируется и настраивается с помощью файла *php.ini*. Этот файл может находиться в одном из нескольких каталогов вашей операционной системы. Если вы запустили PHP с помощью PHP-FPM, как это описано выше, файл конфигурации *php.ini* можно найти в */etc/php5/fpm/php.ini*. Как это ни странно, *php.ini* не используется интерпретатором PHP, когда вызывается из командной строки, – он использует свой файл *php.ini*, который обычно можно найти в */etc/php5/cli/php.ini*. Если вы собрали PHP из исходных текстов, расположение файла *php.ini*, скорее всего, определяется значением каталога *\$PREFIX*, заданным при конфигурировании сборки. Я предполагаю, что вы запускаете PHP с помощью PHP-FPM, как было описано выше, но все, что будет касаться оптимизации применимо к любому из файлов *php.ini*.



В целях безопасности просканируйте файл `php.ini` с помощью инструмента PHP `Iniscan` (<https://github.com/psecio/iniscan>), написанного Крисом Корнуттом (Chris Cornutt). Файл `php.ini` использует формат INI. Вы можете больше узнать о формате INI из Википедии (<https://ru.wikipedia.org/wiki/.ini>).

Память

Первое очем следует позаботиться при запуске PHP – определить объем памяти для каждого PHP-процесса. Параметр `memory_limit` в файле `php.ini` определяет максимальное количество системной памяти, который сможет использовать PHP-процесс.

Его значение по умолчанию 128м, и оно, обычно, подходит большинству малых и средних PHP-приложений. Тем не менее, если запускается крошечное PHP-приложение, можно сэкономить системные ресурсы, путем снижения этого значения до, например, 64м. Для приложений, интенсивно использующих память (например, сайт на Drupal), можно задать более высокое значение, например, 512м, чтобы повысить его производительность. Выбираемое значение определяется количеством доступной системной памяти. Выяснение объема выделяемой памяти является скорее искусством, чем наукой. Я задаю себе следующие вопросы, перед тем как определить лимит объема памяти для PHP и количество процессов PHP-FPM, которые я могу себе позволить:

Сколько всего памяти я могу выделить PHP?

Сначала я определяю, сколько всего системной памяти я могу выделить PHP. Например, я работаю с виртуальной машиной Linode, имеющей 2 ГБайт общей памяти. Однако, другие процессы (например, nginx, MySQL или memcache) также выполняются на этой машине и потребляют часть общей памяти. Я полагаю, что могу уверенно выделить для PHP 512 МБ памяти.

Сколько памяти в среднем занимает один PHP-процесс?

Далее, определяю, сколько памяти, в среднем, занимает один PHP-процесс. Это требуется для контроля использования памяти процессами. Если вы работаете в командной строке, запустите `top`, чтобы увидеть в реальном времени статистику

выполнения процессов. Также можете вызвать функцию PHP `memory_get_peak_usage()` при завершении PHP-сценария и вывести максимальный объем памяти, использованный текущим сценарием. В любом случае, запустите один и тот же сценарий несколько раз (чтобы «разогреть» кэши) и найдите среднее значение потребленной памяти. Обычно PHP процессы используют 5–20 МБайт памяти (ваш диапазон значений может отличаться). Если сценарий осуществляет выгрузку файлов, обрабатывает графические данные или просто интенсивно использует память, значения, очевидно, будут выше.

Сколько процессов PHP-FPM я могу себе позволить?

Я определил, что могу выделить для PHP 512 МБайт памяти, и каждый PHP-процесс в среднем потребляет около 15 МБайт. Я делю общий объем памяти на количество памяти, потребляемой каждым PHP-процессом, и определяю, что могу себе позволить 34 процесса PHP-FPM. Это значение является оценочным и уточняется экспериментальным путем.

Достаточно ли у меня системных ресурсов?

Наконец, я спрашиваю себя, достаточно ли системных ресурсов, чтобы запустить PHP-приложение и обработать ожидаемый трафик. Если да, это отлично. Если нет, мне нужен сервер с большим объемом памяти, получив его, я вернусь к первому вопросу.



Воспользуйтесь стресс-тестами Apache Bench (<http://bit.ly/apache-bench>) или Siege (<http://www.joedog.org/siege-home/>) для испытания PHP-приложений в схожих с производственными условиями. Если ваше PHP-приложение не имеет достаточных ресурсов, лучше узнать об этом до его ввода в производственную эксплуатацию.

Zend OPcache

Разобравшись с выделением памяти, настроим PHP-расширение Zend OPcache. Оно предназначено для *кэширования байт-кода*. Что такое кэширование байт-кода? Давайте сначала рассмотрим, как типичный PHP-сценарий обрабатывает каждый HTTP-запрос. Во-первых, nginx

направляет HTTP-запрос PHP-FPM и PHP-FPM перенаправляет запрос одному из дочерних процессов PHP. Процесс PHP находит соответствующий PHP-сценарий, читает его в память, компилирует в байт-код и выполняет скомпилированный байт-код, чтобы получить HTTP-ответ. HTTP-ответ возвращается в nginx и nginx отсылает его HTTP-клиенту. При обработке каждого HTTP-запроса выполняется масса лишних операций.

Этот процесс можно ускорить путем кэширования скомпилированного байт-кода. После этого можно читать и выполнять скомпилированный байт-код из кэша, а не искать сценарий на диске, читать его в память и компилировать при каждом HTTP-запросе. Расширение Zend OPcache встроено в PHP, начиная с версии 5.5.0. Ниже приведены мои параметры из *php.ini* для настройки и оптимизации расширения Zend OPcache:

```
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
```

Объем памяти (в мегабайтах), выделенный под кэш байт-кода. Он должен быть достаточно большим, чтобы обеспечить хранение скомпилированного байт-кода всех PHP-сценариев приложения. Если PHP-приложение включает лишь несколько сценариев, его значение может быть небольшим, например 16 МБайт. Если же PHP-приложение включает большое число сценариев, используйте большее значение, например 64 МБайт.

```
opcache.interned_strings_buffer = 16
```

Объем памяти (в мегабайтах), используемой для хранения интернированных строк. Что, черт возьми, такое, эти интернированные строки? Это было и моим первым вопросом. Если интерпретатор PHP обнаруживает несколько экземпляров идентичных строк, он сохраняет в памяти только одну строку и использует указатель на нее, при повторном использовании строки. При этом экономится память. По умолчанию, интерпретатор PHP разделяет интернированные строки разных

PHP-процессов. Этот параметр позволяет всем процессам из одного пула PHP-FPM хранить свои интернированные строки в общем буфере, поэтому на одну и ту же интернированную строку могут ссылаться несколько процессов из пула PHP-FPM. Это ведет к еще большей экономии памяти. Значение по умолчанию для этого параметра 4 МБайт, но я предпочитаю увеличивать его до 16 МБайт.

```
opcache.max_accelerated_files = 4000
```

Максимальное количество PHP-сценариев, которые могут быть сохранены в кэше байт-кода. Можно использовать любое число между 200 и 100000. Я использовал 4000. Это число должно быть больше числа файлов в PHP-приложении.

```
opcache.validate_timestamps = 1
```

Когда этот параметр включен, интерпретатор PHP проверяет PHP-сценарии на наличие изменений в них через интервал времени, указанный в параметре `opcache.revalidate_freq`. Если этот параметр сброшен, PHP не проверяет PHP-сценарии на наличие изменений, и для применения изменений следует очистить кэш байт-кода вручную. Я рекомендую включать этот параметр во время разработки и отключать во время эксплуатации.

```
opcache.revalidate_freq = 0
```

Как часто (в секундах) интерпретатор PHP проверяет скомпилированные PHP-файлы на наличие изменений. Преимущество кэширования заключается в том, что позволяет избегать повторной компиляции PHP-сценариев при каждом запросе. Этот параметр определяет, как долго кэш байт-кода считается свежим. По истечении этого интервала времени PHP проверит PHP-сценарии на наличие изменений и при их обнаружении – скомпилирует и обновит байт-код в кэше. Я использую интервал в 0 секунд. Такое значение указывает, что PHP-файлы должны проверяться при каждом запросе, если и только если не был выключен параметр `opcache.validate_timestamps`. Это означает, что PHP будет проверять файлы перед обработкой каждого запроса (что очень удобно). Полезность этого параметра на этапе эксплуатации является спорной, так как параметр `opcache.validate_timestamps` в любом случае отключен.

```
opcache.fast_shutdown = 1
```

Этот параметр указывает расширению Zend OPcache использовать более быструю последовательность переключения путем делегирования разборки объектов и высвобождения памяти диспетчеру памяти Zend Engine. Документация к этому параметру отсутствует, поэтому *просто включите его*.

Выгрузка файлов

Использует ли PHP-приложение выгрузку файлов? Если нет, отключите возможность выгрузки для повышения безопасности приложения. Если приложение использует выгрузку файлов, лучше установить максимальный размер выгружаемых файлов, который будет поддерживать ваше приложение. Неплохо, также установить максимальное количество одновременных операций выгрузки, поддерживаемых приложением. Следующие настройки *php.ini* я использую для моих приложений:

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

По умолчанию, PHP позволяет выгружать до 20 файлов в одном запросе. Каждый выгружаемый файл должен иметь размер не более 2 МБайт. Вам, наверняка, не понадобятся 20 одновременных операций выгрузки, я обычно разрешаю не более трех в одном запросе, но вы можете изменить этот параметр на значение, которое лучше подходит вашему приложению.



Если вы собираетесь поддерживать выгрузку очень больших файлов, убедитесь, что веб-сервер сконфигурирован должным образом. Возможно, потребуется изменить параметр *client_max_body_size* в конфигурации виртуального хоста сервера nginx, в дополнение к изменениям в файле *php.ini*.

Если мои PHP-приложения используют выгрузку файлов, им обычно требуется выгружать файлы гораздо большего размера, чем 2 МБайт. Я устанавливаю параметр *upload_max_filesize* равным 10M или выше, в зависимости от требований конкретного приложения. Не устанавливайте это значение слишком большим, иначе веб-сервер

(например, nginx) может выразить недовольство HTTP-запросом, имеющим слишком большое тело или превысившим время, отводимое на обработку.

Максимальное время выполнения

Параметр `max_execution_time` в файле `php.ini` определяет максимальную продолжительность времени выполнения PHP-процесса до его завершения. По умолчанию, это продолжительность составляет 30 секунд. Но 30 секунд – это очень много. Мы все хотим, чтобы наши приложения были супер-быстрыми (выполнялись в течение миллисекунд). Я рекомендую изменить значение, уменьшив до 5 секунд:

```
max_execution_time = 5
```



Изменить этот параметр можно в любом сценарии, вызвав функцию `set_time_limit()`.

Вы спросите: «А что делать, если PHP-сценарий должен выполняться дольше?». Он не должен этого делать. Чем дольше выполняется сценарий, тем дольше посетителям веб-приложения придется ждать ответа. Если вам необходимы действия, выполнение которых занимает длительное время (например, изменение размера изображения или создание отчетов), выделите эти действия в отдельный рабочий процесс.



Я использую функцию PHP `exec()` для вызова bash-команды `at`. Это позволяет выполнять длительные операции в отдельных процессах, сделать их неблокирующими, чтобы они не задерживали текущий PHP-процесс. Если вы используете функцию PHP `exec()`, позаботьтесь об экранировании аргументов среды с помощью функции PHP `escapeshellarg` (<http://php.net/manual/function.escapeshellarg.php>).

Предположим, что нужно запустить отчет и вывести результаты в PDF файл. Выполнение этой задачи может занять 10 минут. Конечно,

весьма нежелательно ждать завершения выполнения PHP-запроса в течение 10 минут. Вместо этого, мы создадим отдельный PHP-файл *create-report.php*, который попытит 10 минут и, в конце концов, сгенерирует нам отчет. Однако наше веб-приложение затратит миллисекунды на запуск отдельного фонового процесса и вернет клиенту, например, такой HTTP-ответ:

```
<?php  
exec('echo "create-report.php" | at now');  
echo 'Отчет на рассмотрении...';
```

Автономный сценарий *create-report.php* запускается в фоновом режиме и может по завершении внести изменения в базу данных или отправить отчет получателю по электронной почте. Нет абсолютно никаких причин ждать завершения длительных заданий в главном PHP-сценарии, сдерживая работу пользователя.



Если количество фоновых процессов станет значительным, вам лучше воспользоваться выделенной очередью заданий. PHP Resque (<https://github.com/chrisboulton/php-resque>) – отличный менеджер очереди заданий, основанный на оригинальном менеджере очереди заданий Resque (<https://github.com/blog/542-introducing-resque>) из GitHub.

Обслуживание сеансов

Обработчик сеансов по умолчанию может замедлить выполнение больших приложений, потому что хранит данные сеансов на диске. Это приводит к выполнению ненужных файловых операций чтения и записи, которые занимают достаточно много времени. Вместо этого, разгрузите обработку сеансов, воспользовавшись хранилищами данных в памяти, такими как Memcached (<http://memcached.org/>) или Redis (<http://redis.io/>). Это дает дополнительное преимущество в виде облегчения масштабируемости. Если данные сеансов хранятся на диске, это делает невозможным горизонтальное масштабирование PHP-приложения. Если же данные сеансов хранятся в центральном хранилище данных Memcached или Redis, они становятся доступными из любого количества распределенных PHP-FPM серверов.

Установите расширение PECL Memcached (<http://pecl.php.net/package/memcached>) для доступа к хранилищу данных Memcached из

PHP. Теперь можно сменить место хранения сеансов по умолчанию на Memcached, добавив следующие строки в файл *php.ini*:

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

Буферизация вывода

Передача по сети большего количества данных в меньшем количестве пакетов выполняется быстрее, чем передача меньшего количества данных в большем количестве пакетов. Другими словами, отправляйте контент пользователям меньшим количеством пакетов, чтобы уменьшить общее число HTTP-запросов.

Для этого следует включить буферизацию вывода. По умолчанию буферизация вывода в PHP включена (кроме запуска из командной строки). Выходной буфер накапливает до 4096 байт перед отправкой содержимого обратно веб-серверу. Ниже приводятся рекомендуемые мной параметры *php.ini*:

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



Если вы решите изменить размер буфера вывода, убедитесь, что его размер кратен 4 (для 32-битных систем) или 8 (для 64-битных систем).

Кэш Realpath

PHP поддерживает кэш путей файлов, используемых в PHP-приложении, который нужен, чтобы не приходилось постоянно искать путь подключаемого файла при каждом включении файла. Этот кэш называется *кэшем Realpath*. Для больших PHP-приложений, использующих множество отдельных файлов (Drupal, компоненты Composer и т. д.), можно улучшить производительность путем увеличения размера кэша Realpath.

Размер кэша Realpath по умолчанию равен 16к. Точное определение требуемого размера не является тривиальной задачей, но можно воспользоваться следующим трюком. Сначала, увеличьте размер кэша Realpath до какой-то невероятно большой величины, например

256к. Затем выведите фактический размер кэша Realpath, поместив перед выходом из PHP-сценария оператор `print_r(realpath_cache_size());`. Установите размер кэша Realpath равным этому фактическому значению. Установить размер кэша Realpath можно в файле `php.ini`:

```
realpath_cache_size = 64k
```

Что дальше

Наш сервер работает на полную мощность, и мы готовы приступить к развертыванию PHP-приложения для промышленной эксплуатации. В следующей главе мы обсудим несколько способов автоматизации развертывания PHP-приложений.

ГЛАВА 9.

Развертывание

У нас есть готовый сервер с работающим на нем веб-сервером nginx и PHP-FPM. Теперь нужно развернуть PHP-приложение на сервере. Есть много способов ввода приложений в эксплуатацию. FTP стал популярным способом развертывания PHP-кода много лет назад. FTP по-прежнему работает, но в последние времена появились более безопасные и более предсказуемые способы развертывания. В этой главе описано, как просто, предсказуемо и обратимо развертывать приложения с помощью современных инструментов автоматизации развертывания.

Управление версиями

Я полагаю, что вы уже используете системы управления версиями кода, не так ли? Если это так, отлично. Если нет, оставьте на время то, что вы сейчас делаете, и займитесь установкой такой системы. Я предполагаю управлять версиями моего кода с помощью Git (<http://git-scm.com/>), но существует и другое программное обеспечение для управления версиями кода, например, Mercurial (<http://mercurial.selenic.com/>). Я пользуюсь программным обеспечением Git, потому что знаю его и потому что оно беспроблемно работает с популярными интернет-репозиториями, такими как Bitbucket (<https://bitbucket.org/>) и GitHub (<https://github.com/>).

Система управления версиями – бесценный инструмент для разработчиков PHP-приложений, потому что позволяет отслеживать изменения в коде. Мы можем отмечать моменты времени в качестве релизов, возвращаться к предыдущим состояниям и экспериментировать с новыми функциями в отдельных областях, не оказывая влияния на находящийся в эксплуатации код. И, что более важно, система управления версиями помогает автоматизировать развертывание PHP-приложений.

Автоматизация развертывания

Необходимо так автоматизировать развертывание приложений, чтобы этот процесс стал простым, предсказуемым и обратимым. Вас не должна беспокоить сложность процесса развертывания. Сложность отпугивает, а тем, что пугает, редко пользуются.

Сделайте развертывание простым

Сведем процесс развертывания к выполнению простой команды. Простой процесс развертывания перестает пугать, а значит, вы, вероятнее, всего выберите его для ввода кода в эксплуатацию.

Сделайте развертывание предсказуемым

Сделайте процесс развертывания предсказуемым. Предсказуемый процесс менее страшен, потому что точно известно, что он делает. Он не должен приводить к неожиданным побочным эффектам. Если при его выполнении происходит ошибка, процесс развертывания прерывается, оставляя всю имеющуюся кодовую базу без изменений.

Сделайте развертывание обратимым

Сделайте процесс развертывания обратимым. Если вы случайно внесете ошибку в код, должна существовать простая односторонняя команда, которая вернет код в предыдущее стабильное состояние. Это ваша страховочная сетка. Обратимость процесса развертывания снимает нервное напряжение при процессе ввода кода в эксплуатацию. Если вы что-то испортили, можно просто откатиться к предыдущей версии.

Capistrano

Capistrano (<http://capistranorb.com/>) – это программное обеспечение для автоматизированного развертывания приложений, превращающее эту процедуру в простой, предсказуемый и обратимый процесс. Capistrano выполняется на локальном компьютере и общается с удаленным сервером посредством SSH. Capistrano изначально предназначался для развертывания приложений на языке Ruby, но с его помощью можно разворачивать приложения, написанные на любом языке программирования, включая PHP.

Как это работает

Вы устанавливаете Capistrano на локальной рабочей станции. Capistrano развертывает ваши PHP-приложения на удаленном сервере, посыпая команды SSH с локальной рабочей станции на удаленный сервер. Capistrano организует развертывание приложений в каталоги на удаленном сервере. Capistrano поддерживает пять и более каталогов развертывания приложения, чтобы дать возможность вернуться к одной из более ранних версий. Кроме того Capistrano создает текущий каталог `current/`, который является символьной ссылкой на актуальное развернутое приложение. Структура каталога, контролируемого Capistrano на сервере, будет выглядеть примерно так, как это показано в примере 9.1.

Пример 9.1. Пример структуры каталогов

```
/  
home/  
    deploy/  
        apps/  
            my_app/  
                current/  
                releases/  
                    release1/  
                    release2/  
                    release3/  
                    release4/  
                    release5/
```

При развертывании новой версии приложения, Capistrano сначала получит последнюю версию из хранилища на Git. Затем поместит код приложения в новый каталог. И, наконец, изменит символьскую ссылку `current/`, связав ее с каталогом последнего релиза. Если потребуется вернуться к предыдущей версии, Capistrano переназначит символьскую ссылку `current/`, связав ее с каталогом предыдущего релиза. Capistrano является элегантным и простым решением для развертывания, он делает развертывание PHP-приложений простым, предсказуемым и обратимым.

Установка

Устанавливать Capistrano нужно на локальном компьютере. *Не устанавливайте Capistrano на удаленные серверы.* Вам понадобятся также `ruby` и `gem`. У пользователей OS X они уже имеются. Пользователи Linux могут установить `ruby` и `gem` с помощью соответствую-

ших менеджеров пакетов. После установки `ruby` и `gem`, установите Capistrano командой:

```
gem install capistrano
```

Настройка

После установки Capistrano нужно инициализировать проект для Capistrano. Откройте терминал, перейдите в корневой каталог проекта, и выполните следующую команду:

```
cap install
```

Эта команда создаст файл с именем *Capfile*, каталог с именем *config/* и еще один каталог с именем *lib/*. Теперь корневой каталог проекта должен содержать следующие файлы и каталоги:

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

Файл *Capfile* является главным конфигурационным файлом Capistrano и объединяет конфигурационные файлы, расположенные в каталоге *config/*. Каталог *config/* содержит файлы с настройками для каждой из сред удаленного сервера (например, среды для тестирования, промежуточной среды или среды для промышленной эксплуатации).



Конфигурационные файлы Capistrano написаны на языке Ruby. Тем не менее, в них легко разобраться и при необходимости внести изменения.

По умолчанию, Capistrano предполагает наличие у приложения нескольких сред. Например, можно иметь отдельные среды для разработки и эксплуатации. Capistrano предоставляет отдельный файл конфигурации для каждой из сред в каталоге *config/deploy/*. Capistrano также поддерживает конфигурационный файл *config/deploy.rb*, который содержит общие для всех сред параметры.

Для каждой из сред поддерживается понятие *ролей*. Например, в среде эксплуатации может иметься фронтальный веб-сервер (роль *web*), сервер приложений (роль *app*) и сервер баз данных (роль *db*). Только самые крупные приложения требуют такой архитектуры. Небольшие PHP-приложения обычно используют только одну машину, которая запускает веб-сервер (*nginx*), сервер приложений (PHP-FPM) и сервер базы данных (*MariaDB*).

В этом примере я покажу только роль *web*, и не буду касаться ролей *app* и *db*. Роли Capistrano позволяют организовать выполнение задач только на серверах, которые соответствуют нужной роли. Сейчас нас это не должно особенно волновать. Тем не менее, я намерен следовать концепции сред серверов Capistrano. В этом примере используется среда *production*, но описываемые шаги будут в равной степени применимы к любым другим средам (например, *staging* или *testing*).

Файл config/deploy.rb

Рассмотрим файл *config/deploy.rb*. Этот конфигурационный файл содержит общие для всех сред параметры (например, для сред *staging* и *production*). Большинство интересующих нас настроек Capistrano находится в этом файле. Откройте файл *config/deploy.rb* в текстовом редакторе и измените следующие параметры:

:application

Имя PHP-приложения. Должен содержать только буквы, цифры и подчёркивания.

:repo_url

URL-адрес хранилища Git. Должен указывать на хранилище Git и хранилище должно быть доступно с удаленного сервера.

:deploy_to

Абсолютный путь к каталогу развертывания PHP-приложения на удаленном сервере. В данном случае это будет путь */home/deploy/apps/my_app*, как показано в примере 9.1.

:keep_releases

Количество хранимых предыдущих релизов, чтобы иметь возможность вернуться к одному из них.

Файл config/deploy/production.rb

Этот файл содержит настройки только для среды *production*. Он определяет роли для этой среды и в нем перечислены серверы, от-

носящиеся к каждой роли. Мы используем только роль `web` и у нас есть только один сервер, выполняющий эту роль. Давайте используем сервер, которым мы обзавелись в главе 7. Заменим все содержимое конфигурационного файла `config/deploy/production.rb` следующей строкой (не забудьте изменить IP-адрес):

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

Аутентификация

Прежде, чем приступить к развертыванию приложения с помощью Capistrano, нужно наладить аутентификацию между локальным компьютером и удаленными серверами, а так же между удаленными серверами и хранилищем Git. Мы уже рассматривали настройку аутентификации парными ключами SSH между локальным компьютером и удаленным сервером. Теперь необходимо настроить аутентификацию парными ключами SSH между удаленными серверами и хранилищем Git.

Воспользуйтесь приведенными выше инструкциями для создания открытых и закрытых ключей SSH для каждого удаленного сервера. Хранилище Git должно иметь доступ к открытым ключам каждого из серверов. Хранилища GitHub и BitBucket позволяют добавить несколько открытых ключей SSH к учетной записи. В конечном счете, нужно получить возможность переносить содержимое хранилища Git на удаленные серверы, не вводя пароля.

Подготовка удаленного сервера

Мы почти готовы к развертыванию приложения. Но прежде нужно подготовить удаленный сервер. Войдите на удаленный сервер с помощью SSH и создайте каталог для развертывания PHP-приложения. Этот каталог должен быть доступен для чтения и записи пользователю `deploy`. Я обычно создаю каталог для своих приложений внутри домашнего каталога пользователя `deploy`, следующим образом:

```
/  
home/  
  deploy/  
    app/  
      my_app/
```

Виртуальный хост

Символическая ссылка `current` ссылается на каталог, содержащий последнюю версию приложения. Смените корневой каталог докумен-

тов виртуального хоста веб-сервера, чтобы он указывал на каталог `curent/`. С учетом приведенной выше схемы каталогов, корневым каталогом документов виртуального хоста должен быть `/home/deploy/apps/my_app/curent/public/`. Это предполагает, что PHP-приложение содержит каталог `public/`, который служит корневым каталогом документов. Перезагрузите веб-сервер, чтобы применить изменения в настройках виртуального хоста.

Программные зависимости

Удаленному серверу не требуется Capistrano, но ему нужен Git. А также необходимо программное обеспечение для запуска PHP-приложения. Установить Git можно с помощью следующих команд:

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

# CentOS
sudo yum install git;
```

Обработчики Capistrano

Capistrano позволяет запускать команды (или *обработчики*) в выбранные моменты времени при развертывании приложений. Многие PHP-разработчики управляют зависимостями приложений с помощью менеджера зависимостей Composer. Мы можем устанавливать зависимости с помощью Composer при каждом развертывании, используя обработчики Capistrano. Откройте конфигурационный файл `config/deploy.rb` в текстовом редакторе и добавьте в него следующий код на языке Ruby:

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

Зависимости приложения теперь будут устанавливаться автоматически, при каждом развертывании. Более подробную информацию об обработчиках Capistrano можно найти на сайте Capistrano (<http://bit.ly/cap-flow>).



Если ваш проект использует менеджер зависимостей Composer, убедитесь, что Composer установлен на удаленных серверах.

Развертывание приложения

А теперь самое интересное! Отправьте самый свежий код приложения в репозиторий Git. Затем откройте терминал на локальном компьютере и перейдите в корневой каталог приложения. Если все сделано правильно, развернуть PHP-приложение можно с помощью следующей односрочной команды:

```
cap production deploy
```

Откат к предыдущей версии приложения

Если вдруг в новой версии обнаружилась ошибка, можно вернуться к предыдущей версии с помощью следующей односрочной команды:

```
cap production deploy:rollback
```

Дополнительные материалы

Я затронул лишь малую часть темы развертывания. Инструмент развертывания Capistrano имеет гораздо больше возможностей. Мне лично он нравится больше всего, но существует множество других инструментов развертывания, например:

- Deployer (<http://deployer.io/>)
- Magallanes (<http://mageweb.com/>)
- Rocketeer (<http://rocketeer.autopergamene.eu/>)

Что дальше

Мы подготовили сервер и автоматизировали развертывание PHP-приложения с помощью Capistrano. Далее мы рассмотрим вопросы соответствия поведения PHP-приложения нашим ожиданиям. Для этого мы используем *тестирование* и *профилирование*.

ГЛАВА 10.

Тестирование

Тестирование является важной частью разработки PHP-приложений, но этой частью часто пренебрегают. Я думаю, что многие разработчики не занимаются тестированием, потому что считают тестирование ненужным, требующим слишком много времени, и дающим слишком мало преимуществ. Другие разработчики вообще не знают, как организовать тестирование, потому что существует масса инструментов тестирования и у разработчиков не хватает времени на их изучение.

В этой главе я попытаюсь развеять связанные с тестированием заблуждения. Мне хочется помочь вам почувствовать себя уверенно и комфортно при тестировании PHP-кода. Я хочу, чтобы вы рассматривали тестирование, как неотъемлемую часть рабочего процесса и в самом начале, и в середине и в конце процесса разработки приложений.

Почему мы тестируем?

Мы пишем тесты, чтобы убедиться, что PHP-приложения работают и будут работать в дальнейшем, в соответствии с предъявляемыми к ним требованиями. Да, именно так просто. Как часто вы испытывали беспокойство перед вводом приложения в эксплуатацию? До того, как я начал тестировать свой код, я каждый раз жутко волновался перед вводом очередного релиза в эксплуатацию. Будет ли мой код работать? Вылезут ли ошибки? Все, что я мог поделать, это скрестить пальцы и надеяться на лучшее. Но это же не метод. Такое беспокойство и напряжение, как правило, заканчивается крахом. Тесты же позволят избежать неопределенности и добавляют уверенности при написании и развертывании кода.

Ваш злой босс может заявить, что на написание тестов не хватает времени. В конце концов, время это деньги. Это недальновидный подход. Установка инфраструктуры тестирования и написание тестов

занимает время, но это мудрое вложение средств, которое принесет дивиденды в будущем. Тесты помогут писать код, который правильно заработает с первого раза. Тестирование позволит нам непрерывно двигаться вперед, не нарушая существующего кода. Согласен, мы будем продвигаться медленнее, чем, если бы мы не использовали тесты, но в будущем не потратим много часов на выявление и исправление ошибок, которые были в свое время пропущены. В долгосрочной перспективе, тесты экономят деньги, предотвращают потери времени и укрепляют доверие к нам.

Когда мы тестируем?

Я знаю, что многие разработчики начинают заниматься написанием тестов в самом конце процесса разработки. Эти разработчики понимают важность тестирования, но считают тесты тем, чем им придется заняться, а не тем, чем они хотят заниматься. Они обычно тянут с тестированием до самого конца процесса разработки приложения. На скорую руку настрочат несколько тестов, чтобы успокоить руководство, и считают на этом свою работу законченной. Это не правильно. Тестирование должно быть в центре внимания перед разработкой, в процессе разработки и после разработки.

Перед

Установите и настройте инструменты тестирования до начала разработки приложения. Какие именно инструменты тестирования вы выберете, не имеет значения. Отнеситесь к их установке так, как будто они являются жизненно важной частью вашего приложения. Это физически и психологически облегчит тестирование приложения в процессе его разработки. В этот же период хорошо бы встретиться с руководством проекта и определить общие принципы работы приложения.

В процессе

Пишите и запускайте тесты сразу по готовности любой части приложения. Вы добавили новый PHP-класс? Протестируйте его немедленно, потому что, вероятней всего, вы забудете сделать это позднее. Тестирование при разработке поможет писать надежный и стабильный код, а также быстро находить и исправлять ошибки в новом коде, нарушающие работу существующего кода.

После

Вам, скорее всего, не удастся полностью протестировать все функции приложения во время разработки. Если вы найдете ошибку уже после запуска приложения, напишите новый тест, чтобы убедиться, что ошибка действительно исправлена. Тесты не являются чем-то сделанным раз и навсегда. Тесты постоянно изменяются и улучшаются, так же, как само приложение. Если вы изменили код приложения, не забудьте поправить соответствующие тесты.

Что мы тестируем?

Мы тестируем мелкие фрагменты приложения. PHP-приложения состоят из классов, методов и функций. Мы должны проверить каждый общедоступный класс, метод и функцию, чтобы убедиться, что их поведение, изолированное от внешних воздействий, соответствует ожиданиям. Убедившись, что каждая часть работает без ошибок сама по себе, можно быть уверенными, что они хорошо будут работать после интеграции в приложение. Такие тесты называются *модульными тестами*.

К сожалению, тестирование каждого отдельного фрагмента приложения не гарантирует правильной работы всего приложения. Поэтому нам придется также провести тестирование приложения на макроуровне, с помощью автоматизированных средств, проверяющих соответствие поведения всего нашего приложения ожиданиям. Применяемые при этом тесты называются *функциональными*.

Как мы тестируем?

Мы уже знаем почему, когда и что тестиировать. А теперь перейдем обсудим, *как* тестиировать код. В среде PHP-разработчиков популярны несколько подходов к тестированию. Одни предпочитают модульное тестирование. Другие – прием разработки через тестирование (Test-Driven Development, TDD). А третья – разработку, основанную на функционировании (Behavior-Driven Development, BDD). *Эти подходы не являются взаимоисключающими.*

Модульное тестирование

Самым популярным способом тестирования PHP-приложений является модульное тестирование. Как я уже упоминал выше, мо-

дульные тесты предназначены для сертификации отдельных классов, методов и функций в отрыве от остального приложения. Де-факто стандартным PHP-фреймворком для модульного тестирования является PHPUnit (<https://phpunit.de/>), написанный Себастьяном Бергманном (Sebastian Bergmann, <https://sebastian-bergmann.de/>). Фреймворк PHPUnit Себастьяна придерживается архитектуры тестов XUnit.

Существуют также альтернативные механизмы модульного тестирования, такие как PHPSpec. Однако большинство популярных PHP-фреймворков используют тесты PHPUnit. Если вы намерены заниматься разработкой и распространением PHP-компонентов, вам необходимо научиться читать, писать и запускать тесты PHPUnit. Я расскажу, как устанавливать, писать и запускать модульные тесты PHP-кода в конце этой главы.

Разработка через тестирование (TDD)

Разработка через тестирование подразумевает создание тестов *перед* разработкой прикладного кода. Эти тесты определяют цели и описывают, как приложение *должно* вести себя. По мере добавления функциональности тесты начинают успешно выполняться. TDD помогает выстроить цели разработки – вы заранее знаете, что будете разрабатывать, и как код должен работать.

Но это не значит, что вы должны сначала написать все тесты, а лишь потом приступить к разработке приложения. Вместо этого можно написать несколько тестов, а затем разработать связанную с ними функциональность. Пишите тесты и разрабатывайте. Пишите тесты и разрабатывайте. Процесс TDD является итеративным. Двигайтесь вперед небольшими шагами (спринтами), пока не завершите работу над приложением.

Разработка, основанная на функционировании (BDD)

Разработка, основанная на функционировании, предполагает написание сценариев, описывающих поведение приложения. Выделяют два типа BDD: SpecBDD и StoryBDD.

SpecBDD – это один из типов модульного тестирования, использующий гибкий и близкий человеческому языку для описания функционирования приложения. SpecBDD – альтернатива инструментам модульного тестирования, таким как PHPUnit. В отличие от архитек-

туры XUnit, PHPUnit использует в качестве тестов SpecBDD удобочитаемые сценарии, описывающие функционирование. Тест PHPUnit может называться, например, `testRenderTemplate()`. Эквивалентный тест SpecBDD может быть назван `itRendersTheTemplate()`. Этот тест SpecBDD может использовать вспомогательные методы с именами `$this->shouldReturn()`, `$this->shouldBe()` и `$this->shouldThrow()`. Тесты SpecBDD используют описания, которые гораздо проще читать и понимать, чем альтернативные инструменты XUnit. Наиболее популярным инструментом тестирования SpecBDD является PHPSpec (<http://www.phpspec.net/>).



Пишите тесты StoryBDD описывающие только бизнес-логику, а не конкретную ее реализацию. Хороший тест StoryBDD должен подтвердить, что «итог корзины покупок увеличивается при добавлении товара в корзину». Плохой тест StoryBDD подтверждает, что «итог корзины увеличивается при отправке HTTP-запроса PUT с телом `product_id=1&quantity=2` по URL-адресу `/cart`». Первый тест является общим и описывает бизнес-логику верхнего уровня. Второй – чересчур специфичен и описывает конкретную реализацию.

Инструменты StoryBDD используют те же удобные для восприятия человеком сценарии, что и инструменты тестирования SpecBDD. Инструменты StoryBDD, однако, ориентированы на тестирование функций верхнего уровня, а не их реализаций на нижнем уровне. Например, тест StoryBDD должен подтвердить, что код создает отчет в формате PDF и отправляет его по электронной почте. С другой стороны, тест SpecBDD должен подтвердить, что конкретный метод класса, генерирующий PDF, создает корректный PDF-файл при заданном наборе входных параметров. Разница в масштабах. Тест StoryBDD больше соответствует видению задачи руководителем проекта (например, «приложение должно создать отчет и отправить его по электронной почте»). Тест SpecBDD же соответствует видению задачи разработчиком (например, «метод класса должен получить массив данных и записать его в PDF-файл»). Инструменты тестирования StoryBDD и SpecBDD не являются взаимоисключающими. Часто они используются совместно для построения всеобъемлющего набора тестов. Обычно, вы вместе с руководителем проекта пишете тесты StoryBDD, определяющие общие функции приложения, а затем при реализации функций приложения пишете тесты SpecBDD. Наиболее

популярным инструментом тестирования StoryBDD является Behat (<http://behat.org/>).

PHPUnit

Давайте поговорим о том, как устанавливать, писать и запускать тесты PHPUnit. Развертывание инфраструктуры займет совсем немного времени, но это значительно упростит написание и запуск тестов PHPUnit. Прежде чем углубиться в PHPUnit, познакомимся с некоторыми терминами. Тесты PHPUnit объединяются в *комплекты тестов* (test cases), а комплекты группируются в *наборы тестов* (test suites). PHPUnit запускает наборы тестов с помощью *стартера тестов* (test runner).

Комплект тестов – это PHP-класс, наследующий класс `PHPUnit_Framework_TestCase`. Каждый комплекс тестов содержит общедоступные методы, имена которых начинаются с `test`, которые должны подтвердить правильность конкретных сценариев. Каждый тест может быть либо подтвержден, либо отвергнут. Вашей целью является подтверждение всех тестов.



Имя класса тестов должно заканчиваться на `Test`, имя содержащего его файла должно заканчиваться на `test.php`. Например, класс тестов `FooTest` должен содержаться в файле `FooTest.php`.

Набор тестов является коллекцией связанных между собой тестов. Если разрабатывается отдельный PHP-компонент, можно обойтись одним набором тестов. Если тестируется большое PHP-приложение с несколькими подсистемами или компонентами, лучше разделить тесты на несколько наборов.

Назначение стартера тестов соответствует его названию. Это – инструмент запуска наборов тестов и вывода результатов. По умолчанию стартер тестов PHPUnit запускается из командной строки командой `phpunit`.

Структура каталогов

Я предпочитаю организовывать PHP-проекты следующим образом. Корневой каталог проекта содержит подкаталог `src/` с исходным

кодом. Здесь же имеется каталог *tests/*, содержащий тесты. Ниже приведен пример структуры каталогов:

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

src/

Каталог с исходными кодами PHP-проекта (т. е., PHP-классы).

tests/

Каталог с PHPUnit-тестами PHP-проекта. Этот каталог содержит файл *bootstrap.php*, подключаемый к PHPUnit перед выполнением модульных тестов.

composer.json

В этом файле перечислены зависимости PHP-проекта, управляемые Composer, в том числе фреймворк тестирования PHPUnit.

phpunit.xml

Этот файл содержит параметры настройки стартера тестов PHPUnit.

.travis.yml

Этот файл содержит параметры настройки веб-службы тестирования непрерывной интеграции Travis CI.



Просмотрите исходный код понравившегося вам компонента или фреймворка на GitHub, и вы увидите, что он организован аналогичным образом.

Установка PHPUnit

Первое, что потребуется сделать – установить PHPUnit и профилировщик Xdebug. PHPUnit нужен для запуска тестов. Профилировщик Xdebug генерирует полезную информацию об охвате кода.

Проще всего установить фреймворк PHPUnit с помощью Composer. Откройте терминал, перейдите в корневой каталог проекта и выполните следующую команду:

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

Эта команда загрузит фреймворк PHPUnit в каталог *vendor/* проекта и обновит файл *composer.json*, добавив зависимость *phpunit/phpunit*. Выполняемый двоичный файл *phpunit* будет помещен в каталог *vendor/bin/* проекта. Вы можете добавить этот каталог в переменную окружения *PATH* или ссылаться на *vendor/bin/phpunit* всякий раз, при запуске тестов из командной строки. Классы фреймворка PHPUnit будут автоматически загружены в PHP-приложение посредством других зависимостей, управляемых Composer.

Установка Xdebug

Расширение PHP Xdebug несколько сложнее в установке. Если вы установили PHP с помощью менеджера пакетов, установить Xdebug можно тем же способом (пример 10.1).

Пример 10.1. Установка Xdebug

```
# Ubuntu
sudo apt-get install php5-xdebug

# CentOS
sudo yum -y --enablerepo=epel,remi,remi-php56 install php-xdebug
```

Если вы установили PHP из исходных текстов, расширение Xdebug можно будет с помощью команды *pecl*:

```
pecl install xdebug
```

После этого измените файл конфигурации *php.ini*, указав путь к скомпилированному расширению Xdebug.



Найти каталог расширений PHP можно с помощью команды *php-config --extension-dir* ИЛИ *php -i | grep extension_dir*.

Добавьте следующую строку в файл *php.ini*, указав путь к расширениям PHP в вашей системе:

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

Перезапустите PHP и дело сделано. О профилировщике Xdebug мы поговорим в главе 11.

Настройка PHPUnit

А теперь настроим PHPUnit с помощью файла *phpunit.xml* нашего проекта.

```
<?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>
```

Параметры настройки PHPUnit представлены атрибутами корневого XML-элемента `<phpunit>`. Наиболее важным, на мой взгляд, является параметр `bootstrap`, определяющий путь к PHP-файлу (относительно файла *phpunit.xml*), который будет подключен перед запуском PHPUnit. Мы настроили автозагрузку зависимостей приложения с помощью Composer в файле *bootstrap.php*, поэтому они доступны PHPUnit-тестам. Также файл *bootstrap.php* определяет путь к набору тестов (т.е., каталог, с соответствующими тестами). PHPUnit обрабатывает все PHP-файлы в этом каталоге, имена которых заканчиваются на *Test.php*. И, наконец, в этом конфигурационном файле, в элементе `<filter>`, перечислены каталоги, которые будут включены в анализ охвата кода. В предыдущем примере XML элемент `<white list>` требует от PHPUnit оценить охват только для кода, находящегося в каталоге *src/*.

В сущности, назначение этого конфигурационного файла состоит в том, чтобы собрать параметры PHPUnit в одном месте. Это облегчает нам жизнь, потому что отпадает необходимость задавать эти настройки при каждом запуске `phpunit` из командной строки. Кроме того, этот конфигурационный файл позволяет применять те же настройки и для

удаленных серверов непрерывного тестирования, таких как Travis CI. После внесения изменений в конфигурационный файл *phpunit.xml*, замените содержимое файла *tests/bootstrap.php* следующим кодом:

```
<?php
// Включить автозагрузчик Composer
require dirname( DIR ) . '/vendor/autoload.php';
```



Не забудьте перед запуском тестов PHPUnit определить зависимости Composer.

Класс Whovian

Прежде, чем начать писать модульные тесты, нам понадобится что-то, что мы будем тестировать. Ниже приводится гипотетический PHP-класс Whovian, имеющий непоколебимое мнение о персонаже телешоу Би-Би-Си. Поместите это определение класса в файл *src/Whovian.php*:

```
<?php
class Whovian
{
    /**
     * @var string
     */
    protected $favoriteDoctor;
    /**
     * Конструктор
     * @param string $favoriteDoctor
     */
    public function __construct($favoriteDoctor)
    {
        $this->favoriteDoctor = (string)$favoriteDoctor;
    }
    /**
     * Say
     * @return string
     */
    public function say()
    {
        return 'Лучший доктор ' . $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(
                    'Никоим образом! %s - лучший доктор!',
                    $this->favoriteDoctor
                )
            );
        }
        return 'Я согласен!';
    }
}

```

Конструктор класса `Whovian` настраивает любимого доктора для экземпляра. Метод `say()` возвращает строку с именем любимого доктора экземпляра. А метод `respondTo()` получает заявление от другого экземпляра `Whovian` и реагирует на него соответствующим образом.

Класс теста `WhovianTest`

Модульные тесты для класса `Whovian` будут находиться в файле `test/WhovianTest.php`. Группа связанных между собой тестов называется *набором тестов*. В нашем примере все тесты, находящиеся в каталоге `test/`, будут принадлежать одному набору. Каждый файл класса в каталоге `test/` называется *комплектом тестов*, а методы класса, имена которых начинаются с `test` (например, `testThis` или `testThat`) являются отдельными тестами. Каждый отдельный тест использует утверждения для проверки заданного условия. Утверждение может быть принято или отвергнуто.



Список утверждений PHPUnit можно просмотреть на сайте PHPUnit (<http://bit.ly/php-unit>). Некоторые из утверждений не документированы. Все доступные утверждения можно найти в исходном коде на GitHub (<http://bit.ly/phpu-qh>).

Каждый комплект тестов PHPUnit представлен классом, наследующим `PHPUnit_Framework_TestCase`. Давайте объявим комплект тестов `WhovianTest`, поместив его объявление в файл `test/WhovianTest.php`:

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

class WhovianTest extends PHPUnit_Framework_TestCase
{
    // Отдельные тесты помещаются сюда
}
```

Напомню, что модульные тесты проверяют поведение общедоступных методов ожидаемому. Мы будем тестировать три общедоступных метода класса `Whovian`. Напишем модульный тест для проверки, что аргумент метода `__construct()` становится предпочтаемым доктором экземпляра. Затем напишем модульный тест, проверяющий, что метод `say()` возвращает имя предпочтаемого доктора. И, наконец, напишем два теста для метода `respondTo()`. Первый тест проверит совпадение возвращаемого значения метода со строкой 'я согласен!', если входной аргумент соответствует имени его предпочтаемого доктора. А второй тест проверит возбуждение исключения методом, если входной аргумент не соответствует предпочтаемому доктору.

Тест 1: `__construct()`

Наш первый тест должен подтвердить, что конструктор устанавливает любимого доктора для экземпляра `Whovian`:

```
public function testSetsDoctorWithConstructor()
{
    $whovian = new Whovian('Питер Капальди');
    $this->assertEquals('Питер Капальди', 'favoriteDoctor',
$whovian);
}
```

Этот тест создает новый экземпляр `Whovian`, передавая единственный строковый аргумент: 'Питер Капальди'. Чтобы подтвердить, что свойство `favoriteDoctor` экземпляра `$whovian` равно строке 'питер Капальди', используется метод-утверждение `assertEquals()`.

Почему мы проверяем имя любимого доктора с помощью утверждения `assertEquals()`, а не метода получения значения (например, `getFavoriteDoctor()`)? Когда мы пишем тест, то тестируем *отдельно только один конкретный метод*. В идеале тест не должен

полагаться на другие методы. В этом конкретном примере, мы тестируем метод `__construct()` и подтверждаем, что он присваивает свой аргумент свойству `$favoriteDoctor` объекта. Утверждение `assertAttributeEquals()` позволяет проверить внутреннее состояние объекта, не полагаясь на непроверенный метод получения.



Метод-утверждение `assertAttributeEquals()` принимает три аргумента. Первый – ожидаемое значение, второй – имя свойства и последний – проверяемый объект. Следует отметить, что метод `assertAttributeEquals()` можно применять также для проверки защищенных свойств, что достигается при помощи возможностей механизма рефлексии в PHP.

Тест 2: `say()`

Наш следующий тест подтверждает возвращение методом `say()` экземпляра `Whovian` строки с именем любимого доктора:

```
public function testSaysDoctorName()
{
    $whovian = new Whovian('Дэвид Теннант');
    $this->assertEquals('Лучший доктор - Дэвид Теннант', $whovian->say());
}
```

Здесь для сравнения двух значений используется утверждение `assertEquals()`. В первом аргументе утверждению передается ожидаемое значение. Во втором – проверяемое значение.

Тест 3: `respondTo()` при согласии

Теперь проверим ответ экземпляра `Whovian` при согласии с другим экземпляром `Whovian`:

```
public function testRespondToInAgreement()
{
    $whovian = new Whovian('Дэвид Теннант');

    $opinion = 'Дэвид Теннант - лучший доктор, точка';
    $this->assertEquals('Я согласен!', $whovian->respondTo($opinion));
}
```

Этот тест завершается успехом, потому что метод `respondTo()` экземпляра `Whovian` получает в качестве аргумента строку с именем любимого доктора.

Тест 4: respondTo() при несогласии

Ну а что, если Whovian *не согласен*? Выведите его как можно скорее, пока он не подрался с другим фанатом! Ладно, просто вызовем исключение. Давайте протестируем это:

```
/**  
 * @expectedException Exception  
 */  
public function testRespondToInDisagreement()  
{  
    $whovian = new Whovian('Дэвид Теннант');  
  
    $opinion = 'Никоим образом! Мэтт Смит - лучший доктор!';  
    $whovian->respondTo($opinion);  
}
```

Тест будет считаться пройденным, если сгенерирует исключение. При тестировании этого условия применяется аннотация `@expectedException`.



PHPUnit предоставляет несколько аннотаций для проверки определенных тестов. Более подробную информацию об аннотациях PHPUnit можно найти в документации PHPUnit на странице (<http://bit.ly/phpunit-docs>).

Запуск тестов

После того как будут написаны все тесты, нужно запустить набор тестов. Сделать это несложно. Откройте терминал и перейдите в корневой каталог проекта (в этом же каталоге расположен конфигурационный файл `phpunit.xml`). Мы будем использовать выполняемый файл PHPUnit, установленный с помощью Composer. Выполните следующую команду, чтобы запустить тестирование:

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

Ключ `-c` задает путь к конфигурационному файлу PHPUnit. Результаты тестирования будут выведены в терминал, как показано на рис.10.1.

Эти результаты говорят, что:

1. PHPUnit прочитал конфигурационный файл;
2. выполнение тестов заняло 24 миллисекунды;
3. использовано 3,5 Мбайт памяти;

4. успешно выполнено пять тестов и получено пять подтверждений.

```
1.bash
Josh's-MacBook-Pro:test-example josh$ vendor/bin/phpunit -c phpunit.dist.xml
PHPUnit 4.3.3 by Sebastian Bergmann.

Configuration read from /Users/josh/Repos/modern-php/test-example/phpunit.dist.xml

Time: 24 ms, Memory: 3.50Mb

OK (5 tests, 5 assertions)
Josh's-MacBook-Pro:test-example josh$
```

Рис. 10.1. Результаты тестирования

Охват кода

Теперь мы знаем, что тестирование завершилось успехом. Тем не менее, можно ли быть уверенным, что тестирование было настолько всеобъемлющим, насколько это вообще возможно? Может быть, мы забыли что-то протестировать. Узнать, какой именно код был протестирован (или остался непроверенным) можно из отчета об охвате кода (рис. 10.2). Мы уже задали в конфигурационном файле PHPUnit путь или пути к файлам с исходным кодом. Все PHP-файлы, находящиеся в каталогах, перечисленные в белом списке, будут включены в отчет PHPUnit. Получить отчет об охвате кода можно при каждом тестировании:

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



Рис. 10.2. Отчет PHPUnit об охвате кода тестированием

Это та же команда, которую мы уже использовали выше, но в нее добавлен ключ `--coverage-html`, определяющий путь к каталогу отче-

та об охвате кода. После выполнения этой команды, откройте вновь созданный файл `coverage/index.html` в браузере и вы увидите сведения об охвате кода тестированием. В идеале желательно иметь сто-процентный охват во всех панелях. Однако стопроцентного охвата обычно *не удается* достичь и его получение не может считаться обязательным требованием. Какой охват считать хорошим – вещь субъективная и индивидуальная для каждого проекта.



Используйте отчет об охвате кода тестированием как руководство к действию для улучшения вашего кода. Не ограничивайтесь только требованием достижения высоких процентов охвата.

Непрерывное тестирование с помощью Travis CI

Иногда даже самые лучшие PHP-разработчики забывают написать тесты. Поэтому так важна автоматизация процесса тестирования. Лучшие тесты подобны хорошим средствам резервного копирования: их не видно, о них можно забыть, но они есть. *Тесты должны запускаться автоматически*. Моей любимой службой непрерывного тестирования является Travis CI, потому что она имеет собственные обработчики в репозиториях GitHub. Travis CI дает возможность запускать тесты при каждой отправке кода в репозиторий GitHub. Кроме того, Travis CI выполняет тесты под несколькими версиями PHP.

Установка

Если вы раньше никогда не использовали Travis CI, перейдите на сайт <https://travis-ci.org> (проект для общедоступных репозиториев) или на сайт <https://travis-ci.com> (проект для закрытых репозиториев). Войдите в свою учетную запись GitHub. Следуя инструкциям на экране, выберите репозиторий, код которого должен тестироваться с помощью Travis CI.

Затем создайте конфигурационный файл Travis CI `.travis.yml` в корневом каталоге приложения. Не забудьте о точке в начале имени файла! Сохраните конфигурационный файл Travis CI и отправьте его в репозиторий GitHub. Ниже приводится пример конфигурации Travis CI:

```
language: php
php:
  - 5.4
  - 5.5
  - 5.6
  - hhvm
install:
  - composer install --no-dev --quiet
script: phpunit -c phpunit.xml --coverage-text
```

Конфигурация Travis CI записана в формате YAML и содержит следующие параметры:

language

Язык приложения. В данном случае `php`. Значение чувствительно к регистру букв!

php

Travis CI выполнит тесты под перечисленными ниже версиями PHP. Параметр важен, если приложение должно поддерживать несколько версий PHP.

install

Определяет bash-команду, выполняемую Travis CI перед запуском тестов приложения. Здесь Travis CI дается указание установить зависимости проекта от Composer. Обязательно добавьте ключ `--no-DEV`, чтобы избежать установки ненужных при разработке зависимостей.

script

Определяет bash-команду, выполняемую Travis CI для запуска тестирования. По умолчанию, это команда `phpunit`. Этот параметр позволяет переопределить команду по умолчанию. В данном примере Travis CI указывается использовать пользовательский конфигурационный файл PHPUnit и генерировать результаты охвата в виде обычного текста.

Запуск

Travis CI автоматически запускает тесты приложения при каждой отправке кода в репозиторий GitHub и высылает результаты по электронной почте. Что, круто? Конечно, существует много других параметров настройки среды тестирования Travis CI (например, для установки PHP-расширений, использования настроек `ini` и так далее). Подроб-

ную информацию о конфигурировании Travis CI для работы с PHP можно найти на странице [http://bit.ly/bbuild-php](http://bit.ly/build-php).

Дополнительные материалы

Следующие ссылки помогут вам больше узнать о тестировании PHP-приложений:

- <https://phpunit.de/>
- <http://www.phpspec.net/docs/introduction.html>
- <http://behat.org/>
- <https://leanpub.com/grumpy-phpunit>
- <https://leanpub.com/grumpy-testing>
- <http://www.littlehart.net/atthekeyboard/>

Что дальше

В этой главе мы узнали почему, когда и как писать тесты. Тестирование приложений укрепляет доверие и способствует созданию более предсказуемого кода. Однако, тесты не позволяют анализировать производительность приложений. Вот почему необходимо еще и профилирование. О нем я и собираюсь рассказать в следующей главе.

ГЛАВА 11.

Профилирование

Профилирование предназначено для анализа производительности приложений. Это отличный способ диагностики проблем производительности и выявления узких мест в приложениях. Другими словами, если приложение работает медленно, с помощью профилировщика можно выяснить, почему это происходит. Профилировщики позволяют пройтись по стеку вызовов PHP, увидеть, какие функции или методы вызывались, в каком порядке, сколько раз, с какими аргументами и сколько времени заняло их выполнение. А также отображают использование памяти и загрузку процессора на всех этапах обработки запроса.

Когда следует использовать профилировщик

Нет никакой необходимости сразу приступать к профилированию приложения. Профилирование требуется, когда имеются проблемы с производительностью, которые не удается диагностировать другими способами. Как узнать, что проблемы связаны именно с производительностью? Иногда проблемы очевидны (например, запрос к базе данных занимает слишком много времени). Но бывают случаи, когда ответы на такие вопросы не лежат на поверхности.

Обнаружить проблемы производительности можно с помощью оценочных инструментов, таких как Apache Bench (<http://bit.ly/apache-bench>) и Siege (<http://www.joedog.org/siege-home/>). Оценочные инструменты позволяют проверить производительность *извне*, то есть оценить производительность приложения при работе пользователя через браузер. Оценочные инструменты дают возможность задать количество одновременно работающих пользователей и общее число запросов поступающих на определенный URL-адрес приложения. После завершения проверки, оценочный инструмент сообщит

о количестве запросов в секунду, при котором приложение еще сохраняло устойчивость (среди прочих статистических данных). Если найден конкретный URL-адрес, который выдерживает лишь небольшое количество запросов в секунду, это может говорить о проблемах с производительностью. Если проблема не очевидна, используется профилировщик.

Типы профилировщиков

Существует два типа профилировщиков. Профилировщики первого типа могут применяться только при разработке, а второго типа – также и при эксплуатации.

- **Xdebug** (<http://xdebug.org/>), написанный Дериком Ретансом (Derick Rethans) – популярный инструмент профилирования PHP, но его можно использовать только во время разработки, потому что он потребляет много системных ресурсов. Результаты, выдаваемые профилировщиком Xdebug неудобочитаемы, поэтому требуется дополнительное приложение для анализа и отображения результатов. Хорошими приложениями визуализации результатов профилировщика Xdebug являются KCacheGrind (<http://kcachegrind.sourceforge.net/>) и WinCacheGrind (<http://sourceforge.net/projects/wincachegrind/>).
- **XHProf** (<http://xhprof.io/>) – популярный PHP-профилировщик от Facebook. Предназначен для профилирования при разработке и эксплуатации. Результаты, выдаваемые профилировщиком XHProf также неудобочитаемы, но Facebook предоставляет приложение XHGUI визуализации и анализа результатов профилировщика. Я еще вернусь к XHGUI позже в этой же главе.



Оба профилировщика – Xdebug и XHProf – являются PHP-расширениями. Их можно установить с помощью менеджера пакетов операционной системы. Также их можно установить с помощью `pecl`.

Xdebug

Xdebug – один из самых популярных профилировщиков PHP, своей популярностью он обязан прежде всего простоте анализа стека вызо-

вов приложения при поиске узких мест и проблем с производительностью. Инструкция по установке Xdebug приводится в примере 10.1.

Настройка

Параметры настройки Xdebug хранятся в файле *php.ini*. Ниже приводятся рекомендуемые мной настройки профилировщика Xdebug. Не забудьте изменить каталог для сохранения результатов профилирования на приемлемый в вашем случае. Перезагрузите PHP после сохранения настроек:

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

```
xdebug.profiler_enable = 0
```

Указывает, что Xdebug не должен запускаться автоматически. Нам не нужен автоматический запуск Xdebug при каждом запросе, потому что это приведет к резкому снижению производительности и затруднит разработку.

```
xdebug.profiler_enable_trigger = 1
```

Указывает, что Xdebug должен запускаться по требованию. Мы можем активировать профилирование запроса, добавив в запрос параметр `XDEBUG_PROFILE = 1` для любого URL-адреса PHP-приложения. Обнаружив этот параметр, Xdebug выполнит профилирование текущего запроса и сгенерирует отчет в каталоге вывода, заданном параметром `xdebug.profiler_output_dir`.

```
xdebug.profiler_output_dir = /path/to/profiler/results
```

Путь к каталогу для сохранения результатов работы профилировщика. Отчеты профилировщика для сложных PHP-приложений могут иметь солидный размер (например, 500 Мбайт и больше). Удостоверьтесь, что определили в этом параметре путь, действительно существующий в файловой системе.



Я рекомендую хранить результаты профилировщика в корневом каталоге PHP-приложения. Это позволит легко находить и просматривать результаты профилирования при разработке.

Включение

Так как параметр `xdebug.profiler_enable` имеет значение 0, профилировщик Xdebug не запускается автоматически. Чтобы запустить профилирование, в любой запрос к приложению нужно добавить параметр `XDEBUG_PROFILE = 1`. Например, URL-адрес HTTP-запроса может иметь следующий вид: `/users/show/1?XDEBUG_PROFILE=1`. Когда Xdebug обнаружит в параметре `XDEBUG_PROFILE`, он активирует и запустит профилирование текущего запроса. Результаты профилирования будут сохранены в каталоге, указанном в параметре `xdebug.profiler_output_dir`.

Анализ

Профилировщик Xdebug генерирует результаты в формате CacheGrind. Для анализа результатов профилировщика вам понадобится приложение, отображающее данные в формате CacheGrind. Ниже приводится список приложений для просмотра файлов в формате CacheGrind:

- WinCacheGrind для Windows (<http://sourceforge.net/projects/wincachegrind/>);
- KCacheGrind для Linux (<http://kcachegrind.sourceforge.net/>);
- WebGrind для веб-браузеров (<http://code.google.com/p/webgrind/>).

Пользователи Mac OS X могут установить KCacheGrind с помощью Homebrew, выполнив следующую команду:

```
brew install qcachegrind
```



Homebrew (<http://brew.sh/>) – это менеджер пакетов для OS X. Мы поговорим о Homebrew в Приложении А.

XHProf

XHProf – новейший профилировщик PHP-приложений. Он создан в Facebook и предназначен для профилирования как во время разработки, так во время эксплуатации. Он не собирает столько же сведе-

ний, как профилировщик Xdebug, но потребляет меньше системных ресурсов, что позволяет применять его в эксплуатационном окружении.

Установка

Проще всего установить XHProf – с помощью менеджера пакетов операционной системы (если вы установили PHP таким же способом):

```
# 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;
```

Добавьте следующие строки в файл *php.ini* и перезапустите PHP для загрузки новых расширений:

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

XHGUI

XHProf особенно удобно использовать в паре с XHGUI, сопутствующем веб-приложением от Facebook для просмотра и анализа результатов профилировщика XHProf. XHGUI – это веб-приложение PHP и требует наличия:

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

Я предполагаю, что все требуемые зависимости уже установлены. Я также предполагаю, что веб-приложение XHGUI находится в каталоге */var/sites/xhgui/*. Имейте в виду, что путь к этому каталогу на вашем сервере, скорее всего, будет другим:

```
cd /var/sites;
git clone https://github.com/perf-tools/xhgui.git;
```

```
cd xhgui;
php install.php;
```

Веб-приложение XHGUI имеет каталог *webroot/*. Смените корневой каталог документов виртуального хоста веб-сервера на этот каталог.

Настройка

Откройте конфигурационный файл *config/config.default.php* XHGUI в текстовом редакторе. По умолчанию XHProf собирает данные только об 1% всех HTTP-запросов. Это нормально для эксплуатации, но во время разработки желательно чаще собирать данные. Увеличить частоту сбора данных можно, изменив следующие строки в конфигурационном файле *config/config.default.php*:

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

Замените приведенные выше строки на следующие:

```
'profiler.enable' => function() {
    return true; // <-- Запускать для каждого запроса
},
```



XHProf предполагает, что PHP-приложение запущено на одном сервере. XHProf также предполагает, что база данных MongoDB не требует аутентификации. Если ваш сервер MongoDB требует аутентификации, измените параметры подключения к базе данных MongoDB в файле *config/config.default.php*.

Включение

Вы должны подключить файл *external/header.php* веб-приложения XHGUI в самом начале PHP-приложения. Для этого достаточно добавить параметр *auto_prepend_file* в конфигурационный файл *php.ini*:

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

Или определить его в конфигурации виртуального хоста nginx:

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

Или определить в конфигурации виртуального хоста Apache:

```
php_admin_value auto-prepend_file "/var/sites/xhgui/external/header.php"
```

Перезапустите PHP, и XHProf начнет сбор и сохранение информации в базе данных MongoDB. Вы можете просмотреть и проанализировать результат работы XHProf, перейдя по URL-адресу виртуального хоста XHGUI.

Профилировщик New Relic

Еще один популярный профилировщик – New Relic (<https://newrelic.com/>). На самом деле это веб-служба, которая использует нестандартный демон операционной системы и PHP-расширение для подключения к PHP-приложению и передачи отчетных данных в веб-службу. В отличие от Xdebug и XHProf, PHP-профилировщик New Relic не является бесплатным. Тем не менее, мне он очень нравится и я рекомендую его вам, если ваш бюджет позволяет его приобретение. Как и XHProf, PHP-профилировщик New Relic можно запускать в эксплуатационной среде и он предоставляет обзор производительности вашего приложения практически в режиме реального времени с помощью очень удобной онлайн-панели. Более подробную информацию о New Relic можно найти на странице <http://bit.ly/new-relic-php>.

Профилировщик Blackfire

На момент написания этой книги компания Symfony тестировала новый PHP-профилировщик Blackfire (<https://blackfire.io/>). Он обеспечивает уникальные средства визуализации для поиска узких мест приложения. Я слышал, что этот профилировщик будет хорошей альтернативой Xdebug и XHProf. Следите за его появлением.

Дополнительные материалы

Я надеюсь, что описал в этой главе профилирование в достаточном объеме, чтобы вы, прочтя ее, могли комфортно себя чувствовать при поиске, установке и использовании PHP-профилировщика, лучше всего подходящего вашему приложению. Приведенные ниже ссылки позволяют вам поближе познакомиться с PHP-профилированием:

- <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>

Что дальше

На данный момент мы уже рассмотрели множество особенностей современного языка PHP, в том числе новые возможности, хорошие методики, настройку, подготовку, развертывание, тестирование и профилирование. Я надеюсь, что при чтении книги у вас возникла масса интересных идей, которые вы сможете реализовать в будущих PHP-приложениях.

А теперь я хочу отнять у вас немного времени на обсуждение будущего PHP. Экосистема PHP бурно развивается. Будущее языка PHP представляется блестящим, благодаря концептуальным проектам, таким как PHP 7 (<https://wiki.php.net/rfc/php7timeline>), HHVM (<http://hhvm.com/>), Hack (<http://hacklang.org/>) и PHP-FIG (<http://www.php-fig.org/>). Давайте, например, рассмотрим HHVM и Hack, и выясним, что они значат для будущего PHP.

ГЛАВА 12.

HHVM и Hack

Что вы думаете о приложениях Facebook, я же, размышляя о них, не испытываю ничего, кроме благодарности блестящей команде разработчиков Facebook. Разработчики Facebook Open Source (<https://code.facebook.com/projects/>) создали за последние несколько лет множество важнейших проектов, два из которых оказали особенно значительное влияние на PHP-сообщество.

Первым из них является HHVM или *Hip Hop Virtual Machine* (<http://hhvm.com/>). Это – альтернативный движок PHP, выпущенный в октябре 2013 года. Его динамический (Just-In-Time, JIT) компилятор по производительности во много раз превосходит PHP-FPM. И действительно, компания WP Engine, недавно мигрировавшая на HHVM, увеличила производительность в 3.9 раз (<http://bit.ly/engine-box>) относительно традиционной установки Wordpress. Разработчики MediaWiki также перешли на HHVM и добились значительного уменьшения времени отклика и улучшения пропускной способности (<http://www.mediawiki.org/wiki/HHVM>).

Второй такой проект – Hack (<http://hacklang.org/>) – новый серверный язык, который является модификацией языка PHP. Hack в основном совместим с PHP, дополняя его статической типизацией, новыми структурами данных и серверной проверкой типов в режиме реального времени. Сами разработчики Hack предпочитают называть Hack диалектом PHP, а не новым языком.

HHVM

Начиная с 1994 года, под *интерпретатором языка PHP* подразумевали только Zend Engine (<http://www zend com/en/community/php>). Движок Zend Engine и был языком PHP. Это был единственный интерпретатор языка PHP. 4 февраля 2004 года Марк Цукерберг (Mark Zuckerberg) создал небольшое приложение, названное им

Thefacebook. Мистер Цукерберг и его растущая компания в основном использовали язык PHP, потому что его легко освоить, и он обеспечивает простоту развертывания. Применение языка PHP позволяло Facebook оперативно привлекать новых разработчиков для расширения и внесения новшеств в платформу.

Перенесемся на несколько лет вперед. Facebook становится настоящей империей. Им пользуется масса людей. Он так огромен, что традиционный движок Zend Engine начинает сдерживать его дальнейшее развитие. Команда Facebook обслуживает быстро растущую базу пользователей (к 2007 году в Facebook уже был зарегистрирован каждый десятый житель планеты) ищет способы увеличения производительности, не связанные с постройкой новых центров обработки данных и покупкой дополнительных серверов.

PHP в Facebook

Язык PHP традиционно интерпретируется, а не компилируется. Это значит, что исходный код на PHP продолжает оставаться исходным кодом на PHP до тех пор, пока не будет пропущен через интерпретатор при выполнении из командной строки или по запросу веб-сервера. Интерпретатор читает сценарий на PHP и преобразует в набор кодов Zend Opcodes (<http://php.net/manual/internals2.opcodes.php>) (машинных команд), выполняемых Zend Engine. К сожалению, интерпретируемые языки работают медленнее, чем компилируемые, потому что выполняют преобразование в машинный код при каждом выполнении. А это дополнительные затраты системных ресурсов. Команда Facebook осознала эту проблему и в 2010 году начала работу над компилятором PHP-в-C++, получившим название НРНРс.

Компилятор НРНРс преобразует код на языке PHP в код на языке C++. Затем компилирует код C++ в выполняемый файл, который развертывается на производственных серверах. Проект НРНРс был весьма успешным, он позволил улучшить производительность Facebook и снизить нагрузку на серверы. Однако, рост производительности, который позволяла идея, лежащая в основе проекта НРНРс, приблизился к своему потолку, кроме того не было достигнуто 100% совместимости с языком PHP, процесс компиляции требовал больших временных затрат, что значительно затрудняло обратную связь при разработке. Все это убедило команду Facebook в необходимости гибридного решения, обеспечивающего лучшую производительность и не снижающего скорость разработки из-за больших затрат времени на компиляцию.

И команда Facebook начал работать над проектом, получившим название HHVM. HHVM преобразует и кэширует PHP-код в промежуточный байт-код и использует JIT-компилятор для преобразования и оптимизации кэша байт-кода в машинный код x86_64. JIT-компилятор HHVM позволяет такую оптимизацию производительности на низком уровне, которой просто невозможно достичь при компиляции кода PHP в C++ с помощью HPHPc. Кроме того, HHVM уменьшает затраты времени на обратную связь при разработке, так как компилирует байт-код в машинный, только когда PHP-сценарии запрашиваются веб-сервером, то есть так же, как при использовании обычного интерпретируемого языка. И, что самое удивительное, в ноябре 2012 года производительность HHVM превысила производительность HPHPc (<http://bit.ly/hhvm-evo>) и продолжает расти (рис. 12.1).

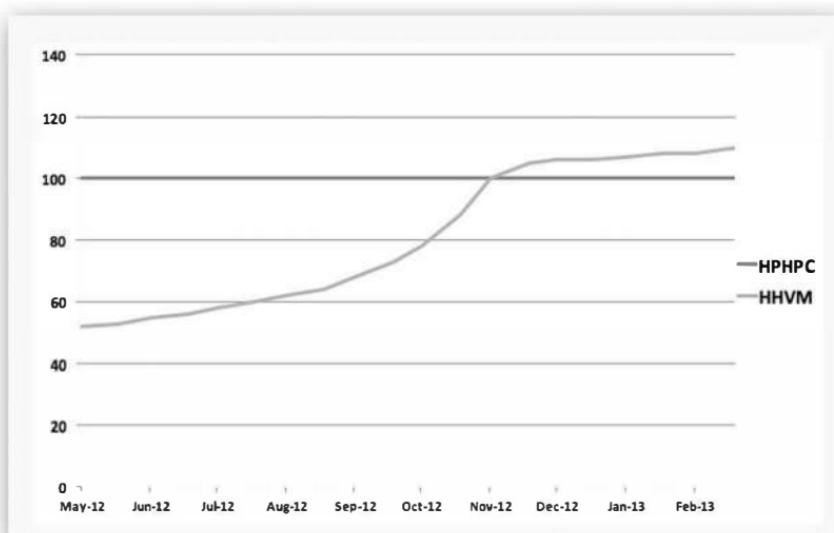


Рис. 12.1. Сравнение производительности HHVM с производительностью HPHPc

Проект HPHPc признан устаревшим вскоре после того, как HHVM показал более высокую производительность, и в настоящее время Facebook отдает предпочтение HHVM, как PHP-интерпретатору.



Не надо бояться HHVM! Пусть его внутренняя реализация сложна, но, в конце концов, HHVM это просто замена более привычных выполняемых файлов `php` и `php-fpm`:

- С помощью выполняемого файла `hhvm` вы запускаете на PHP-сценарии точно так же, как с помощью выполняемого файла `php`.
- Вы используете выполняемый файл `hhvm` для создания сервера FastCGI точно так же, как используете выполняемый файл `php-fpm`.
- HHVM использует конфигурационный файл `php.ini` точно так же, как традиционный Zend Engine. Он даже использует те же INI-директивы.
- HHVM имеет встроенную поддержку многих популярных PHP-расширений.

Совместимость HHVM с Zend Engine

Оригинальный компилятор HPHPc не был полностью совместим с языком PHP (т. е. с Zend Engine). Facebook стремится к полной совместимости, потому что это позволит HHVM стать подходящей заменой для Zend Engine.

Разработчики Facebook тестируют HHVM с самыми популярными фреймворками, чтобы обеспечить совместимость с действующим кодом на PHP 5. Команда Facebook близка к достижению 100% совместимости. Однако Facebook сообщил на трекере HHVM (<http://bit.ly/fb-hhvm>) о ряде еще оставшихся нерешенными важных вопросов. HHVM все еще не на 100% совместим с традиционным Zend Engine, но такая совместимость становится ближе день ото дня. Facebook, Baidu и Википедия уже используют HHVM. С HHVM также могут работать Wordpress, Drupal и многие другие популярные PHP-фреймворки.

Будет ли HHVM правильным выбором для меня?

Использование HHVM не станет правильным выбором абсолютно для всех. Существуют более простые способы улучшения производительности приложений. Сокращение количества HTTP-запросов и оптимизация запросов к базе данных, являются лежащими на поверхности средствами, позволяющими заметно повысить производительность приложений и сократить время отклика. Если вы еще не сделали такой оптимизации, сделайте ее, прежде чем принимать

решение о переходе на HHVM. HHVM от Facebook необходим тем разработчикам, которые уже сделали оптимизацию, но все еще не добились нужной производительности. Если вы рассматриваете вопрос о переходе на HHVM, приведенные ниже ресурсы помогут вам принять правильное решение:

Расширения

(<http://bit.ly/fb-extensis>)

Просмотрите список PHP-расширений, совместимых с HHVM.

Совместимость с фреймворками

(<http://hhvm.com/frameworks/>)

Список наиболее популярных фреймворков, совместимых с HHVM.

Отслеживание проблем

(<http://bit.ly/fb-hhvm>)

Список известных проблем HHVM.

Часто задаваемые вопросы

(<https://github.com/facebook/hhvm/wiki/FAQ>)

Ознакомьтесь с часто задаваемыми вопросами по HHVM.

Блог

(<http://hhvm.com/blog>)

Следите за последними новостями HHVM.

Установка

Установка HHVM в популярных дистрибутивах Linux довольно проста. Двигок HHVM изначально разрабатывался для Ubuntu (моего любимого дистрибутива Linux), поэтому в следующих примерах я буду использовать Ubuntu.



Facebook предоставляет готовые пакеты для некоторых других дистрибутивов Linux, в том числе для Debian и Fedora. Для прочих дистрибутивов Linux есть возможность собрать HHVM из исходных текстов.

Воспользовавшись инструкциями Facebook (<http://bit.ly/fb-prebuilt>) установить HHVM в последней версии Ubuntu можно с помощью менеджера пакетов Aptitude:

```
wget -O - \
  http://dl.hhvm.com/conf/hhvm.gpg.key | \
  sudo apt-key add -; \
echo deb \
  http://dl.hhvm.com/ubuntu trusty main | sudo tee /etc/apt/sources.list.d/hhvm.list;
sudo apt-get update;
sudo apt-get install hhvm;
```

Если вы станете от этого счастливее, поменяйте последнюю строку на приведенную ниже, чтобы установить последнюю ночную сборку:

```
sudo apt-get install hhvm-nightly;
```

При этом будет добавлен открытый ключ GNU Privacy Guard (GPG) HHVM для проверки пакетов. Он добавит репозиторий пакетов HHVM в локальный список репозиториев. И, наконец, установит HHVM с помощью Aptitude, как любой другой пакет программного обеспечения. Выполняемый файл HHVM устанавливается в `/usr/bin/hhvm`.

Настройка

HHVM использует конфигурационный файл `php.ini` так же, как это делает Zend Engine. Этот файл находится по умолчанию в `/etc/hhvm/php.ini` и содержит множество тех же настроек, что и используемые в Zend Engine. Полный список HHVM-директив `php.ini` можно найти на странице <http://docs.hhvm.com/manual/ini.list.php>.

Если вы запускаете HHVM в качестве сервера FastCGI, добавьте INI-директивы, связанные с сервером, в файл `/etc/hhvm/server.ini`. Полный список серверных директив HHVM можно найти на странице <https://github.com/facebook/hhvm/wiki/INI-Settings>. Страница вики HHVM не вдается в детали, поэтому обратитесь к страницам поддержки HHVM сообществом:

- StackOverflow (<http://stackoverflow.com/questions/tagged/hhvm>);
- Каналы IRC (<http://webchat.freenode.net/?channels=hhvm>);
- Страница в Facebook (<https://www.facebook.com/hhvm>).

Файла `/etc/hhvm/server.ini` по умолчанию должно быть достаточно для начала работы. Его содержание выглядит так:

```
; параметры php
pid = /var/run/hhvm.pid
; параметры hhvm
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
```

Наиболее примечательными настройками являются `hhvm.server.port = 9000` и `hhvm.server.type = fastcgi`, они указывают, что HHVM должен запускаться как сервер FastCGI и прослушивать порт 9000.

Когда запускается выполняемый файл `hhvm`, ему можно передать путь к файлам конфигурации с помощью ключа `-c`. Если `hhvm` используется для запуска сценариев из командной строки, достаточно указать только конфигурационный файл `/etc/hhvm/php.ini`:

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

Если же файл `hhvm` играет роль сервера FastCGI, нужно передать файлы `/etc/hhvm/php.ini` и `/etc/hhvm/server.ini`:

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

Расширения

HHVM не может использовать PHP-расширения, скомпилированные для Zend Engine, если расширения не используют Zend Extension Source Compatibility Layer (<http://bit.ly/ext-zend-comp>) от Facebook. К счастью, большинство PHP-расширений, которыми мы обычно пользуемся, изначально поддерживаются HHVM. Другие PHP-расширения сторонних производителей (например, расширение GeoIP) может быть скомпилировано отдельно и загружено в HHVM как динамическое расширение. Найти список совместимых с HHVM PHP-расширений можно на GitHub (<http://bit.ly/int-extension>).

Мониторинг HHVM с помощью Supervisord

HHVM отличный сервер, но и он не идеален. Я рекомендую управлять мастер-процессом HHVM с помощью монитора процессов Supervisord (<http://supervisord.org/>), который начинает следить за процессом HHVM при загрузке и автоматически перезапускает его при возникновении ошибок.

Установите Supervisord:

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



Если вы еще не знакомы с Supervisord, у Криса Фидао (<http://fideloper.com/>) есть отличное руководство по нему (<http://bit.ly/c-fidaq>).

Затем, убедитесь, что конфигурационный файл `/etc/supervisor/supervisord.conf` содержит следующие две строки:

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

Эти две строки позволяют создать конфигурационные файлы в каталоге `/etc/supervisor/conf.d/` для каждого контролируемого приложения. Далее создайте файл `/etc/supervisor/conf.d/hhvm.conf`:

```
[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
```

Наиболее важными параметрами являются:

command

Supervisord вызывает эту команду, чтобы запустить процесс HHVM. Ключ `-m` позволяет запустить HHVM в режиме сервера. Ключ `-c` используется для определения путей к конфигурационным файлам HHVM `php.ini` и `server.ini`.

autostart

Требует запустить HHVM при запуске Supervisord (например, при загрузке системы).

autorestart

Требует перезапускать процесс HHVM, если он завершился.

startretries

Предельное число попыток перезапустить процесс HHVM, прежде чем прекратить такие попытки.

`user`

Пользователь, владеющий процессом HHVM. В целях безопасности я рекомендую использовать непrivилегированного пользователя. В этом примере, я также использую непrivилегированного пользователя `deploy`, созданного в примере 7.1.



Создайте каталог `/home/deploy/logs` вручную, потому что Supervisord не сделает это за вас.

После редактирования конфигурационных файлов Supervisord запустите следующие две команды, чтобы применить изменения:

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

Получить список процессов, контролируемых Supervisord, можно с помощью команды:

```
sudo supervisorctl
```

С помощью следующих команд можно запустить, остановить или перезапустить программу Supervisord. В этом примере `hhvm` представляет имя программы, указанной в начале файла `/etc/supervisor/conf.d/hhvm.conf`:

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

Итак, мы установили HHVM и контролируем его с помощью Supervisord. Но нам по-прежнему необходим веб-сервер для передачи запросов в HHVM. Напомню, что HHVM запускает сервер FastCGI точно так же, как это описано в главе 7 для PHP-FPM. Мы будем использовать FastCGI с HHVM для обработки PHP-запросов, отправленных сервером nginx.

HHVM, FastCGI и Nginx

HHVM взаимодействует с веб-сервером (например, nginx) по протоколу FastCGI. Для передачи PHP-запросов серверу FastCGI HHVM необходимо создать виртуальный хост nginx. В следующем примере приходится определение виртуального хоста nginx:

```
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 $request_filename;
        fastcgi_pass 127.0.0.1:9000;
    }
}
```



С этого момента, я предполагаю, что nginx установлен и работает. Инструкции по установке nginx приводятся в главе 7.

Предполагая, что вы следовали инструкциям по установке nginx, приведенным в главе 7, создайте файл `/home/deploy/apps/example.com/current/public/index.php` со следующим содержимым:

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

Убедитесь, что домен `example.com` соответствует IP-адресу вашего сервера, и перейдите на страницу <http://example.com/index.php> в браузере. Вы должны увидеть слово «HipHop» в окне браузера.



Определить соответствие любого доменного имени любому IP-адресу можно, внеся изменения в локальный файл `/etc/hosts`. Например, следующая строка ставит в соответствие доменному имени `example.com` IP-адрес `192.168.33.10`:

```
192.168.33.10 example.com
```

Мои поздравления! Вы установили HHVM как сервер FastCGI, который сможет запускать PHP-приложения. Но это не самое главное. А знаете, что самое главное? Язык Hack. HHVM может работать и с ним тоже.

Язык Hack

Hack (<http://hacklang.org/>) – это серверный язык, очень похожий на PHP. Разработчики Hack даже называют Hack диалектом PHP. Зачем Facebook создал что-то, так похожее на PHP? Facebook создал язык Hack по некоторым причинам. Язык Hack добавляет новые удобные структуры данных и интерфейсы, недоступные в языке PHP. И, что более важно, Hack вводит *статическую типизацию*, чтобы помочь писать более предсказуемой и стабильной код. Статическая типизация обеспечивает раннее выявление ошибок в процессе разработки.

Стоят ли новые структуры данных, интерфейсы, и статическая типизация времени, потраченного на изучение новшеств набора инструментов языка? Может быть. Но помните, что Facebook есть Facebook. В нем тысячи разработчиков работают над созданием гигантской кодовой базы. Если Facebook сможет оптимизировать даже малую часть процесса разработки, он дважды пожинает плоды, повысив эффективность работы разработчиков и стабильность выполнения кода.

Я не рекомендую, отложив все дела, немедленно браться за перевод существующих PHP-приложений на Hack. Однако, если вы начинаете работу над новым проектом и у вас есть время для установки и изучения языка Hack, то почему бы и нет. Вы, безусловно, выиграете от наличия новых структур данных и статической типизации.

Перевод с PHP на Hack

Чтобы преобразовать код на языке PHP в код на языке Hack, замените <?php на <?hh. Это все. Это код на языке PHP:

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

А это эквивалентный код на языке Hack:

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

Facebook сделал переход от языка PHP к языку Hack очень простым, потому что понимает, что преобразование большого объема существующего кода не станет быстро решаемой задачей. Начните миграцию кодовой базы со смены <?php на <?hh. Затем, добавьте несколько статических типов. Потом, примените некоторые новые структуры данных языка Hack. Переходите на язык Hack постепенно и безболезненно, не выбиваясь из графика разработки.

Что такое типы?

Прежде, чем перейти к сравнению динамической и статической типизации, будет нeliшним дать определение понятию *типов*. Большинство PHP-программистов полагают, что тип определяется видом данных, присваиваемых переменной. Например, выражение `$foo = "bar"` указывает, что значение переменной `$foo` является строкой. Выражение `$bar = 14` указывает, что значение переменной `$bar` является целым числом. Да, эти примеры демонстрируют типы, но они не передают полного определения типов.

Tip – это некая этикетка, которая присваивается свойствам приложения для подтверждения, что их поведение должно соответствовать существующим нормам и требованиям. Я перефразировал превосходное определение Криса Смита (Chris Smith) типов в программировании (<http://bit.ly/prog-types>).

Определения типов можно расширить, добавив синтаксические аннотации, которые уточняют идентичность переменных, аргументов или возвращаемых значений. Аннотации типов (или *подсказки*) используются как в PHP, так и в Hack. Вы, наверное, встречали код, подобный следующему:

```
<?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];
    }
}
```

Это произвольный пример, но он использует синтаксические подсказки для определенных свойств приложения. Например, в описании метода `addWidget()` используется подсказка `Widget` перед аргументом `$widget`, чтобы сообщить PHP, что аргументом метода будет экземпляр класса `Widget`. Интерпретатор PHP проверяет это требование. Если аргумент не является экземпляром класса `Widget`, код вызовет ошибку. В этом примере тип объявляет наши ожидания: метод `addWidget()` получит аргумент только класса `Widget`.

Приведенные выше простые примеры (например, `$foo = "bar"`) и пример `WidgetContainer` демонстрируют применение типов. В первом примере переменная имеет строковый тип, хотя явно это нигде не указывается. PHP достаточно умен, чтобы *вывести* тип переменной как строковый, основываясь на синтаксисе кода. Во втором примере тип задается с помощью аннотации, которая явно определяет требование метода `addWidget()`, и PHP обеспечивает соблюдение этого требования, основанного на явной подсказке, а не на предположении. Типы программы выполняют функцию тестов, гарантируя более предсказуемое выполнение.



Типы являются чем-то большим, чем предполагаемые идентичности и аннотации. Тем не менее, с этими двумя их проявлениями вы будете сталкиваться наиболее часто при написании кода на PHP и Hack. Больше узнать о типах в программировании можно в книге Бенджамина С. Пирса (Benjamin C. Pierce) «*Types and Programming Languages*»¹ (<http://bit.ly/pl-pierce>).

¹ Пирс Б. «Типы в языках программирования», ISBN: 978-5-7913-0082-9, 2014, Добросвет. – Прим. ред.

Если вы думаете, что подсказки в PHP являются статическими типами, я сейчас развею ваше заблуждение. И статические, и динамические типы помогают писать предсказуемый код, но они используют разные системы проверки типов. Основное различие между статическими и динамическими типами в том, когда проверяются типы и как проверяется корректность программы.

Статическая типизация

Предсказуемое поведение программ со статической типизацией основывается на применении в коде подсказок, аннотаций или других определений типов, в зависимости от конкретного языка. Если

программа со статической типизацией компилируется успешно, можно быть уверенным, что она будет выполняться должным образом.

Заметили, что я использовал слово *компиляция*? Языки со статической типизацией обычно компилируются. Проверка типов и сообщения об ошибках вменяются в обязанность компилятору языка. И это неплохо, потому что компилятор выявляет ошибки программы, связанные с типами, во время компиляции, до ввода приложения в эксплуатацию. К сожалению, компилируемые языки предполагают длительный цикл обратной связи. Чтобы выявить ошибки программа должна быть скомпилирована, а сложные программы требуют много времени на компиляцию. Это замедляет разработку.

Достоинством статической типизации является повышенная стабильность, потому что при компиляции программы проходят дополнительную проверку. Тем не менее, мы все равно должны писать тесты для проверки *правильности* работы программы. Если программа компилируется, это означает, что программа выполнит свой код. Но это не означает, что ее поведение будет соответствовать нашим ожиданиям. То есть, статическая типизация делает ненужным лишь связанные с типами модульные тесты, необходимые для программ с динамической типизацией.

Динамическая типизация

В отличие от статической типизации, динамическая типизация не может обеспечить контроль кода во время компиляции, потому что типы не проверяются до момента выполнения программы. Программы с динамической типизацией обычно интерпретируются. PHP – это интерпретируемый язык с динамической типизацией. Это означает, что при *каждом* запуске PHP-сценария из командной строки или через веб-сервер, интерпретатор читает PHP-код, преобразует в набор байт-кодов и выполняет.

Как же найти ошибки, если код на PHP не компилируется? Ошибки возникают во время выполнения. Это одновременно и хорошо, и плохо. Хорошо, потому что ускоряется обновление кода. Написанный код можно сразу запустить. Обратная связь практически мгновенна. Но, к сожалению, теряется достоверность и предсказуемость, присущие статической проверке типов. Большое значение приобретают модульные тесты, проверяющие соответствие типов и результатов выполнения. Тесты должны охватывать всевозможные варианты выполнения. Это срабатывает в ситуациях, когда можно предвидеть ошибки, а что делать в непредвиденных ситуациях. Для непредвиден-

ных ситуаций, приводящих к ошибкам PHP, мы должны предусмотреть вывод понятных сообщений и их журналирование.

Двойной подход языка Hack

Статическая типизация является сильнейшим аргументом в пользу языка Hack. Но еще более интересной его особенностью является поддержка статической и динамической типизации. Напомню, что Hack в основном совместим с обычным языком PHP. А это значит, что Hack поддерживает все нюансы динамической типизации PHP. Это возможно благодаря использованию JIT-компилятора HHVM в Hack. Hack проверяет типы, если они явно указаны. HHVM читает код Hack, оптимизирует его и кэширует промежуточный байт-код. Файл с кодом на Hack превращается в машинный код x86_64 и выполняется по требованию. Он выбирает в себя лучшее из обоих подходов. Мы получаем точность и безопасность статической типизации с автономной проверкой типов Hack (подробнее об этом ниже), и гибкость и быстроту обновления динамической типизации благодаря JIT-компилятору HHVM.



Имеются несколько особенностей языка PHP, не поддерживаемых языком Hack. Они перечислены на странице <http://docs.hhvm.com/manual/hack.unsupported.php>. Эти особенности поддерживаются HHVM только при выполнении обычного PHP-кода.

Контроль типов в Hack

Hack поддерживает автономную проверку типов, которая выполняется в фоновом режиме и контролирует типы *в режиме реального времени*. Это огромное преимущество. В нем состоит главная причина, почему Facebook создал язык Hack. Мгновенная проверка типов Hack обеспечивает точность и безопасность статической типизации без длительного цикла обратной связи. Если вы используете Hack, но не пользуетесь его проверкой типов, то вы неправильно его используете.

Рассмотрим настройку механизма проверки типов Hack для приложения. Во-первых, я предполагаю, что виртуальная машина HHVM уже установлена и запущена. Если нет – обращайтесь к инструкциям по установке в разделе «HHVM» выше. Затем создайте пустой файл с именем `.hhconfig` в корневом каталоге проекта. Он укажет механизму

проверки типов Hack каталог для анализа. Механизм проверки типов будет отслеживать состояние файлов в этом каталоге и запускать проверку соответствующих файлов при каждом обнаружении изменений в файловой системе. Запустите проверку типов Hack, выполнив команду `hh_client` в корневом каталоге проекта или в одном из его подкаталогов.

Утилита проверки типов Hack имеет несколько ограничений. Согласно электронной документации для языка Hack (<http://bit.ly/hack-hhvm>):

Утилита проверки типов предполагает наличие глобального автозагрузчика, способного загружать любой класс по требованию. Это значит, что все имена классов и функций должны быть уникальными. Она не имеет ни малейшего понятия о проверке импорта или чего-нибудь в этом роде. Более того, она не поддерживает условных определений функций и классов – она должна быть в состоянии статически определить, что имеется, а что отсутствует. Конечно, вполне возможно существование проекта, который отвечал бы этим требованиям и не имел глобального автозагрузчика, и утилита проверки типов будет прекрасно работать с таким проектом, но проекту с автозагрузчиком отдается предпочтение.

Утилита проверки типов не поддерживает смешивание кода на Hack с разметкой HTML. Отслеживание и статический анализ этих сложных переключений не поддерживается, в частности потому, что большая часть современных программ не использует такую возможность. Код на Hack может выводить разметку в браузер с помощью `echo`, если вывод достаточно прост, или с помощью механизмов шаблонов или XHP в более сложных сценариях.

Режимы Hack

Код на Hack можно писать в трех режимах: `strict`, `partial` или `decl`. Начиная новый проект на языке Hack я рекомендую использовать режим `strict`. Если вы переводите существующий PHP-код на Hack или ваш проект использует код на обоих языках, используйте режим `partial`. Режим `decl` позволяет интегрировать унаследованный PHP-код без типизации в кодовую базу Hack-кода в режиме `strict`. Режим объявляется в начале файла, сразу после открывающего тега (как показано в следующих примерах). Имена режимов чувствительны к регистру букв:

```
<?hh // strict
```

Режим `strict` требует надлежащего аннотирования всего кода. Утилита проверки типов Hack будет отлавливать все возмож-

ные ошибки, связанные с типами. Также этот режим запрещает включение постороннего кода (например, унаследованного PHP-кода) в Hack-код. Ознакомьтесь с аннотациями типов в Hack, прежде чем переходить в режим strict. Среди прочих требований, все массивы в Hack должны быть типизированы, вы не сможете использовать массивы, без типизации. А также необходимо аннотировать возвращаемые типы для функций и методов.

```
<?hh // partial
```

Режим `partial` (выбран по умолчанию) позволяет использовать PHP-код, без преобразования его в Hack-код. Режим `partial` также не требует аннотирования *всех* аргументов функций или методов. Допускается аннотировать часть аргументов, не вызвав неудовольствие утилиты проверки типов Hack. Если вы еще только знакомитесь с Hack или хотите конвертировать существующий PHP-код, этот режим, пожалуй, лучше всего подойдет вам.

```
<?php // decl
```

Режим `decl` позволяет Hack-коду в режиме `strict` вызывать код, не использующий типизацию. Часто Hack-код использует унаследованный PHP-класс без типизации. В этом случае, унаследованный PHP-код должен быть объявлен в режиме `decl`, чтобы новый Hack-код смог использовать его.

Синтаксис Hack

Hack поддерживает аннотации типов для свойств классов, аргументов методов и типов возвращаемых значений. Эти аннотации проверяются автономной утилитой проверки типов в соответствии с режимом каждого файла.



Полный список доступных аннотаций типов можно найти на странице http://docs.hhvm.com/manual/hack_annotations.types.php

Вернемся к нашему примеру `WidgetContainer` и добавим аннотации типов. Код на Hack после внесения изменений будет выглядеть следующим образом:

```
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. }
```

Аннотации свойств

В строке 4 объявлено свойство `$widgets` с аннотацией `Vector<Widget>`. Эта аннотация определяет следующее:

- Свойство имеет тип `vector` (<http://bit.ly/vector-tv>) (аналог массива с числовыми индексами).
- Свойство должно содержать только экземпляры класса `Widget`.

Аннотации аргументов

Эта аннотация уже знакома тем, кто пользовался подсказками типов в PHP. Аргумент метода `__construct()` в строке 6 объявлен с аннотацией `array<Widget>`. Такая аннотация определяет следующее:

- Аргумент должен быть массивом.
- Аргумент должен содержать только экземпляры `Widget`.

В отличие от аннотации свойства в строке 4, эта аннотация допускает передачу в аргументе массива с числовыми индексами или ассоциативного массива. Далее выполняется обход элементов массива

в цикле и их значения добавляются в структуру данных `Vector`. Чтобы аргумент мог быть только массивом с числовыми индексами или только ассоциативным массивом, следовало бы использовать аннотацию `array<int, Widget>` или `array<string, Widget>`, соответственно.

Аннотации типов возвращаемых значений

В строках 13 и 20 определены типы возвращаемых методами значений. Метод `addWidget()` возвращает свой экземпляр класса (подробнее об этом ниже). Метод `getWidget()` возвращает экземпляр класса `Widget`. Аннотации типов возвращаемых значений помещаются *после* закрывающей скобки сигнатуры метода и *до* открывающей скобки тела метода.



Исключением из правил является метод `__construct()`. Можно предположить, что конструктор возвращает значение типа `void`. Но это не так. Нельзя применять аннотацию типа к значению, возвращаемому конструктором.

Некоторым разработчикам нравится использовать *цепочки методов*. Если метод класса возвращает свой экземпляр класса, можно по цепочке вызвать сразу несколько методов, например:

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

Для этого в Hack имеется аннотация `this` типа возвращаемого значения. Мы использовали такую аннотацию в методе `addWidget()` в строке 13 предыдущего примера.

Структуры данных Hack

Статическая типизация является широко разрекламированной особенностью. Но вместе с тем Hack предоставляет новые структуры данных и интерфейсы, которых нет в языке PHP. Они способны сократить время разработки, если учесть поиск обходных путей для их реализации в обычном языке PHP. Ниже перечислены некоторые из новых структур данных и интерфейсов языка Hack:

- Коллекции: векторы, отображения, множества и соответствия (<http://docs.hhvm.com/manual/en/hack.collections.php>)
- Обобщения (<http://docs.hhvm.com/manual/en/hack.generics.php>)

- Перечисления
(<http://docs.hhvm.com/manual/en/hack.enums.php>)
- Формы
(<http://docs.hhvm.com/manual/en/hack.shapes.php>)
- Кортежи
(<http://docs.hhvm.com/manual/en/hack.tuples.php>)

Многие из этих структур данных дополняют, уточняют и расширяют возможности языка PHP. Например, коллекции в Hack устраниют неопределенность массивов PHP. Обобщения позволяют создавать структуры данных для обработки однородных значений заданного типа, который определяется в момент создания экземпляра обобщенного класса. Они позволяет обходиться без проверок типов внутри класса методом `instanceof` PHP. Перечисления полезны для создания набора именованных констант без помощи абстрактных классов. Формы помогают контролировать типы структур данных, которые должны иметь фиксированный набор ключей. А кортежи позволяют использовать массивы фиксированной длины.

Только не подумайте, что я призываю вас сразу же приступить к использованию всех этих новых структур данных. Признаюсь, некоторые из них имеют весьма ограниченную и специфичную область применения. Некоторые из этих структур дублируют (и расширяют) возможности других структур. Я рекомендую познакомиться с новыми структурами данных, но использовать их только там, где они действительно необходимы.



Лично я самыми полезными из всех структур данных в языке Hack считаю коллекции. Они ведут себя более адекватно и предсказуемо, чем массивы PHP. Замена массива PHP одной из коллекций, я полагаю, будет хорошим решением.

HHVM и Hack против PHP

Если HHVM и Hack представляют собой настолько превосходное решение, то стоит ли вообще продолжать использовать PHP? Я часто и многим задаю этот вопрос. А также спрашиваю, сколько, по их мнению, продержится еще язык PHP. Ответы неоднозначны, нет четкого разделения на черное и белое. Скорее цвет ответов нейтрально серый.

HHVM стал первым реальным конкурентом Zend Engine, традиционной среде языка PHP. Сравнительные тесты PHP 5.x показали, что HHVM обеспечивают лучшую производительность и более эффективно использует память, чем Engine Zend. Я думаю, что это известие застало команду разработчиков ядра PHP врасплох. Фактически, появление HHVM стало причиной возобновления интереса разработчиков PHP к повышению производительности и снижению затрат памяти. В настоящее время уже идут работы над PHP 7, который планируется выпустить в конце 2015 года PHP 7 обещает стать достойным конкурентом или даже превзойти HHVM. Оправдаются эти ожидания или нет, можно только догадываться. Однако, суть в том, что появление HHVM создало конкуренцию, а конкуренция помогает обеим сторонам. И HHVM и Zend Engine продолжат развиваться, а PHP-разработчики от этого только выигрывают. Ни HHVM, ни Zend Engine не собираются почивать на лаврах или признавать себя побежденными. Я думаю, что они еще долго будут сосуществовать и подпитывать друг друга энергией конкуренции.

Язык Hack, на мой взгляд, значительно превосходит язык PHP. И на это есть несколько причин. Во-первых, язык Hack был создан в Facebook для решения реальных проблем. Он конкретно ориентирован. У него есть цель. Его разработку не курировал комитет. Язык PHP, напротив, разрабатывался по частям в течение длительного времени. PHP призван удовлетворять различные потребности, и он находится под контролем комитета, члены которого не склонны к единогласным решениям. По сравнению с PHP 5.x, язык Hack обладает строгой проверкой типов данных и поддерживает унаследованный от PHP код. Я считаю, что лучшие особенности Hack в конечном итоге будут перенесены в PHP. И наоборот. Действительно, команда разработчиков языка Hack объявила, что намерена сохранить совместимость с Zend Engine. Опять же, я считаю, что конкуренция приведет к улучшению обоих языков, и они будут сосуществовать в симбиозе.

Примером такого симбиоза является официальная спецификация языка PHP. До недавнего времени, язык PHP был языком Zend Engine из-за отсутствия альтернативных реализаций. Появление HHVM подвело нескольких разработчиков Facebook на анонсирование спецификации языка PHP (<http://bit.ly/fb-spec>). Эта спецификация гарантирует поддержку нынешними и будущими реализациями PHP (Zend Engine, HHVM и так далее) фундаментальных особенностей языка и стала значимым событием для PHP-сообщества.



Информацию об официальной спецификации PHP можно найти на GitHub: <https://github.com/php/php-lang-spec>.

Дополнительные материалы

Мы лишь слегка затронули огромную тему HHVM и языка Hack. В этой книге просто не хватит страниц, чтобы описать все предлагаемые этими двумя инициативами новшества. Вместо этого, я приведу ссылки на полезные ресурсы по этой теме:

- <http://hhvm.com>
- <http://hacklang.org>
- @ptarjan в Twitter
- @SaramG в Twitter
- @HipHopVM в Twitter
- @HackLang в Twitter

ГЛАВА 13.

Сообщество

Самым лучшим помощником в освоении языка PHP должно стать PHP-сообщество. Оно разнообразно, изменчиво и глобально. Я рекомендую присоединиться к PHP-сообществу, чтобы получать информацию и делиться ею с другими PHP-разработчиками. Там всегда найдется чему поучиться и ваше активное участие в PHP-сообществе является лучшим способом продолжить обучение. Это также отличный способ для общения и оказания помощи другим разработчикам.

Местная группа PHP-разработчиков

Советую найти местную группу PHP-разработчиков (PHP User Group или PUG) и присоединиться к ней. Такие группы существуют во многих городах. Найти свою локальную группу PUG можно на странице <http://phpug>. Присоединение к локальной группе PUG станет лучшей возможностью познакомиться и пообщаться с другими PHP-разработчиками из вашего местного сообщества.

Если поблизости не окажется PUG, в запасе остается еще несколько вариантов действий. Вы сами можете организовать PUG. Если вы не живете в джунглях, я уверен, что где-то поблизости найдутся PHP-разработчики, которые хотели бы присоединиться к PUG. В противном случае, вы можете присоединиться к NomadPHP (<https://nomadphp.com/>) – группе пользователей Интернета с ежемесячными форумами и непрерывным оперативным обменом информацией по всем вопросам, касающимся языка PHP.

Конференции

Ежегодно проходит множество конференций посвященных PHP. Конференции дают прекрасную возможность познакомиться и пообщаться с лучшими специалистами PHP-сообщества. Вы можете вы-

слушать мнения других разработчиков и высказать свое. Вы будете в курсе самых актуальных возможностей разработки и самых современных методик. Конференции также послужат неплохим поводом для небольшого отпуска. Список предстоящих конференций по PHP можно найти по адресу <http://php.net/conferences/>.

Наставничество

Если вы новичок в PHP и вам необходим совет или помощь, вы можете найти себе наставника на сайте <http://phpmantoring.org>. Многие опытные PHP-разработчики готовы пожертвовать своим временем, чтобы помочь новичкам. А если вы сами являетесь опытным PHP-разработчиком, подумайте о том, чтобы записаться в наставники. Имеется много начинающих PHP-разработчиков, которые не знают, как и с чего начать, и ваше наставничество будет иметь для них неоценимое значение.

Будьте в курсе

Язык PHP часто меняется. Ниже приводится список ресурсов, которые помогут вам оставаться в курсе новых возможностей языка PHP и современных методик.

Сайты

- <http://php.net>
- <http://php.net/docs.php>
- <http://www.php-fig.org>
- <http://www.phptherightway.com>

Списки рассылок

- <http://php.net/mailing-lists.php>

Твиттер

- @official_php
- @phpc

Подкасты

- <http://voicesoftheelephant.com>
- <http://looselycoupled.info>

- <http://elephantintheroom.io>
- <http://phptownhall.com>
- <http://devhell.info>
- <http://www.phpclasses.org/blog/category/podcast/>
- <http://threedevsandaliamaybe.com/>

Юмор

- @phpbard
- @phpdrama

ПРИЛОЖЕНИЕ А.

Установка PHP

Linux

Linux – моя любимая среда разработки. У меня есть ноутбук Macbook Pro с операционной системой OS X, но разработку я веду в виртуальной машине Linux. PHP легко устанавливается в Linux с помощью менеджера пакетов, такого, например, как `aptitude` в Ubuntu Server или `yum` в CentOS.

Здесь мы рассмотрим только установку PHP для работы из командной строки. Настройку PHP-FPM и веб-сервера nginx мы уже рассматривали в главе 7.

Менеджеры пакетов

Большинство дистрибутивов Linux имеет собственные менеджеры пакетов. Например, в Ubuntu есть `aptitude`. В CentOS и Red Hat Enterprise Linux (RHEL) – `yum`. Менеджер пакетов – это удобный инструмент поиска, установки, обновления и удаления программного обеспечения в операционной системе Linux.



Некоторые менеджеры пакетов Linux устанавливают не самые последние версии программного обеспечения. Например, Ubuntu 14.04 LTS предлагает установить PHP 5.5.9, а это уже не последняя версия PHP 5.6.3 (по состоянию на декабрь 2014 года).

К счастью, есть возможность дополнить источники программного обеспечения, используемые менеджером пакетов по умолчанию, репозиториями сторонних производителей, содержащими более свежие версии программных пакетов. Мы используем дополнительный репозиторий программного обеспечения для Ubuntu и CentOS с самой последней версией PHP. Прежде чем двигаться дальше, зареги-

стрируйтесь с привилегиями суперпользователя `root` или получите эти привилегии с помощью `sudo`. Это необходимо для установки программного обеспечения с помощью менеджеров пакетов Linux.

Ubuntu 14.04 LTS

Ubuntu не содержит последней версии PHP в своих репозиториях по умолчанию. Поэтому необходимо добавить поддержку персональных архивов пакетов (Personal Package Archive или PPA). Термин PPA используется только в Ubuntu, но идея везде одна: использование репозитория программного обеспечения от сторонних производителей для расширения выбора программного обеспечения, предоставляемого Ubuntu по умолчанию. Ондржей Сури (Ondřej Surý) поддерживает отличный PPA с самой последней стабильной версией PHP. Этот PPA называется `ppa:ondrej/php5-5.6`.

1. Добавление программных зависимостей

Прежде, чем добавить PPA Ондрея Сури, необходимо убедиться, что в операционной системе имеется выполняемый файл `add-apt-repository`. Этот выполняемый файл входит в пакет `python-software-properties`. Введите следующую команду в окне терминала и нажмите **Enter**. Введите свой пароль, если потребуется:

```
sudo apt-get install python-software-properties
```

Эта команда установит пакет Python Software Properties, включающий выполняемый файл `add-apt-repository`. Теперь можно добавить любой PPA.

2. Добавление PPA `ppa:ondrej/php5-5.6`

Этот PPA расширяет доступный выбор программного обеспечения в Ubuntu за пределы репозиториев, доступных в Ubuntu по умолчанию. Введите следующую команду в окне терминала и нажмите **Enter**. Введите свой пароль, если потребуется:

```
sudo add-apt-repository ppa:ondrej/php5-5.6
```

Эта команда добавит PPA Ондрея Сури в список источников программного обеспечения Ubuntu. Она также загрузит открытый GPG-ключ PPA и добавит его в локальный набор GPG-ключей. Открытый GPG-ключ позволит Ubuntu удостовериться, что пакеты в PPA не были подменены и были собраны и подписаны их первоначальным автором.

Ubuntu кэширует список всех доступных программ. При добавлении новых источников программного обеспечения, необходимо обновить кэш Ubuntu. Введите следующую команду в окне терминала и нажмите **Enter**. Введите свой пароль, если потребуется:

```
sudo apt-get update
```

3. Установка PHP

Теперь можно использовать менеджер пакетов `aptitude` для установки последней стабильной версии PHP из PPA Ондрея Сури. Но прежде важно узнать, какие PHP-пакеты будут доступны и что они делают. PHP распространяется в двух формах. Одной из форм является пакет CLI, который позволяет использовать PHP в командной строке (мы будем использовать этот пакет). Есть несколько других пакетов PHP, которые объединяют PHP с веб-серверами Apache или nginx (мы рассмотрели их в главе 7). Здесь мы выберем пакет PHP CLI.

Сначала установим пакет PHP CLI. Введите следующую команду в окне терминала и нажмите **Enter**. Введите свой пароль, если потребуется:

```
sudo apt-get install php5-cli
```

Менеджер пакетов Linux также содержит пакеты для PHP-расширений, которые могут быть установлены отдельно. Установим некоторые из них прямо сейчас. Введите следующую команду в окне терминала и нажмите **Enter**. Введите свой пароль, если потребуется:

```
sudo apt-get install php5-curl php5-gd php5-json php5-mcrypt  
php5-mysqlnd
```

Удостоверьтесь, что PHP успешно установлен, выполнив команду:

```
php -v
```

В результате в окне терминала должен появиться примерно следующий вывод:

```
PHP 5.5.11-3+deb.sury.org-trusty+1 (cli) (built: Apr 23 2014 12:15:16)  
Copyright (c) 1997-2014 The PHP Group  
Zend Engine v2.5.0, Copyright (c) 1998-2014 Zend Technologies  
with Zend OPcache v7.0.4-dev, Copyright (c) 1999-2014,  
by Zend Technologies
```

CentOS 7

Как и Ubuntu, CentOS и RHEL содержат не самую последнюю стабильную версию PHP в своих репозиториях по умолчанию. RHEL очень придирчиво подходит к программному обеспечению, входящему в официальный дистрибутив, потому что гордится собственным высоким уровнем безопасности и стабильности, поэтому обновления программного обеспечения добавляются в официальный дистрибутив с большой задержкой.

Мы не входим в 500 компаний рейтинга журнала Fortune, поэтому можем себе позволить установку последней стабильной версии PHP в дистрибутив CentOS/RHEL Linux. Чтобы сделать это, используем репозиторий EPEL (*Extra Packages for Enterprise Linux*, <https://fedoraproject.org/wiki/EPEL>). EPEL характеризует себя следующим образом:

... Fedora Special Interest Group, которая создает, поддерживает и управляет высококачественными наборами дополнительных пакетов для Enterprise Linux, в том числе, Red Hat Enterprise Linux (RHEL), CentOS, Scientific Linux (SL) и Oracle Enterprise Linux (OEL).

Репозиторий EPEL не имеет отношения к официальным дистрибутивам CentOS/RHEL Linux, но может быть добавлен к репозиториям CentOS/RHEL по умолчанию. А это именно то, что мы и собираемся сделать.

1. Добавление репозитория EPEL

Сообщим системе CentOS/RHEL, что она должна использовать репозиторий EPEL. Введите следующие команды в окне терминала по очереди, нажимая клавишу **Enter** после каждой команды. Введите свой пароль, если потребуется:

```
sudo rpm -Uvh \
http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-5.noarch.rpm;
```

```
sudo rpm -Uvh \
http://rpms.famillecollet.com/enterprise/remi-release-7.rpm;
```

Эти команды добавят сторонние репозитории EPEL и remi к репозиториям системы CentOS/RHEL. После их выполнения в каталоге */etc/yum.repos.d* должны появиться файлы *epel.repo* и *remi.repo*.

2. Установка PHP

Теперь установим последнюю версию PHP из репозиториев EPEL и remi. Как упоминалось выше, в описании установки PHP в Ubuntu, PHP распространяется в двух формах. Одной из них является пакет CLI, позволяющий использовать PHP в командной строке.

Сначала установим пакет PHP CLI. Введите следующую команду в окне терминала и нажмите **Enter**. Введите свой пароль, если потребуется.

```
sudo yum -y --enablerepo=epel,remi,remi-php56 install php-cli
```

Далее, установим несколько дополнительных PHP-расширений. Получить полный список PHP-расширений можно с помощью менеджера пакетов `yum`. Введите следующую команду в окне терминала и нажмите **Enter**:

```
yum search php
```

Получив список PHP-расширений, установите их, как показано в следующем примере. Имена ваших пакетов могут отличаться:

```
sudo yum -y --enablerepo=epel,remi,remi-php56 \
install php-gd php-mbstring php-mcrypt php-mysqlnd php-opcache
php-pdo
```

Обратите внимание на ключ `--enablerepo`. Он указывает `yum` установить пакеты из репозиториев EPEL, remi и remi-php56. Без этого ключа `yum` будет использовать только репозитории по умолчанию.

Удостоверьтесь в успешной установке PHP. Введите следующую команду в окне терминала и нажмите **Enter**:

```
php -v
```

В результате должен появиться вывод примерно следующего:

```
PHP 5.6.3 (cli) (built: Nov 16 2014 08:32:30)
Copyright (c) 1997-2014 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2014 Zend Technologies
    with Zend OPcache v7.0.4-dev, Copyright (c) 1999-2014,
        by Zend Technologies
```

OS X

OS X устанавливает PHP «из коробки», но это, обычно, не последняя версия, и она, как правило, не содержит нужных PHP-расширений. Я советую игнорировать версию PHP, поставляемую вместе

с OS X, и использовать одну из сторонних сборок PHP. Существует много способов установки PHP в OS X, но я рекомендую два способа: MAMP и Homebrew.

MAMP

MAMP – лучший способ установить PHP в OS X, если вас пугает работа в командной строке. MAMP (аббревиатура Mac, Apache, MySQL и PHP) обеспечивает традиционный набор программного обеспечения для веб-разработки, включающий себя веб-сервер Apache, сервер баз данных MySQL и PHP. MAMP – это приложение OS X с графическим интерфейсом. Многие пользователи предпочитают привычный графический интерфейс, потому что он обеспечивает удобный метод «наведи и щелкни» при установке и настройке программного обеспечения MAMP (рис. А.1). Приложение MAMP находится в каталоге */Applications*, и для его запуска нужно дважды щелкнуть на иконке приложения. У него имеется обычный инсталлятор пакетов (.pkg) OS X, что значительно упрощает его установку и использование. Его даже можете перетащить в Dock OS X для быстрого доступа.

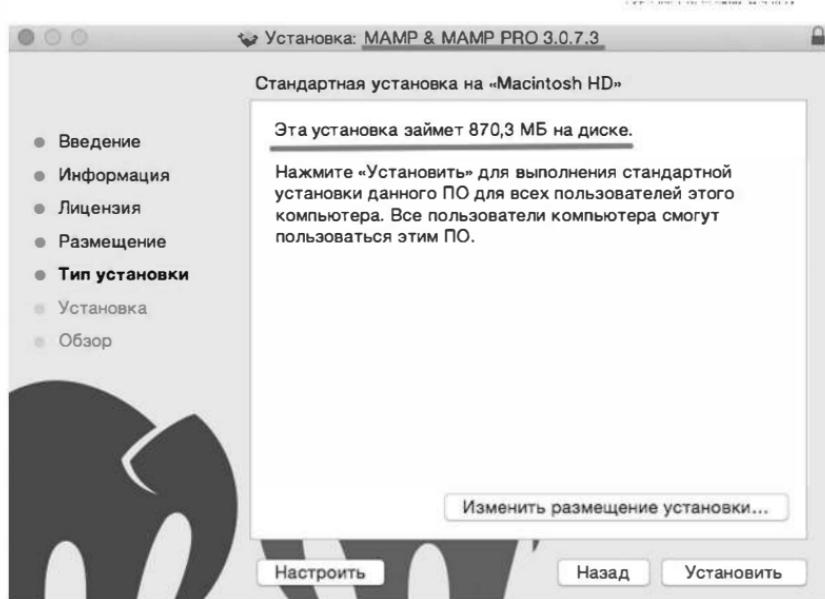


Рис. А.1. Установка MAMP

Установка

Загрузите инсталлятор пакета MAMP (*.pkg*) со страницы <http://www.mamp.info>, и запустите двойным щелчком. Следуйте инструкциям на экране.

Когда установка MAMP завершится, найдите приложение MAMP в каталоге */Applications* и запустите двойным щелчком на значке приложения. После запуска MAMP, щелкните на кнопке **Start Servers** (Запустить серверы), чтобы запустить серверы Apache и MySQL (рис. А.2). Это действительно очень просто.



Рис. А.2. Интерфейс MAMP

«*А как же PHP?*», – спросите вы. MAMP встраивает PHP в веб-сервер Apache в виде модуля `mod_php`. Не вдаваясь в излишние подробности, можно сказать, что PHP автоматически будет использоваться после запуска веб-сервера Apache. Порядок развертывания PHP был описан в главе 7.

После запуска серверов Apache и MySQL, откройте веб-браузер и перейдите по адресу <http://localhost:8888>. Если установка MAMP прошла успешно, вы увидите страницу приветствия MAMP.

Обычно веб-сервер Apache использует порт 80. Но MAMP выбирает для сервера Apache порт 8888. Аналогично, MySQL обычно прослушивает порт 3306. Но MAMP определяет для MySQL порт 8889. Вы можете изменить порты MAMP по умолчанию в настройках приложения MAMP. Корневым каталогом документов веб-сервера Apache является */Applications/MAMP/htdocs*. Любые PHP-файлы в этом каталоге доступны браузеру по адресу: <http://localhost:8888>.

Если вы собираетесь часто использовать MAMP, перейдите в настройки приложения MAMP (рис. А.3) и установите флажок **Start Servers when starting MAMP** (Запускать серверы при запуске MAMP). Затем добавьте приложение MAMP во вкладке **Login Items**, в настройках вашей учетной записи OS X. Благодаря этому серверы Apache и MySQL будут запускаться автоматически после входа в OS X.

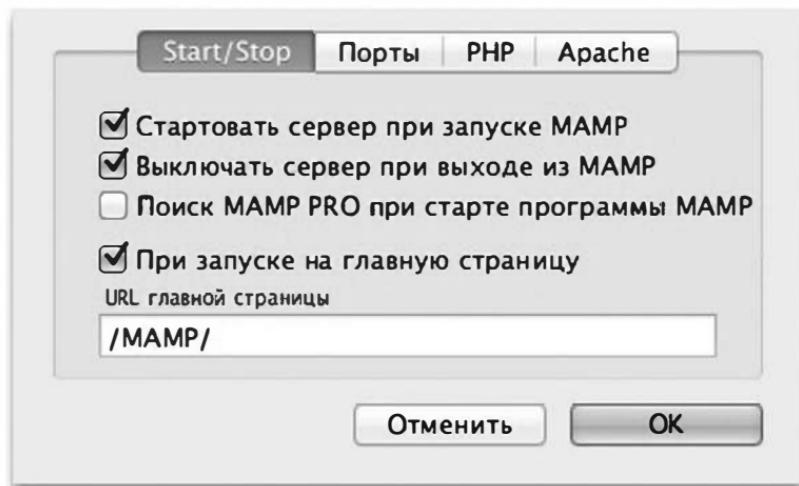


Рис. А.3. Настройка приложения MAMP

Расширения

Имеется возможность загрузки дополнений MAMP для локальной установки нескольких разных версий PHP. MAMP часто обновляется и, скорее всего, поставляется в комплекте с последней версией PHP. Но

если по какой причине это не так или если вам понадобится старая версия PHP, перейдите на сайт MAMP и загрузите нужную версию PHP.

Ограничения

Бесплатная версия MAMP обеспечивает только один виртуальный хост Apache и не предусматривает инструментов простого изменения конфигурации PHP и PHP-расширений. MAMP содержит только самое основное и самое необходимое для разработки на PHP под OS X.

Имеется и платная версия MAMP «Pro», позволяющая создавать несколько виртуальных хостов Apache, упрощающая редактирование конфигурационного файла `php.ini` и настройку PHP-расширений. MAMP Pro прекрасная вещь, не поймите меня превратно. Но вместо траты денег на приобретение MAMP Pro, полезней изучить несколько базовых операций командной строки, что даст вам возможность использовать прекрасный менеджер пакетов Homebrew (<http://brew.sh/>).

Homebrew

Homebrew (<http://brew.sh/>) – это менеджер пакетов OS X, аналог менеджеров пакетов `aptitude` в Ubuntu и `yum` в RHEL. Homebrew обеспечивает просмотр, поиск, установку, обновление и удаление любых пакетов программного обеспечения в OS X. Однако *Homebrew является приложением командной строки*. Если вы не знакомы с работой в командной строке в OS X, вам удобней будет работать с MAMP.

Homebrew использует для установки программных пакетов, объединенных в *формулы*. В Homebrew изначально имеются формулы для большого числа программных продуктов, не входящих в дистрибутив OS X. Например, Homebrew содержит формулы для `wget`, `phploc`, `phpmd` и `php-code-sniffer` (и это всего лишь небольшая часть полного списка). Если встроенных формул Homebrew недостаточно, можно подключиться к сторонним репозиториям формул для расширения выбора доступного программного обеспечения. Homebrew, лично для меня, является предпочтительным средством установки PHP в OS X.

Инструменты командной строки Xcode

Перед установкой Homebrew необходимо установить набор инструментов XCode Command-Line Tools, предоставляемый компанией Apple Inc (бесплатно). Этот набор инструментов включает компилятор `gcc`, необходимый для сборки и установки пакетов программного обеспечения. Если вы используете OS X Mavericks 10.9.2 или новее,

откройте окно терминала OS X, введите следующую команду и нажмите клавишу **Enter**:

```
xcode-select --install
```

Эта команда откроет модальное окно, как показано на рис. А.4.



Рис. А.4. Установка инструментов командной строки XCode

Щелкните на кнопке **Install** (Установить), чтобы начать установку XCode Command-Line Tools. Щелкните на кнопке **Agree** (Принять), когда появится лицензионное соглашение программного обеспечения. После окончания установки щелкните на кнопке **Done** (Завершить) и перейдите к следующему шагу.

Если вы используете старую версию OS X, посетите Apple Developer Portal (<https://developer.apple.com/>), загрузите и запустите инсталлятор пакета (.pkg) XCode Command-Line Tools.

Установка

После установки XCode Command-Line Tools введите следующую команду в окне терминала OS X и нажмите **Enter**:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/homebrew/go/install)"
```



Эта команда выполняет код Ruby, который загружается с удаленного URL-адреса. Необходимо всегда проверять полученный из удаленного источника код перед его выполнением, независимо от легитимности источника.

Права доступа к каталогу

Менеджер пакетов Homebrew после загрузки помещает программное обеспечение в каталог */usr/local/Cellar*. Это символьическая ссылка на локальный каталог с установленными выполняемыми файлами */usr/local*. Ваша учетная запись в OS X должна иметь права доступа к каталогу */usr/local*, чтобы иметь возможность использовать программное обеспечение, установленное с помощью менеджера пакетов Homebrew.

Сделаем вашу учетную запись OS X владельцем каталога */usr/local*. Для этого введите следующую команду в окне терминала OS X и нажмите **Enter**. Введите свой пароль, если потребуется:

```
sudo chown -R `whoami` /usr/local
```

Команда `chown` означает «change the owner» («изменить владельца») указанного каталога, флаг `-R` команды означает «распространить изменения рекурсивно на все подкаталоги» указанного каталога, а аргумент `whoami` будет динамически заменен OS X именем вашей учетной записи. После выполнения этой команды, ваша учетная запись OS X будет владеть (и, следовательно, иметь доступ) каталогу */usr/local*.

Переменная окружения PATH

Добавьте каталог */usr/local* в переменную окружения `PATH`. Эта переменная хранит список каталогов, в которых выполняется поиск при запуске программ по имени, без указания абсолютного пути в файловой системе. Например, при вводе команды `wget`, OS X выполнит ее поиск во всех каталогах, перечисленных в `PATH`. Иначе пришлось бы вводить команду `/usr/local/wget`, чтобы запустить программу `wget`. Введите следующую команду в окне терминала OS X и нажмите **Enter**:

```
echo 'export PATH="/usr/local/bin:$PATH"' >> ~/.bash_profile
```

Подключение репозиториев формул

Перед тем как приступить к установке PHP с помощью Homebrew, необходимо подключить дополнительные репозитории с формулами для установки PHP, которых нет в репозиториях Homebrew по умолчанию.

Сначала добавим репозиторий `homebrew/dupes`. Этот репозиторий содержит формулы для программного обеспечения, которое уже присутствует в OS X, только более свежие, чем те, что поставляются с

OS X. Введите следующую команду в окне терминала OS X и нажмите **Enter**:

```
brew tap homebrew/dupes
```

Затем добавим репозиторий `homebrew/versions`. Этот репозиторий содержит набор версий программного обеспечения, уже имеющегося в OS X. Введите следующую команду в окне терминала OS X и нажмите **Enter**:

```
brew tap homebrew/versions
```

Наконец, добавим репозиторий `homebrew/php`. Этот репозиторий содержит формулы для PHP, которые не включены в репозиторий по умолчанию. Репозиторий Homebrew по умолчанию не поддерживается разработчиками PHP. А репозиторий `homebrew/php` содержит программное обеспечение, поддерживаемое разработчиками PHP. Введите следующую команду в окне терминала OS X и нажмите **Enter**:

```
brew tap homebrew/php
```

Установка PHP

Итак, мы установили менеджер пакетов Homebrew, настроили права доступа в файловой системе, обновили переменную окружения `PATH` и подключили дополнительные репозитории формул. Теперь пришло время установки PHP. В репозитории имеются формулы для каждой из версий PHP и PHP-расширений. Homebrew обеспечивает простой способ поиска доступных формул. Введите следующую команду в окне терминала и нажмите **Enter**:

```
brew search php
```

После запуска команды должен появиться длинный список формул Homebrew для установки PHP. Находим последнюю стабильную версию PHP (PHP 5.5.x соответствует `php55`, PHP 5.6.x соответствует `php56` и так далее). Я выбираю `php56`, так как последней стабильной версией является PHP 5.6.x (по состоянию на декабрь 2014 года). Введите следующую команду в окне терминала и нажмите **Enter**:

```
brew install php56
```

Установка займет некоторое время, так что можете отойти попить кофе и вернуться через несколько минут. После установки пакета PHP, проверьте успешность установки, выполнив команду `php -v`

в окне терминала. Результатом должен быть вывод полного имени и номера версии PHP, установленного с помощью Homebrew.

Установка расширений PHP

Менеджер пакетов Homebrew позволяет устанавливать PHP-расширения отдельно от интерпретатора PHP. Поиск PHP-расширений выполняется аналогично поиску версий PHP. Если считать, что была выбрана версия php56, введите следующую команду в окне терминала и нажмите клавишу **Enter**:

```
brew search php56
```

После запуска команды должен появиться длинный список расширений PHP 5.6, каждое из которых будет иметь префикс `php56-`. Найдя нужное расширение, введите следующую команду в окне терминала и нажмите **Enter**. Замените формулу в данном примере на формулу расширения, которые вы хотите установить:

```
brew install php56-intl php56-mcrypt php56-xhprof
```

Менеджер пакетов Homebrew предлагает множество других функций. Введите `brew` в окне терминала и нажмите **Enter**, чтобы увидеть полный список команд Homebrew. Также можно ознакомиться с полной документацией Homebrew на сайте <http://brew.sh>.

Сборка из исходных текстов

Предварительно скомпилированный выполняемый PHP-файл, предоставляемый менеджером пакетов вашей операционной системы, может оказаться не самой последней версией или не той версией, которая нужна вам. В этом случае соберите PHP из исходных текстов. Да, звучит страшновато. Я тоже долго не мог решиться перед первой попыткой сборки PHP. Но, уверяю вас, это не так страшно, как кажется.

Процесс сборки прост. Загрузите и извлеките исходный код PHP. Настройте его и проверьте установку всех зависимостей. Затем выполните сборку выполняемых файлов PHP. Скачать. Настроить. Собрать. Три простых шага.

Компиляция PHP из исходного кода обеспечивает гибкость, позволяет точно подстроить PHP под свои нужды. Хотя и существует множество способов настройки PHP, я опишу только настройку PHP,

применяющую мной в собственных проектах. Дополнительно я обычно добавляю в PHP поддержку:

- OpenSSL;
- Кэширования байт-кода;
- FPM (управление процессами FastCGI);
- Абстракцию баз данных PDO;
- Шифрование;
- Многобайтовые строки;
- Работа с изображениями;
- Сетевые сокеты;
- Curl.

Будем при сборке PHP ориентироваться на этот список. Постарайтесь выполнять сборку вместе со мной. Если вы первый раз собираете PHP из исходного кода, я рекомендую делать это на виртуальной машине. Установить виртуальную машину можно локально с помощью VMWare, Parallels или VirtualBox. Или же воспользоваться удаленной виртуальной машиной, совсем недорого предоставляемой DigitalOcean, Linode и другими веб-хостами с почасовой тарификацией. Если вы запутаетесь, можете уничтожить виртуальную машину, вновь восстановить ее и попробовать еще раз с самого начала.

Теперь глубоко вздохните, выдохните, откройте терминал и (самое главное) не бойтесь делать ошибки.

Получение исходного кода

Начнем с загрузки исходного кода PHP. Найдите последнюю стабильную версию исходного кода PHP на странице <http://www.php.net/downloads.php>. В моем случае последней стабильной версией была версия 5.6.3, но в вашем случае она может быть другой. Вводите приведенные ниже команды в окне терминала и нажимайте клавишу Enter после ввода каждой команды.

Каталог src/

Сначала создадим каталог *src/* в домашнем каталоге. Сюда нужно скопировать исходный код, загруженный с сайта PHP.net. Перейдем в каталог *src/* командой *cd*, чтобы сделать его текущим рабочим каталогом:

```
mkdir ~/src;
cd ~/src;
```

Загрузка исходного кода

Далее, используем команду `wget` для загрузки исходного кода PHP в архиве `tar.gz`. Загруженный файл будет сохранен в каталог `~/src/php.tar.gz`:

```
wget -O php.tar.gz http://www.php.net/get/php-5.6.3.tar.gz/  
from/this/mirror
```

Извлечем исходный код PHP из архива командой `tar` и перейдем с помощью команды `cd` в каталог с разархивированным исходным кодом:

```
tar -xzvf php.tar.gz;  
cd php-*;
```

Настройка PHP

Исходный код PHP загружен. Теперь настроим его. Прежде всего, необходимо установить несколько программных зависимостей. Откуда я знаю, какие зависимости установить? Я запускаю команду `./configure` (описана в следующем подразделе). В случае отсутствия необходимого программного обеспечения `./configure` завершится с ошибкой. Установите недостающие зависимости и запустите `./configure` еще раз. Устанавливайте и повторяйте запуск, пока `./configure` не отработает без ошибок.

К счастью я уже разобрался с программными зависимостями PHP, необходимыми для команды `./configure`. Давайте сейчас установим их. Я приведу команды для дистрибутивов Linux Ubuntu/Debian и CentOS/RHEL, а вы воспользуетесь командами для своего дистрибутива Linux.



Если по какой-либо причине команда `./configure` выдаст сообщение о прочих отсутствующих зависимостях, найти пакеты отсутствующих зависимостей можно по адресу: <http://packages.ubuntu.com/> (для Ubuntu) или <https://fedoraproject.org/wiki/EPEL> (для CentOS).

Сборка основного программного обеспечения

Это основное программное обеспечение необходимо для сборки PHP в вашей операционной системе. В его состав входят: `gcc`, `automake` и другое программное обеспечение для разработки:

```
# Ubuntu  
sudo apt-get install build-essential;  
  
# CentOS  
sudo yum groupinstall "Development Tools";  
  
libxml2
```

Библиотека libxml2 используется для поддержки функций PHP, связанных с XML:

```
# Ubuntu  
sudo apt-get install libxml2-dev;  
  
# CentOS  
sudo yum install libxml2-devel;
```

openssl

Библиотека openssl служит для обертывания потоков HTTPS в PHP, что *так же* важно, не так ли?

```
# Ubuntu  
sudo apt-get install libssl-dev;  
  
# CentOS  
sudo yum install openssl-devel;
```

Curl

Библиотека libcurl используется для поддержки функций PHP, работающих с HTTP:

```
# Ubuntu  
sudo apt-get install libcurl4-dev;  
  
# CentOS  
sudo yum install libcurl-devel;
```

Работа с изображениями

Системные библиотеки GD, JPEG, PNG и другие, связанные с обработкой изображений. К счастью, они собраны в единственный пакет. Они необходимы для работы с изображениями в PHP:

```
# Ubuntu  
sudo apt-get install libgd-dev;  
  
# CentOS  
sudo yum install gd-devel;
```

Mcrypt

Системная библиотека `mcrypt` используется для поддержки функций шифрования и дешифрования в PHP. По какой-то причине в CentOS отсутствует пакет Mcrypt по умолчанию, поэтому в этом дистрибутиве необходимо добавить пакет из стороннего репозитория EPEL:

```
# Ubuntu
sudo apt-get install libmcrypt-dev;

# CentOS
wget http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-
release-6-8.noarch.rpm;
sudo rpm -Uvh epel-release-6*.rpm;
sudo yum install libmcrypt-devel;
```

Команда `./configure`

Установив программные зависимости, перейдем к настройке процедуры сборки PHP. Введите следующую команду `./configure` в окне терминала и нажмите **Enter**:

```
./configure
--prefix=/usr/local/php5.6.3
--enable-opcache
--enable-fpm
--with-gd
--with-zlib
--with-jpeg-dir=/usr
--with-png-dir=/usr
--with-pdo-mysql=mysqlnd
--enable-mbstring
--enable-sockets
--with-curl
--with-mcrypt
--with-openssl;
```

Это длинная команда с большим количеством ключей. Но пусть она вас не смущает. Каждый ключ команды имеет свое назначение. Список всех доступных ключей можно получить, выполнив команду `./configure --help`. Обойдем все строки команды `./configure` и рассмотрим назначение каждой из них:

```
--prefix=/usr/local/php5.6.3
```

Ключ `--prefix` определяет путь к каталогу в файловой системе, куда будут сохранены скомпилированные выполняемые файлы PHP, включая библиотеки и конфигурационные фай-

лы. Я предлагаю помещать сборки PHP и связанные с ними файлы в один родительский каталог только из соображений удобства организации. Учетная запись пользователя должна иметь разрешение на запись в этот каталог. Если у вас нет такого разрешения, укажите в ключе `--prefix` каталог в своем домашнем каталоге (например, `~/local/php-5.5.13`). В любом случае удостоверьтесь, что каталог в ключе `--prefix` существует, *перед* выполнением команды `./configure`.

`--enable-opcache`

Ключ `--enable-opcache` включает встроенную в PHP систему кэширования байт-кода. Рекомендую всегда добавлять этот ключ, потому что кэширование байт-кода значительно повышает производительность.

`--enable-fpm`

Ключ `--enable-fpm` включает встроенный FastCGI Process Manager. Он позволяет запускать PHP как процесс FastCGI, доступный через TCP-порт или локальный сокет Unix. FPM быстро становится популярным способом запуска PHP (особенно с веб-сервером nginx). Несомненно, я рекомендую добавить этот ключ.

`--with-gd`

Ключ `--with-gd` предоставляет PHP-интерфейс к библиотеке GD для работы с изображениями. Добавьте этот ключ, если планируете работать с изображениями из PHP.

`--with-zlib`

Ключ `--with-zlib` предоставляет PHP-интерфейс к библиотеке Zlib, используемой для сжатия данных и необходимой библиотеке GD для создания и обработки изображений PNG. Этот ключ нужен, только если добавлен ключ `--with-gd`.

`--with-jpeg-dir`

Ключ `--with-jpeg-dir` указывает путь к каталогу в файловой системе с библиотекой JPEG. Этот ключ нужен, только если добавлен ключ `--with-gd`.

`--with-png-dir`

Ключ `--with-png-dir` указывает путь к каталогу в файловой системе с библиотекой PNG. Этот ключ нужен, только если добавлен ключ `--with-gd`.

```
--with-pdo-mysql=mysqlnd
```

Ключ `--with-pdo-mysql` включает программный интерфейс PDO для базы данных MySQL, использующий встроенный в PHP драйвер MySQL. Этот ключ нужен, только если используется MySQL.

```
--enable-mbstring
```

Ключ `--enable-MBstring` включает поддержку многобайтовых (читай «Юникод») строк. Этот ключ включается практически всегда.

```
--enable-sockets
```

Ключ `--enable-sockets` включает поддержку сокетов для взаимодействий с удаленными компьютерами через TCP-сокеты. Этот ключ включается практически всегда.

```
--with-curl
```

Ключ `--with-curl` включает PHP-интерфейс к библиотеке curl, с помощью которой можно отправлять и получать HTTP-запросы. Этот ключ включается практически всегда.

```
--with-mcrypt
```

Ключ `--with-mcrypt` включает PHP-интерфейс к библиотеке Mcrypt для шифрования и дешифрования данных. Хотя эта возможность не является обязательной, она используется все большим числом PHP-компонентов. Я настоятельно рекомендую добавить этот ключ.

```
--with-openssl
```

Ключ `--with-openssl` включает PHP-интерфейс к библиотеке openssl. Это необходимо для использования оберток потоков HTTPS в PHP. Технически данная возможность не является обязательной, но фактически всегда включается. Сделайте это и Эдвард Сноуден (Edward Snowden) будет вами гордиться. Добавьте этот ключ.

Сборка и установка PHP

Настройка PHP и установка его программных зависимостей была тяжелым этапом. После этого дорога пойдет под гору. Если команда `./configure` выполнилась успешно, введите следующую команду в окне терминала и нажмите **Enter**:

```
make && make install
```

Она выполнит компиляцию PHP и это займет некоторое время. Сейчас самое время выпить чашечку кофе или даже две. В конце концов, выполнение команды завершится и PHP будет установлен. Не так уж и ужасно, верно?

Скомпилированные выполняемые файлы PHP будут скопированы в подкаталог *bin/* заданного в ключе *--prefix* каталога. Выполняемый файл *php-fpm* будет скопирован в подкаталог *sbin/* заданного в ключе *--prefix* каталога. Добавьте каталоги *bin/* и *sbin/* в переменную окружения *PATH*, чтобы ссылаться на выполняемый файл *php* по имени, без использования абсолютного пути.

Создание файла *php.ini*

Не стоит забывать о файле *php.ini*. Он не создается автоматически. В репозитории PHP на GitHub имеется файл *php.ini*, предназначенный для локальной разработки. В нашем же случае файл *php.ini* должен находиться в подкаталоге *lib/* заданного в ключе *--prefix* каталога. Давайте создадим его. Введите следующие команды в окне терминала, нажимая клавишу *Enter* после ввода каждой команды.

Сначала перейдем в подкаталог (команда *cd*) *lib/*. Путь может быть другим, если вы задали другой путь в ключе *--prefix* команды *./configure*:

```
cd /usr/local/php5.6.3/lib
```

Затем загрузим файл *php.ini* из репозитория GitHub:

```
curl -o php.ini \
  https://raw.githubusercontent.com/php/php-src/master/php.ini-
development
```

Вот и все. Вновь установленный интерпретатор *php* готов к работе. Выполняемый файл *php-fpm* был подробно рассмотрен выше, при описании методики развертывания PHP в главе 7.

Windows

Да, интерпретатор PHP можно запустить в Windows. Но я все же рекомендую воспользоваться виртуальной машиной Linux. Скорее всего, ваше приложение будет выполняться на сервере, действующем под управлением Linux, к тому же вам нужно настроить локальную среду, чтобы она точно соответствовала эксплуатационной. Но если вам все же необходимо использовать именно Windows, сделать это можно следующим образом.

Скомпилированные файлы

Добрые люди выложили на сайте PHP.net готовые скомпилированные файлы PHP для Windows, которые можно загрузить, перейдя по адресу: <http://php.net/windows>. Загрузите нужную версию PHP (в виде ZIP-архива) и распакуйте загруженный архив в выбранный каталог. Я распаковал его в каталог *C:\PHP*. Скопируйте файл *php.ini-production* в тот же каталог, переименовав его в *php.ini*. Больше никаких действий для запуска интерпретатора PHP из командной строки Windows не требуется. Выполнить любой PHP-сценарий можно, добавив аргументы, как показано ниже:

```
C:\PHP\php.exe -f "C:\path\to\script.php" -- -arg1 -arg2 -arg3
```



Чтобы избавить себя от излишнего ввода с клавиатуры, добавьте каталог с выполняемым файлом интерпретатора PHP в переменную окружения PATH (<http://bit.ly/addtopath>) и добавьте расширение .php в переменную окружения PATHEXT.

WAMP

Также можно загрузить и установить WAMP (<http://www.wampserver.com/ru/>) – пакет для быстрого развертывания локальной среды разработки на PHP. Как и его коллега MAMP для OS X, WAMP является пакетом программного обеспечения вида «все включено», который предоставляет стандартную среду для веб-разработки с установкой «из коробки». Эта среда включает веб-сервер Apache, сервер баз данных MySQL и PHP. В нем предусмотрен инсталлятор для Windows, который выполнит установку. WAMP также обеспечивает всплывающее конфигурационное меню для области уведомлений в панели задач Windows, с помощью которой можно быстро и удобно запустить, остановить или перезапустить серверы Apache и MySQL. Как и MAMP, WAMP встраивает PHP в веб-сервер Apache в виде модуля Apache `mod_php`. Если работает сервер Apache, можно использовать и PHP.

WAMP является отличным средством быстрой установки среды разработки на компьютере с операционной системой Windows. Однако, как и MAMP, он ограничивает выбор только тем программным обеспечением и теми расширениями, которые включены в WAMP. Дополнительные версии PHP можно загрузить отдельно с сайта WAMP.

Более подробную информацию можно найти по адресу <http://www.wampserver.com/>.

Zend Server

Другим решением вида «все включено» является Zend Server. Имеется бесплатная и платная версии пакета. Как и WAMP, он включает веб-сервер Apache, последнюю версию интерпретатора PHP, популярные PHP-расширения, сервер баз данных MySQL и собственные средства отладки Zend в одном удобном установочном пакете. Загрузите инсталляционный файл (.exe), запустите его и следуйте инструкциям на экране. Более подробную информацию можно найти по адресу <http://www zend com/en/products/server/>.

ПРИЛОЖЕНИЕ Б.

Локальная среда разработки

До этого момента мы часто касались темы подготовки к работе производственных серверов и развертывания приложений для дальнейшей эксплуатации. Однако нигде не была раскрыта тема разработки приложений *на локальном компьютере*. Какими инструментами пользоваться? Как сделать среду разработки соответствующей эксплуатационной среде? Здесь будут приведены ответы на эти вопросы.



Убедитесь, что в виртуальной машине установлена та же операционная система, что и на производственном сервере (я отдаю предпочтение операционной системе Ubuntu Server). Для предотвращения появления ошибок при развертывании и эксплуатации приложений, вызванных несоответствием программного обеспечения разных операционных систем, важно использовать одну и ту же операционную систему при локальной разработке и на производственном сервере.

Многие начинающие PHP-разработчики используют набор программ по умолчанию для своей операционной системы, как правило, содержащий старые версии Apache и PHP. Я настоятельно рекомендую не пользоваться программным обеспечением по умолчанию. Многие пользователи OS X (включая меня) были в шоке, когда после обновления OS X исчезли с таким трудом настроенные конфигурационные файлы Apache. Держитесь подальше от встроенного программного обеспечения, оно часто бывает устаревшим и может быть переписано при модернизации операционной системы. Создайте локальную среду разработки на *виртуальной машине*, надежно изолированной от операционной системы локального компьютера. Виртуальная машина – это программный эмулятор операционной системы.

Например, в OS X можно создать виртуальную машину, работающую под Ubuntu или CentOS. Виртуальная машина ведет себя так же как отдельный компьютер.

VirtualBox

Существуют масса программного обеспечения для создания и управления виртуальными машинами. Среди них есть коммерческие продукты (например, VMware Fusion, (<http://www.vmware.com/products/fusion>) или Parallels (<http://www.parallels.com/products/desktop/>)) и продукты с открытым исходным кодом (например, VirtualBox (<https://www.virtualbox.org/>)). Следует отметить, что VirtualBox – весьма солидный продукт. Его работа соответствует его рекламе, и он бесплатен. VirtualBox, может быть, не так красиво выглядит, как и его коммерческие альтернативы, но он хорошо справляется со своей работой. Загрузить VirtualBox для OS X или Windows можно по адресу <https://www.virtualbox.org>. При установке VirtualBox используется интерфейс, соответствующий вашей операционной системе (рис. Б.1).



Рис. Б.1. Установка VirtualBox

Vagrant

Хотя VirtualBox и позволяет создавать виртуальные машины, но не предусматривает удобного пользовательского интерфейса для запуска, комплектования, остановки и уничтожения виртуальных машин. Для этого можно использовать средство виртуализации Vagrant (<https://www.vagrantup.com/>), которое дает возможность создавать, запускать, останавливать и уничтожать виртуальные машины VirtualBox одной командой. Он дополняет (и абстрагирует) VirtualBox, предоставляя удобный интерфейс командной строки. Загрузить Vagrant для OS X и Windows можно по адресу <https://www.vagrantup.com>. При установке Vagrant также используется графический интерфейс, соответствующий вашей операционной системе.

Команды

После установки, появляется возможность использовать команду `vagrant` в окне терминала для создания, комплектования, запуска, остановки и уничтожения виртуальных машин VirtualBox. Чаще всего используются следующие команды Vagrant:

`vagrant init`

Создает новый конфигурационный сценарий `Vagrantfile` в текущем рабочем каталоге. Этот сценарий используется для настройки свойств виртуальной машины и спецификации комплектования.

`vagrant up`

Создает и/или запускает виртуальную машину.

`vagrant provision`

Комплектует виртуальную машину на основе сценариев комплектования. Комплектование мы рассмотрим ниже в этом же приложении.

`vagrant ssh`

Регистрирует пользователя в виртуальной машине через SSH.

`vagrant halt`

Останавливает виртуальную машину.

`vagrant destroy`

Уничтожает виртуальную машину.



Я рекомендую создать псевдонимы для следующих команд Vagrant, потому что вводить их приходится часто. Поместите их в файл `~/.bash_profile` и перезапустите терминал:

```
alias vi="vagrant init"
alias vu="sudo echo 'Starting VM' && vagrant up"
alias vup="sudo echo 'Starting VM' && vagrant up
--provision"
alias vp="vagrant provision"
alias vh="vagrant halt" alias vs="vagrant ssh"
```

Боксы

Мы установили VirtualBox и Vagrant. Что дальше? Необходимо выбрать *бокс для Vagrant*, как исходное состояние виртуальной машины. Бокс для Vagrant – это предварительно подготовленная виртуальная машина, которая обеспечивает фундамент, на который мы поместим свой сервер и построим свое PHP-приложение. Некоторые боксы содержат по-спартански минимальный набор программ и используются в качестве чистых листов. Другие включают полные наборы программ для обслуживания определенного типа приложений. Список доступных боксов можно найти по адресу <https://vagrantcloud.com>.

Я обычно выбираю по-спартански минимальный бокс `ubuntu/trusty64` (<https://vagrantcloud.com/ubuntu/boxes/trusty64>), а затем использую Puppet для доукомплектования бокса конкретным набором программ, необходимым моему приложению. Если вы найдете бокс для Vagrant, содержащий все необходимые вам инструменты, имеет смысл использовать именно его в целях экономии времени.

Инициализация

Выбрав бокс, перейдите в нужный рабочий каталог в окне терминала. Выполните инициализацию нового файла *Vagrantfile* с помощью следующей команды:

```
vagrant init
```

Откройте новый файл *Vagrantfile* в текстовом редакторе. Этот файл содержит код Ruby, но он интуитивно понятен. Найдите параметр `config.vm.box` и замените его значение именем вашего бокса для Vagrant. Например, если я захочу использовать бокс Ubuntu, то дол-

жен буду задать значение `ubuntu/trusty64`. После внесения изменений строка в файле `Vagrantfile` будет иметь вид:

```
config.vm.box = "ubuntu/trusty64"
```

Затем раскомментируйте следующую строку, чтобы получить доступ к виртуальной машине из веб-браузера в локальной сети по IP-адресу `192.168.33.10`:

```
config.vm.network "private_network", ip: "192.168.33.10"
```

И, наконец, создайте виртуальную машину следующей командой:

```
vagrant up
```

Эта команда загрузит удаленный бокс для Vagrant (при необходимости) и создаст новую виртуальную машину VirtualBox, основанную на боксе для Vagrant.

Комплектование

Если вы не выбрали бокс для Vagrant, предоставляющий полный набор программ, ваша виртуальная машина ничего не умеет делать. Вы должны **укомплектовать** виртуальную машину программным обеспечением для запуска PHP-приложения. По крайней мере, нам потребуется веб-сервер, PHP и, наверное, база данных. Тема комплектования виртуальных машин слишком обширна для этой книги. Я лишь смогу указать направление, в котором нужно двигаться. Укомплектовать виртуальную машину Vagrant можно с помощью Puppet или Chef. И Puppet, и Chef можно подключить и настроить в конфигурационном файле `Vagrantfile`.



Эрика Хейди (Erika Heidi) (<http://erikaheidi.com/>) создала отличную презентацию NomadPHP (<http://www.bit.ly/1zUJmqb>) по Vagrant и вспомогательным инструментам, таким как Puppet и Chef. Она также написала книгу «*Vagrant Cookbook*» (<https://leanpub.com/vagrantcookbook>), которую можно приобрести на LeanPub.

Puppet

Если прокрутить файл `Vagrantfile` вниз, можно увидеть следующую секцию (она может быть закомментирована):

```
config.vm.provision "puppet" do |puppet|
  puppet.manifests_path = "manifests"
  puppet.manifest_file = "default.pp"
end
```

Если раскомментировать ее, Vagrant выполнит комплектование виртуальной машины с помощью Puppet, основываясь на ваших манифестах Puppet. Более подробную информацию о Puppet можно найти по адресу <http://puppetlabs.com>.

Chef

Если вы предпочитаете инструменты комплектования Chef, раскомментируйте следующую секцию в файле *Vagrantfile*:

```
config.vm.provision "chef_solo" do |chef|
  chef.cookbooks_path = "../my-recipes/cookbooks"
  chef.roles_path = "../my-recipes/roles"
  chef.data_bags_path = "../my-recipes/data_bags"
  chef.add_recipe "mysql"
  chef.add_role "web"

  # Также можно указать свои атрибуты JSON:
  chef.json = { mysql_password: "foo" }
end
```

Подготовьте свои поваренные книги (*cookbook*), роли (*role*) и рецепты (*recipe*). Vagrant укомплектует виртуальную машину в соответствии с ними. Более подробную информацию о Chef можно найти по адресу <https://www.chef.io/chef/>.

Синхронизация каталогов

Часто бывает полезно отобразить каталог проекта на локальной машине в каталог на виртуальной машине. Например, можно отобразить локальный каталог проекта в каталог */var/www* виртуальной машины. Если виртуальным хостом веб-сервера виртуальной машины будет */var/www/public*, каталог локального проекта будет обслуживаться веб-сервером виртуальной машины. Любые изменения в локальном каталоге будут *немедленно* переноситься на виртуальную машину. Раскомментируйте следующую строку в файле *Vagrantfile* для определения синхронизированных каталогов на локальной и виртуальной машинах:

```
config.vm.synced_folder ".", "/vagrant_data"
```

Первый аргумент (.) – это локальный путь относительно конфигурационного файла *Vagrantfile*. Второй аргумент (/vagrant_data) – это абсолютный путь в виртуальной машине, к которому будет привязан локальный каталог. Каталог виртуальной машины во многом зависит от конфигурации виртуального хоста веб-сервера виртуальной машины. Пользователи OS X должны включить опцию NFS синхронизации каталогов. Измените строку config.vm.synced_folder, как показано ниже:

```
config.vm.synced_folder ".", "/vagrant_data", type: "nfs"
```

Затем раскомментируйте следующие строки, чтобы увеличить размер памяти виртуальной машины до 1024 Мбайт:

```
config.vm.provider "virtualbox" do |vb|
  # Не загружайтесь в режиме headless
  # vb.gui = true

  # Используйте VBoxManage для настройки ВМ.
  # Например, чтобы изменить объем памяти:
  vb.customize ["modifyvm", :id, "--memory", "1024"]
end
```

Быстрый старт

Puppet и Chef нелегко освоить, особенно новичкам в Vagrant. Существуют инструменты, которые помогут вам начать работу с Vagrant, не требуя написания манифестов Puppet и Chef.

Homestead от Laravel

Homestead (<http://laravel.com/docs/4.2/homestead>) – это абстракция над Vagrant, предварительно подготовленный бокс Vagrant с полным набором программного обеспечения, включающим:

- Ubuntu 14.04
- PHP 5.6
- HHVM
- Nginx
- MySQL
- Postgres
- Node (вместе с Bower, Grunt и Gulp)
- Redis
- Memcached
- Beanstalkd
- Laravel Envoy

Homestead отлично подходит для любых PHP-приложений. Я использую Homestead на локальной машине для разработки приложений на Slim и Symfony. Более подробную информацию о Homestead можно найти по адресу <http://laravel.com/docs/4.2/homestead>.

Сайт PuPHPet

Сайт PuPHPet (<https://puphpet.com/>) идеально подходит для тех, кто не умеет писать манифести Puppet. Это интерактивный сайт, который создает конфигурацию Puppet автоматически (рис. Б.2). Вы загружаете полученную конфигурацию Puppet и выполняете команду `vagrant up`. Это на самом деле очень просто.

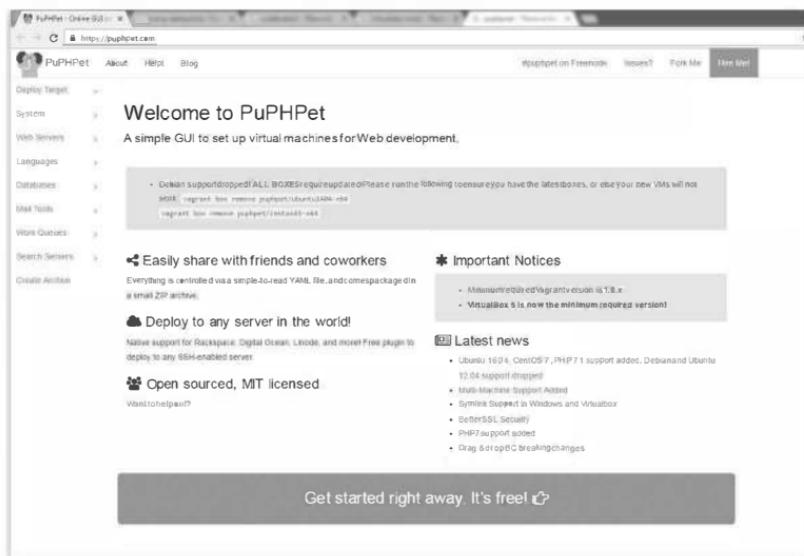


Рис. Б.2. PuPHPet

Vaprobash

Vaprobash (<http://fideloper.github.io/Vaprobash/>) похож на PuPHPet. Это не интерактивный сайт, но работать с ним так же просто. Вы загружаете файл *Vagrantfile* с сайта Vaprobash и раскомментируете строки с нужными инструментами. Вам нужен nginx? Раскомментируйте строку, содержащую nginx. Требуется MySQL? Раскомментируйте строку, содержащую MySQL. Хотите Elasticsearch? Раскомментируйте строку, содержащую Elasticsearch. Когда все будет готово, выполните команду `vagrant up` в окне терминала и Vagrant укомплектует вашу виртуальную машину.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

A

Apache Bench 198, 232

B

behavior-driven development (BDD) 216
Bitbucket 206
Blackfire 238
BOM (byte order mark) 67

C

Capistrano
автентификация 211
виртуальный хост 211
использование 208
настройка 209
откат приложения 213
подготовка удаленного сервера 211
подключения 212
программные зависимости 212
развертывание приложения 213
удобство 208
установка 208
файл config/deploy.rb 210
CentOS
создание непривилегированного
пользователя 180
nginx 190
обновление 179
установка PHP 269
установка PHP-FPM 185

Chef 292

Composer
закрытые хранилища 96
назначение 88
пример проекта 93
установка 89
установка компонентов 90
файл composer.lock 94
скрипт scan.php 93

D

DRY (Do not repeat yourself) 41

F

Forge 194

G

Git 22

H

HipHop Virtual Machine (HHVM)
выбор 243
использование 243
настройка 245
полезность 23, 242
разработка 242
расширения 244
совместимость с Zend Engine 243
установка 244
Homebrew 273
Homestead 293
Homestead от Laravel 293
HTTP-сервер
запуск 57
настройка 57
недостатки 59
обнаружение 58
полезность 56
скрипты маршрутизации 58

K

KCacheGrind 233

L

Linode 174, 178

M

MAMP (Mac, Apache, MySQL и PHP) 270

Mercurial 206

N

nginx

- взаимодействие с HHVM 248
- настройки виртуального хоста 191
- установка 190

O

OS X 270

P

PaaS

- или укомплектование 177
- полезность 175

Packagist 85, 99

PHP Code Sniffer (phpcs) 72

PHP-CS-Fixer 72

PHP Framework Interop Group (PHP-FIG)

- автозагрузка 75
- деятельность 63
- миссия 63
- рекомендации или правила 64
- создание 63
- стандарт автозагрузки 34

PHPUnit

- гипотетические классы тестов 224
- запуск тестов 227
- настройка 222
- основные термины 219
- покрытие кода 228
- структура каталогов 219
- установка 220
- установка Xdebug 221

PHP-ключевые слова 69

PHP-сообщество

- источники 263

PHP-сообщество

- изменения в языке 263
- локальная группа PUG (PHP User Groups) 262

- наставничество 263

- полезность 262

PSR (PHP standards recommendation)

- PSR-1 Базовый стиль кода 66

- PSR-2 строгий стиль кода 68

- PSR-3 Интерфейс регистратора 73

важность 65

полезность 65

публикация рекомендаций 63

PUG

- конференции 262

PUG (PHP User Group) 262

Puppet 291

S

Seige 198

SpecBDD 217

SQL-запросы 112

SSH-аутентификация с помощью парных ключей 181

StoryBDD 218

Supervisord 246

T

Travis CI 229

U

Ubuntu

- настройка виртуального хоста 193

- обновление программного обеспечения 179

- создание непривилегированного

- пользователя 180

- установка nginx 190

- установка PHP 267

- Установка PHP-FRM 184

V

VirtualBox 288

W

WAMP 285

WinCacheGrind 233

Windows 284

X

XHGUI 233

XHProf 233, 235

Z

Zend Engine 22, 240, 243

Zend OPcache

- включение 54
- использование 55
- настройка 55, 198
- полезность 53

Zend Opcodes 241**Zend Server** 286**А****автозагрузка**

- автозагрузчик стандарта PSR4 34
 - важность 76
 - компонентов 94
 - назначение 75
 - написание автозагрузка PSR-4 78
 - определение 64
 - пространства имен 67
- алгоритм хеширования bcrypt** 118
- анонимные функции** 48
- аргумент \$context** 74
- аутентификация с помощью парных ключей** 181

Б**базы данных**

- обеспечение безопасности учетных данных 133
 - подготовленные выражения 134
 - подключение и DSN 131
 - расширение PDO 132
 - результаты запроса 137
 - транзакции 139
- библиотека HTML Purifier** 111
- брандмауэр** 184

В

- ввод, санитаризация** 109, 110
- верблюжья нотация** 67
- видимость** 71
- виртуальная машина** 287
- виртуальный выделенный сервер** 173
- виртуальный хост** 190, 211
- внешние источники данных** 110
- выделенный сервер** 174

Г**генераторы**

- достоинства и недостатки 47
 - использование 46
 - полезность 44
 - создание 45
- глобальное пространство имен** 33
- Гутман Энди** 21

Д**данные**

- SQL-запросы** 112
 - обертывание потоком 146
 - потоки 146
 - проверка 114
 - санитарование ввода 110
 - санитарование текста в формате HTML 111
 - сведения из профилей пользователей 113
 - хорошие методики 109
- даты, время и часовые пояса**
- PHP-классы** 125
 - класс DateInterval** 127
 - класс DatePeriod** 129
 - Класс DateTime** 125
 - класс DateTimeZone** 128
 - компонент nesbot/carbon 130
 - установка часового пояса по умолчанию 125

движок шаблонов smarty/smarty 115**динамические типы, определение** 23**З****завершение строк Unix (LF)** 69**загрузка файлов, настройка** 201**закрытые хранилищ** 96**замыкание**

- возможности** 48
- или анонимные функции** 48
- прикрепление состояния** 50
- создание** 48

И**идентификаторы** 147**имена**

компонентов 90
 пакетов 98
 производителей 98
 стандарты 67
 имена констант 67
 именованный заполнитель 135
 импортирование
 импорт нескольких 32
 пространства имен или трейты 43
 инструмент PHP Iniscan 197
 инструменты командной строки Xcode 273
 интерпретируемый язык 198
 интерфейс регистратора
 написание 73, 74
 стандарты 73
 интерфейсы
 важность 34
 гибкость кода 39
 идея 34
 полезность 64
 рекомендации по регистратору 73
 итераторы 44

К

кеш байт-кода 53
 класса определение 70
 классическая модель наследования 39
 ключевые слова 69
 ключевое слово if 72
 ключевое слово case 72
 ключевое слово catch 72
 ключевое слово catch 72
 ключевое слово do while 72
 ключевое слово else 72
 ключевое слово elseif 72
 ключевое слово extends 70
 ключевое слово for 72
 ключевое слово foreach 72
 ключевое слово implements 70
 ключевое слово try 72
 ключевое слово use 31
 ключевое слово use func 32
 ключевое слово while 72
 кодировка UTF-8 67, 144
 компилятор HHVM 241
 компиляторы just-in-time (JIT)
 HHVM 240

полезность 23
 комплектование
 nginx 189
 PHP-FRM 184
 автоматизация 193
 делигирование 194
 необходимые навыки 177
 обзор 177
 подходы 177
 установка сервера 178
 через PaaS 177
 компонент nesbot/carbon 130
 компонент Whoops 166
 компоненты
 автозагрузка 94
 закрытые хранилища 96
 или фреймворки 83
 имена 90
 использование 80, 88
 определение 81
 организация файловой системы 99
 поиск/выбор 87
 пример проекта 92
 создание 98
 установка 91
 установка Composer 89
 характеристики 81
 контроль вывода 115
 коэффициент производительности 118
 Кэш Realpath 204
 кэширование байт-кода 53

Л

Лердорф Расмус 21
 локальная среда разработки
 Homestead 294
 PuPHPet 294
 Vapro bash 294
 Vagrant 289
 VirtualBox 288
 полезность 21
 синхронизация каталогов 292
 цель 287

М

магический метод `_autoload()` 65
 магический метод `_invoke()` 49

Максимальное время выполнения 202
 менеджер процессов PHP-FPM
 возможности 184
 глобальная конфигурация 185
 конфигурации пулов 186
 установка 184
 менеджеры пакетов 265
 метод addDocument() 36
 метод addRoute() 52
 метод bindTo() 51
 метод dispatch() 53
 метод getContent() 36
 метод GetId() 36
 метод makeRange() 46
 метод spl_autoload_register() 65
 многобайтовые строки 143
 модульное тестирование
 определение терминов 216
 полезность 217
 фреймворки для тестирования 217
Мониторинг HHVM с помощью Supervisord
 246

Н

назначение имен классам в стиле Zend 30
 настройка
 Zend OPcache 198
 буферизация вывода 204
 загрузка файлов 201
 кэш Realpath 204
 максимальное время выполнения 202
 обслуживание сессий 203
 память 197
 полезность 196
 файл php.ini 196
 настройка буферизации вывода 204
 настройка памяти 197
 настройки кеширования 55
 непривилегированный пользователь 180

О

обозначение периода 127
 обработчики шаблонов 115
 обработчик шаблонов Twig 76
 обслуживание сессий, настройка 203
 объектно-ориентированное
 программирование 35
 определение методов 70
 определения пулов 186
 откат 213
 оценочный инструмент 232
 ошибки и исключение
 регистрация исключений 161
 ошибки и исключения
 возбуждение исключений 157
 исключений 157
 обработчики исключений 161
 обработчики ошибок 164
 ошибки 162
 перехват исключений 159
 при разработке 166
 различия 156, 162
 сообщения 163

П

пакетов имен 98
 пароли
 отключение аутентификации 183
 поддержание безопасности 116
 правильная обработка 116
 Программный интерфейс хеширования
 119
 хеширование с помощью bCrypt 118
 хранение 118
 подготовленные выражения 134
 подготовленные заранее операторы PDO
 113
 подключения 212
 подстановка имен 134
 подстановки 74
 пользователь root 180
 потоки
 введение 146
 контекст 150
 обертывание 146
 определение терминов 145
 полезность 145
 пользовательские фильтры 153
 фильтры 150
 префикс @ 156
 проблемы производительности 232
 программные зависимости 212
 проект Facebook Open Source 240
 пространство имен производителя 29
 пространства имен 33, 69, 99

автозагрузка 34
 глобальные 33
 импорт и псевдонимы 30
 компонента 99
 назначение 25
 несколько в одном файле 33
 объявление 28, 69
 полезность 28
 пространство имен производителя 29
 пространство имен
 или файловая система 30
 импорт нескольких 32
 протокол FastCGI 248
 протокол системного журнала RFC 5424 73
 профилирование
 Blackfire 238
 New Relic 238
 Xdebug 233
 XHProf 235
 возможности 232
 когда требуется 232
 типы профилировщиков 233
 профилировщик New Relic 238
 профилировщик Xdebug
 анализ 235
 включение 234
 использование с Zend OPcache 54
 настройка 234
 недостатки 233
 установка 221, 234
 псевдоним
 определение 30
 пользовательский 31
 по умолчанию 32
 псевдоним по умолчанию 31

P

развертывание
 автоматизация 207
 подход 207
 с помощью Capistrano 207
 управление версиями 206

разделяемые серверы 172
 размер буфера 204
 разработка через тестирование (TDD) 216
 расширение mbstring 143
 расширение PDO 131
 регистратор monolog/monolog 73, 75

регулярные выражения 112
 релизы, нумерация версий 92

C

Сайт PuPHPet 294
 санация адреса электронной почты 113
 санирование входные данные в формате
 HTML 111
 сведения из профилей пользователей 113
 связанные параметры 135
 серверные скрипты, общее описание 20
 символы пробелов 69
 символы табуляции 68
 скрипт маршрутизатора 58
 скрипты командной строки 96
 совместимость
 автозагрузка 64
 интерфейсы 64
 стиль 65
 совместимость фреймворков
 автозагрузка 64
 интерфейсы 64
 стиль 65
 состояния, прикрепление 50
 специальные символы
 многобайтные строки 143
 санирование текста в формате HTML 111
 спецификация
 определение терминов 22
 спецификация периода 127
 стандарт строк 69
 стандарты
 PSR-1: Базовый стиль кода 66
 PSR-2: Строгий стиль кода 68
 PSR-3: Интерфейс регистратора 73
 PSR-4: Автозагрузчики 75
 важность 63
 совместимость фреймворков 64
 стандарты файлов 69
 стартер из командной строки 219
 стиль кода
 автоматизация совместимости 72
 видимость 71
 имена 67
 ключевые слова 69
 кодировка UTF-8 67
 определение классов 70
 определение методов 70

- отступы 68
- пространства имен 69
- стандартизация 66
- теги 67
 - файлы и строки 69
- стиль кода PSR-1
 - Базовый стиль кода 66
- стиль кода PSR-2
 - Строгий стиль кода 68
- стандарты
 - PHP-FIG 63
- стресс-тесты 198
- строка DSN 132
- Суракки Зеев 21
- схема Semantic Versioning 92
- схемы 147

- T**
- теги PHP 67
- тестирование
 - важность 214
 - когда тестировать 216
 - модульное 216
 - непрерывное 229
 - разработка, основанная на функционировании (BDD) 217
 - с помощью PHPUnit 219
 - с помощью Travis CI 229
- тиปизация

 - динамическая 253
 - контроль типов 254
 - определение и термины 250
 - статическая 252
 - статическая или динамическая 23

- транзакции, поддержка PDO 139
- трейты
 - использование 43
 - определение классов при копиляции 44
 - определение терминов 39
 - полезность 39
 - создание 41

- У**
- удобочитаемые сценарии 218
- управление версиями
 - общедоступные хранилища кодов 106
 - полезность 206
- программное обеспечение 22
- семантика версий 91
- управляющие конструкции 72
- установка
 - CentOS 7 268
 - Homebrew 273
 - MAMP (Mac, Apache, MySQL and PHP) 270
 - OS X 269
 - Ubuntu 14.04 LTS 266
 - Windows 284
- из исходных текстов 277
- Инструменты командной строки Xcode 273
- менеджеры пакетов 265
- среда разработки 265
- установка сервера 178
 - SSH-аутентификация с помощью парных ключей 181
 - безопасность 180
 - брэндмаэр 184
 - обновление программного обеспечения 179
 - отключение входа по паролям 183
 - первый вход 179

- Ф**
- файл auth.json 97
- файл composer.json 100
- файл config/deploy/production.rb 210
- файл config/deploy.rb 210
- файл .htaccess 58
- файл php.ini 196
- файл README 103
- Феррара Энтони 119
- фрейворки
 - или компоненты 83
- фреймворки
 - выбор 84
 - назначение 84
 - популярные 84
- фронт-контроллер 58
- функции
 - анонимные 48
 - замыкания 48
- функциональное тестирование 217
- функция exec() 202
- функция filter_input() 113

функция `filter_var()` 113
функция `htmlentities()` 111, 115

X

хеширование
алгоритмы 118
или шифрование 117

хорошие методики
`DRY` 41
базы данных 131
даты, время и часовые пояса 125
или наилучшие методики 109
компоненты 80
контролирование вывода 115
многобайтные строки 143
опереление трейтов 39
ошибки и исключения 156
пароли 116
полезность 109
потоки 145
проверка данных 110
санирование ввода 109
стандарты 62

хостинг
виртуальный выделенный сервер (VPS)
 173
выбор тарифного плана 176
выделенный сервер 174
компании 172
платформы как сервис (PaaS) 175
подходы 172
разделяемые серверы 172

хранилище EPEL (Extra Packages for Enterprise Linux) 268

Ц

цели 147

Ч

часовой пояс UTC 129

Ю

Юникод 144

Я

язык Hack
 динамическая типизация 253
 или PHP 259
 конвертирование PHP 250
 контроль типов 254
 обратная совместимость 240
 особенности 23
 полезность 250, 254, 260
 режимы 255
 синтаксис 256
 статическая типизация 252
 структуры данных 258

язык PHP
 HTTP-сервер 56
 Zend OPcache 53
 версия PHP 7 24
 генераторы 44
 движки 22
 замыкания 48
 или HHVM/Hack 259
 интерпритируемый язык 241
 интерфейсы 34
 история 21
 конвертирование в Hack 250
 пространства имен 25
 спецификация 22
 существенные и несущественные
 особенности 25
 трейты 39
 эволюция 20



ОБ ОБЛОЖКЕ

На обложке книги «Современный PHP» изображен австралийский ибис (*Threskiornis spinicollis*). Водится в Австралии, Новой Гвинее и некоторых частях Индонезии. Изображение взято из книги «Woods Illustrated Natural History».

Австралийские ибисы – это большие птицы, достигают 75 сантиметров в длину. Отличительной их особенностью являются жесткие перья на шее, появляющиеся во взрослом возрасте, из-за которых птица получила свое английское название straw-necked (соломенная шея). Австралийские ибисы имеют длинные изогнутые клювы, через которые они просеивают воду, поедая насекомых, моллюсков и лягушек. Фермеры радуются появлению ибисов на своих землях, потому что птицы питаются вредителями растений: насекомыми, кузнечиками, сверчками и саранчой.

Эти птицы ведут кочевой образ жизни, стаями перелетая в новые места обитания. Предпочитают неглубокие пресноводные водоемы, пастбища, болота, лагуны и луга. В период размножения ибисы строят большое чашевидное гнездо из прутьев и тростника на высоких деревьях расположенных вблизи водоемов. Гнездятся колониями, часто вместе с австралийскими белыми ибисами. Их легко обнаружить, стоящими на голых ветвях высоких деревьев, трудно не обратить внимания на их поразительный силуэт на фоне неба.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся на грани исчезновения. Все они важны для нашего мира. Чтобы узнать, как можно помочь их сохранению, посетите страницу animals.oreilly.com.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: www.aliants-kniga.ru.

Оплотные закупки: тел. +7 (499) 782-38-89.

Электронный адрес: books@aliants-kniga.ru.

Джош Локхарт

Современный PHP

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Рагимов Р. Н.*

Научный редактор *Киселев А. Н.*

Корректор *Синяева Г. И.*

Верстка *Паранская Н. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16. Гарнитура «Петербург».

Печать офсетная. Усл. печ. л. 17,67.

Тираж 200 экз.

Веб-сайт издательства: www.dmk.ru

«Несколько лет я искал книгу по PHP, которую мог бы рекомендовать как отражающую современное состояние языка и сообщества. С появлением книги «Современный PHP», я, наконец, могу уверенно сказать это.»

Эд Финклер, разработчик и автор, *Funkatron.com*

«В программировании не изменяются только константы. Меняется PHP и вместе с ним должен меняться подход к разработке приложений. Джош привел инструменты и идеи, знание которых поможет писать современный код на PHP.»
Кэл Эванс, Twitter: @oreillymedia, facebook.com/oreilly.

PHP переживает ренессанс, хотя это трудно заметить, просматривая устаревшие материалы. Из данной книги вы узнаете, как PHP превратился в зрелый полнофункциональный объектно-ориентированный язык, с пространствами имен и постоянно растущей коллекцией библиотек компонентов.

Джош Локхарт, создатель популярного ресурса «PHP The Right Way», демонстрирует новые возможности языка на практике. Вы узнаете о передовых методах проектирования и конструирования приложений, работы с базами данных, обеспечения безопасности, тестирования, отладки и развертывания. Если вы уже знакомы с языком PHP и желаете расширить свои знания о нем, то эта книга для вас!

В этой книге вы:

- узнаете об особенностях современного языка PHP, таких как пространства имен, трейты, генераторы и замыкания;
- научитесь находить, использовать и создавать PHP-компоненты;
- ознакомитесь с передовыми приемами поддержки безопасности приложений, работы с базами данных, обработки ошибок и исключений и многими другими;
- овладеете инструментами и методами развертывания, настройки, тестирования и профилирования PHP-приложений;
- познакомитесь с виртуальной машиной HVVM и языком Hack, созданным в Facebook и оцените их влияние на современный язык PHP;
- узнаете, как создать локальную среду разработки, эквивалентную реальному серверу.

Интернет-магазин:

www.dmkpress.com

Книга – почтой:

orders@aliants-kniga.ru

Оптовая продажа:

“Альянс-книга”

тел.(499)782-38-89

books@aliants-kniga.ru

O'REILLY®



www.dmk.ru