www.phparch.com

February 2016
VOLUME 15 - ISSUE 2

php[architect]

Workflow Powerups

Docker for PHP

Software Branching Strategies: A Git Overview

Build a Local CI with Docker for PHP

ALSO INSIDE

Extensible Applications with Symfony Event Dispatcher

Leveling Up:
Using Code to Help You Code Better

Security Corner:
Defaulting to Secure

Community Corner:
Renewal

Education Station:
Get Some Git Extras

finally{}:
Building a Conference Schedule

WE'RE HIRING

QA

SOFTWARE QUALITY ASSURANCE ENGINEER

test
develop
collaborate
improve

self starter
security minded
experienced
attention to detail

APPLY	nexc.es/218WJØY

NEXCESS™
beyond hosting.

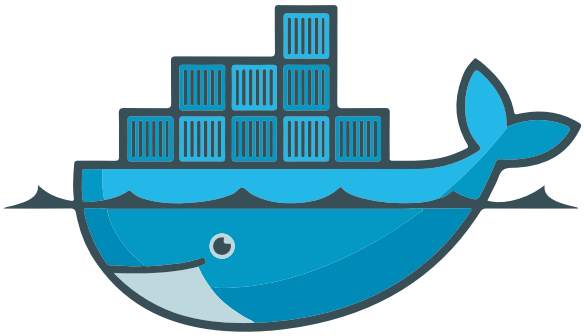
NEXCESS.NET | USA 1.866.639.2377 | UK 0.808.120.7609 | AU 1.800.765.472

php[architect]

CONTENTS

FEBRUARY 2016
Volume 15 - Issue 2

Workflow Powerups



Columns

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith, Eli White

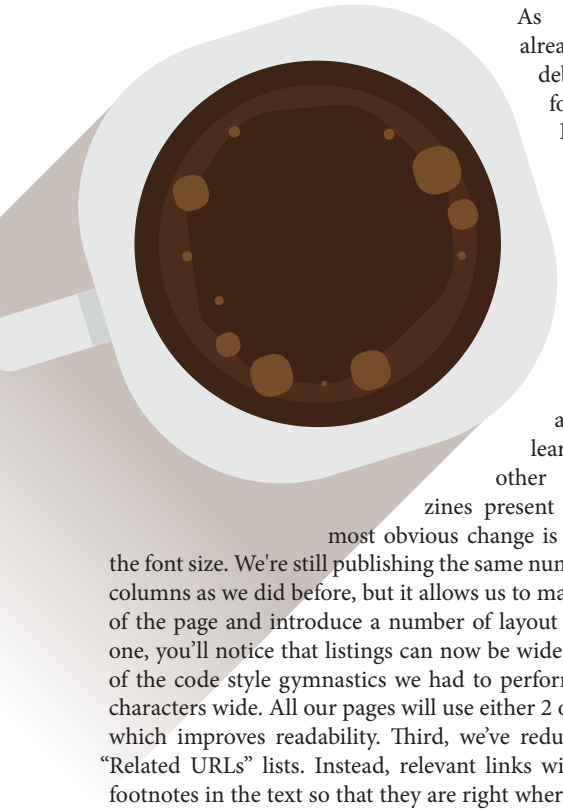
php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Workflow Powerups

This issue has some big changes in store. First, I want to welcome Cal Evans as our new contributor to *Community Corner*. Joe Devon did a great job in shaping it and making it a great resource for keeping up with people and events in the PHP community—both old and new. Thank you, Joe. I look forward to Cal's column each month.



As you've no doubt already noticed, we're debuting a new layout for the magazine. If you're reading the ePub or Mobi versions, you'll have to download the PDF version or wait for your print copy to arrive. We surveyed similar journals and magazines to learn and adapt how other professional magazines present their content. The most obvious change is that we've reduced the font size. We're still publishing the same number of features and columns as we did before, but it allows us to make much better use of the page and introduce a number of layout improvements. For one, you'll notice that listings can now be wider eliminating much of the code style gymnastics we had to perform to stay under 60 characters wide. All our pages will use either 2 or 3 column layouts, which improves readability. Third, we've reduced the use of the "Related URLs" lists. Instead, relevant links will be presented via footnotes in the text so that they are right where you need them. If you want a flexible presentation that allows you to change font sizes you still have the option of reading the eBook versions.

What do you think of the new layout? Let me know at oscar@phparch.com. I'm sure we'll find things we want to tweak and adjust.

In an odd coincidence, this month's topic is on tools that can improve your workflows. If you're like me, your TODO list changes day-to-day as new tasks become a priority or clients adjust their requirements. If you don't have solid processes for adapting to such changes quickly, you'll be overwhelmed and frustrated. Good processes are indispensable. That's not to say that you shouldn't revisit your toolbox to learn about and integrate new ways of doing things. On the contrary, if there's a tool that helps you to be more productive, you should explore it.

In this issue, we have a collection of tools you may have heard about that are worth a look. Docker and containers are very popular now. Ben Hosmer will show you how you can start using it to set up a development environment in *Docker for PHP*. Have you examined how you are using Git? In *Software Branching Strategies: A Git Overview*, Georgiana Gligor explores the variety of workflows you can use with it. Nicola Pietroluongo shares how to setup a testing environment with Docker in *Build a Local CI with Docker for PHP*.

Matthew Setter will help you be more productive with Git when you *Get Some Git Extras* in *Education Station*. If you haven't evaluated your code quality objectively, David Stockton collects the tools you need to check out in *Leveling Up: Using Code to Help You Code Better*.

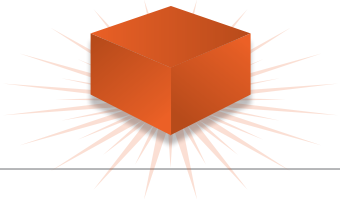
Also in this issue, Juan Manuel Torres writes about building more flexible solutions with Symfony in *Extensible Applications with Symfony Event Dispatcher*. As I mentioned above, Cal Evans will be writing *Community Corner* for us, and this month he shares his plans for the column.

With the php[tek] schedule just published, Eli gives you a peek behind the curtain to see what goes in to *Building a Conference Schedule*. If you want to become a conference speaker or are an old hand at it, he shares an honest look at the factors beyond your title and abstract that go into being selected to speak.

Did you find a great new tool this month? Do you have one you want to see in a future issue? Don't hesitate to contact me or any of our authors with constructive feedback!

Download this issue's code package:

http://phpa.me/Feb2015_code



Extensible Applications with Symfony Event Dispatcher

Juan Manuel Torres

Last month we learned how to create simple, yet elegant, CLI applications using a couple of Symfony components. In addition to the components, we also focused our attention on some best practices and how we can write better code by learning the ins and outs of Symfony. This month we take on another Symfony component: the Event Dispatcher[1].

This article introduces different ways to extend an application, what the Mediator Design Pattern is and how the Symfony Event Dispatcher implements it. To demonstrate how to use it we will refactor the application from last month to add a custom event, and create a subscriber to extend the functionality of the `SayCommand`.

When to Use the Event Dispatcher

The Event Dispatcher follows a design pattern called the Mediator[2]. The objective of the Event Dispatcher is to allow communication between different parts of your application without coupling them. This way, an application can be extended, and it does not have to know about the details of the classes or functions that do it.

The Event Dispatcher can be useful when creating new applications, refactoring legacy code, or when you want to add the means of extending existing code with minimal changes.

Many applications, including the Symfony HTTP Kernel (which is at the heart of many PHP projects), use the Event Dispatcher to allow extension and bring together different core components.

Common Ways to Extend Code Behavior

Before I introduce the Mediator pattern, I would like to give an introduction to various methods for extending behavior in Object-Oriented Programming.

Inheritance

One of the most traditional ways to extend an application is by using inheritance. As long as the class is not a final class, it can be extended, and the original behavior of the class can be overwritten.

Take, for example, the `Car` class in Listing 1.

If we wanted to extend the behavior of the class car using inheritance—e.g., build a turbo car—we could generate a new class that extends the original. The new `TurboCar` class overwrites the appropriate methods to extend the functionality. In this case, the `startEngine` and `accelerate` methods are overwritten.

LISTING 1

```
01. <?php
02. class Car
03. {
04.     public function operateVehicle() {
05.         $this->startEngine();
06.         $this->accelerate();
07.     }
08.     public function startEngine() {/* ... */}
09.     public function accelerate() {/* ... */}
10. }
11.
12. Class TurboCar extends Car
13. {
14.     public function startEngine() {/* new implementation */}
15.     public function accelerate() {/* new implementation */}
16. }
17.
18. $car = new Car();
19. $turboCar = new TurboCar();
```

1 *Symfony Event Dispatcher Component:* http://phpa.me/symfony-event-dispatcher

2 *Mediator Design Pattern:* https://sourcemaking.com/design_patterns/mediato

Composition

An alternative to inheritance is composition. Composition is a way to create complex objects from simple ones. If we were to use composition for the `Car` class, we would abstract the functionality that changes and place it into a set of specialized classes. Listing 2 shows how to use composition by abstracting some of the operate vehicle functionality into a new set of classes that implement the `VehicleControlInterface`.

In this case, we extend the behavior of our class without using inheritance, and we do not need an additional `TurboCar` class. Instead, all we do is move the functionality of the operation of the vehicle to a new family of classes that are in charge of doing only one job. Then we use constructor injection to pass an instance of a class that implements the `VehicleControlInterface`.

In Object-Oriented Programming (OOP), this functionality is known as polymorphism (*poly* = many, *morph* = form). A single car class can use any vehicle control object as long as it implements the `VehicleControllInterface`. By using different types of *Vehicle Controls* we can make the car behave like a regular car, a turbo car, or any implementation of the `VehicleControlInterface`.

An Alternative Way to Extend Behavior

When using composition, the car must have an intimate knowledge of how to use the *Vehicle Controls*, specifically in that they conform to the `VehicleControlInterface`. This knowledge is proper in this context, but what if we wanted to extend the behavior beyond the functionality described by the interface? What if we wanted to know the speed of the car or send a distress call if the engine blows up? The Mediator pattern gives us a way to address these questions and more.

LISTING 2

```
01. <?php
02. class Car
03. {
04.     protected $vehicleControl;
05.
06.     public function __construct(VehicleControlInterface $sc) {
07.         $this->vehicleControl = $sc;
08.     }
09.
10.     public function operateVehicle() {
11.         $this->vehicleControl->startEngine();
12.         $this->vehicleControl->accelerate();
13.     }
14. }
15.
16. interface VehicleControlInterface
17. {
18.     public function startEngine();
19.     public function accelerate();
20. }
21.
22. class CarControl implements VehicleControlInterface
23. {
24.     public function startEngine() {/* ... */}
25.     public function accelerate() {/* ... */}
26. }
27.
28. class TurboCarControl implements VehicleControlInterface
29. {
30.     public function startEngine() {/* ... */}
31.     public function accelerate() {/* ... */}
32. }
33.
34. $car = new Car(new CarControl());
35. $turboCar = new Car(new TurboCarControl());
```

The Mediator Design Pattern

Design Patterns are blueprints to solutions to common problems. These problems appear over and over when developing software projects. In practice, design patterns are not finished solutions, but provide a template to build a solution.

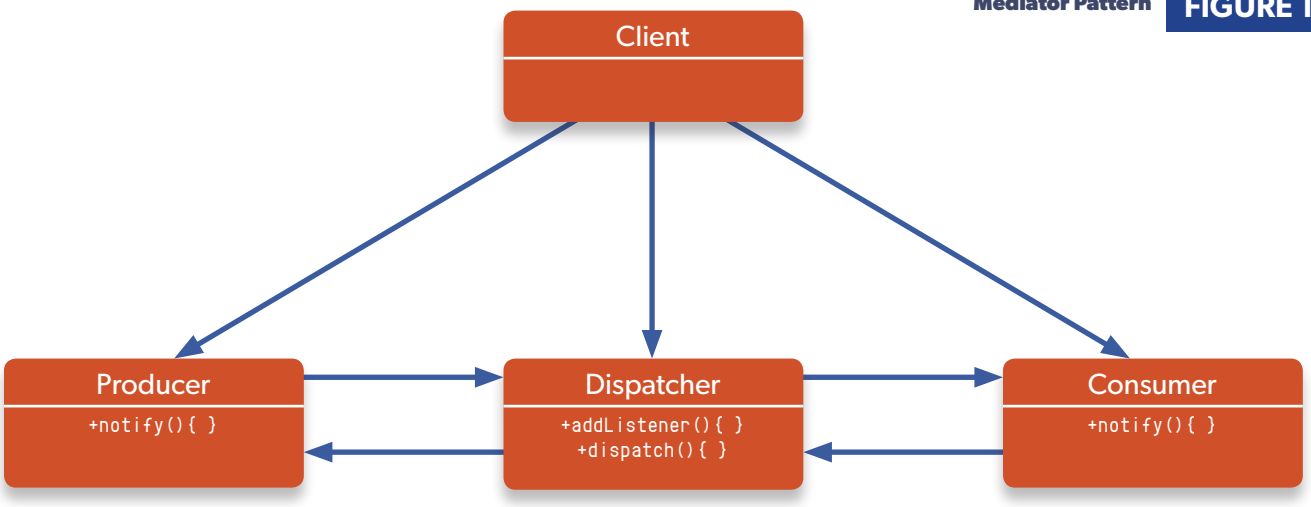
Design patterns can be classified as Creational, Behavioral, and Structural. The Mediator pattern falls in the Behavioral category, and its primary intent is to allow two classes to communicate without knowing anything about each other. By doing so, the Mediator pattern promotes loose coupling between classes, enabling a broad range of interactions between them. To allow loose coupling, this pattern defines an intermediary class as *mediator* or *dispatcher*, and its job is to be a central hub for all communications.

There are many examples on the web and in books trying to explain how this pattern works, but the one I like the most is the *producer, dispatcher, consumer* example.

In this example, the producer is represented by "Farmer Bill"; Farmer Bill produces different types of fruits and vegetables throughout the year .

In the middle, we have the dispatcher (mediator), who handles all communications between the producer and the consumer.

Mediator Pattern **FIGURE 1**



On the other side, we have consumers. You can think of the consumer as a grocery store, a restaurant, or any entity that wants to do something with fruits and vegetables, such as labeling, selling, or genetically altering them.

In Figure 1, the role of adding a consumer request with the dispatcher is in the hands of the **client**. The client is the one that sets up and puts this whole process in motion.

Registration/Subscription

The consumer must be registered ahead of time with the dispatcher to *listen* for specific *events*. This way the dispatcher can notify consumers whenever a product they are interested in is available; see Figure 2.

NOTE: *The producer and consumer both know about the dispatcher, but producers and consumers do not have to know about each other.*

Event Dispatching

Whenever Farmer Bill harvests *x* product, he **notifies** the dispatcher by sending a message saying "the **event** `harvest` just happened." Farmer Bill has the option to send along with this event message an **event object**; this object may contain information about the `harvest` event and even the product itself. Farmer Bill does not have to know anything about what happens next once he has notified the dispatcher of the event.

In Figure 3, Farmer Bill harvests and sends carrots and apricots to the dispatcher. At this point the dispatcher handles all **notifications** to the consumers that have registered, or ***subscribed***, to listen for `harvest carrots` and `harvest apricots`.

Two-way Communication

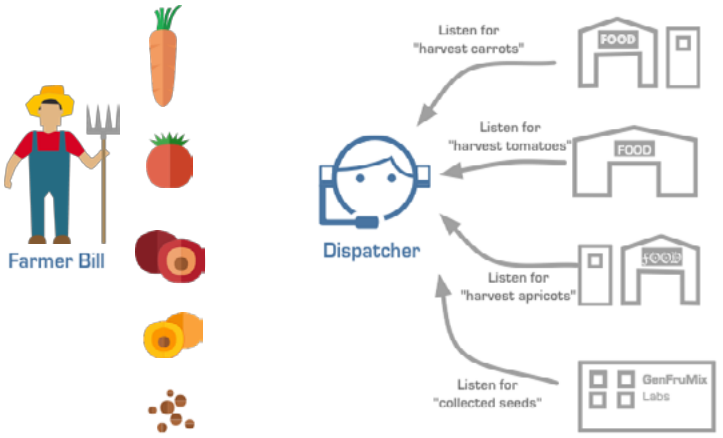
The Mediator pattern can also be used to establish a two-way communication between the producer and the consumer, again without them knowing about each other. This type of communication is very powerful, as the producer can expect some things to change after the event has been dispatched.

Suppose that Farmer Bill has both plum and apricot trees; he has collected the seeds from the fruit and is planning to use them to plant more apricot and plum trees. Before he plants the seeds, a message is sent to the dispatcher notifying of the event `collected seeds` along with a collection of seeds. The dispatcher sends the seeds to GenFruMix Labs, a genetic lab that requested the dispatcher to inform them anytime there are seeds available.

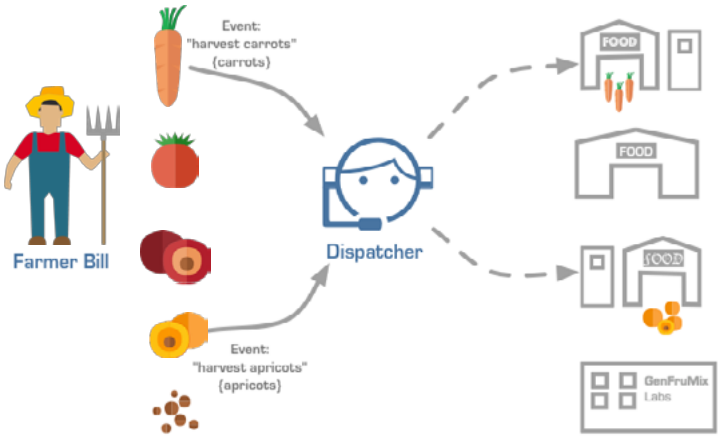
GenFruMix Labs genetically modifies the original seeds and creates a new collection of pluot seeds. Pluot is a delicious fruit that is the result of genetically crossing plums and apricots. These new seeds are sent back to the dispatcher and given back to Farmer Bill, who takes the new seeds and plants them. The result is a different type of tree from the ones originally intended, but the new trees will produce a much tastier fruit!

In real life, this is an unlikely scenario, but it does help us visualize how the dispatcher can be used to extend and modify behavior within an application.

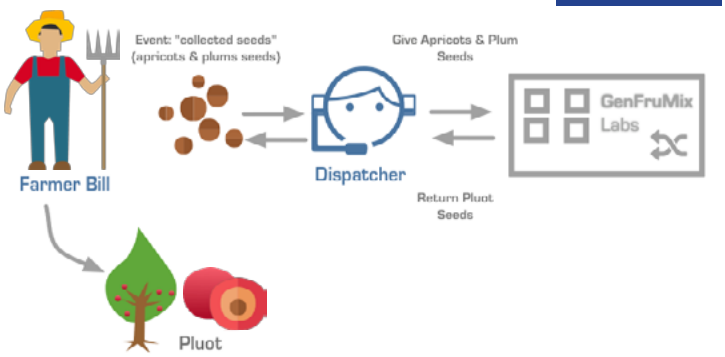
Subscribing to events **FIGURE 2**



Dispatching events **FIGURE 3**



Dispatching events **FIGURE 4**



The Symfony Event Dispatcher

The Symfony Event Dispatcher implements the Mediator pattern, providing the means to decouple *producers* and *consumers*.

In Symfony, the consumers are referred to as *listeners*. Listeners are callables [3]. They can be objects or functions.

The component at its core defines two interfaces, the `EventDispatcherInterface` and the `EventSubscriberInterface`, and a class called `Event`.

3 php.net Callables: http://php.net/language.types.callable

The `EventDispatcherInterface` has methods to add, remove, get, and check for existing listeners. In addition to these, the interface also defines a method to dispatch events, and two methods to add and remove `subscribers`; see Listing 3. A `subscriber` is a specialized `listener` that must implement the `EventSubscriberInterface`, which we will learn more about later.

Listing 4 has three listeners; these are simple callables that represent consumers in the "Farmer Bill" example. The first listener is a "label factory" and it attaches a label to the product (think about those annoying stickers that are on your apples or tomatoes). The second listener is a grocery store, and as soon as it has a product it is listening for, it will put it up for sale. The third listener is GenFruMix Labs; this replaces the product "apricot and plum seeds" with "pluot seeds".

The component comes with three implementations for the `EventDispatcherInterface`: the `EvenentDispatcher`, the `ContainerAwareEventDispatcher`, and the `ImmutableEventDispatcher`. The implementation you will most likely use for your projects is the `EventDispatcher`. The other two are an extension and a wrapper of this one.

Listing 5 shows how the event dispatcher adds the listeners, from Listing 4, via the `EventDispatcher::addListener()` method. The first parameter is the event that it will listen for. The second parameter is the callable, and the third parameter is a priority value, and it dictates the order in which listeners get called. The higher the number, the higher the priority (zero is used as the default value). In this case, the label is placed on the product before it is sent to other listeners.

Note: *The* `LabelListener` *can be reused in several locations; as long as there is a product in the event class, it will work properly.*

Every time a producer needs to dispatch an event, it calls the dispatcher method `EventDispatcher::dispatch()` along with a name for that event; see Listing 6. As an optional parameter, the producer may choose to use an `Event` object. If no event object is passed to the method, one is created internally; each listener gets this event object plus the event name and an instance of the event dispatcher.

Listing 6 represents the producer code, and in this example Farmer Bill harvests **carrots** and collects **apricot and plum seeds**. For each of these events the dispatcher (which is the same object created in Listing 4) is used to dispatch the appropriate event. Because we had listeners listening for these events, we can see in the output that all products are labeled, the grocery store sells carrots, and Farmer Bill ends up planting Pluot trees.

```
Grocery is selling: Carrots produced by Farmer Bill
GenFruMix is mixing: Apricot and Plum Seeds produced by
Farmer Bill
Farmer Bill plants Pluot seeds
```

If you remove the code that adds the `$labelListener` the behavior of the program is going to change, as labels will not be added to the products.

```
Grocery is selling: Carrots
GenFruMix is mixing: Apricot and Plum Seeds
Farmer Bill plants Pluot seeds
```

Moreover, if we do not add the `$genFruMixListener` Farmer Bill only plants plum and apricot seeds.

Event Dispatcher Interface **LISTING 3**

```php
<?php
namespace Symfony\Component\EventDispatcher;

interface EventDispatcherInterface
{
    public function dispatch($eventName, Event $event = null);

    public function addListener($eventName, $listener, $priority = 0);

    public function addSubscriber(EventSubscriberInterface $subscriber);

    public function removeListener($eventName, $listener);

    public function removeSubscriber(EventSubscriberInterface $subscriber);

    public function getListeners($eventName = null);

    public function hasListeners($eventName = null);
}
```

Listener examples **LISTING 4**

```php
<?php
use Symfony\Component\EventDispatcher\EventDispatcher;
use Symfony\Component\EventDispatcher\GenericEvent;

// Listener definition, these can be functions or classes.
// These represent the consumers.
$labelListener = function(GenericEvent $event){
    $product = $event['product'];
    $event['product'] = $product
        . ' produced by Farmer Bill';
};
$groceryListener = function(GenericEvent $event){
    echo "Grocery is selling: " . $event['product'] . PHP_EOL;
};
$genFruMixListener = function(GenericEvent $event){
    echo "GenFruMix is mixing: " . $event['product'] . PHP_EOL;
    $event['product'] = 'Pluot seeds' . PHP_EOL;
};
```

Dispatcher example **LISTING 5**

```php
<?php

// Setup dispatcher, this code is setup by the client
$dispatcher = new EventDispatcher();
$dispatcher->addListener('harvest.carrots', $groceryListener);
$dispatcher->addListener('harvest.carrots', $labelListener, 1);
$dispatcher->addListener('collected.seeds', $genFruMixListener);
$dispatcher->addListener('collected.seeds', $labelListener, 1);
```

Producer example **LISTING 6**

```php
<?php
use Symfony\Component\EventDispatcher\GenericEvent;

$event = new GenericEvent();
$event['product'] = 'Carrots';
$dispatcher->dispatch('harvest.carrots', $event);

$event['product'] = 'Apricot and Plum Seeds';
$dispatcher->dispatch('collected.seeds', $event);
print "Farmer Bill plants ".$event['product'];
```

```
Grocery is selling: Carrots
Farmer Bill plants Apricot and Plum Seeds
```

Generic Events Versus Custom Events

In most cases, you will be creating custom objects that extend the `Event` class. These objects contain any information relevant to the event and the means to stop the use of the Dispatcher from dispatching any events via the `Event::stopPropagation()` method.

The Event Dispatcher Component provides a `GenericEvent` class that implements `ArrayAccess` and `IteratorAggregate`, allowing you to use these events using array syntax. This is the same event object utilized in the example above, and as you can see it can be quite easy to use.

While it is convenient to use the `GenericEvent`, you may want to consider developing custom events with the appropriate setters and getters for each event. Doing so makes it easier for you or other developers working on the listeners to find out what information is available via the event, and provide type hinting in some IDEs. Unless you have documented all the properties available in the generic event, it will be hard to keep track of this information.

Listener classes **LISTING 7**

```
01. <?php
02. use Symfony\Component\EventDispatcher\GenericEvent;
03.
04. class LabelListener
05. {
06.     public function labelProduct(GenericEvent $event){
07.         $product = $event['product'];
08.         $event['product'] = $product  . ' produced by Farmer Bill';
09.     }
10. }
11. class GroceryListener
12. {
13.     public function onHarvest(GenericEvent $event){
14.         print "Grocery is selling: " . $event['product'] . PHP_EOL;
15.     }
16. }
17.
18. class GenFruMixListener
19. {
20.     public function onCollectSeeds(GenericEvent $event){
21.         print "GenFruMix is mixing: " . $event['product'] . PHP_EOL;
22.         $event['product'] = 'Pluot seeds' . PHP_EOL;
23.     }
24. }
```

Listeners and Subscribers

Listeners

In most cases, listeners come in the form of ***Listener Classes*** or ***Subscribers***. If you create classes for the listeners in Listing 4, we will end up with Listing 7.

Moreover, these listeners would be added to the dispatcher as shown in Listing 8.

Adding listener classes **LISTING 8**

```
01. <?php
02. $groceryListener = new GroceryListener();
03. $labelListener = new LabelListener();
04. $genFruMixListener = new GenFruMixListener();
05.
06. $dispatcher = new EventDispatcher();
07. $dispatcher->addListener('harvest.carrots', [$groceryListener, 'onHarvest']);
08. $dispatcher->addListener('harvest.carrots', [$labelListener, 'labelProduct'], 1);
09. $dispatcher->addListener('collected.seeds', [$genFruMixListener, 'onCollectSeeds']);
10. $dispatcher->addListener('collected.seeds', [$labelListener, 'labelProduct'], 1);
```

For listeners, the client is responsible for telling the dispatcher what event each listener is registered to.

Subscribers

Subscribers are classes that provide an alternative way to listen for events. Subscribers are unique in that they can tell the dispatcher exactly which events they are listening for. All subscribers must implement the `EventSubscriberInterface`. This interface defines a static method that must return an array describing the methods in the class, that should be called by the dispatcher for each event they subscribe.

The `EventSubscribers::getSubscribedEvents()` must return an array with one or more of the following options:

```
['eventName' => 'methodName']
['eventName' => ['methodName', $priority]]
['eventName' => [['methodName1', $priority], ['methodName2']]]
```

In Listing 9 all classes from Listing 7 have been moved to a single subscriber.

Adding a subscriber to the dispatcher is easier than adding listeners, since you do not have to be concerned about setting the event name or a priority:

```
$dispatcher = new EventDispatcher();
$dispatcher->addSubscriber(new FarmSubscriber());
```

Creating Custom Events

While it is not necessary to pass any information to the listeners, most of the time you will want to. For this purpose, you can create a custom event class that extends the `Symfony\Component\EventDispatcher\Event`. This class holds all of the information that is related to the specific event. For example, any `FarmEvent` class would contain a product, and a simple implementation of such an event would look like Listing 10.

Producers would create events and pass the product using construction injection.

```
$event = new FarmEvent($product);
```

Listeners take those events and can use the setters and getters to use or modify the data from the events.

```
$product = $event->getProduct();
$product = 'New product';
$event->setProduct($product);
```

LISTING 9 Event Subscriber

```
01. <?php
02. use Symfony\Component\EventDispatcher\GenericEvent;
03. use Symfony\Component\EventDispatcher\EventSubscriberInterface;
04.
05. class FarmSubscriber implements EventSubscriberInterface
06. {
07.    public static function getSubscribedEvents() {
08.       return [
09.          'harvest.carrots' => [
10.             ['onHarvest'], ['labelProduct', 1],
11.          ],
12.          'collected.seeds' => [
13.             ['onCollectSeeds'], ['labelProduct', 1]
14.          ]
15.       ];
16.    }
17.
18.    public function labelProduct(GenericEvent $event){
19.       $product = $event['product'];
20.       $event['product'] = $product . ' produced by Farmer Bill';
21.    }
22.
23.    public function onHarvest(GenericEvent $event) {
24.       print "Grocery is selling: " . $event['product'] . PHP_EOL;
25.    }
26.
27.    public function onCollectSeeds(GenericEvent $event){
28.       print "GenFruMix is mixing: " . $event['product'] . PHP_EOL;
29.       $event['product'] = 'Pluot seeds' . PHP_EOL;
30.    }
31. }
```

LISTING 10 Custom farm event

```
01. <?php
02. use Symfony\Component\EventDispatcher\Event;
03.
04. class FarmEvent extends Event
05. {
06.    private $product;
07.
08.    public function __construct($product) {
09.       $this->product = $product;
10.    }
11.
12.    public function getProduct() {
13.       return $this->product;
14.    }
15.
16.    public function setProduct($product) {
17.       $this->product = $product;
18.    }
19. }
```

Extending the Say Command

To demonstrate how to use the dispatcher in an existing application, we will use one of the commands from the article ***Writing Better CLI Tools with Symfony Components***[4]. In this article, we learned how to build basic commands and how to reuse them in other commands. We built three commands:

1. `SayCommand`: writes to the output anything that is passed as an argument.
2. `FortuneCommand`: it opens a file called fortunes, which contains several thousand quotes. The file is exploded into an array, and a random quote is selected and used in the `SayCommand`.
3. `ElephantFortuneCommand`: displays an ASCII elephant with a simple speech bubble containing a fortune. This command uses the `FortuneCommand` to write the quote to the output.

Before we begin, install the event dispatcher using Composer:

```
composer require symfony/event-dispatcher
```

We are going to refactor the `SayCommand` to take an event dispatcher and dispatch a single event before writing the argument to the output. In this case, the command acts as the producer. See Listing 11.

Lines 26–29 display the modification needed in the `execute` method to dispatch a single event if an event dispatcher object has been set. A new method, `setDispatcher`, has been created and must be used by the client to inject an event dispatcher to the command; alternatively, this could have been done via constructor injection.

Since the `SayCommand` can dispatch events, instead of using other commands to display the fortunes or the ASCII elephant, we will create a single Event Subscriber to modify the behavior of the say command. Listing 12 shows the new `SaySubscriber`.

Now to wrap it all up we must modify the app file, which is the entry point to the CLI application, and set everything up.

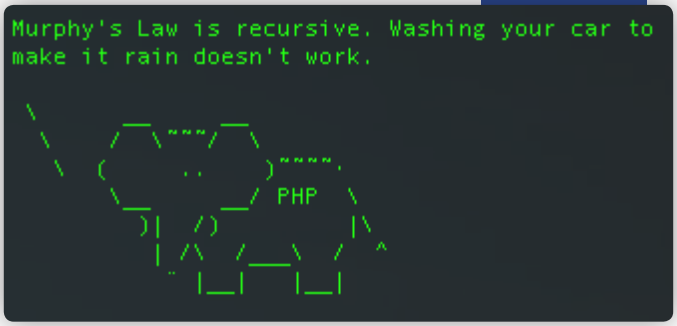
```
use PHPArch\Command\SayCommand;
use Symfony\Component\Console\Application;
use Symfony\Component\EventDispatcher\EventDispatcher;
use PHPArch\Event\SaySubscriber;

$dispatcher = new EventDispatcher();
$dispatcher->addSubscriber(new SaySubscriber());

$application = new Application();
$sayCommand = new SayCommand();
$sayCommand->setDispatcher($dispatcher);
$application->add($sayCommand);
$application->run();
```

Conclusion

The Symfony Event Dispatcher component is a simple yet powerful component that provides the means to extend the behavior of your application without it knowing anything about these extensions. This component gives you or third-party developers the ability to create *plugins* that *hook* onto different parts of the application life cycle.

We looked extensively at how the basic event dispatcher worked but did not study the alternative implementations provided by the component, specifically, the `ContainerAwareEventDispatcher`,

FIGURE 5



4 *See the January issue:* https://www.phparch.com/magazine/2016-2/january

which relies on the Symfony Dependency Injection component. This implementation of the dispatcher is worth using in larger projects as it provides the means to register listeners as services that get instantiated only when they are needed.

For more information on how to use the Dispatcher, see the Symfony documentation or take a look at projects that use the dispatcher such as the Symfony HTTP Kernel.

Juan Manuel *lives in San Diego, California where he works for MindTouch as Senior Software Engineer. His passions include photography, reptiles and SOLID programming. @onema.*

Requirements:

- PHP: 5.3+
- Composer
- Symfony Event Dispatcher Component
- Symfony Console Component

Refactored SayCommand **LISTING 11**

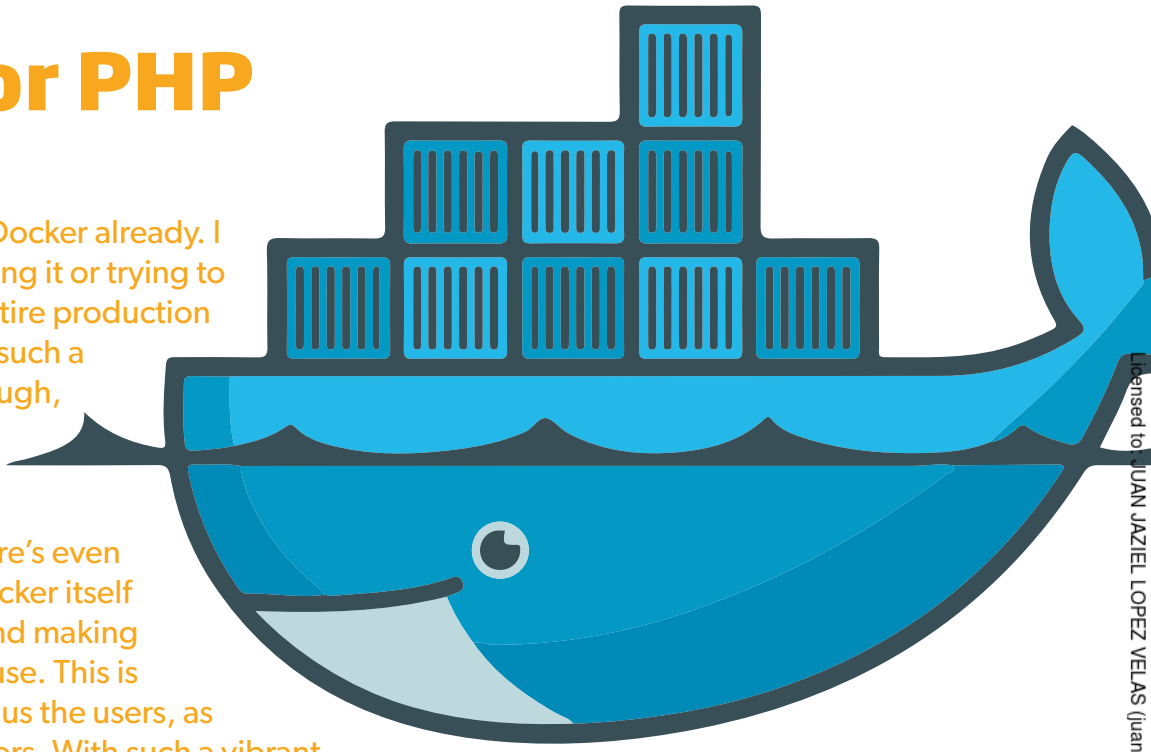
```
<?php
// src/Command/SayCommand.php
namespace PHPArch\Command;
use PHPArch\Event\SayEvent;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\EventDispatcher\EventDispatcherInterface;

class SayCommand extends Command
{
    /**
     * @var EventDispatcherInterface
     */
    private $dispatcher;
    protected function configure(){
        // ...
    }

    public function execute(InputInterface $input, OutputInterface $output) {
        if (isset($this->dispatcher)) {
            $event = new SayEvent($input);
            $this->dispatcher->dispatch('say.before', $event);
        }
        $phrase = $input->getArgument('phrase');
        $output->writeln($phrase);
    }

    public function setDispatcher(EventDispatcherInterface $dispatcher) {
        $this->dispatcher = $dispatcher;
    }
}
```

Say Event Subscriber **LISTING 12**

```
<?php
namespace PHPArch\Event;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
class SaySubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents(){
        return ['say.before' => [
            ['onBeforeSaySetFortunes', 10],
            ['onBeforeSaySetElephant']
        ]
        ];
    }

    public function onBeforeSaySetFortunes(SayEvent $event){
        $input = $event->getInput();
        $dir = __DIR__.'/../Resources/fortunes';
        $fortunes = explode('%', file_get_contents($dir));
        $count = count($fortunes)-1;
        $rand = rand(0, $count);
        $fortune = $fortunes[$rand];
        $input->setArgument('phrase', $fortune);
    }

    public function onBeforeSaySetElephant(SayEvent $event){
        $input = $event->getInput();
        $phrase = $input->getArgument('phrase');
        $dir = __DIR__.'/../Resources/elephant';
        $elephant = file_get_contents($dir);
        $input->setArgument('phrase', $phrase.$elephant);
    }
}
```

Docker for PHP

Ben Hosmer



You've probably heard of Docker already. I won't spend any time hyping it or trying to get you to migrate your entire production system to it right now. For such a relatively new project, though, it has garnered a good bit of attention and numerous startups are blossoming around it. There's even a bit of competition for Docker itself around Linux containers and making them easier to adopt and use. This is definitely a good thing for us the users, as well as system administrators. With such a vibrant and competitive ecosystem, we benefit from the rapid development process and the attention being given to containers right now.

You might be surprised to know that container support has been in the Linux kernel since 2.6.24, which was released in January of 2008. In fact, if you're using Systemd[1], you've already got built-in container functionality. If you aren't familiar with containers, they're virtualization on the operating system level that allows isolation of Linux systems on a single host. Containers are similar to a virtual machine, because they allow you to run different operating systems and applications, and separate them from the host machine. Unlike virtual machines, containers don't have the overhead of an entire operating system. Containers work by sharing the kernel of the host machine that they're running on, while allowing complete isolation of processes. This is what allows you to run two different version of PHP on the same machine at once.

In this article, I'll show you how you can start using Docker[2] today for your local development needs. I'll pose some questions to get you thinking about your current development workflow, and hopefully help you see where Docker can fit in and even replace some of your local tools. I'll explain some of the basic concepts of using Docker for local development and how this model differs from using virtual machines. I'll show you how to build two containers, a database and a web server. Then I'll illustrate how easy it is to launch them. Later I'll describe how you can orchestrate multiple containers to launch a typical LAMP stack. Last, I'll show you how to use the Docker hub to pull down official container images that are already built, which you can either use as written or modify and build upon for your own needs.

Why Docker?

Having kernel support and container functionality is one thing, but making it easy to use is another. This is where Docker excels. If you aren't using a Linux system, don't worry—you'll see later how you can experiment with Docker. By managing the lower-level processes and helping to solidify the de facto container specification[3], Docker has moved the Linux container out of the domain of the seasoned system administrator or kernel developer, and made it a useful and reliable tool for the busy PHP developer. While this article focuses on LAMP stack development, the concepts and practices can be applied to almost any language and development platform.

What Does Your Current Local Development Look Like?

Are you like me, having started with MAMP quite a few years ago? I graduated to using the default Apache/PHP that came installed on my Mac, and crossed my fingers hoping that nothing broke after each system update. After that, I used Homebrew to install different versions of PHP. Until I started using Docker, I used a combination of Vagrant boxes provisioned specifically for each client.

Docker can help to:

- Emulate production in Development, Test, and Stage.
- Onboard new developers.
- Support multiple versions and simplify switching between them.
- Simplify your local development setup.
- Better utilize local system resources.

There are some pretty good reasons for not switching your entire production system over to Docker right now. Operations may not yet be knowledgable enough about how Docker works. Maybe you

1 *SystemD:* http://www.freedesktop.org/wiki/Software/systemd

2 *Docker:* http://www.docker.com

3 *Open Container Initiative:* https://www.opencontainers.org

aren't sure about the security of containers. Using Docker to host mission-critical applications requires a different approach, just like using it for development. Operations will need to get up to speed on how to use it and how to deploy it.

At this point, I hope I've led you to wonder whether you could start using it in development and begin to share the knowledge you gain with the rest of your team. I won't promise it solves all of the above issues, but like Vagrant a few years back, it can get you closer to the ideal state where development, test, stage, and production are as close as possible.

With Docker, each container runs in a separate space. If you've used `chroot` environments before, you'll find Docker similar in some respects, but much easier to set up and use. You can have one web server running the exact same version of Ubuntu for the client still using PHP 5.2 and then switch to PHP 7 as quickly as you can type `$ docker start`.

Because of this, you have to approach working with containers with a different mindset. They aren't virtual machines and shouldn't be treated as such. If you find you're logging in to your containers and running multiple processes within them, you should stop and reevaluate why. The general consensus is to use one container for each process. Translating this to our LAMP development environment, we've now got Linux running on the host, sharing the kernel with any distribution you like—Debian, Ubuntu, CentOS, and, for the brave, even Arch—all on one single host machine. Next, the web server runs in one container and the database in another container. You need two containers at least. Why? Docker watches for a process and when that process ends, the container stops. As long as that process continues to run, the container stays running. A typical virtual machine mindset might use supervisord[4], put the database and web server in one container and run the supervisor process, and then in turn use that to keep other processes running. However, by doing it that way you're losing portability and not taking full advantage of the container. You also have a very large container that could be difficult to troubleshoot and transfer to other developers, or even a shared development hosting platform. Embracing the Unix philosophy of "do one thing and do it well," you'll gain much more flexibility by separating your containers into individual units of functionality.

4 *Supervisord: http://supervisord.org*

Getting Started

If you aren't using Linux natively, you'll need some helper software to get up and running. Because Docker works by using the Linux kernel on the host machine, it is only available for Linux. Don't worry if you're using a Mac or Windows box, though: there are numerous ways for you to still use Docker.

Some options are:

1. Boot2Docker, http://boot2docker.io
   - Boot2Docker gives you access to Docker commands from your native shell. It actually launches a lightweight Linux virtual machine and runs Docker inside that.
2. Kitematic, https://kitematic.com
   - Kitematic works in a similar fashion to Boot2Docker.
3. Virtualbox, https://www.virtualbox.org and Vagrant, http://vagrantup.com
   - Using a Linux virtual machine, you install Docker there. This works underneath like Kitematic and Boot2Docker: you just have to install and configure Docker yourself in the guest.
4. Docker Toolbox, https://www.docker.com/toolbox
   - Docker Toolbox installs Kitematic and is the Boot2Docker replacement. It had just been released at the time of this writing.

All four options allow you to continue to use your native development tools, like your preferred text editor and IDE. If you prefer a GUI, you might like Boot2Docker or Kitematic. If you're already familiar with Vagrant, you could download a base box of your preferred operating system, and install Docker there. Docker Installation[5] documentation is available for a wide variety of operating systems. Whichever one you choose, review the documentation for each and install it before continuing any further. In the next section, we'll build our first container.

5 *Docker installation: https://docs.docker.com/installation/*

Creating Your First Container

The Dockerfile

You could quite easily use the official MariaDB Container[6] and skip building one from scratch. I'm going to explain the `Dockerfile` and building process here, using one of my own, just so that you're familiar with it. If you want to skip this building section, you could just use the official images provided at the Docker Hub[7]. Using others' images is covered in a later section. It is important, though, to have a basic understanding of how to build your own containers. As you start to use Docker, you'll inevitably want to customize your container for your own team's needs. I'd highly recommend sticking to officially provided containers, though. The security implications of blindly running an un-trusted container can be grave. Despite ongoing attention to the security of containers by the community at large, you're still running a container as root within its host operating system.

The `Dockerfile`, the basis for building your containers, is written in a fairly concise manner and contains all of the instructions that Docker will need to build your container. This is where you specify your operating system, any packages you want installed, and the final command to run when it starts.

Let's start with a database.

Listing 1 shows my own MariaDB database `Dockerfile`.

You can find more in-depth information about Dockerfile[8] specifications from Docker's documentation, but I'll also break down this specific `Dockerfile`.

Let's start with the `from centos:centos6` line. Docker works by adding layers to base images. Images can be considered similar to images in the virtual machine world, but they're much more lightweight. In this example, I'm getting the `centos6` image from the official `centos` repository.

Next is the `maintainer` section. Here you can identify who maintains this particular `Dockerfile` for future reference.

The `ADD mariadb.repo /etc/yum.repos.d/` copies the `mariadb.repo` file, which is

6 *MariaDB Container: https://hub.docker.com/_/mariadb/*

7 *Docker Hub: http://hub.docker.com*

8 *Dockerfile: https://docs.docker.com/reference/builder/*

LISTING 1

```
01. from centos:centos6
02.
03. MAINTAINER Tyrell Wellick <t.wellick@evilcorp.com>
04.
05. Add mariadb.repo /etc/yum.repos.d/
06.
07. RUN yum -y update --disablerepo=mariadb && yum -y install MariaDB-server MariaDB-client && yum clean all -y
08. ADD server.cnf /etc/my.cnf.d/server.cnf
09. ADD start /usr/bin/start-mysql
10. RUN chmod +x /usr/bin/start-mysql
11.
12. # A more permanent place to store log files and the database
13. VOLUME ['/mysql-data/']
14.
15. EXPOSE 3306
16. CMD ["/usr/bin/start-mysql"]
```

located relative to the `Dockerfile` itself locally to the container. If you aren't familiar with Linux repositories, this is a `yum`[9] based text file containing metadata for the operating system to use to install the database. Here, it is being copied to the container's `/etc/yum.repos.d/` directory. This is specific to your container's operating system. This example is using the CentOS Linux server operating system. I am also installing MariaDB from the official MariaDB repository[10].

You'll need a file called `mariadb.repo` with the contents shown below. You'll also need `server.cnf` with your MariaDB configuration to copy into the container.

```
[mariadb]
name = MariaDB
baseurl = http://yum.mariadb.org/10.1/centos6-amd64
gpgkey=https://yum.mariadb.org/RPM-GPG-KEY-MariaDB
gpgcheck=1
```

`RUN` instructs the container to actually run the specified command when it is being built. In this case I'm using the CentOS package manager, yum, to install MariaDB. This is what you would type in the terminal if you were actually installing the database manually.

RUN and Container Size

Each `RUN` command adds an additional layer to your container. I've concatenated each separate run command together into one run statement to keep my container size smaller. Tutum[11] has more information about reducing your container's size.

In the MariaDB `Dockerfile` notice the difference between:

```
RUN yum -y update --disablerepo=mariadb && yum -y install
 MariaDB-server MariaDB-client && ...
```

and

```
RUN yum -y update --disablerepo=mariadb
RUN yum -y install MariaDB-server MariaDB-client
...
```

After building the container with the run commands concatenated, my resulting image size is 876.7 MB. I used the `$ docker images` command to get information about my image:

```
REPOSITORY TAG      IMAGE ID       CREATED         VIRTUAL SIZE
mariadb    latest   8a641cd01987   8 minutes ago   876.7 MB
```

Here is the size of the image after putting each run command on a separate line:

```
REPOSITORY TAG    IMAGE ID     CREATED             VIRTUAL SIZE
mariabig   latest f0b910f39ec9 About a minute ago  917.7 MB
```

By simply initiating the `run` operations in one command, I've saved 41MB on my container size. This is a somewhat trivial amount for this example, but you can imagine that with more complex `run` operations the savings can add up.

Up next, `VOLUME ['/mysql-data/']` creates a volume within the container. We'll talk more about using volumes shortly with an explanation on ephemeral data and shared file systems.

The `EXPOSE 3306` allows access to port 3306, the default MySQL port on the container, so that we can access it from outside the container. By default, Docker only allows network access from within containers, so this needs to be specified.

The final directive, `CMD`, is the process that Docker will watch for and start when the container starts. I've created a small helper shell script that checks to see if the `/mysql-data` directory is empty. If it is, it initializes a new database there. If it isn't empty, it simply starts the `mysql` daemon.

It looks like Listing 2.

Earlier I mentioned that containers are stateless and ephemeral. This means that any data stored within them should be considered disposable and will disappear when the container is removed. Needing to reimport client production databases on a daily basis could be counterproductive, so I use a shared file system between the container and the host to house the actual database files. This also allows me to access the logfiles for any troubleshooting needs that might arise. I can easily start, stop, and remove the container without worrying about losing the data. Because no data is stored within this container it can also be used as is in production or on another developer's machine without moving databases back and forth.

After creating a `Dockerfile` and the necessary configuration for it, it's time to build a Docker image. If you're a Vagrant user, images can be considered loosely analogous to boxes.

Within the same directory as your `Dockerfile`, use the `docker build` command to build an image from the `Dockerfile`:

```
docker build --tag=benhosmer/mariadb .
```

The `--tag` flag lets you give your image a humanized name that you can easily reference later.

As you watch the build command run, you can see the packages being installed and the configuration taking place.

After the build command completes, you will now have a database container ready to use!

Let's start a database container now and connect to it:

```
docker run -v /home/vagrant/mysql-data:/mysql-data -d \
 -p 3306:3306 --name="db.dev" benhosmer/mariadb:latest
```

If you're using Vagrant or native Linux, you'll need to set up a host name in your `/etc/hosts` file so you can connect, something like this:

```
192.168.33.123 db.dev
```

9 `yum`: *https://fedoraproject.org/wiki/Yum*

10 *MariaDB repository:* *http://phpa.me/mariaDBdl*

11 *Tutum:* *http://wp.me/p4gI1Q-eD*

FIGURE 1

```
CONTAINER ID        IMAGE                      COMMAND                  CREATED             STATUS              PORTS                    NAMES
38def25e83a6        benhosmer/mariadb:latest   "/usr/bin/start-mysql"   5 seconds ago       Up 3 seconds        0.0.0.0:3306->3306/tcp   db.dev
```

You can then connect to your database using a MySQL client:

```
mysql -u root -p -h db.dev
```

NOTE: This is from your local machine, not the virtual machine itself, although you could install mysql there too and forego the `/etc/hosts` modifications and just connect from `localhost`, like this: `$ mysql -u root -p`. Docker will return a hash of your container when it starts.

You can verify your container is running with `docker ps`, sample output is in Figure 1.

With my `Dockerfile` and helper script, I set the user name as `root` and the password as `root`. Obviously you'll want to change this for production to something a bit more secure!

Here is a breakdown of the flags for the `$docker run` command:

- `-d`: Start the container in daemonized mode. This keeps the container running as long as the process continues to run and leaves your terminal ready for new commands.
- `-p 3306:3306`: Forward port 3306 on the host to port 3306 on the container.
- `-v /home/vagrant/mysql-data:/mysql-data`: Share the host's file system and mount it within the container at `/mysql-data`
- `--name="db.dev"`: This is an arbitrary name you can give your container. It makes it easier to reference it during stop, start, and linking.
- `benhosmer/mariadb:latest`: The repository, image, and version to use for this particular container.

LISTING 2

```
01. #!/bin/bash
02.
03. USERNAME=root
04. PASSWORD=root
05. DATABASE=*
06. REMOTE=%
07.
08. # test if DATA_DIR has content
09. if [[ ! "$(ls -A /mysql-data)" ]]; then
10.     echo "Initializing MariaDB at /mysql-data"
11.     # Copy the data that we generated within the container to
12.     # the empty DATA_DIR.
13.     cp -R /var/lib/mysql/* /mysql-data
14.     rm -rf /var/lib/mysql
15.     chown -R mysql:mysql /mysql-data
16.     usermod -a -G games mysql
17.     usermod -d /mysql-data mysql
18.     mysql_install_db
19.     service mysql start
20.     mysql -e "CREATE USER 'root'@'%' IDENTIFIED BY 'root';"
21.     mysql -e "GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' WITH GRANT OPTION;"
22.     mysql -e "GRANT ALL PRIVILEGES ON $DATABASE.* TO '$USERNAME'@'$REMOTE'  IDENTIFIED BY '$PASSWORD'"
23.     mysql -e "FLUSH PRIVILEGES;"
24.     mysql -e "select user,host from mysql.user;"
25.     service mysql stop
26. fi
27.
28. mysqld --user=mysql
```

NGINX/PHP Etc.

Now that you've got a database container, the next step is a web server. I'm particularly fond of NGINX[12], so I'll use that as an example here. You could also use the official PHP[13] container and skip the building process.

Listing 3 shows the NGINX `Dockerfile`:

Again, build the container using the `Dockerfile`:

```
docker build --tag=benhosmer/nginx .
```

After the container is built, start it (command all on one line):

```
docker run -d -v /vagrant/docroot:/docroot/drupal           \
-v /vagrant/conf.d/:/conf.d/ -v /vagrant/logfiles/:/logfiles \
-p 80:80 --link="db.dev:db.dev" --name="drupal.dev"         \
benhosmer/nginx:latest
```

These `run` commands can get tedious. Later on I'll show you how to use `docker compose` to save some typing!

This run command is similar to the first run that we used to start the database container, with one exception: `--link="db.dev:db.dev"`. This flag allows network communication between containers. In this example, I'm linking the new NGINX container with the previously started `db.dev` database container using the `--name="db.dev"`

LISTING 3

```
01. from centos:centos6
02.
03. MAINTAINER Tyrell Wellick <t.wellick@evilcorp.com>
04.
05. Add nginx.repo /etc/yum.repos.d/
06.
07. RUN rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm && yum -y update && yum -y install nginx php-cgi php-dom php-gd php-pdo php-mbstring php-mysql php-pear spawn-fcgi pear && pear channel-discover pear.drush.org && pear install drush/drush && rm -rfv /etc/nginx/conf.d/ && rm -v /etc/nginx/nginx.conf && yum clean all -y
08.
09. ADD nginx.conf /etc/nginx/nginx.conf
10. ADD spawn-fcgi.conf /etc/sysconfig/spawn-fcgi
11. ADD php.ini /etc/php.ini
12.
13. VOLUME /conf.d
14. VOLUME /docroot
15. VOLUME /logfiles
16. EXPOSE 80
17. CMD service spawn-fcgi start && /usr/sbin/nginx
```

12 NGINX: http://nginx.org

13 Official PHP Container: https://hub.docker.com/_/php/

that I supplied when I started it. Using `--name`, as I mentioned earlier, gives you an easy way to reference your container. When Docker starts a container, it assigns it a random IP address within the local Docker network. By using `--name` along with `--link`, you don't have to worry about determining the IP address of each container so that they can communicate with each other: Docker's own DNS handles that for you. If you stop a container, even though it might have a different IP address, you can always find it again by using just its name.

I've also opened port 80 between the host and the container, and set up a shared document root, configuration directory, and logfiles directory between the container and my host.

You now have a web server and database container running. In this example, I've mapped NGINX's `conf.d` directory to a local file system. This way I can add sites and adjust the configuration, and then simply restart the container, which is equivalent to `# service nginx restart` as you normally would when making a configuration change. I can edit these files locally and never need to log in to the container itself.

You can stop the containers using `$ docker stop`:

```
docker stop drupal.dev db.dev
```

This is Docker's *shutdown* command. The container is basically paused; you can think of this as the equivalent of the `vagrant halt` or `suspend` command if you're used to using Vagrant.

You can completely delete a container using the `$docker rm` command like this:

```
docker rm drupal.dev
```

This deletes the container and is equivalent to Vagrant's `vagrant destroy`.

Try it now, but notice how much faster it is.This is the benefit of Docker's shared kernel resources. You don't need to boot an entire operating system just to use a container.

Using Official Images

Both MariaDB[14] and PHP[15] maintain official images at https://hub.docker.com. Official images are blessed by the Docker project and generally maintained by the software project that they reflect or by someone heavily involved with the project. Both projects have excellent documentation about how to use their images. For example, unlike my example MariaDB `Dockerfile`, where I use a simple shell script to set the root database password, the official MariaDB image uses environment variables that you can set at container run time using the `-e` flag:

```
docker run... -e MYSQL_ROOT_PASSWORD=mysecretpassword...
```

The official PHP container even includes a version with Apache already installed.

Because Docker works by layering images, you could, for example, start with the official PHP image and build upon it to install the `php-gd` extension:

```
FROM php:5.6-fpm
RUN apt-get install php5-gd
```

You pull images from other repositories using `$ docker pull`:

```
docker pull mariadb
```

By default, if you don't include a repository specification in your pull, Docker pulls from the official Docker hub library. The location pattern might look like this if you were pulling from your own repository:

```
docker pull benhosmer/mariadb
```

This lets you use your own private images or even your own locally hosted Docker repository.

After pulling, and just like launching a container you built yourself, you start the container using the `repositoryname/image:version` convention:

```
docker run --name="db.dev" mariadb:5.5
```

Here I've omitted the `repositoryname` for brevity, since Docker will automatically use the official one if you don't supply a name.

Orchestrating Containers

You might be tempted to create some shell scripts that start and stop your container groups. This is what I initially did, because typing the `$ docker run` commands can get quite tedious, especially if you have multiple containers that rely on one another. You don't have to any more. Using Docker Compose[16], you can orchestrate the building and starting of multiple containers with a few commands. See the docs to install Docker Compose.

Listing 4 is a simple `docker-compose.yml` file that gets our web stack up and running:

Now, in the same directory as your `docker-compose.yml`, use `$ docker-compose up` to build and then start your web stack with one command.

14 MariaDB Image: https://hub.docker.com/_/mariadb/

15 PHP Image: https://hub.docker.com/_/php/

16 Docker Compose: https://docs.docker.com/compose/

Closing

I switched to using Docker for my local development needs almost a year and a half ago. Previously I struggled with a combination of Homebrew, Vagrant, and native Linux, and all of the woes that come with supporting multiple database and PHP versions for different clients. While the initial setup and tooling of Docker does require a bit of effort, I've found the savings in time and headaches to be well worth it. The ability to almost instantly support not only a Drupal 6 site still running on PHP 5.3 but also a new client using Symfony in the amount of time it takes me to type `docker start` has improved my development workflow tremendously.

I don't have to wait 20 minutes or more to provision a Vagrant box over slow airport internet connections. I'm able to better utilize my systems resources by not needing four or more virtual machines running throughout the day.

Local development machine configuration for new developers has gone from three to four days of setup and training to a few hours.

You might not be ready to integrate Docker into the entire life cycle of your application just yet. But you may find, as I have, that investing in the knowledge and practice of using Docker for local development will prepare you for that future integration or for those times when a client requests it. For myself, I also like the streamlined and simplified development environment and the fact that a fellow developer can duplicate bugs easily because we're working in the same environment.

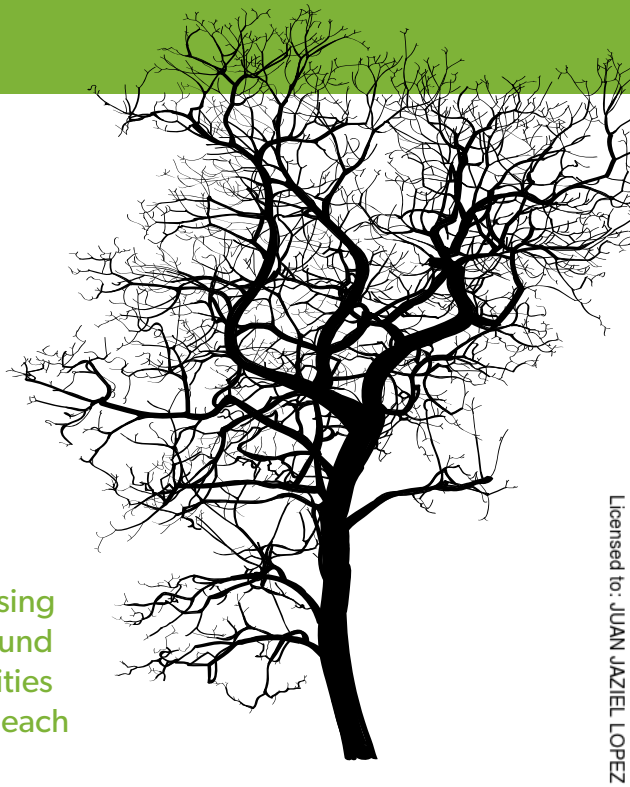
LISTING 4

```
01. web:
02.   build: /vagrant/dockerfiles/nginx
03.   ports:
04.    - "80:80"
05.   volumes:
06.    - /vagrant/docroot:/docroot
07.    - /vagrant/nginx/conf.d:/conf.d
08.    - /vagrant/nginx/logfiles:/logfiles
09.   links:
10.    - db
11.
12. db:
13.   build: /vagrant/dockerfiles/mariadb
14.   ports:
15.    - "3306:3306"
16.   volumes:
17.    - /vagrant/mariadb/mysql-data:/mysql-data
```

Ben Hosmer *is a Drupal Developer, Cloud Evangelist, and System Administrator at RadiantBlue Technologies. He's active in the Drupal Community and authored Getting Started with SALTSTACK[17], a configuration management tool available for free download.*

17 Getting Started with SALTSTACK

Software Branching Strategies: A Git Overview



Georgiana Gligor

Git has become a part of our daily lives as developers. Using branches is now cheaper than ever, but how efficiently are we using this powerful tool? Given that the repository is the common ground where people with so many (sometimes conflicting) responsibilities meet, it is important to employ a branching strategy that makes each of them efficient in performing their duties.

Moreover, teams and people constantly improve, so even if you are already using a strategy, it's never too late to adjust your team's workflow to one that suits you better.

To master git we need to understand the various concepts involved, so that we can see them as building blocks of our branching strategy. After we walk through the main concepts, we are going to take a deeper look at the branching models that have become popular, and describe the criteria for choosing one over the others.

Team Roles

Be it a small or large team delivering software, there are some roles that are distinct in terms of responsibilities, skill set, and means of achieving their goals. Most of the time the same person is wearing several of these hats simultaneously, but I feel it is important to describe them to ensure we are aware of the specific mindset with which they look at the codebase.

Developer

This person is tasked with developing new features, as well as altering and enriching existing ones. His main focus is to make frequent commits that add portions of the required functionality. It is important for him to be able to commit at any time, even half-baked, non-functional code, in order to work independently and to not lose work. As Ben Sandofsky says, these checkpoint commits are "just a cloud-backed undo buffer."

Tester

In order to verify that the code achieves the desired functionality, someone needs to spend time doing the checks. It can be a dedicated QA person, or you can have developers doing BDD, writing functional tests, and running them. How you set this up is highly dependent on your team's working style, but it is a critical aspect that should not be neglected.

Bug Hunter

Once you have detected problems with the functionality, someone needs to hunt down the source of the issue. First, they need to consistently reproduce the problem, trace the source, and separate the change that introduced it. The commits should represent the big-picture history of the code changes, one at the time, like "the authentication feature was added, then there was a small typo fixed in three places, after which we had the sharing articles on Facebook, then a change in terms and conditions happened," etc. The most important thing is to be able to figure out why a certain line of code is there (how it was introduced or changed), which is easier when the commit history is clean, each commit representing one logical aspect.

Package Builder

Software applications are a balanced mix of functional code, third-party dependencies, and application configuration. The end result of a package builder's work is a deliverable containing the code at a certain point in time, enriched with application configuration (think "verbose logs," "CSS minification turned off," etc.).

Deployer

Once a package with the code and configurations is obtained, it needs to be installed in the target environment. By *deployer*, we refer to the person responsible for creating/maintaining the environment, configuring it to receive the build, and installing the build handed over by the package builder. In the happiest scenarios this work is automated, and it's not performed by a real person.

The codebase does not even exist in the eyes of the deployer; the software is just a set of files that need to be installed according to a given set of rules.

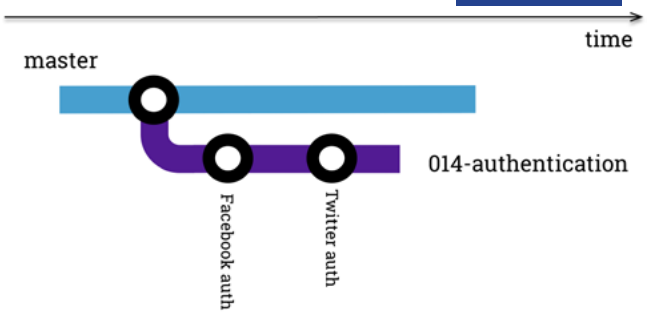
Integrator

In some teams touching sensible long-lived branches is not done by developers, but by a gatekeeper. The integrator is a specialized person responsible for reviewing the code handed over by developers, performing the merge action itself, and any subsequent activities, such as running the functionality checks and deploying it to an environment where others can check as well. Some of these tasks are usually part of the automation process frequently seen in teams doing continuous integration.

Branching Concepts

I aim to introduce a few general concepts that, once clarified, will aid the understanding of the various branching strategies described further. To maintain clarity, it is important to use a common

FIGURE 1



vocabulary when addressing different aspects, so we dedicate a section to clarifying some terminology. When walking through them, please keep in mind that they are not mutually exclusive, so one strategy can encompass several of these concepts at the same time.

Master

When a git repository is first initialized, there is a branch already created and available for use, called "master." In SVN times it used to be called "trunk," while others call it "mainline." Since any repository will use at least one branch, it is common practice to keep master as a long-lived branch in the repository. I will use the terms "master," "trunk," and "mainline" interchangeably for the remainder of this article.

Public Versus Local Branches

The public branches are the ones which have been published (using git push) to the repository. In git lingo, they are also known as remote branches. This means that any person with appropriate permissions can contribute to them.

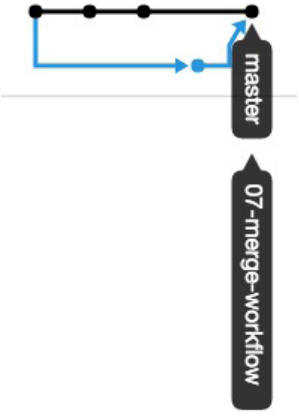
A local branch lives on a person's own machine. The contributor makes changes, commits them, but never publishes the branch to the repository.

Since the history of public branches is seen and possibly used by others (cherry-picking, for example), it is considered bad practice to alter their history using rebase. On the other hand, local branches are a perfect place to clean things up with this method. As Linus put it, "clean your own stuff & don't clean other people's stuff."

Long-lived vs Ephemeral Branches

Most people keep the master branch in their repository, assigning it different purposes (see stable and unstable trunk below). Release branches are around for the time the corresponding software version is supported. We call this type of branch "long-lived." Most of the time, the changes seen in this type of branches are large, like "authentication enabled" or "share articles on Facebook."

FIGURE 2



The ephemeral branches are created for very specific purposes, with a short lifespan, that can be removed shortly after being integrated into a long-lived branch. They can be either public or local. The rules for creating and removing them are quite strict. In fact, most of these rules make different strategies that we'll discuss a bit further.

Topic Branches

For every topic (new feature, bugfix, etc.), a new ephemeral branch is started. Small commits adding partial implementation pieces happen here.

Let's quickly go through the steps needed to start a new topic branch. Assuming we are starting from the master branch, we move to master and grab the latest.

```
$ git checkout master
$ git pull origin master
```

We are going to work on task #14, which is about adding authentication to our project. Our internal convention is to use the `<task-no>-<task-name>` method for naming branches, so we are going to spin off a branch called `014-authentication`:

```
$ git checkout -b 014-authentication
```

It is common to see a lot of commits on topic branches. Developers prefer to put the current unstable state into revision control for fear of losing their work. Nobody produces the perfect code every time, so checkpoints are better than having to rewrite something that was not committed anywhere.

Merge Workflow

The changes made on a topic branch are integrated into a long-lived branch using `git merge`[1]. The implications are very important in terms of commit history: every commit made by the developer will be seen in the long-lived branch where the feature lands.

```
# starting the topic branch, which may be public
$ git checkout -b 025-merge-workflow
# commit some changes
$ git commit -m "introduced explanatory section in readme"
$ git commit -m "new useful articles section"
$ git push origin 025-merge-workflow

# changes happen on the mainline
$ git checkout master
$ git commit -m "improved explanatory opening paragraph"
$ git push origin master

# grab latest from mainline
$ git checkout 025-merge-workflow
$ git pull origin master # might need conflict solving
$ git push origin 025-merge-workflow
```

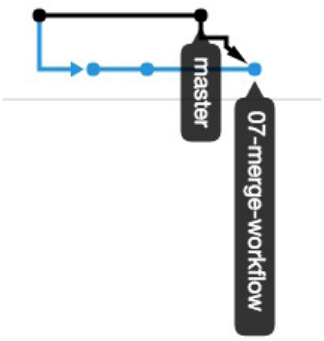
Figure 2 shows the history for integrating a topic branch. You can view the final history in the sample merge repository [2].

```
# integrate the topic branch
$ git checkout master
$ git pull origin master
$ git merge 025-merge-workflow
Updating c328fbb..8c3315d
Fast-forward
 README.md | 7 +++++++
 1 file changed, 7 insertions(+)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
$ git push origin master
```

1 *Git merge documentation:* https://git-scm.com/docs/git-merge

2 *Git merge workflow sample repository:* https://github.com/tekkie/git-merge-workflow/commits/master

FIGURE 3



As you can see in Figure 3, the merge commits are visible in the history. This makes the bug hunter's life quite problematic, as she cannot use tools like git bisect efficiently, even if `git log --first-parent` can improve navigating history. If the number of contributors is small, they can share this information verbally, clarifying the hidden meanings in the history. On the other hand, the topical branch can be shared with more teammates. Furthermore because it's public, there is historical information about the existence of a topic branch, and your continuous integration server can use it to check the sanity of the code.

Rebase Workflow

With a Rebase Workflow[3], the `git rebase`[4] command is used to forward-port local commits to the upstream head. Its power comes from rewriting commit history. Below we are describing a scenario in which we employ a local topic branch, apply its change onto master, and publish it. The resulting master history will show no trace of the originating topic branch. You can view the final history in the sample rebase repository [5].

```
# work on a local topical branch, single change
$ git checkout -b 02-rebase-workflow
# do NOT push to remote
$ git commit -m "added explanatory paragraph"!

# other things are published on master
$ git checkout master
$ git commit -m "added series notice"
$ git push origin master

# we rebase the topical branch single change on
# the master branch
$ git fetch origin
$ git rebase origin/master
$ git checkout master
$ git rebase 02-rebase-workflow
First, rewinding head to replay your work on top of it...
Fast-forwarded master to 02-rebase-workflow.
$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 337 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:tekkie/git-rebase-workflow.git
   5817048..c63367b  master -> master
```

A second scenario that I consider useful is squashing several commits from the topic branch. This means that the selected commits will be seen as a single one in the history, as a result of this operation. In the end they will be applied onto the master branch using rebase.

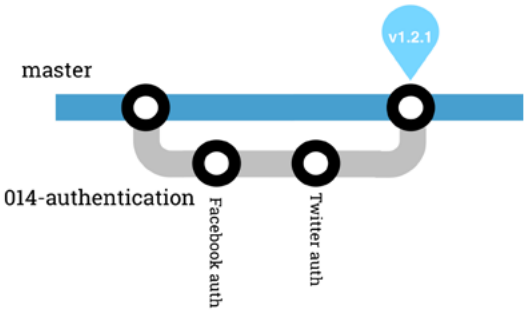
```
# perform multiple changes on topical branch
$ git checkout -b 02-rebase-workflow
# first change
$ git commit -m "improved explanatory paragraph"
# second change
$ git commit -m "added useful links section"

$ git rebase -i master
# squash 2nd change into the first one, to have one
#  monolithical change
pick d217456 improved explanatory paragraph
squash 81971ea added useful links section

# integrate monolith changes from topical branch onto master
$ git checkout master
$ git rebase 02-rebase-workflow
$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 529 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:tekkie/git-rebase-workflow.git
   c63367b..c02e513  master -> master

# we're done working on the local topical branch, let's
# remove it
$ git branch -d 02-rebase-workflow
Deleted branch 02-rebase-workflow (was c02e513).
```

The manual page of git rebase is full of examples and explanations for derived workflows; check them for more advanced usage. Rebase is very powerful, as it effectively rewrites history. This allows contributors to preserve a flat, readable history, and allows easy reversal of an individual public commit (which might have been a few commits back in the local topical branch). It is dangerous (and definitely not recommended) to use this technique on published branches. A drawback of using this approach is the fact that the topic branch is local, so only one contributor will be able to work on it. It also requires the contributor to have a better-than-average familiarity with git; most people approach rebase being intimidated by the complexity behind it. It is also more difficult to ask for a peer review using pull requests, since work lives in the unpublished branch, but patching can be used in this case.

Stable Trunk, aka Trunk-Based Development

The key concept with Trunk Based Development[6] is that master branch should always be stable. Developers develop features on short-lived local topic branches, which get tested and integrated back into the

FIGURE 4



3 *A Rebase-based workflow:* http://phpa.me/22Wy8gN

4 *Git rebase documentation:* https://git-scm.com/docs/git-rebase

5 *Git rebase workflow sample repository:* https://github.com/tekkie/git-rebase-workflow/commits/master

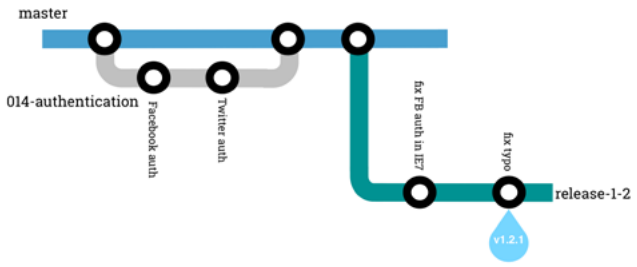
6 *Trunk-based Development:* http://phpa.me/thoughworks-trunk-based-dev

mainline. Even hotfixes are dealt with on topic branches, tested, and reintegrated, just like the new features. Master is used to create builds of the software; any such build is a release candidate, and a select few of them will become releases. Figure 4 shows a stable trunk history.

Unstable Trunk

The master branch is not guaranteed to be stable, and the main development is developers can integrate their work here at any time, without the promise of stability. Both new functionality and bugfixes are applied to the mainline. Some time before the release, a new branch is spun off master, the code is stabilized there, and after that only select commits are picked from the mainline and applied to the release branch. Release candidates and releases are built from this release branch, not from the mainline. You can see this flow in Figure 5.

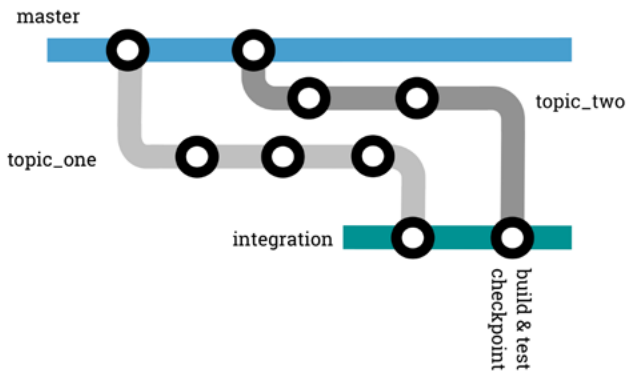
FIGURE 5



Integration Pipelines

Master is stable and its last commit contains the last release. Developers work in parallel, each on their topic branch which was created from the mainline. In order to prepare a new release, a separate integration branch is created, and desired topic branches are applied to it. Software is built from this branch and tested for stability, and release candidates are prepared off it. When there is agreement on the release contents, this integration branch is merged back to master and the release is prepared. Usually the integration branch is discarded after use, or reset to reflect the latest release.

FIGURE 6



This concept is particularly suited for agile teams, who start their topic branches at the beginning of the sprint. The topics which are done make it to the integration branch, and those not yet complete will be easily carried over to the next sprint.

Branching Strategies

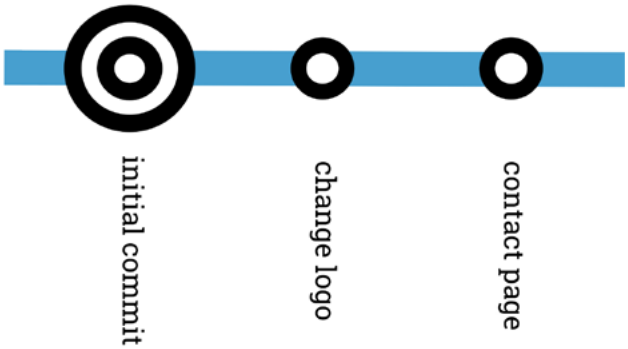
Let's see how these concepts play together in practice. We start by introducing a very simple workflow, then explain the agile ones, after which we take a look at the more elaborate strategies focused on testing and releases.

Lone Wolf

Imagine you're the new hire of Acme Company, and that you've been assigned to handle their public-facing website. You discover that the site is a bunch of static files hosted on a server, and there is no such thing as version control, because nobody cares about it. After you copy the files onto your development machine, you spin off a fresh repository and add them all to git.

Any further commits you make will be made by yourself, and you're the same person doing the build (CSS and JS minification), the release preparation, and the actual deployment. Your release process consists of copying the files to the production machine via FTP, or a git pull from master if you're lucky. A sample timeline of your work looks similar to the one in Figure 7.

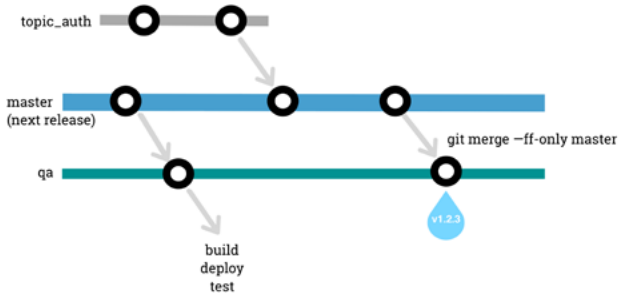
FIGURE 7



GitHub Flow

GitHub Flow[7] is a stable trunk approach, where the master branch is public, long-lived, and can be (and often is) deployed to production very frequently. For several releases a day to happen, centering the workflow around master is a smart approach (see Figure 8).

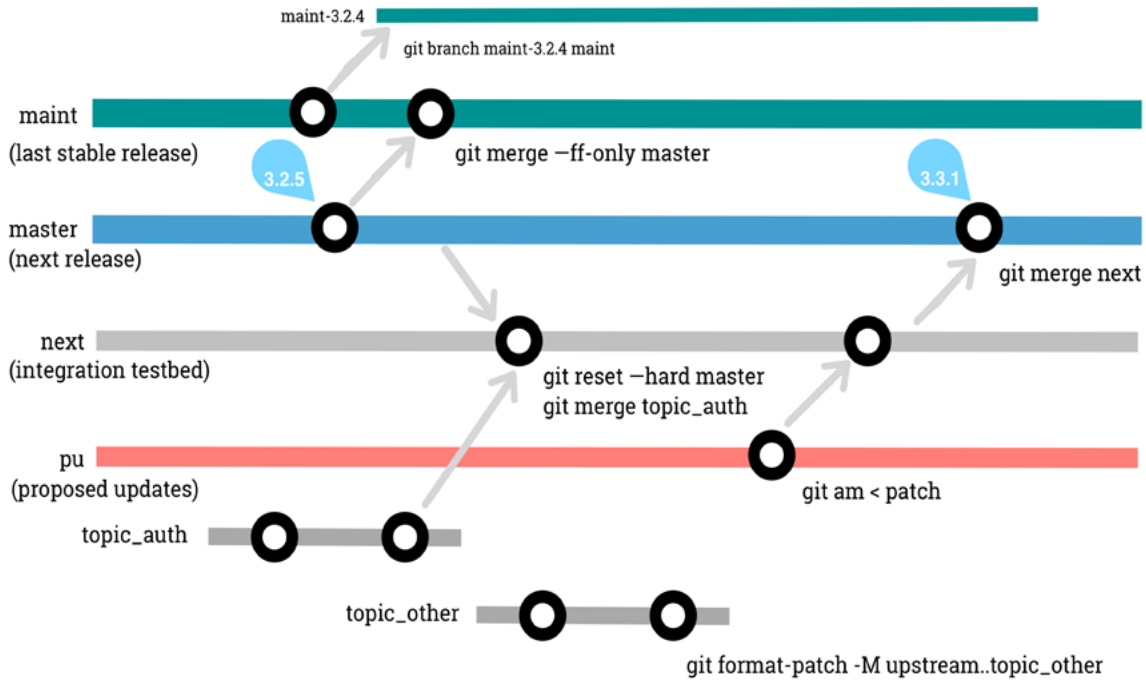
FIGURE 8



The topic branches are where the development happens, and they are public. They are reviewed by peers using pull requests. Once

7 *GitHub Flow:* http://phpa.me/1KanVS4

FIGURE 9



signed off, they are merged into master and deployed immediately to production. The merge workflow is preferred, probably because the very frequent builds and test runs give teammates trust in the code and they don't feel the need for a very clean history.

This approach is very well-suited for software-as-a-service teams that have complete control over the releases, and can move forward production at any time. Bugfixes are applied onto the latest release, just like an ordinary feature, not onto an older release, thus, the team does not have to maintain older releases at all.

Atlassian Simple Git Workflow

Atlassian Simple Git Workflow[8] is very similar to the previous workflow, this one uses a stable master branch that is deployable at any time. Unlike GitHub Flow, the commit history is kept clean as topic branches are rebased from master frequently during their lifetime. All feature branches are public, so if teammates contribute, the remote changes are also integrated locally using rebase. As a topic is finalized, its corresponding branch is added to the list of reviews using pull requests, after which it is integrated back to master using an explicit merge.

Using rebase makes for a cleaner commit history, simplifying the bug hunter's life while keeping the advantages of the GitHub Flow. This flow is also suited to agile teams.

Git Workflow for Agile Teams

It builds on the GitHub Flow concept of using master as the stable source of truth for production. To address the QA process that most companies have, this flow recommends having a separate integration branch called "qa" that is fast-forwarded from master with the latest bunch of work that needs to be checked. It serves a dual purpose: validating work, and doing releases. When new commits reach qa, a checkpoint build is triggered, and the testers can go ahead to validate it. After obtaining the QA blessing, a new release is created by tagging.

Gitworkflows

In Gitworkflows[9], the last stable release lives on the `maint` branch, and it's used for bugfixing it; `master` is an unstable branch containing the code prepared for the next release. An unstable integration branch called `next` serves as a testbed for features that will be promoted to `master`. To test integration of items that are not yet stable, a second unstable integration branch called `pu` (proposed updates) is used.

Developers who have access to the repository integrate their work using merges, while occasional contributors send patches which are tested and applied by core team members.

Long-lived releases might even have their own maintenance branches, spun off `maint` right after the new version tag is created, but before master is merged to `maint`, see Figure 9.

This is a very versatile workflow that accommodates a wide range of scenarios for distributed teams with numerous members of different skill sets. It is fit for both agile and traditional teams, because it favors none in particular. Its flexibility comes mainly from using a small number of branches to keep track of releases, favoring the use of tags; maintenance release branches are an exception rather than a rule. Integration branches can be reset to illustrate a new set of components, and to test against those.

GitLab Flow

While the GitHub and Atlassian flows are centered around agile teams, allowing developers to deliver fast and often, and assuming very frequent production deploys happen, GitLab's version[10] improves on the environment, integration, and release management.

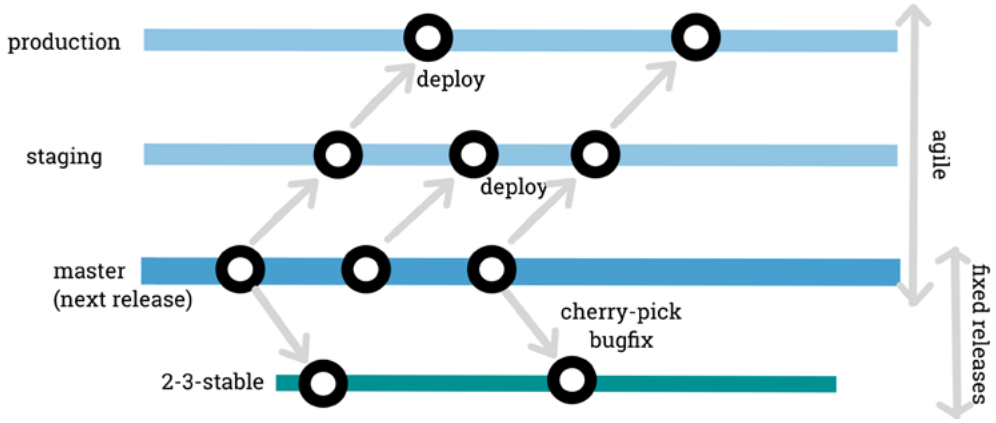
Master is used to hold the latest tested code, and there are public long-lived branches dedicated to specific environments, so there is a production reflecting the current live code at any given time. A pull from master to production will trigger a deploy in that

8 Atlassian Simple Git Workflow: http://wp.me/p2q5Zc-79M

9 gitworkflows: http://phpa.me/1ZXDEx8

10 GitLab Flow: http://phpa.me/22WxU9m

FIGURE 10



environment; see Figure 10.

For people who need to maintain multiple releases of their software and distribute those regularly and predictably, GitLab recommends spinning off a dedicated release branch from master, say 2-3-stable for the 2.3.x releases. After it is created, a build is prepared and publicized. Bugfixes are applied to it by cherry-picking them from master and doing a new build with the minor version number increased.

Development is similar to other flows: topic branches are derived from master, rebase workflow is recommended, and work is checked by others with pull requests. An interesting difference is that merging topics back to master is not done by developers, but by integrators.

Some people consider the branch per environment to be an anti-pattern, because there is always the risk of making changes in the environment branch and might feel like an added complication.

Git Flow

There are several long-lived public branches used by this model. The main branch on which new features are worked on is called `develop`, while `master` is used to hold releases. Features are public, and get integrated back using an enforced merge workflow.

This flow achieved the most fame of all. Even if they don't use it, almost all teams using git have heard of it. In addition to the examples presented by Vincent[11], the presence of git extensions to help enforce the rules and minimize typing was key to adoption. The flow is more complex because it tries to mostly clarify scenarios from a packagist and bug hunter perspective, for more enterprise-like development flows. It feels more at home for teams involved in traditional releases that happen every few weeks (or months), as the release and hotfix processes are heavier and more costly in terms of time. It is less apt for continuous integration and delivery teams, as the payload of releasing and hotfixing adds up, and the entire release management process is not needed for those whose only release is the latest.

Choosing Your Strategy

Most of the time the choice of branching strategy comes from the development team, as they are the ones using git most of their workday. Time can validate their choice, but I've often seen teams moving away from their initial choice because it did not fit their real workflow. If you are in the market for a branching strategy, I would like you to start instead from your release model and from the long-term maintenance that you need to offer. I have found that most projects fall under one of these two categories:

Our latest release is the only one we need to maintain, and we release frequently.

Here we usually meet the continuous delivery teams (oftentimes agile). They tend to produce software that they install on their own hardware, and expose it to the world as a service. Therefore, only the active version from production really matters. The emphasis is placed on a master branch that is releasable at all times, and by automating delivery it is possible to have several releases a day. All fixes are applied directly on top of the latest version. GitHub's and Atlassian's are variations of the stable trunk model that are frequently seen in these cases.

We have to support several older releases in parallel, and we release at specific intervals.

For example, the Symfony framework has a clear process[12], roadmap, and maintains long-term support versions. Other companies package and ship desktop software to customers every few months. Others might have specific deploy windows after each three-week sprint. Because releases are less frequent than in the previous group, they are heavier in terms of features and, therefore, more risky. So greater emphasis is placed on testing integration of new features, as well as packaging and deploying. You only have a small window, so it'd better work well, so we repeat this process many times in lower environments. Using branches per environment like GitLab Flow suggests might fit your project, while Vincent Driessen's git flow and the highly flexible gitworkflows offer more targeted solutions for managing releases.

Georgiana Gligor *is an application architect that enjoys crafting efficient large-scale solutions and has been using PHP for more than a decade, and is a living proof that geek girls are an asset to any team. @gbtekkie*

Required Software

- Git—https://git-scm.com

Related URLs

- Lorna Jane's branching strategy presentation—http://phpa.me/1ZoPLqd
- Git team workflows: merge or rebase?—http://phpa.me/atlassian-merge-rebase
- Linus on clean history—http://lwn.net/Articles/328438/

11 Git Flow: http://phpa.me/nvie-git-flow

12 Symfony framework release process: http://phpa.me/symfony-release-process

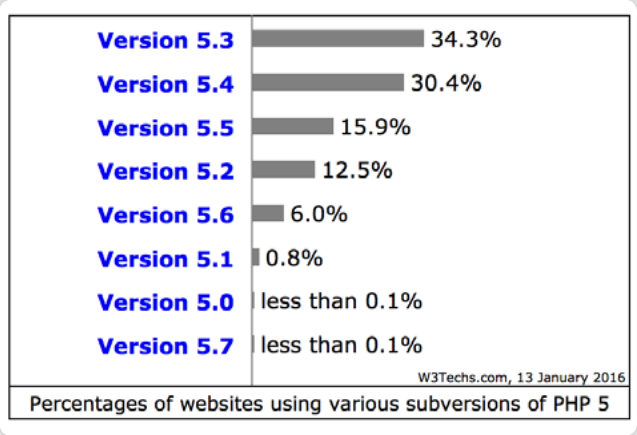
Build a Local CI with Docker

Nicola Pietroluongo

If you struggle to find a fast way to test your code against several PHP versions, this article is for you. This step-by-step guide will show you how to build a local test environment with technologies like Docker, Behat, or PHPUnit in less than 10 minutes.

PHP 7 is around the corner, but the highest percentage of websites still run on older versions—see Figure 1[1].

Percentages of websites using various PHP 5 versions **FIGURE 1**



Supporting only new releases will simplify the developer's life, and will enhance the user experience, opening the doors to a new era of PHP evolution. But what about existing code or applications with back compatibility requirements?

Even if you decide to use external tools like Travis, Jenkins, Codeship, or others, running tests can be tedious and time-consuming. Building a "craft made" local Continuous Integration test environment can be an alternative, and clarifies the approach of other tools.

Project Overview

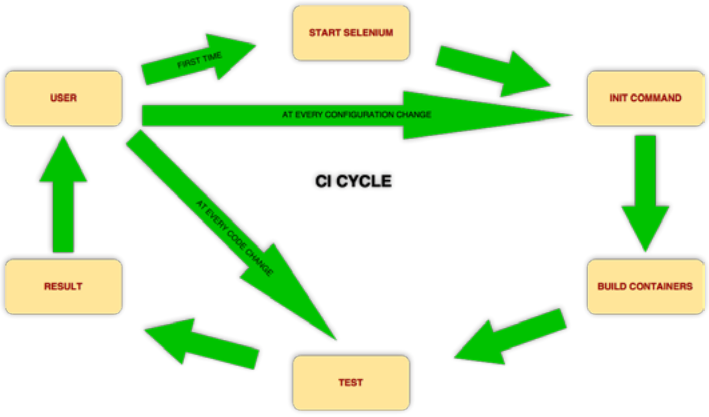
The project will include the following technologies:

- Docker
- Bash scripts
- PHPUnit[2]
- Behat[3] and Mink[4]
- Selenium[5]

You can easily integrate testing tools like SimpleTest, phpspec, or others.

Docker will be used to create multiple test PHP containers. The Bash script will be the "glue" between every operation, providing a useful command line tool. Selenium will be the stand-alone server for functional testing. Figure 2 illustrates the CI process cycle we will create:

CI cycle **FIGURE 2**



By the end of the article, the final folder structure will resemble Figure 3.

Final folder structure **FIGURE 3**

```
.
├── bin
│   ├── app
│   ├── behat
│   ├── phpunit
│   └── test.sh
├── docker
│   ├── php-5.4
│   │   └── Dockerfile
│   ├── php-5.5
│   │   └── Dockerfile
│   ├── php-5.6
│   │   └── Dockerfile
│   ├── php-7.0
│   │   └── Dockerfile
│   └── shared.sh
├── features
│   └── ... Behat features ...
├── src
│   └── ... project files ...
├── tests
│   └── .. test files ...
├── vendor
│   └── .. dependencies ...
├── behat.yml.dist
├── composer.json
├── composer.lock
└── phpunit.xml.dist
```

Please note: it's important to keep the same naming convention as the image above.

If you don't have any projects with PHPUnit and Behat, you can use a sample with:

```
git clone -b start-sample https://github.com/niklongstone/docker-php-ci.git
```

After you've cloned it, run `composer install`.

Docker Images

As our first goal, we need to create the Docker images for every PHP version we want to test.

1 *Credits:* http://phpa.me/php-usage-nov2015
2 *PHPUnit:* https://phpunit.de
3 *Behat:* http://docs.behat.org
4 *Mink:* http://mink.behat.org/en/latest/
5 *Selenium:* http://www.seleniumhq.org

PHP Versions

Instead of starting from scratch, It's easy to begin with the official Docker PHP CLI images available in Docker Hub[6]). Our CI will use the following images:

- php:5.4-cli
- php:5.5-cli
- php:5.6-cli
- php:7.0-cli

Extending a Docker Image

Our CI can't use a Docker image as it is; we need to extend the base PHP image, adding functionalities on top of that. The extension process allows us to add any custom non-default PHP libraries our project could require. One example is installing the `mbstring` library required by Behat.

A generic Dockerfile based on php 5.4-cli image could be:

```
# File: docker/php-5.4/Dockerfile
FROM php:5.4-cli

# Other custom operations

CMD ["php","-v"]
```

Installing Additional Libraries

The official PHP images are based on Debian Jessie[7], which means we can use commands like `apt-get install` for further installations. In addition, every image has two functions to configure extensions: `docker-php-ext-install` and `docker-php-ext-configure`.

For example, to add the gd extension, we can use Listing 1.

LISTING 1

```
01. # File: docker/php-5.4/Dockerfile
02. FROM php:5.4-cli
03.
04. # Gd extension
05. apt-get update && apt-get install -y \
06.    libfreetype6-dev \
07.    libjpeg62-turbo-dev \
08.    libpng12-dev \
09.    && docker-php-ext-configure gd --with-freetype-dir=/usr/include/ \
10.    --with-jpeg-dir=/usr/include/ \
11.    && docker-php-ext-install gd
12.
13. CMD ["php","-v"]
```

First Docker Image

At this point, we know how to build our own `Dockerfile` adding additional libraries. Because our project needs only the mbstring extension, our first `Dockerfile` might be as follows:

```
# File: docker/php-5.4/Dockerfile
FROM php:5.4-cli

# Additional extension
docker-php-ext-install mbstring

CMD ["php","-v"]
```

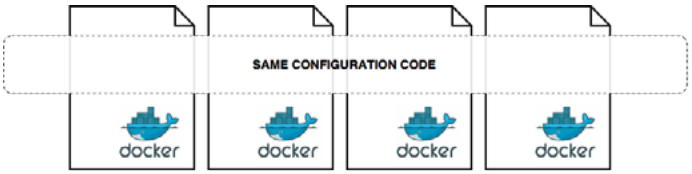
6 Docker Hub: https://hub.docker.com/_/php/

7 Debian Jessie: https://wiki.debian.org/DebianJessie

Configuration Script

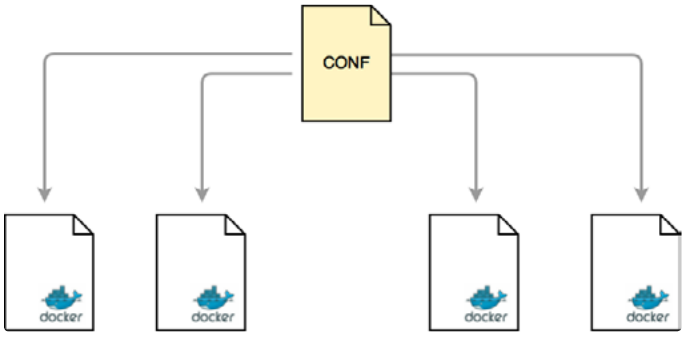
To load additional extensions in every image, we need to insert the same code snippet in each `Dockerfile`. This copy and paste redundancy won't handle configuration changes quickly.

FIGURE 4 Same configuration code



To avoid this repetitious task, we can make the configuration "global," then share it between each Docker container.

FIGURE 5 Distributed configuration



Shared configuration file:

```
#!/bin/sh
# File: docker/shared.sh

docker-php-ext-install mbstring
```

We must assign executable permission to the `shared.sh` *with:*

```
$ chmod +x shared.sh
```

Final Docker Files

Now, we need to include and run the configuration script inside the `Dockerfile`. Unfortunately, Docker doesn't support an `INCLUDE` statement, but with the `COPY` and `RUN` commands we can achieve our goal.

Our first complete `Dockerfile` will look like this:

```
# File: docker/php-5.4/Dockerfile
FROM php:5.4-cli

# Copies the shared.sh: COPY <src> <dest>
COPY shared.sh shared.sh

# Executes the shared.sh file
RUN ./shared.sh

CMD ["php","-v"]
```

Finally, we can create all the others' `Dockerfile` with the process described above, changing only the first `FROM` line. For example, the `docker/php-5.5/Dockerfile` will be:

LISTING 2

```
01. #!/bin/bash
02. #: Name        : app
03. #: Description : Provides automation to PHP code testing
04.
05. #Custom vars
06. images="php-5.4 php-5.5 php-5.6 php-7.0"
07. docker_folder=docker
08. shared_script_path=docker/shared.sh
09. tag_prefix=niklongstone
10.
11. #Default var
12. script_name=${0##*/}
13.
14. #Prints the help
15. function print_help {
16.   echo "USAGE: $script_name [OPTION]"
17.   echo "OPTIONS:"
18.   echo "   init: initialize the repository folder"
19.   echo "   build: build all the docker PHP containers"
20.   echo "   help: this help"
21. }
22.
23. #Shares with every Docker container the same shared.sh script
24. function init {
25.   echo $docker_folder/*/ | xargs -n 1 cp $shared_script_path
26.   echo The install script is correctly shared
27. }
28.
29. #Build all the Docker containers looping in all the docker folders
30. function build {
31.   for image in "${images[@]}"
32.   do
33.     docker build -t $tag_prefix/$image $docker_folder/$image/
34.   done
35. }
36.
37. #Handles the input option
38. case "$1" in
39.     init)       # app init
40.         init
41.         ;;
42.     build)      # app build
43.         build
44.         ;;
45.     help)       # app help
46.         print_help
47.         ;;
48.     *)          # default option
49.         print_help
50.         ;;
51. esac
```

```
# File: docker/php-5.5/Dockerfile
FROM php:5.5-cli

COPY shared.sh shared.sh
RUN ./shared.sh

CMD ["php","-v"]
```

A COPY Problem

As you probably noticed, the source path of `COPY`, in our `Dockerfile`, is inside the context of the build. Docker does not allow adding a path like `../shared.sh`, so we must place a `shared.sh` file inside each `docker/php-.../` folder. Is this another repetition? No, it's not like before, because we can automate the operation with the following shell command:

```
echo docker/*/ | xargs -n 1 cp docker/shared.sh
```

In a nutshell, the `echo` command will output all the directories inside the `docker/` path, and the `xargs` will prepend the `cp` command for each argument received. You can find more information about xargs at http://phpa.me/man-xargs

Bash Script

The Bash script `app` will centralize and simplify operations like configuration sharing, start containers, and run the test.

At the moment, we will implement our CLI script with two functionalities:

- `app init` to copy the `shared.sh` files inside every `docker/..` folder
- `app build` to build the Docker images before the test.

In Listing 2 you can see the first version of the `app` script.

Note: every Bash file in this project needs executable permission: `chmod +x bin/app`

Selenium

Running tests with PHPUnit or another testing tool is quite easy in our CI, but problems may arise when there is a new technology like Selenium in our stack.

Having a Selenium server accessible from all the PHP Docker instances will allow us to centralize its administration. To support this requirement, we can use a Docker Selenium container.

Starting a Docker Selenium stand-alone Chrome server is very easy—it's just one line of command:

```
docker run -d -p 4444:4444 selenium/standalone-chrome:2.47.1
```

Let's integrate Selenium into our `app` with the code in Listing 3.

You can find more information about Selenium and Docker in the official repository[8].

LISTING 3

```
01. #Custom vars
02. ...
03. selenium_image=selenium/standalone-chrome:2.47.1
04. ...
05.
06. #Prints the help
07. function print_help {
08.   echo "USAGE: $script_name [OPTION]"
09. ...
10.   echo "    selenium: run docker selenium instance"
11. ...
12. #Starts the Selenium container
13. function selenium {
14.     # Runs the selenium image with the name 'selenium' and
15.     # exposes the port 4444 for connections
16.     docker run -d -p 4444:4444 --name selenium $selenium_image
17. }
18. #Handles the input options
19. case "$1" in
20. ...
21.     selenium)
22.         selenium
23.         ;;
24. ...
```

8 Selenium and Docker: https://github.com/SeleniumHQ/docker-selenium

Don't forget to change your Behat configuration to support the remote Selenium connection. An extraction of a sample `behat.yml.dist` file is shown below:

```
extensions:
...
      selenium2:
        browser: chrome
        wd_host: selenium:4444
...
```

Adding the Test

The final step is to add the test. The command that runs a PHPUnit test following a Behat test is:

```
./bin/phpunit
./bin/behat
```

Note: you can change the above command according to your project settings, for example, using `./vendor/bin/phpunit` instead of `./bin/phpunit` or adding other CLI specific options.

Writing the above declarations in a `test.sh` Bash file will organize our CI test configuration.

The `bin/test.sh` file will be as follows:

```
#!/bin/sh
# File: bin/test.sh

# Enters in the container project's folder, by convention
'/webapp'
cd /webapp

# Runs the tests
./bin/phpunit
./bin/behat
```

Container Behavior Summary

At this stage, a few things are worth pointing out before proceeding. Each container:

- should start with a command line request;
- should be linked to the Selenium server;
- should mount the last version of our project files;
- should run the test command;
- should be stopped to allow the second test to run free.

All the above actions are combined with a single command:

```
docker run -t --rm -v $(pwd):/webapp --name CONTAINER_NAME \
       --link selenium:selenium \
       IMAGE_NAME COMMAND
```

Let's take a look at the `run` options:

- `-t` allocates a pseudo tty
- `-rm` removes the container when it exits
- `--name` identifies the container
- `-v` mounts our root directory in the `/webapp` container path
- `--link` creates a connection between containers
- `IMAGE_NAME` will be a PHP image
- `COMMAND` will be our `test.sh` script.

For more information about the `run` command, please refer to the Docker official documentation[9].

To apply the `docker run` command for each container, we can use a Bash `for` statement.

Listing 4 is another update for the `app` script:

LISTING 4

```
14. #Custom vars
15. ...
16. test_script_path=bin/test.sh
17. ...
18. #Prints the help
19. function print_help {
20.   echo "USAGE: $script_name [OPTION]"
21. ...
22.   echo "    test: run the test case"
23. ...
24. #Runs the container and trigger the test
25. function run_test {
26.   for image in "${images[@]}"
27.   do
28.     docker run -t --rm -v $(pwd):/webapp --name test-$image \
29.         --link selenium:selenium $tag_prefix/$image \
30.         /webapp/$test_script_path
31.   done
32. }
33. ...
34. #Handles the input options
35. ...
36.     test)
37.         run_test
38.         ;;
39. ...
```

Let's Play

At this point our CI is complete. To play with it we can:

1. Run the Selenium server with: `./bin/app selenium`. This operation is necessary only once; to subsequently stop/start the server we can use respectively: `docker stop selenium`, `docker start selenium`.
2. Insert the shared configuration in every Docker folder with `./bin/app init` and build all the Docker PHP container with `./bin/app build`. The `init` and `build` operation must be repeated on every `shared.sh` change.
3. Run the test and see the result with `./bin/app test` each time we made a modification to our project code.

Improvements

Let's put some sugar in our script by adding more useful functionalities.

Show the PHP Version Before Tests

The first upgrade of our script will output the PHP version before every test. The tty allocation in the `docker run` command supports colored output. To change the color we can use the `tput` command as below:

```
#!/bin/sh
# File: bin/test.sh

#Outputs PHP version in magenta color
tput setaf 5
php -v
tput sgr0

#Enter in the project's folder
....
```

You can find more information about `tput` online[10]. The resulting

9 Docker RUN: https://docs.docker.com/reference/run/

10 `tput`: http://phpa.me/man-tput

output is shown in Figure 6.

Coloured PHP version **FIGURE 6**

Run Tests on Commit

To automate the test on every `git commit` we can use a Git Hook[11]. A Git hook can fire off custom scripts when certain actions occur. In our case, we will trigger the `./bin/app test` before a single commit. We need to create a file under the `.git/hooks/` of our project called `pre-commit` with the following simple snippet:

```
#!/bin/sh
# File: .git/hooks/pre-commit

./bin/app test
```

then make the file executable with:

```
$ chmod +x .git/hooks/pre-commit
```

Use a Different Selenium Server

The third upgrade involves the Selenium server. The Selenium Docker repository offers other interesting containers, for example, the stand-alone server with a VNC connection. The selenium/standalone-chrome-debug:2.47.1 exposes a VNC connection with an X11 terminal to inspect and debug processes.

The standalone-chrome-debug uses the 5900 port for VNC; this means we need to expose another port in our `docker run selenium` command to establish the VNC connection. We can output the Selenium server IP to facilitate inspections.

The new `selenium` function might look like Listing 5:

Note: if you use docker-machine, you will find the correct IP via `docker-machine ip name-of-your-vm`

The Selenium Grid Hub deserves to be mentioned because it enables you to run your tests against different browsers in parallel. For the setup, see the official documentation[12].

Run Tests with Custom PHP Versions

Our CI environment runs the test against all the PHP versions declared in the `default_images` variable of the `app` script. It's not so flexible, because it does not allow you to specify the arbitrary versions you want to test. We can use an environment variable called `PHP_IMG` to override the `default_images` value.

The `app` script can be modified as follows:

```
# Custom vars
default_images="php-5.4 php-5.5 php-5.6 php-7.0"
...
#Default var
...
# If PHP_IMG is not specified, it will be set with the
default value
: ${PHP_IMG:=$default_images}
images=($PHP_IMG)
...
```

With the above modification, for example, we can use:

```
PHP_IMG="php-5.4 php-5.5" ./bin/app test
```

to test the code only with PHP 5.4 and 5.5.

PHP CI Projects

You've seen how to get started building a testing pipeline with Docker. There's no reason to implement it all from scratch: there are a number of open-source CI projects that you can reuse.

- PHPCI: https://github.com/block8/phpci
- JoliCi: https://github.com/jolicode/JoliCi
- DUnit: https://github.com/Vectorface/dunit

While hosted solutions are very convenient, now you know how to set up a custom CI environment that can be tailored to your exact needs.

Nicola *is a London-based Software Engineer with many years of experience. He's built web applications for big companies across Europe, and is now creating and contributing to awesome PHP web projects. If you like this article, come and say "Hi" on Twitter @niklongstone.*

Requirements:

- Docker—https://www.docker.com
- Git—https://git-scm.com
- Composer—http://getcomposer.org

LISTING 5

```
01. #Custom vars
02. ...
03. selenium_image=selenium/standalone-chrome-debug:2.47.1
04. ...
05. function selenium {
06.   # Runs Selenium
07.   docker run -d -p 4444:4444 -p 5900:5900 --name selenium $selenium_image
08.   # Output the IP
09.   SELENIUM_IP=$(docker inspect --format='{{.NetworkSettings.IPAddress}}' selenium)
10.   echo Selenium IP = $SELENIUM_IP
11. }
```

11 Git Hooks: http://phpa.me/customizing-git-hooks

12 Selenium Docker: https://github.com/SeleniumHQ/docker-selenium

Get Some Git Extras

Matthew Setter

Welcome back to another month of Education Station. You may well be wondering what new, exciting, and interesting repository I've found to share with you this month. What is it that I've found to make your development workflow even simpler and less complicated than it already is? What library have I found to pique your interest and reignite the hacker juices that flow through your veins?

If you're even a casual reader of the column, you'll know that, of course, I've found one. But this month, it's not one that is strictly related to PHP. Even so, it's one that can help you do more with less and be more productive on the command line with Git. What is it? Allow me to intrigue you further.

You keep your code under source control, right? And if you're like the majority of developers[1], your version control tool of choice is Git. Sure, you've used CVS and Subversion at one point, perhaps even dabbled with Mercurial and SourceSafe. But they've all given way to Git.

If the hype and statistics are anything to go on, this is a fair assumption to make. It's taken off, and there's no stopping it. However, I think for some, perhaps most, of us, to use Git is to have a masochistic tendency. Why? Because despite all that Git gives us, it's a tough tool to use. Many of us rave about it, proudly extolling its virtues. But we all know that Git isn't a tool for the beginner.

I make that assertion for several reasons. First, there's the sheer depth of functionality that it has on offer. From committing code, cheap and easy branching, staging multiple changes across multiple files, and reviewing revision history to *interactive rebasing*, *merging*, and *cherry picking*—Git is feature rich.

But think about the number of options these commands have. Run any of the following if you're not sure:

```
git help log
git help branch
git help rebase
```

Figure 1 is a sample from Git branch:

In the output of any of these, you'll see that they contain a cornucopia of command switches and sub switches. Let's take the example of `git log`. You can see in Figure 2 that the basic man page isn't that complicated. Git log shows commit logs.

The command synopsis is not too complicated, right? Erm, no. You can specify everything from how to format the output, whether to follow file renames, whether to show the log size, how to limit commits, how to find commits, and more.

Despite this complexity, great strides have been made to bring about greater simplicity and to make them more developer-friendly, easier to understand, and friendlier to use. What's more, sites like http://git-scm.com have taken this even further by providing excellent, rapidly searchable documentation. This documentation is friendly and welcoming—for beginners and seniors alike. If you are starting out with Git today, you're starting at a point in time where you don't need to be a more seasoned veteran like you once needed to be.

That's not to say that Git couldn't be made easier. To illustrate this point, this month, I give you Git Extras [2]. Git Extras is a set of utilities that builds upon the functionality that you have come to know and love, but Git Extras makes it easier and exposes functionality that you might not have yet learned about.

FIGURE 1

```
NAME
       git-branch - List, create, or delete branches

SYNOPSIS
       git branch [--color[=<when>] | --no-color] [-r | -a]
               [--list] [-v [--abbrev=<length> | --no-abbrev]]
               [--column[=<options>] | --no-column]
               [(--merged | --no-merged | --contains) [<commit>]] [--sort=<key>]
               [  points at <object>] [<pattern>...]
       git branch [  set upstream |   track |   no track] [ l] [ f] <branchname> [<start point>]
       git branch (  set upstream to=<upstream> |  u <upstream>) [<branchname>]
       git branch --unset-upstream [<branchname>]
       git branch (-m | -M) [<oldbranch>] <newbranch>
       git branch (-d | -D) [-r] <branchname>...
       git branch --edit-description [<branchname>]
```

FIGURE 2

```
NAME
       git-log - Show commit logs

SYNOPSIS
       git log [<options>] [<revision range>] [[--] <path>...]

DESCRIPTION
       Shows the commit logs.

       The command takes options applicable to the git rev-list command to control what is shown and
       how, and options applicable to the git diff-* commands to control how the changes each commit
       introduces are shown.

OPTIONS
       --follow
           Continue listing the history of a file beyond renames (works only for a single file).
```

1 *Git & Mercurial Popularity:* http://programmers.stackexchange.com/q/128851

2 *Git Extras:* https://github.com/tj/git-extras

If you have a look at the list of commands, which you can find at http://phpa.me/git-extras-commands, you'll see there are commands like the following:

Command	Task
`git feature | refactor | bug | chore`	Help you work with branches.
`git effort`	Show you the effort invested in source files.
`git contrib`	Show you a developer's contributions to a project.
`git summary`	Provide a concise project summary.
`git missing`	Show which commits are on one branch but not on another.
`git obliterate`	Completely remove a file from a repository.
`git graft`	Merge commits from one branch to another.

Sound like something you're keen to explore? Awesome. Then let's dive in and install it.

Installing Git Extras

The package maintainers have done a great job of making it easy to install, regardless of your platform. If you're on a Debian-based Linux distribution, you can install it using Apt by running the following:

```
sudo apt-get install git-extras
```

If you're on a Mac, you can install it via Homebrew by running the following:

```
brew install git-extras
```

If you're on Windows, not surprisingly, it will take a little extra effort. But don't worry, it's not too involved. Here's a summary of what you need to do. First, clone the project locally by running the following:

```
git clone https://github.com/tj/git-extras.git
```

Then switch to the cloned directory and run the following:

```
install.cmd "C:\git"
```

Please note, you'll need Git 2.0 for this to work.

If you're a bit of a purist and like to do everything by hand, building from source, that's fine. Clone the repository, change to the cloned directory, and update to the latest tag by running the following:

```
git checkout $(git describe --tags $(git rev-list --tags \
 --max-count=1))
```

Then to install the package, run the following:

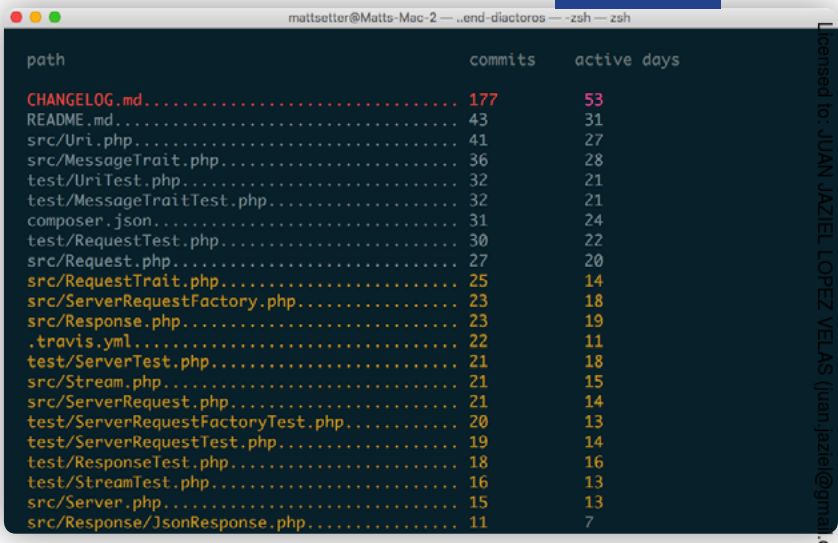
```
sudo make install
```

With that done, we're now ready to have a play and see how we can make our lives easier.

Git Effort: What Are the Most Active Files?

Let's say that you're keen to know which files have received the most love on your project, both in terms of time and commits. Sometimes this can be handy. To do that, we can run `git effort`. Git effort displays the summary in two forms—first unsorted, then sorted. The example you see in Figure 3 is the sorted version from running it on the Zend Diactoros[3] project.

FIGURE 3



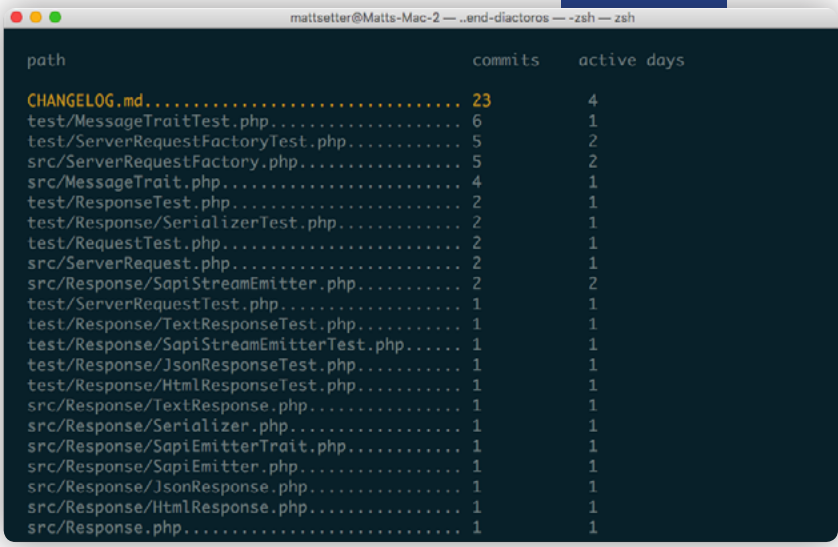
In it, you can see the path, number of commits, and active days. Git effort is a simple wrapper over `git log`, so you might be happy to know that you can also provide some of Git log's switches to further refine the information returned, specifically options from the "Commit Limiting" section of the documentation.

Let's do that now, starting by filtering commits to those made from the 1st of December 2015 to the 20th of January 2016. To do that, we'd pass the `since` and `until` switches as follows:

```
git effort -- --since=01.12.2015 --until=20.01.2016
```

Note the double hyphen separating Git log's switches from Git effort's. You can see the revised, sorted output in Figure 4.

FIGURE 4



3 *Zend Diactoros:* https://github.com/zendframework/zend-diactoros

Git Ignore: Quickly Ignore Files

We all know that to ignore a file or path in Git, you add it to either your local or global `.gitignore` file, right? What if you just don't have the compunction to open the file, add a new path, and save and close it? Sounds a little lazy, but hey, we all get tired, feel unmotivated, or just want a simpler option.

If that's you, use Git ignore to do it. Let's say that I want to add PhpStorm's `.idea` directory to `.gitignore` without editing the file. To do so, I'd just run the following command, and hey, presto! It's been done.

```
git ignore .idea
```

After that's done, you'll see the following confirmation message, which shows that `.idea` has been successfully added:

```
Adding pattern(s) to: .gitignore
... adding '.idea'
```

Git Local-Commits: What Commits Haven't I Pushed Yet?

This is a question I used to ask myself often. OK, a lot. Perhaps often is just a tad bit of an over-exaggeration. But from time to time, you need to know the differences between branches.

Perhaps you're trying to track down a random bug and need to see if it's a difference between a local development branch and a remote one that is causing the issue. Here's an excellent comment to quickly show the changes to you. Running `git local-commits` will show you that. Unfortunately, I don't actually have any local ones. Not yet, anyway.

Git Summary: What's the State of My Repository?

Sometimes, you want to get a quick overview of the repository—information such as how long it's been around for, the total number of commits and files, and the proportional contribution from each contributor. Well, if you need this, then `git summary` is here to help.

Running that command on the `Zend\Diactoros` repository, which is a PSR-7 HTTP Message implementation, gave me the summary that you see in Figure 5. It's slightly truncated. But you can see that the repository's been available for 1 year and 5 months, or active for 106 days. It's had 788 commits across 74 files. And its main contributor is the one and only Mr. Matthew Weier O'Phinney (@mwop), who's contributed 76.4% of the code.

Git Count: Who's Committed the Most?

Even the quietest of us has a competitive streak. We may come from countries that espouse communal or group ideals more than an admiration of the individual. But I'd be willing to wager that in all of us lies that desire to know how we compare to our peers, even if it's only occasionally.

If you have even a flicker of that feeling, or if it's burning bright within you, then `git count` is here to the rescue. Maybe you make it a big thing in your organization to have a running leader board posted for all to see. If you do—and if you want to add the commits to it—then run `git count --all` to see where you stand in relation to your team. Figure 6 shows what the `Zend\Diactoros` leader board looks like.

FIGURE 5

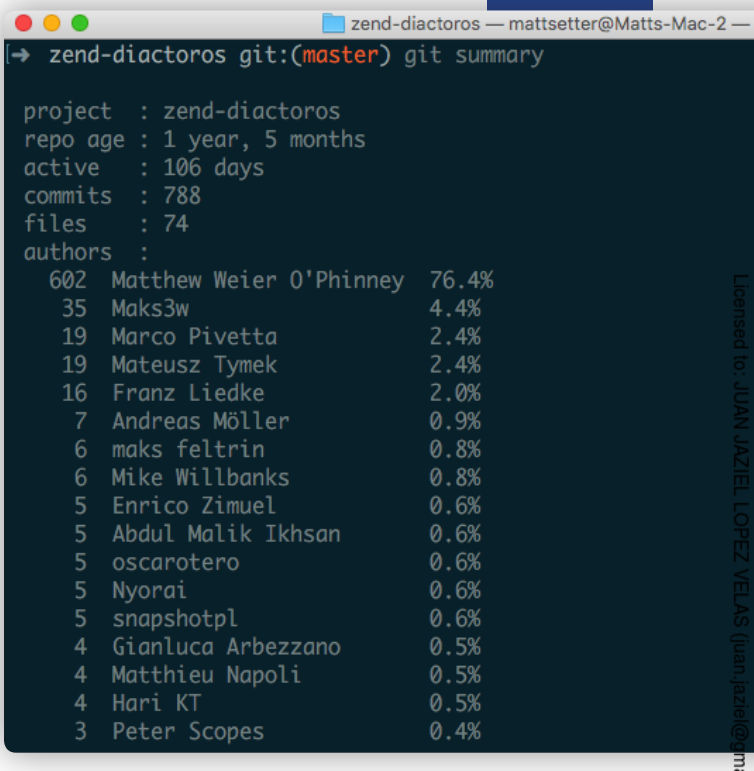
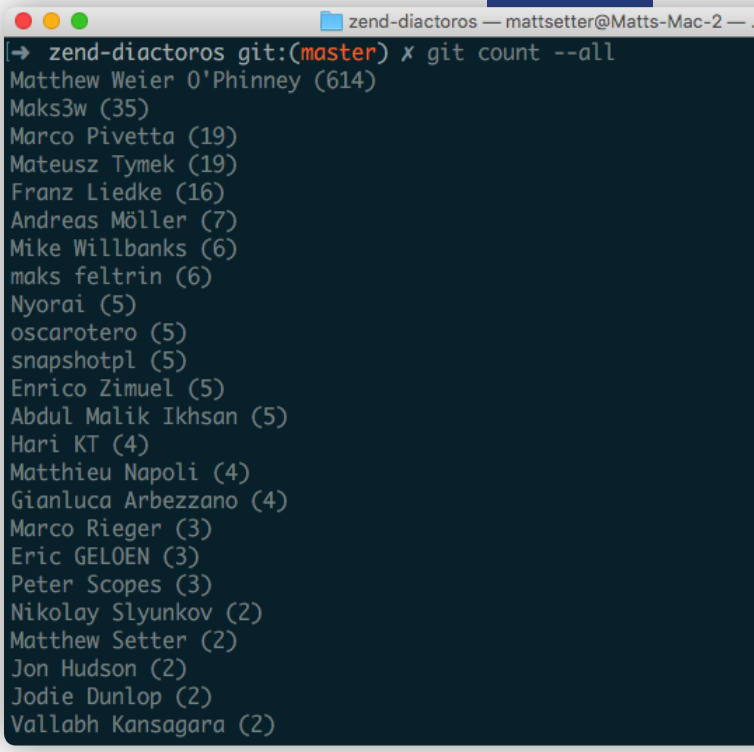


FIGURE 6



Can Branching Be Made Even Simpler?

It sure can! That's the answer to *that* question. How, specifically? Well, there are quite a few workflows around—from simple branching and merging to the more sophisticated, such as GitFlow. But what if you want something simple, with the one difference being that your branches are prefixed with their intent?

If this sounds like you, then Git Extras provides you with the ability to create a new branch from your current branch, prefixed with one of *feature*, *refactor*, *bug*, or *chore*. As I said, there's nothing different exactly, just the prefix. So these will prefix the branches created with `feature/`, `refactor/`, `bug/`, and `chore/`, respectively.

Now, when you push a PR, your team will have a clearer indication about what it's doing. What's more, if you're like me and often maintain multiple branches simultaneously, then you'll be able to make sense of them more effectively. So let's see an example. We're going to create a fictitious feature branch to implement a new feature in `Zend\Diactoros`. To do so, I'd run the following:

```
git feature my-hypothetical-new-feature
```

Git Commits-Since: What's Been Done Since...?

This is a question I've asked myself more than once. Sometimes, out of curiosity, I've been asked to find out. What's been committed since last Friday?

Well, now you can quickly and easily find out by running `git commits-since`, along with a switch or two. Continuing with our `Zend\Diactoros` repository example, let's find out what's been committed since last Friday, which is a week back as I'm writing the column. To do that, I'll run the command:

```
git commits-since last friday
```

Turns out, there's not been any. So all I saw was `(END)`. That's not really helpful. So let's go back further, say back a month. Running

```
git commits-since last month
```

Gives me the output in Figure 7. In it, you can see that Matthew Weier O'Phinney, snapshotpl, and Mateusz Tymek have contributed a total of 21 commits, with about half of them being merge commits. Now, let's go quite a way back to April 2015. I wonder how many commits have been made since then.

Running the following shows that there have been 525:

```
git commits-since april 2015 | wc -l
```

If you're not familiar with `wc -l`, the command `wc` performs word, line and character, and byte counts on the input supplied. Passing the `-l` switch makes it run a line count, which sums the commits, giving me 525.

But getting back to `commits-since`, I believe it's a wrapper around `git log`, so any of the English-like expressions for time limiting for which you can give `git log`, you can also give to commits-since. For more information about the supported options, check out the reference manual at https://git-scm.com/docs/git-log or run `git help log` in your terminal.

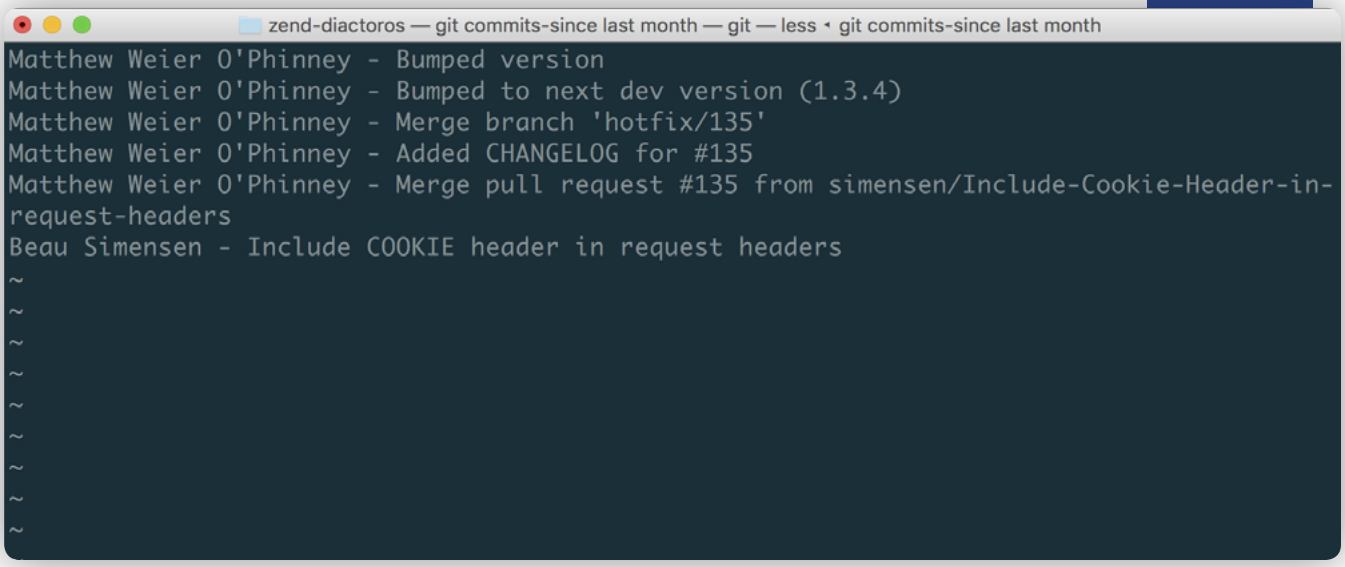
And That's a Wrap

So what do you think? Do you think that with all of these options, you could both simplify and grow your Git command line prowess? I know that I have. I'll admit that, over the years, I've created a number of tools that are similar in purpose to the ones in Git Extras. But it's nice to know that there's a repository that takes my efforts further, one that is maintained by people other than myself—one that has been tested, too.

Like any good repository, by perusing how the commands have been implemented, I've learned so much more, especially about features that I never knew existed. I strongly encourage you to do the same after you've installed it. I'm confident you'll get a lot out of it and grow ever more proficient with Git.

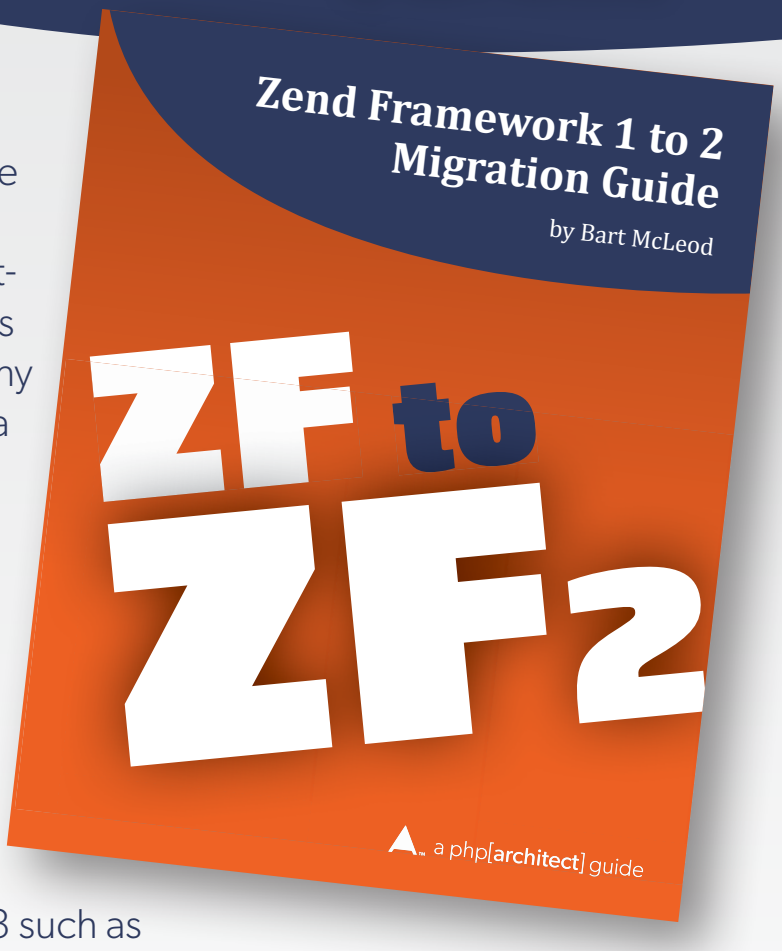
Matthew Setter is a software developer specializing in PHP, Zend Framework, and JavaScript. He's also the host of http://FreeTheGeek.fm, about the business of freelancing as a developer and technical writer, and editor of Master Zend Framework , http://www.masterzendframework.com.

FIGURE 7



Zend Framework 1 to 2 Migration Guide

by Bart McLeod



Zend Framework 1 was one of the first major frameworks for PHP 5 and, for many, introduced object-oriented programming principles for writing PHP applications. Many developers looking to embrace a well-architected and supported framework chose to use it as the foundation for their applications. Zend Framework 2 is a significant improvement over its predecessor. It re-designed key components, promotes the re-use of code through modules, and takes advantage of features introduced in PHP 5.3 such as namespaces.

The first release of ZF1 was in 2006. If you're maintaining an application built on it, this practical guide will help you to plan how to migrate to ZF2. This book addresses common issues that you'll encounter and provides advice on how best to update your application to take advantage of ZF2's features. It also compares how key components—including Views, Database Access, Forms, Validation, and Controllers—have been updated and how to address these changes in your application code.

Written by PHP professional and Zend Framework contributor, coach, and consultant Bart McLeod, this book leverages his expertise to ease your application's transition to Zend Framework 2.

Purchase
http://phpa.me/ZFtoZF2

Renewal

Cal Evans

To improve is to change; to be perfect is to change often.
– Winston Churchill

For the past ten years, I have been active in the PHP community. Some years, it has been my job; other years, it has been my pleasure. Either way, it has been an interesting ten years.

I have seen members of the community rise up to become prominent voices. I have seen others move on to other languages or technologies, or just leave the profession entirely. I've seen families created and future programmers brought into this world. I've mourned with the community as we lost members.

I've watched as our group of Core developers churned. Good developers who contributed a lot have left us and moved on to other projects. I've seen new developers step in to fill the gaps that have been vacated and leave their mark on us and the world. This renewal brings us new ideas, new points of view, and guarantees a constant renewal of the engine that drives PHP's development.

As a community, we are constantly growing and churning as well. It has been interesting to watch these past ten years as the people who were leaders move on to other roles, other communities, or move into another phase of their lives. Many a time, I've watched as men and women who honed their leadership skills in this very community move up the corporate ladder based on those skills. As leaders in our community move up into more responsible roles in their company, they may step back from our community, but they continue influence the community, just in different ways.

Looking forward, I see an interesting future for PHP. We have just been given the gift of one of the greatest releases in the history of PHP, PHP 7. Not content to rest on their laurels, work has already begun on new features slated for the next major release. PHP will continue to grow. With projects like WordPress, Drupal, Joomla!, and Magento based on it, it will not fade in importance or influence anytime in the near future.

We are experiencing renewal not only in the language, but also with two of our large projects, Drupal and Magento. These projects—as well as many others—are celebrating major milestone releases and looking toward their own bright future. PHP's community of projects continues to grow, not only in breadth but in depth.

I have no doubt that this year we will see prominent members of our community step back into the background, just as surely as we will see new members step up and find their own place in the fabric of our community. Projects will fade away, and new ones will be created and form communities of their own.

Renewal is not an event, it is a cycle. Change is not something to be resisted, it is something to be embraced. PHP will continue to evolve, the Core will continue to evolve, and the community will continue to evolve. All of that is a good thing, because as soon as we stop, we begin to die.

So it will be that this column will evolve. As I took over the reins, I thought long and hard about what I wanted to do with it. I could use it as yet another bully pulpit, but honestly, these days I don't have any trouble finding outlets for my thoughts. I think that instead of just using it as an "Old Guy Rant" (I did briefly consider renaming it *My Lawn*) that I am going to seek out community members who actually have something to say and give them an outlet, much like Joe Devon did previously. Some of them you will know, some of them you may be introduced to in this column. Yes, from time to time I will use it for my own rants, but my goal is to find community members you may not know, and let them tell you something.

To paraphrase the late great David Robert "David Bowie" Jones:

I don't know where we're going from here, but I promise it won't be boring.

Cheers!

=C=

Past Events

Ski PHP 2016
January 14–15, Salt Lake City, UT
http://www.skiphp.com

PHPBenelux
January 29–30, Antwerp, Belgium
http://phpbenelux.eu

Upcoming Events

SunshinePHP 2016
February 4–6, Miami, FL
http://sunshinephp.com

Joomla Day UK
February 13, London, Victoria, UK
http://www.joomla-day.uk

PHP UK,
February 18–19, London, UK
http://phpconference.co.uk

DrupalCon Asia
February 18–21, Mumbai, India
https://events.drupal.org/asia2016

ConFoo 2016
February 24–26, Montreal, Canada
http://confoo.ca

Drupalcamp London 2016
March 4–6, London, U.K.
http://drupalcamp.london

Midwest PHP 2016
March 4–5, Minneapolis, MN
http://2016.midwestphp.org

DrupalCon 2016
May 9–13, 2016, New Orleans, LA
https://events.drupal.org/neworleans2016

IPC 2016
May 29–June 2, Berlin, Germany
https://phpconference.com

Lone Star PHP
April 7–9, Dallas, TX
http://lonestarphp.com

New Orleans DrupalCon
May 9–13, New Orleans, LA
https://events.drupal.org/neworleans2016

phpDay 2016
May 12–14, Verona, Italy
http://2016.phpday.it

php[tek]
May 23–27, St. Louis, MO
http://tek.phparch.com

PHP South Coast
June 10-11, Portshmouth, UK
https://cfp.phpsouthcoast.co.uk

Dutch PHP Conference
June 23–25, Amsterdam, The Netherlands
http://www.phpconference.nl

php[cruise]
July 17–24, Baltimore, MD (leaving from)
https://cruise.phparch.com

These days, when not working with PHP, Cal can be found working on a variety of projects like Nomad PHP. He speaks at conferences around the world on topics ranging from technical talks to motivational talks for developers @calevans.

Things We Sponsor

Developers Hangout

Listen to developers discuss topics about coding and all that comes with it.
www.developershangout.io

NEPHP

NorthEast PHP & UX Conference.
August 11-12, 2016
2016.northeastphp.org

NomadPHP

Start a habit of Continuous Learning. Check out the talks lined up for this month.
nomadphp.com

SunshinePHP

Take a break from the cold to enjoy some sunshine and talk about PHP!
2016.sunshinephp.com

DC PHP

The PHP user group for the DC Metropolitan area
meetup.com/DC-PHP

FredWebTech

The Frederick Web Technology Group
meetup.com/FredWebTech

Using Code to Help You Code Better

David Stockton

Last month, we talked about how our code is a liability and how we can determine what problems we need to solve before writing more code. However, liability or not, code is inevitable, so we should always strive to improve our code whenever possible. Fortunately, we have quite a few tools available to help us do that, and this month we're going to talk about some of them.

It has long been the goal of many to make programming a thing of the past and allow non-developers to simply tell the computer what they want—at which point the computer would create the necessary code to solve their problems. This has not yet materialized. This is fortunate for us, because it means we can keep our coding jobs for the foreseeable future, but it is also unfortunate, because software capable of performing such tasks could bring about all sorts of advances we haven't even thought of yet.

However, I don't believe software like that is, in fact, a real possibility. In order for it to work for anything non-trivial, humans would still need to provide requirements to the computer in a clear, unambiguous manner. The requirements would need to be syntactically correct, so that the computer could understand and interpret them correctly. We all know how difficult it is to get clear and unambiguous requirements for building software. As humans, we're quite good at parsing the requirements, but understanding them is a tough one. The requirements we receive are often open to interpretation and assumptions. Many times, we don't even realize we've made assumptions until a bug or defect is opened or a new requirement comes in to clarify or explain what the client really wanted. And often what they really want isn't even known by them until they see the code running and decide that whatever they see doesn't match what they wanted—even if it does precisely what they asked.

How More Code Can Help

So for now—and for the foreseeable future—software is pretty bad at writing software. Even so, that doesn't mean we can't use it to gain insight into our code. As I'm writing this article, I can see a running count of words at the bottom of the screen. Right now, it reads 370 words. While I could go through what I've written and figure out how many words it contains, the software I'm using can do it significantly faster than I could ever hope to. Code can find syntax errors faster than I can. But we can do even more than syntax checking and word counts with software. We can use software to gain insight into our codebase—let it find patterns, use it to help determine ways we can improve our code and our tests, have it identify areas that need better testing or refactoring, and more. This month, we'll look at some of these tools and see what insights they can provide, and we'll learn how to integrate them so you are kept informed and working on improving our code.

PHPUnit

PHPUnit[1] has been the topic of many an article in php[architect] over the years. I will briefly discuss it here. It's a testing framework that helps with ensuring that our code is doing what we expect and not doing what we don't expect. Of course, in order to use this tool, we have a lot of work to do. We must write tests, whether they are at the unit level, integration, functional, or system tests. For now, I'll state that PHPUnit is a tool that you should be using and leave it to other future articles to elaborate more on that.

PHPLOC

The PHPLOC[2] (or PHP lines of code) is a tool that doesn't require any work on your part to use other than running it. It will run across your codebase and give you insight about the makeup of your code. It does a lot more than just check lines of code, however. Since it's written in PHP and knows about PHP, it will give you listings of how many and what types of classes, comments versus lines of code, and more. Take a look at Listing 1 to see a sample output from PHPLOC of some code I work on regularly.

As you can see, about one-third of the overall codebase is comments. The average length of a class is 10 lines of code, but the maximum length is 212. It may be a good idea to look at and refactor that class. Looking farther down, the average method length is just 2 lines of code, but the longest length is 42 lines. Again, this is potentially pointing to a place that might warrant some further investigation and simplification.

The next section is *Cyclomatic Complexity*. Cyclomatic complexity is a measure of how many different linear independent paths through the code exist. For example, a method with no loops and no branches has a cyclomatic complexity of 1. There are no branches and hence, there is only one way through that code. Each branch point in the code, including loops and case statements increase the cyclomatic complexity number by one. For example, a method with two `if` statements would score a 3 for cyclomatic complexity, scoring one for the entrance to the method and one more for each of the `if` statements.

Methods and functions with high cyclomatic complexity are harder to understand and harder to test. They are spots where bugs find places to hide. This means that I should probably look at my code and find ways to reduce the number of conditionals and loops for my method that has a

1 PHPUnit: https://phpunit.de

2 PHPLOC: http://phpa.me/github-phploc

LISTING 1

```
01. phploc 2.1.3 by Sebastian Bergmann.
02.
03. Directories                     621
04. Files                          1592
05.
06. Size
07.   Lines of Code (LOC)         129318
08.   Comment Lines of Code (CLOC)  40834 (31.58%)
09.   Non-Comment Lines of Code (NCLOC)  88484 (68.42%)
10.   Logical Lines of Code (LLOC)  24354 (18.83%)
11.     Classes                    16202 (66.53%)
12.       Average Class Length        10
13.         Minimum Class Length       0
14.         Maximum Class Length     212
15.       Average Method Length         2
16.         Minimum Method Length       0
17.         Maximum Method Length      42
18.     Functions                   1336 (5.49%)
19.       Average Function Length      26
20.     Not in classes or functions 6816 (27.99%)
21.
22. Cyclomatic Complexity
23.   Average Complexity per LLOC    0.08
24.   Average Complexity per Class   2.34
25.     Minimum Class Complexity     1.00
26.     Maximum Class Complexity    78.00
27.   Average Complexity per Method  1.38
28.     Minimum Method Complexity    1.00
29.     Maximum Method Complexity   34.00
30.
31. Dependencies
32.   Global Accesses                 13
33.     Global Constants               0 (0.00%)
34.     Global Variables               0 (0.00%)
35.     Super-Global Variables        13 (100.00%)
36.   Attribute Accesses            3974
37.     Non-Static                  3965 (99.77%)
38.     Static                         9 (0.23%)
39.   Method Calls                 13062
40.     Non-Static                 12972 (99.31%)
41.     Static                        90 (0.69%)
42.
43. Structure
44.   Namespaces                     525
45.   Interfaces                      65
46.   Traits                           2
47.   Classes                       1360
48.     Abstract Classes               0 (0.00%)
49.     Concrete Classes            1360 (100.00%)
50.   Methods                       5363
51.     Scope
52.       Non-Static Methods        5362 (99.98%)
53.       Static Methods               1 (0.02%)
54.     Visibility
55.       Public Methods            5157 (96.16%)
56.       Non-Public Methods         206 (3.84%)
57.   Functions                       50
58.     Named Functions                0 (0.00%)
59.     Anonymous Functions           50 (100.00%)
60.   Constants                       20
61.     Global Constants               0 (0.00%)
62.     Class Constants               20 (100.00%)
63.
64. Tests
65.   Classes                         92
66.   Methods                        261
```

cyclomatic complexity of 34. We'll talk about other tools that also use this metric in a bit.

Next up is the section for Dependencies. Currently, there are no global constants or global variables, but there are thirteen places where my code is accessing Superglobals. I took a look at this one in particular, and fortunately, all of them happen in three methods across three classes. I could refactor these to inject the values I need instead of having those methods reaching out into the Superglobals. Removing these accesses will make the code easier to test and will eliminate a potential source of bugs. Also in this section are the numbers and types of attribute accesses and method calls.

The structure section indicates how many namespaces, interfaces, and traits were found. It splits classes between concrete and abstract. Methods are broken down between public and nonpublic and static and non-static. Additionally, functions and constants are shown. For the output in the listing, I also included the `--count-tests` option, which counts PHPUnit tests and lists the number of classes and methods.

PHPLOC provides interesting statistics relatively quickly on your codebase, but it is up to you to interpret and act on them. For now, let's move on to the next tool.

PHP_Codesniffer

If you're working on a codebase with a team, being able to easily read and understand each class and section of code is important. A few years ago, convincing team members to follow a coding standard was a much bigger challenge than it is today. While there are still developers who rail against coding standards, most have embraced the reality that even if you don't completely agree with all parts of any given coding standard, having a coding standard is better than not having one. Many—if not most—open-source PHP projects define and follow a coding standard. It's common now to use one defined by the PHP-FIG (Framework Interoperability Group). PSR-1[3] and PSR-2[4] are the two. There are also a number of other coding standards that are included with PHP_Codesniffer, such as PEAR and Zend standards.

PHP_Codesniffer[5] allows you to use any of these defined standards or to use them while either adding in or removing certain "sniffs." Or you can also define your own standard using the "sniffs" from any of the standards. The software will run across your codebase and report any errors or warnings for code that doesn't conform to your chosen or defined coding standard. This allows software that is impartial to let developers know when they've submitted code that is not compliant with the coding standards, even going so far as to reject pull requests or commits that are not up to standards (if you so choose).

Additionally, there are a number of tools available that will automatically reformat code to follow your chosen coding standard. I won't be covering these here, however. Choosing a defined coding standard (recommended) or creating your own (not recommended) means that no matter what section of code a developer is working on, the resulting code should look the same, which lowers the barrier to making changes by allowing developers to focus on the code, not on how it's formatted.

---

3 *PSR-1:* http://www.php-fig.org/psr/psr-1/

4 *PSR-2:* http://www.php-fig.org/psr/psr-2/

5 *PHP_CodeSniffer:* https://github.com/squizlabs/PHP_CodeSniffer

PHP Copy/Paste Detector

The PHP Copy/Paste Detector[6] finds copy/pasted code in your software. And it's smarter about it than you might think. First of all, in order to count, duplicate sections of code need to be longer than a certain threshold, so a bunch of small methods that are duplicated won't be reported as duplicated. Secondly, even if the code has had variable names changed, comments added, or been reformatted, PHPCPD can find and report on those duplicate sections. It's actually using PHP to parse tokens to determine blocks of duplicated code, so formatting and variable names don't necessarily come into play. The report for PHPCPD will indicate the files and line ranges for found duplicated code as well as a summary indicating how much of the codebase is made up of copy/pasted code.

While it might be tempting to keep this tool reporting at 0% all the time, that may not be a good idea in all cases. In many situations, including reducing copy/pasted code, the solution is "it depends." Often copy/pasted code can be reduced by refactoring the code into a common method or class and then calling it from the places that require it. However, it is important to ensure that the duplicated code is actually doing the same thing for the same reason. It's often better to deal with duplicated code as opposed to introducing an incorrect abstraction.

PHP Mess Detector

The PHP Mess Detector[7] is another code analysis tool that can tell you about potential problems and issues. This includes reporting on dead code (code that is unreachable), overcomplicated expressions (cyclomatic complexity), and other possible bugs. The PHPMD tool has a number of different rulesets that can be enabled or disabled, each containing several different rules that it uses to detect these (potential) problem areas.

The first set of rules are about writing clean code. It will detect and report on functions and methods that use Boolean arguments. The reason for this is that a Boolean argument often represents a violation of the single responsibility principle (SRP). It's the "S" in "SOLID," which means each of our methods should do one thing and do it well. A Boolean argument may indicate two different responsibilities provided by one method. To fix, the logic around the Boolean flag could be extracted into another method or class.

The "clean code" ruleset also looks for static access to other resources and uses of "else." Static access can be a problem when testing because it makes it difficult to introduce a test double. And according to the PHPMD docs, the "else" keyword is never needed; it recommends the use of a ternary when doing simple assignments. Personally, I don't always follow all of those recommendations, but having the PHPMD report list these as issues gives me an opportunity to revisit code and potentially come up with a better or simpler solution.

Next up are the unused code rules. These detect unused private fields, unused local variables, unused private variables, and unused formal parameters. I've found that the first three of these are always fixable and often indicate either a change in design or a refactor that left a little bit of mess in the codebase. Removing them is nearly always the right thing to do and shouldn't ever result in broken code. The only rule I've had any issue with is unused formal parameters. If the parameter is part of an interface you're implementing, then removing it requires a change to the interface, which would necessitate the change of other classes that implement the interface. This may not be possible. If the method in question doesn't implement a method from an interface, then removing the unused parameter would require finding all callers of the method and changing them. Tools like PhpStorm make this easy, but if the unused parameter is not the last one in the list, it's imperative to update all callers to remove the parameter. If this is not done, it will introduce an error.

The third set of PHPMD rules concern naming. These detect fields, parameters, and local variables that are too long or too short. For instance, variable names like `$id` will be flagged as being too short. Longer variables, over 20 characters, will also be flagged. Additionally, the naming rules will find constants that are not defined in upper case. It will detect and flag PHP 4 style constructors. These are constructors that aren't `__construct`, but rather match the name of the class. Finally, it will flag "getters" for Boolean fields that use the word "get" instead of "is" or "has." For example, a getter named "getValid" on a Boolean field will be flagged with the recommendation that the getter be called `isValid()` or `hasValid()`.

PHP 4 style constructors were deprecated in PHP 7.

Additionally, PHPMD includes a set of rules around design. It will flag code that uses `exit()`, `eval()`, or `goto`. It will flag classes that have more than 15 child classes, and it will flag classes that descend from more than six classes. Both of these indicate that there is likely an unbalanced inheritance hierarchy. Finally, it will look at coupling between objects. This means it will count up dependencies, method parameters that are objects, return types, and thrown exceptions. If there are more than 13 total, the class will be flagged. This detection not only works on formally type-hinted parameters, but it also examines the doc block comments for `@returns`, `@throws`, `@param`, and others. Each of these coupled classes indicates some other class that the class in question must know something about, or that consumers of this class need to be aware of. Keeping this number low means it's easier to work with the code.

The next-to-last set of rules are the so-called "Controversial Rules." They are mostly about coding style, and many would be covered by a PHP_codesniffer ruleset—ensuring class names, property names, method names, parameters, and variables are defined in camel case. The final rule in this set is about accessing super global variables. Where PHPLOC will report a count of how many times Superglobal variables are accessed, PHPMD will give you a report of where all those places in the code are.

The final ruleset for `phpmd` contains the "Code Size Rules." These are the rules that, for me at least, lead to the best improvements in the code, but they also typically take the most effort to resolve. The first is Cyclomatic Complexity. As mentioned in the PHPLOC section, this is how many paths there are through the code. Methods with more than 10 paths will be flagged, and PHPMD will count each `if`, `else if`, `for`, `while`, and `case`, along with 1 for entering into the method.

The next rule is for NPathComplexity. This is similar to cyclomatic complexity, but it is the number of unique paths through the code, not just linear independent paths. For this metric, each added conditional or loop can have a multiplier effect on the number of paths through the code. A score of 200 or higher will result in PHPMD flagging the code.

6 *PHPCPD: http://phpa.me/github-phpcpd*

7 *PHPMD: http://phpmd.org*

The excessive method length rule looks at the lines of code in a method as an indicator that a method may be trying to do too much and recommends refactoring into other helper methods and classes or removing copy/pasted code. Similarly, the excessive class length rule looks at the lines of code in the entire class, again as an indication the class might be doing too much.

The excessive parameter count rule flags methods that have more than 10 parameters. It may indicate that a new object should be created to group like parameters. The excessive public methods rule flags classes that define more than 45 public methods. It indicates that a lot of effort may be needed in order to thoroughly test the class. The recommendation is to break the class into smaller classes that each do less. This rule will also flag classes that contain too many methods (public or otherwise) or too many properties. More than 15 properties or 25 methods may indicate a class that could be reduced in complexity. Both the public method and total method rules will ignore getters and setters by default.

Finally, the excessive class complexity rule totals up all the complexity metrics of the various methods in a class. It gives an indication of the relative amount of time and effort that would be needed to maintain or modify the class.

It is simple to configure PHPMD to use some or all of the rules across any of the rulesets. If a configured rule doesn't make sense for your codebase or you don't agree with it, you can simply disable it. Overall, though, PHPMD provides a great way to automatically detect potential problem areas in your code.

Recommendations for Usage

While it is possible to run all of these tools on your own development machine, chances are that doing so will quickly become tedious and you'll decide to stop running them. Instead, I'd recommend setting up a "build" in Jenkins or other continuous integration server (think TravisCI, Bamboo, or others). These tools can be configured to run any time new source code is checked in, whether that's merged into the mainline or, preferably, when a pull request is submitted. The CI tools can keep track of the various statistics and reports between one run and the next and produce charts and graphs that allow for easy viewing of trends and changes. The build job can be configured to ensure that statistics that indicate problems are increasing will cause a build to fail, indicating that the code should not be merged or accepted until the issues introduced are resolved.

The CI server can report back build status via email, slack, IRC, and other ways to inform you or other developers of the results of running all of these tools. It's then up to you to determine how much effort and when you want to maintain the code to reduce the various errors and warnings presented by these code analysis tools.

Conclusion

I would recommend looking at these tools, trying them out, and taking a look at other tools that are mentioned on the PHP QA Tools page: http://phpqatools.org. In addition to providing installation and configuration for the tools I've talked about, this page also provides information about getting all of these tools (and more) to work in Jenkins. If you're already running a CI server and don't have these tools, consider installing them and integrating the reports into your build. Use the reports to increase the quality and maintainability of your code.

Finally, one tool we didn't get into much this month is PHPUnit. With PHPUnit, it's important, but not always easy, to write tests that are effective and help to ensure the code is doing the right thing. Next month, I'll talk about a tool that can help you determine if your tests are as effective as you think they are. I'm really excited about sharing this with you, so I hope you'll join me next time.

David Stockton is a husband, father, and Software Engineer and builds software in Colorado, leading a few teams of software developers. He's a conference speaker and an active proponent of TDD, APIs and elegant PHP. He's on twitter as @dstockto, youtube at http://youtube.com/dstockto, and can be reached at levelingup@davidstockton.com.

Defaulting to Secure

Chris Cornutt

There's a natural tension in writing a new component or library between making it highly adaptable and work well for users out-of-the-box. For us, "works well" means being secure. Undoubtedly, this is a difficult juggling act and this month I'd like to share how security-by-default is important.

When writing good, SOLID (and DRY) code for your applications, we want to make it the best that we can. Depending on the library or feature that we're creating, this can mean many different things. Usually, though, this includes flexibility. We want to write code that can hold up over time, can roll with the punches, and can adapt to the needs of the rest of the codebase without too much extra effort from developers down the line. We happily build this flexibility into the code and make it easy for other developers to customize the objects they need with a few constructor arguments and method calls. Sounds like an ideal world, right? Well, in most cases it is, but there's one context where you could be doing a disservice to the users of your library or tool … you guessed it: security.

If you're a regular reader of this column, you know that I've talked about some of the shifts in thinking required to integrate development principles that are more secure into your everyday code. With the creation of most other kinds of tools and features, having as much flexibility as possible (within reason, naturally) is a good thing. You give developers all the access they might need to the inner workings of your tool with getters, setters, and injection points. They can craft a custom setup that's just exactly what they need at any given time. As the man once said, however, "with great power comes great responsibility." By providing all of this flexibility to a user, you may be providing them with functionality that could lead to their application being compromised.

Early on, the PHP engine itself did not cater to security by default. We had things like register globals, magic quotes, and session IDs in URL queries. Many, if not all, of these could be toggled via INI settings, so getting a PHP environment that was secure on installation was hardly possible. Luckily, we've learned a lot since those days and many of those features have been removed or deprecated.

Let's look at an example—a basic encryption library. Encryption itself can be a tricky topic to try to cover in an article like this, so I'm only going to use it as a backdrop here. I'm not going to get into guidance on what kind of cryptography is good or bad and give advice on what you should use—that's for another time. Instead, I want to use this as a platform to talk about one of the best things you can do for those who will be using your software: establish secure defaults.

Secure Defaults Defined

While the main point of writing open-source software and releasing it out to the world is to make things that work, there's also value in seeing how something is done. Developers create tools for a reason—to solve problems—and in solving them they usually figure out the best practices. They then take this knowledge and put it into the library so others can benefit from their effort and not have to make the same mistakes. Most of us have written, or will write, our own MVC framework to learn how they work.

Say we create a basic encryption library and give it two basic methods: `encrypt` and `decrypt`. In the back of every developer's mind right now there's probably this nagging voice looking at the next steps past these two obvious methods. They want to implement the rest of the class to allow a user to select whatever algorithm and mode he or she would like to use when encrypting or decrypting data. Pseudo-code starts forming, maybe with constructor arguments for each of these different types and a little logic internally to handle whatever the user selects.

There's just one problem here: if you're a developer who knows about encryption and are making this library for others to use, why aren't you giving them any guidance?

Now, back to our encryption library—if we allow the user to define whatever algorithm and mode combination, our users are not benefiting from our experience in working with the encryption of our own application. We've already solved the problem for our systems, so we have a set of "known goods" we can share with the rest of the world via our code. Instead of making the library uber-flexible and leaving it up to the user to decide what's good or not, be direct and tell them…not in the documentation but in the code itself.

Instead of offering direct access to the algorithm/mode combination, create a series of "levels" of complexity the user can pick from based on how strong they need the encryption level to be. This could even be as general as a "low," "medium," and "high" set of choices, possibly defined as something like a class constant or changed via a method call. The key here is that this grouping of defaults provides the end user with some "known goods values" to work from. They get to stand on your shoulders and use what you've learned to make their application even more secure and with a minimum amount of effort.

For example, PHP's `password_hash` function implements a safe default hash algorithm via the `PASSWORD_DEFAULT` constant. Symfony's Twig templating engine defaults to encoding output; you have to override this behavior to render raw output.

Default, Not Popular

When creating these defaults, be prepared to find that they're not always going to be popular. The needs of some end users may not exactly match the carefully selected defaults you've created. Using your library may be a little frustrating for them. Keep this in mind and think about what kind of functionality you'd like to provide. Maybe allow a more advanced user to tweak the settings manually in an alternative way rather than the simple default-driven method. However, be wary of including something like this—that kind of functionality has a way of being abused.

So, How Do You Pick Those Defaults?

This is the trickiest part of the whole thing. It entirely depends on what you're trying to accomplish. In my example, I've been talking about encryption functionality since the idea of "levels" transfers over easily to that world. What if you're not creating an encryption library with such clearly defined "levels" you can rely on? A few questions you can ask that could help you determine what these levels might be in your code are:

1. Is there any part of the functionality that most users won't need? Sometimes there's a natural split between the pieces that 90% of users will make use of and the other 10% that's only for advanced usage. Use this as a guide on where to place these levels but be sure that there actually is a base (and it's not defaulting to null).

2. What's the "bare minimum" operating environment someone might want for your tool? When thinking about these base-level users, think about what minimum requirements would be needed to give them a secure and working environment. Usually this can be based on whatever the most common use case is for the tool in question—just be sure it follows good practices!

3. Are there any previously defined best practices around this sort of thing you can lean on? As with any kind of development, there's going to be similarities to tools that are already out there. While what you're doing with yours may be a little bit different, there's also something to be said from learning from others. Look around at other tools in the same realm and see if they have settings or defaults already. Hopefully, these are based on some research done by the author, but before blindly using them as a good practice, do a little digging before using them as defaults in your code.

4. Are we providing functionality to use these defaults correctly in a safe way? It's tricky to provide solid, secure functionality with predefined settings and not undermine it with advanced features that would let the user turn those settings right back off. Be sure that a user can't override the defaults you're providing but allow them to augment them to match their needs more closely if necessary.

Conclusion

Next time you're reviewing the components you're creating or the third-party libraries you're pulling in, pause to ask if they are providing secure defaults and see if you're using them securely. Ask yourself these questions and think about the "chunks" of your library or application and see if there are any boundaries to help split it up and even maybe help others write secure code without thinking about it. Share what you've learned in developing the code and help others secure their own apps even more effectively.

Biography

For the last 10+ years, Chris has been involved in the PHP community. These days he's the Senior Editor of PHPDeveloper.org and lead author for Websec.io, a site dedicated to teaching developers about security and the Securing PHP ebook series. He's also an organizer of the DallasPHP User Group and the Lone Star PHP Conference and works as an Application Security Engineer for Salesforce.

January Happenings

PHP Releases

- PHP 7.0.2: http://php.net/archive/2016.php#id2016-01-07-1
- PHP 5.6.17: http://php.net/archive/2016.php#id2016-01-07-3
- PHP 5.5.31: http://php.net/archive/2016.php#id2016-01-07-2

News

Derick Rethans: Re-proposed Adopting a Code of Conduct

After Anthony Ferrara withdrew his RFC proposing that the PHP project adopt a Code of Conduct, Derick Rethans stepped forward to re-propose it. He strongly believes the a CoC is required because the toxic behavior on internals drives people away, him included. This RFC has been very controversial, since it's based on the Contributor Covenant, which some critics point out has some very vague provisions.
http://news.php.net/php.internals/90728

Larry Garfield: Giving Back in 2016

In the latest post to his site Larry Garfield makes a charge to the community—both Drupal and the wider PHP community—to gave back in 2016 and make an effort to contribute in some way back to the projects you use and love. He suggests not only that you get out and give back but that you also do it in somewhat unfamiliar territory.
http://phpdeveloper.org/news/23589

Softpedia.com: Debian Is Moving to PHP 7, and So Are Numerous Other Linux Distributions

According to this new article on the Softpedia site, Debian and several other major Linux distributions will soon be making the move up to PHP 7 for their releases, the latest major version of the PHP language. In fact, PHP 7 has already made it to the unstable releases of the Debian linux distribution for those that would like to test things out.
http://phpdeveloper.org/news/23566

Mark Baker: A Functional Guide to Cat Herding with PHP Generators

In this post to his blog Mark Baker looks at a feature added in PHP 5.5 - generators—and how to use them with some of the array handling functionality PHP provides. He starts with a more "real world" example of using a generator in a handler for GPX files, XML files storing GPS data. He gives an example of the typical file contents and shows a simple generator script (class) that he uses to grab chunks of the file at a time instead of reading it all in and parsing it from there.
http://phpdeveloper.org/news/23570

Laravel News: Automatically Upgrade Your Laravel App with Shift

On the Laravel News site they've posted an interview with Jason McCreary, the lead developer behind the Laravel Shift service, a product that helps you keep your Laravel applications up to date with the latest versions of the framework. They talk about where the idea for Laravel Shift came from originally and how the upgrade process happens (hint: it's automated).
http://phpdeveloper.org/news/23530

Ibuildings Blog: Programming Guidelines—Part 1: Reducing Complexity

On the Ibuildings blog Matthias Noback has kicked off a series that wants to help PHP developers reduce the complexity of their applications. In part one he shares some general tips along with code snippets illustrating the change. The rest of the article is broken up into several changes you can make to reduce complex code.
http://phpdeveloper.org/news/23581

Securi Blog: Security Advisory: Stored XSS in Magento

Securi looks at a stored XSS vulnerability in Magento 1 and 2 that can be used by attackers to take over a site, create admin accounts, and more. Attackers could upload Javascript via an unsanitized email field that could be displayed in the application's admin backend.
https://blog.sucuri.net/?p=14597

Michael Cullum: CFP Tips Roundup

Micahel Collum has collected the CFP Tips that he started sharing on twitter at the end of December. He wrote them to help speakers improve the quality of their submissions. He's one of the organizers of PHP South Coast and had gotten requests to put his advice in a more permanent archive.
http://www.michaelcullum.com/cfp-tips-roundup/

Ethode.com: Fixing Spaghetti: How to Work With Legacy Code

On the Ethode.com blog they've shared some hints for working with legacy code to help you more effectively refactor your way out of the "spaghetti code" you might have right now. These are more general tips and aren't really PHP (or even really web application) specific but they're a good starting place for any refactoring effort.
http://phpdeveloper.org/news/23601

CloudWays Blog: Co-Founder Of PHPwomen, Ligaya Turmelle Talks About PHP, MySQL and php[tek]

The CloudWays blog has posted their latest in their series of interview with members of the PHP community. In their latest they talk with Ligaya Turmelle about PHP Women, Oracle, MySQL and php[tek].
http://phpdeveloper.org/news/23595

Building a Conference Schedule

Eli White

It's been a week since we finished the process of selecting talks and publishing the schedule for our upcoming php[tek] 2016 conference. Every time we publish a schedule, I receive many questions about why someone wasn't accepted. I do my best to answer those, but it's always tricky.

I recently talked to someone who mentioned that they were considering not responding to the Call for Speakers because they had not been accepted in a while. As I told them, not being accepted is not solely based on how great your submissions are. Numerous other factors come into play.

I'd like to explain how we at php[architect] select sessions. The process varies for each conference we put on, and it can be drastically different for conferences put on by other people. However, I wanted to describe our "baseline" to be open about it and help other speakers.

> *Conferences are really like parties, and an A-list party is one where A-list people are in attendance. You figure out who are the really important people to invite and get them to show up as speakers or as guests. Then everybody wants to be there. If you don't know who the important people are, you shouldn't be doing a conference.*
>
> *— Tim O'Reilly*

The Ratings

First, we review submissions as they come in throughout the Call for Speakers. We have a team at php[architect] who go through every submission, whether we get 300 or 600. We rate them all individually on their own merits. The talks are rated in random order for each reviewer to help mitigate any issues with people seeing talks in a specific order, which can help or hurt a specific talk.

We use a half-blind method of rating here. As a reviewer rates a talk, they first see just the title and abstract, and they rate it solely based upon that information using a scale of 1-5. After the initial rating, if the rater chooses, they can see the name of the presenter and adjust his or her rating. For example, this happens when a talk about a specific tool looked good, but after the initial rating, the rater realized the speaker is the tool's creator; this can bump the rating from *good* to *great*.

The raters consider the specific conference and its overall goals, which topics the team feels are hot at the moment, and the specific themes we are highlighting that year. Therefore, a talk that might get a 5-star rating for php[world] may only get a 3.5-star for php[tek]. Those can change from year to year. We don't share the raw reviews with submitters because they aren't useful outside the full context of the conference and every other submission we received.

The Spreadsheet

The next step of the process is less time consuming, but it's much more mentally taxing. I import all the talks and their average rating scores into a spreadsheet for easier sorting. I break them up into three tabs (for our conference purposes): Sessions, Tutorials, and Short Talks. Each category has a specific number of slots available in the conference schedule.

The general process involves tracking how many slots to fill for each category. For example, for php[tek] 2016[1], we needed 38 sessions, 6 tutorials, and 8 short talks. You might think it would be easiest to sort the talks by average rating from high to low and then select the top 38, 6, or 8 talks in the appropriate tab, giving us an amazing schedule. Yes, we would like to do that, but…

For many important reasons, we can't be that straightforward in our selection process. Much more goes into making what we consider the "perfect schedule". This is the point that makes most speakers upset. It's rare, but you can have an amazingly high-rated session and not make it into schedule. Why?

The Budget

Well, there are many reasons. First, we are a company, and we need to hit our bottom line. We have to consider our budget. We spend a significant amount paying for speakers' flights and hotels. For that reason, we prefer to get two talks from any speaker we select. We also have to be careful about how many international speakers we accept because those flights are significantly more expensive (up to 4x as much).

The first thing my co-chair and I do is sit down and look through the list. We start with the tutorials, then move on to the short talks and the sessions. Starting at the top of the list, and for each talk, we ask, "This is their highest-rated tutorial. What's the next highest-rated submission from this speaker? Is it good enough to accept?" We slowly work down the entire list, looking for two talks from each speaker as we go. Sometimes we remove a speaker because his or her highest-rated talk only got 4.67 stars and their next talk only got 3.12 star, which we deemed too low. Note that with the number of amazing submissions we get, we often have over 200 talks rated 4.5 or better.

Eventually we hit a point at which we are *close* to having a full schedule. Typically we reach a painful situation where we have 1 or 2 talk slots left and 5 to 10 people who could fill them, all equally rated. At that point, the next phase begins, which is…

The Balancing

In this step, my co-chair and I take a step back, and brace ourselves for the most painful part of the process. We look at the schedule as a whole, thinking about which topics are covered and which aren't. Do we have too many talks on X? Did we manage to select something on Y? Did we get so many talks on Z that it's obviously a hot, up-and-coming topic, and did we accept enough of them?

Now the *art* of the selection process begins. Talks are added or removed, and we compare how the schedule looks after these changes. We have to make sure we have enough but not too much

1 php[tek]: http://tek.phparch.com

of each topic we need. Some of the more controversial parts of the process occur at this point. We need to look at things that will make the conference "sell". While we want as many new faces as possible, do we have enough "known names" to help us sell tickets? Will people pay extra to see the talks we chose for Tutorial Day? Have we ensured that none of the talks conflict directly with topics covered on Training Day or Tutorial Day, leading attendees to avoid buying tickets for those extra days?

After all this is done, we have a schedule. But it's *still* not final. At this point, we hop on a Google Hangout with all our team members. We go through the schedule again, sometimes once and sometimes up to three times, with fresh eyes, looking for imbalances: advanced vs beginner, topics, speakers, draw, etc. It all plays into our decisions. During this time, talks that were in may be pulled out, and others take their place. This is also when we start begging our CFO to loosen the purse strings a bit, and we often add a few extra speakers, especially those who might only cover one topic just because it is extremely important or the session will be amazing.

Finalization

After all that—which usually happens in the span of only one week—we have what we consider a final schedule. We make one final pass to catch typos, to make sure speakers aren't giving two talks in the same slot or on the same day, and to polish the schedule to get it ready.

Finally, we announce it and cross our fingers, hoping that people really enjoy what we put together. I get the amazing job of telling speakers they were accepted and the heart-breaking job of telling the rest that they weren't. That heart-breaking job is even worse when I talk to people I knew were right on the cusp or even in the running at one point before being removed during the balancing process.

In all honesty, that still doesn't end the process. We have to deal with keynotes. Inevitably, a speaker drops out, which restarts the entire process as we refer back to all our notes to determine how best to fill that gap in the schedule.

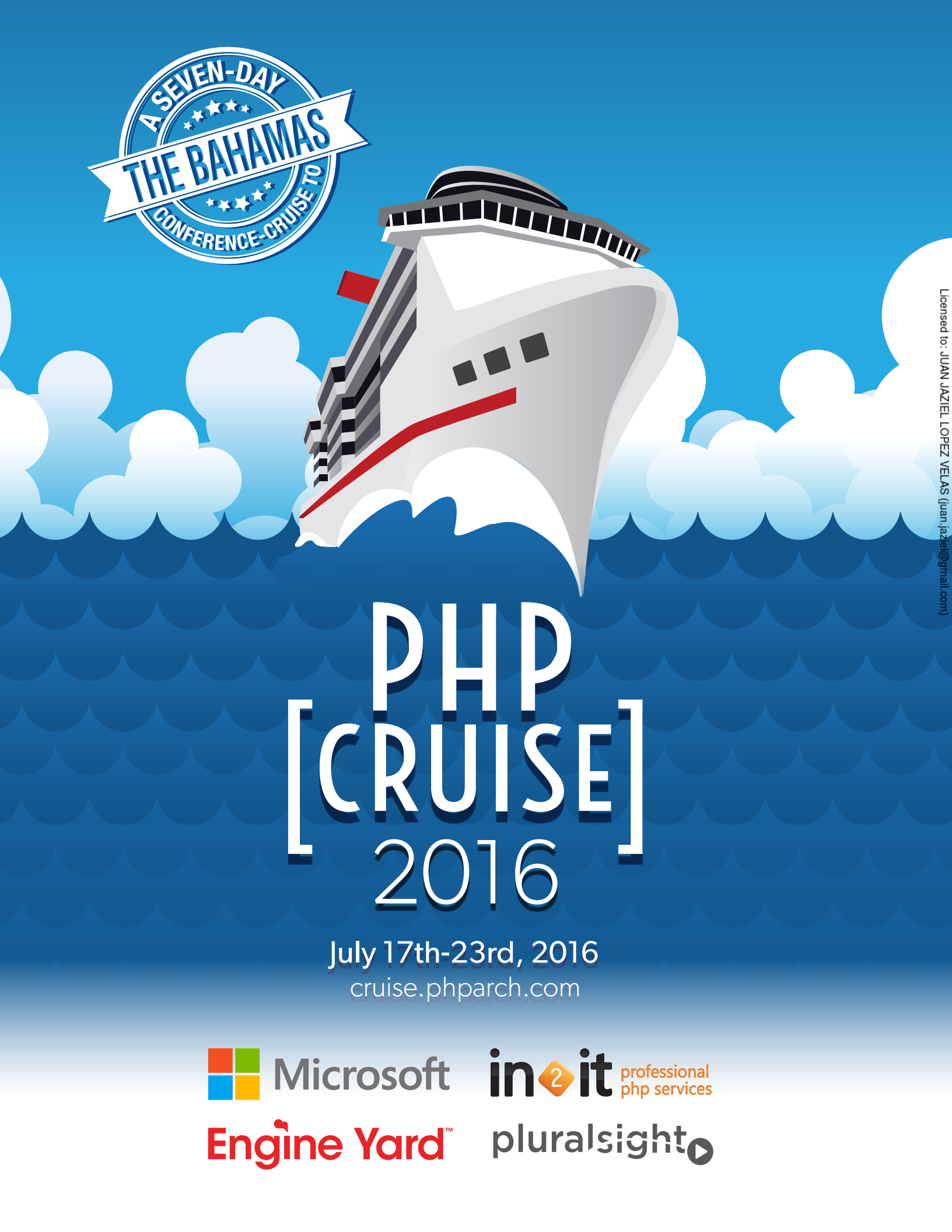
In Conclusion

There you have it: insight into what goes into making a conference schedule. Don't let this discourage you from speaking or submitting talks. In fact I hope it does the opposite. In most cases, the reason that a talk wasn't accepted has less to do with you not being an amazing speaker, or concocting the perfect magical incantation of an abstract that means that you are accepted. Selection is based on so many factors as to be mind-boggling. Personally, I've had years where I was accepted at 15 conferences, followed by a year where I was accepted at zero. It's just part of how everything fluctuates.

I wanted everyone to have an honest look at how and why we select our speakers and presentations. The reality is that we can only accept a small portion of the submitted talks. And by far, the majority of them are awesome talks that we wish we could accept all of them.

Eli White is the Managing Editor & Conference Chair for php[architect] and a Founding Partner of musketeers.me, LLC (php[architect]'s parent company). He has now been the conference chair for two different companies, and a total of 9 conferences past or upcoming. Hopefully for many more as well.

"This porridge is just right." https://en.wikipedia.org/wiki/Goldilocks_and_the_Three_Bears#/media/File:The_Three_Bears_-_Project_Gutenberg_eText_17034.jpg



XML Parsing in PHP

by John M. Stokes

Since its introduction in 1998, XML has become an indispensable technology for exchanging data and operating across heterogeneous systems. Whether you're sharing calendar events, importing syndication feeds, or querying an external API, XML is usually one of the formats available for consumption.

Early versions of PHP provided extensions for working with XML, but it wasn't until the introduction of SimpleXML, DOM, XMLReader, and XMLWriter in PHP 5 that working with XML was streamlined. XML Parsing with PHP, edited and produced by php[architect], provides a comprehensive survey of the classes and functionality available for working with XML. This edition covers parsing and validating XML documents, leveraging XPath expressions, and working with namespaces as well as how to create and modify XML files programmatically. Each chapter contains examples illustrating how to use the different XML extensions at your disposal.

Written by PHP professional John M. Stokes, this book provides an easy-to-use reference for working with XML.

Purchase

http://phpa.me/xmlparsing

PHP SWAG

PHP Drinkware

Tumbler

PHPye Shirts

php[architect]™

Visit our Swag Store where you can buy your own plush friend or other PHP branded gear for yourself.

As always, we offer free shipping to anyone in the USA, and the cheapest shipping costs possible to the rest of the world.

Get yours today!

www.phparch.com/swag

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[architect]

magazine
books
conferences
training

www.phparch.com

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe