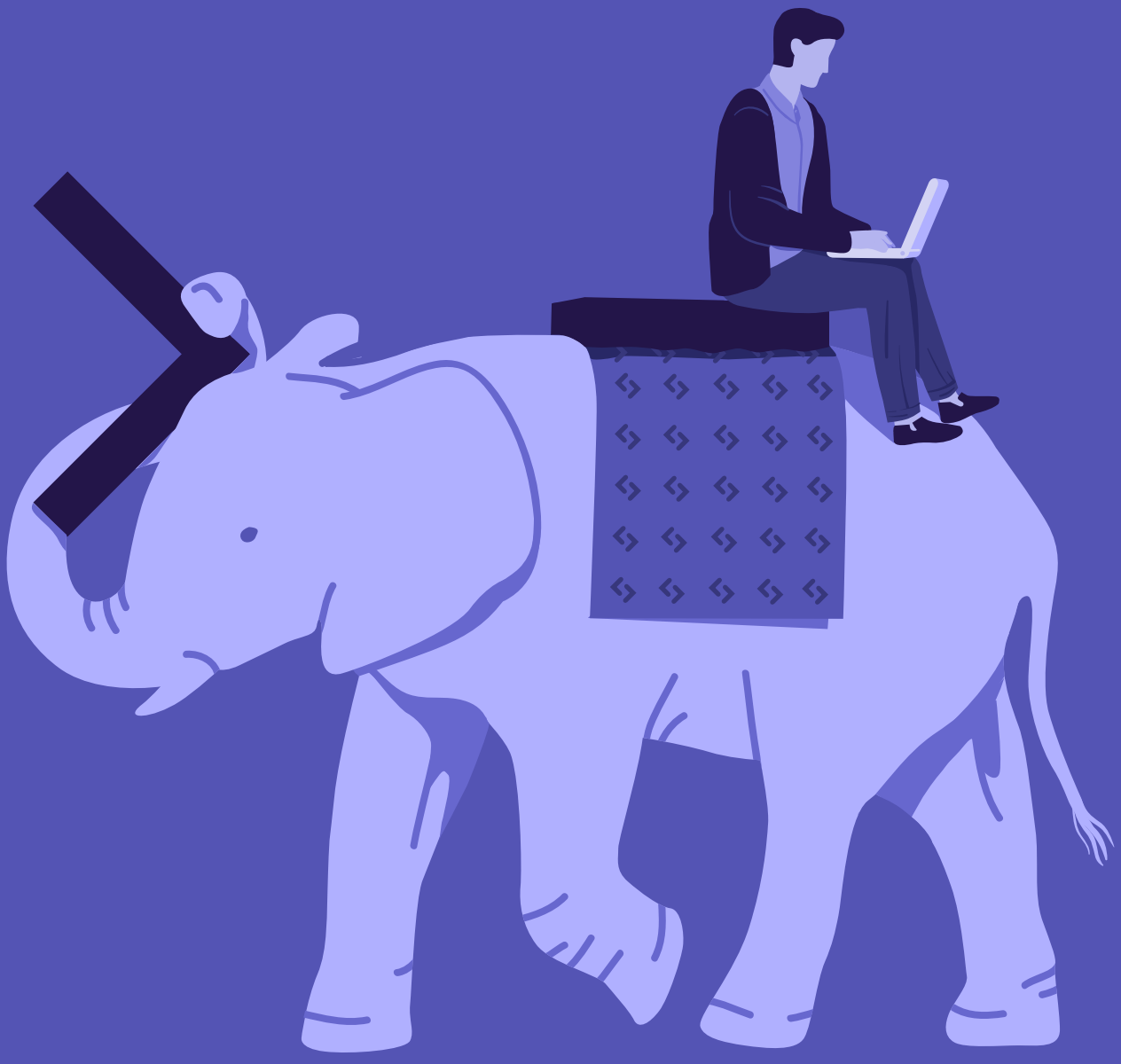


Better PHP Development

Better PHP Development

Cover Design: Natalia Balska

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

Table of Contents

Preface

PHP powers the vast majority of the web today. It is by far the most ubiquitous and accessible mature web language, and welcomes thousands of new developers every day. It is this ease of access that can, admittedly, sometimes give it a bad name - good resources are few and far in between, and the competition is real, driving people to take shortcuts, skip best practices, and learn on-the-fly.

With PHP 7+ came some improvements that make it harder to make mistakes, and 7.2 is making things even safer, even more structured. If you're just getting started with the language, you're in luck. Not only will it be ever harder to slip up and make mistakes, but content such as this—hand picked from the excellent SitePoint PHP channel—will help you get started the right way.

Once you've chewed through these modern introductions and re-introductions into popular tools and concepts which will undoubtedly become a permanent part of your tool belt, why not head on over to the channel and check out some articles or video courses that deal with these topics in more depth? Found something interesting you'd like covered? Let us know. For now, just dive in—it's the first step that's most important, and you're half way through yours. Happy PHPing!

Bruno Škvorc, SitePoint PHP channel editor.

Conventions Used

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk.</p>
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-design-real
➥ -user-testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter 1

How PHP Executes—from Source Code to Render

by Thomas Punt

Peer reviewed by Younes Rafie.

Introduction

There's a lot going on under the hood when we execute a piece of PHP code. Broadly speaking, the PHP interpreter goes through four stages when executing code:

1. Lexing
2. Parsing

3. Compilation
4. Interpretation

This piece will skim through these stages and show how we can view the output from each stage to really see what is going on. Note that while some of the extensions used should already be a part of your PHP installation (such as tokenizer and OPcache), others will need to be manually installed and enabled (such as php-ast and VLD).

Stage 1 - Lexing

Lexing (or tokenizing) is the process of turning a string (PHP source code, in this case) into a sequence of tokens. A token is simply a named identifier for the value it has matched. PHP uses re2c to generate its lexer from the zend_language_scanner.l definition file.

We can see the output of the lexing stage via the tokenizer extension:

```
$code = <<<'code'
<?php
$a = 1;
code;

$tokens = token_get_all($code);

foreach ($tokens as $token) {
    if (is_array($token)) {
 echo "Line {$token[2]}: ", token_name($token[0]), "
↪ ('{$token[1]}')", PHP_EOL;
    } else {
        var_dump($token);
    }
}
```

Outputs:

```
Line 1: T_OPEN_TAG ('<?php
')
Line 2: T_VARIABLE ('$a')
Line 2: T_WHITESPACE (' ')
string(1) "="
Line 2: T_WHITESPACE (' ')
Line 2: T_LNUMBER ('1')
string(1) ";"
```

There's a couple of noteworthy points from the above output. The first point is that not all pieces of the source code are named tokens. Instead, some symbols are considered tokens in and of themselves (such as `=`, `;`, `:`, `?`, etc). The second point is that the lexer actually does a little more than simply output a stream of tokens. It also, in most cases, stores the lexeme (the value matched by the token) and the line number of the matched token (which is used for things like stack traces).

Stage 2 - Parsing

The parser is also generated, this time with Bison via a BNF grammar file. PHP uses a LALR(1) (look ahead, left-to-right) context-free grammar. The look ahead part simply means that the parser is able to look `n` tokens ahead (1, in this case) to resolve ambiguities it may encounter whilst parsing. The left-to-right part means that it parses the token stream from left-to-right.

The generated parser stage takes the token stream from the lexer as input and has two jobs. It firstly verifies the validity of the token order by attempting to match them against any one of the grammar rules defined in its BNF grammar file. This ensures that valid language constructs are being formed by the tokens in the token stream. The second job of the parser is to generate the *abstract syntax tree* (AST) - a tree view of the source code that will be used during the next stage (compilation).

We can view *a form of* the AST produced by the parser using the php-ast extension. The internal AST is not directly exposed because it is not particularly "clean" to work with (in terms of consistency and general usability), and so the

php-ast extension performs a few transformations upon it to make it nicer to work with.

Let's have a look at the AST for a rudimentary piece of code:

```
$code = <<<'code'
<?php
$a = 1;
code;

print_r(ast\parse_code($code, 30));
```

Output:

```
ast\Node Object (
    [kind] => 132
    [flags] => 0
    [lineno] => 1
    [children] => Array (
        [0] => ast\Node Object (
            [kind] => 517
            [flags] => 0
            [lineno] => 2
            [children] => Array (
                [var] => ast\Node Object (
                    [kind] => 256
                    [flags] => 0
                    [lineno] => 2
                    [children] => Array (
                        [name] => a
                    )
                )
                [expr] => 1
            )
        )
    )
)
```

The tree nodes (which are typically of type `ast\Node`) have several properties:

- `kind` - An integer value to depict the node type; each has a corresponding constant (e.g. `AST_STMT_LIST` => 132, `AST_ASSIGN` => 517, `AST_VAR` => 256)
- `flags` - An integer that specifies overloaded behaviour (e.g. an `ast\AST_BINARY_OP` node will have flags to differentiate which binary operation is occurring)
- `lineno` - The line number, as seen from the token information earlier
- `children` - sub nodes, typically parts of the node broken down further (e.g. a function node will have the children: parameters, return type, body, etc)

The AST output of this stage is handy to work off of for tools such as static code analysers (e.g. Phan).

Stage 3 - Compilation

The compilation stage consumes the AST, where it emits opcodes by recursively traversing the tree. This stage also performs a few optimizations. These include resolving some function calls with literal arguments (such as `strlen("abc")` to `int(3)`) and folding constant mathematical expressions (such as `60 * 60 * 24` to `int(86400)`).

We can inspect the opcode output at this stage in a number of ways, including with OPcache, VLD, and PHPDBG. I'm going to use VLD for this, since I feel the output is more friendly to look at.

Let's see what the output is for the following **file.php** script:

```
if (PHP_VERSION === '7.1.0-dev') {
    echo 'Yay', PHP_EOL;
}
```

Executing the following command:

```
 php -dopcache.enable_cli=1 -dopcache.optimization_level=0
↪ -dvld.active=1 -dvld.execute=0 file.php
```

Our output is:

```
line     #* E I O op                           fetch          ext  return  operands
-------------------------------------------------------------------------------------
   3     0  E > > JMPZ                                                    <true>, ->3
   4     1    >   ECHO                                                    'Yay'
         2        ECHO                                                    '%0A'
   7     3    > > RETURN                                                  1
```

The opcodes sort of resemble the original source code, enough to follow along with the basic operations. (I'm not going to delve into the details of opcodes here, since that could be a book in itself.) No optimizations were applied at the opcode level in the above script - but as we can see, the compilation phase has made some by resolving the constant condition (`PHP_VERSION === '7.1.0-dev'`) to `true`.

OPcache does more than simply caching opcodes (thus bypassing the lexing, parsing, and compilation stages). It also packs with it many different levels of optimizations. Let's turn up the optimization level to four passes to see what comes out:

Command:

```
 php -dopcache.enable_cli=1 -dopcache.optimization_level=1111
↪ -dvld.active=-1 -dvld.execute=0 file.php
```

Output:

```
line     #* E I O op                           fetch          ext  return  operands
-------------------------------------------------------------------------------------
   4     0  E >   ECHO                                                    'Yay%0A'
   7     1      > RETURN                                                  1
```

We can see that the constant condition has been removed, and the two `ECHO` instructions have been compacted into a single instruction. These are just a taste of the many optimizations OPcache applies when performing passes over the opcodes of a script. I won't go through the various optimization levels here, though.

Stage 4 - Interpretation

The final stage is the interpretation of the opcodes. This is where the opcodes are run on the Zend Engine (ZE) VM. There's actually very little to say about this stage (from a high-level perspective, at least). The output is pretty much whatever your PHP script outputs via commands such as `echo`, `print`, `var_dump`, and so on.

So instead of digging into anything complex at this stage, here's a fun fact: PHP requires itself as a dependency when generating its own VM. This is because the VM is generated by a PHP script, due to it being simpler to write and easier to maintain.

Conclusion

We've taken a brief look through the four stages that the PHP interpreter goes through when running PHP code. This has involved using various extensions (including tokenizer, php-ast, OPcache, and VLD) to manipulate and view the output of each stage.

I hope this piece has helped to provide you with a better holistic understanding of PHP's interpreter, as well as shown the importance of the OPcache extension (for both its caching and optimization abilities).

Chapter 2

Getting to Know and Love Xdebug

by Bruno Škvorc

It's been 15 years since Xdebug first came out. We think this is the perfect opportunity to re-introduce it to the world, and explain how and why it does what it does.

Xdebug is a PHP extension (meaning it needs to be compiled and installed into a PHP installation) which provides the developer with some features for debugging. They include:

- stack traces - detailed output of the path the application took to reach a given error, including parameters passed to functions, in order to easily track the error down.

- a prettier `var_dump` output which produces color coded information and structured views, similar to VarDumper, along with a a super-globals dumper
- a profiler for finding out where the bottlenecks in your code are, and the ability to visualize those performance graphs in external tools. What this results in is a graph similar to that which Blackfire produces.
- a remote debugger which can be used to remotely connect Xdebug with running code and an end-client like an IDE or a browser to step through breakpoints in code and execute line by line of your application.
- code coverage which tells you how much of your code was executed during a request. This is almost exclusively meant to help with unit tests and finding out how much of your code is test-covered.

How do I use it?

Xdebug comes with a detailed installation page which handles most if not all use cases, but if you'd like to play with the functionality presented below, we recommend using Homestead Improved which comes with the extension pre-installed and activated.

With modern IDEs and Blackfire, is there even a need for Xdebug?

IDEs do provide good code lookup functionality, so the link format functionality's usefulness can seem questionable. There's also loggers of all kinds now which can handle errors and exceptions. Likewise, function traces and profiling are done really well in Blackfire. However, file link formats are just one part of Xdebug, and using Blackfire has its own hurdles - installing the extension, setting up the keys, and then paying to keep trace history. Loggers also need to be used with a lot of foresight, and aren't very easy to add into an application later on.

There's more to Xdebug than just this, though - it's still required for proper unit testing (testing frameworks depend on it for code coverage reports), it's far from easy to get remote break-point debugging going via other means, and it's a tool so old and stable it's been ironed out to near perfection.

If your current tools can handle everything it offers or you don't need the features it offers then of course, there's no need for Xdebug, but I've yet to start a single project that could be completed just as efficiently without it.

Let's Try It Out

I'll assume you have a working Xdebug installation at this point. If not, please consider using Homestead Improved.

Let's make a new project folder with a simple `index.php` file, and echo out a non-existent variable like `$foo`:

```
<?php

echo $foo;
```

This is what we get:

(!) Notice: Undefined variable: foo in /home/vagrant/Code/xdebug2/index.php on line *3*

Call Stack

#	Time	Memory	Function	Location
1	0.0006	357664	{main}( )	.../index.php:0

Turning Xdebug Off

Screens like these are so ubiquitous these days, and such a common default, that most people don't even realize this is already Xdebug-styled. To prove it, let's see how it looks without Xdebug. To disable Xdebug, we edit the file `/etc/php/7.1/fpm/conf.d/20-xdebug.ini` in Homestead Improved, and comment out the first line:

```
;zend_extension=xdebug.so
xdebug.remote_enable = 1
xdebug.remote_connect_back = 1
xdebug.remote_port = 9000
```

```
xdebug.max_nesting_level = 512
```

We need to restart PHP-FPM afterwards:

```
sudo service php7.1-fpm restart
```

Location of ini File

if you're using another development environment with a different PHP installation, your Xdebug ini file might be elsewhere. Consult your system's documentation for the exact location.

Notice: Undefined variable: foo in **/home/vagrant/Code/xdebug2/index.php** on line **3**

Looks quite barren, doesn't it? It's missing the whole call stack. Granted, this information isn't particularly useful at this point since we're only dealing with a single line in a single file, but we'll look at a heavier use later on.

Reactivate Xdebug now by removing the comment in the previously edited file, and let's continue. Don't forget to restart PHP!

File Clickthroughs

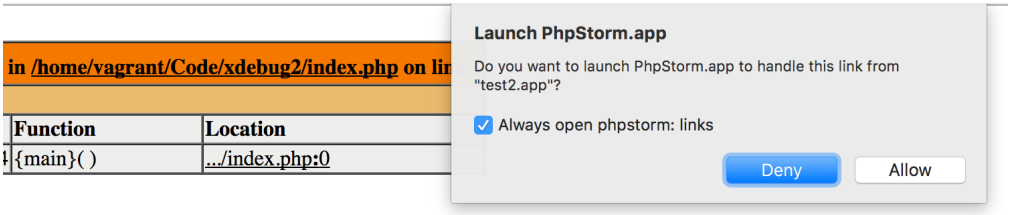
If you're a developer who's fixed on an IDE (like I am on PhpStorm), it would definitely be useful to be able to click on files in the stack trace and go directly to them in the IDE. A non-trivial upgrade in debugging speed, for sure. I'll demonstrate the implementation of this feature for PhpStorm.

First, let's open the `20-xdebug.ini` file we edited previously, and add the following to it:

```
xdebug.file_link_format = phpstorm://open?%f:%l
```

Note that this will work in some browsers, and won't in others. For example, Opera has problems with `phpstorm://` links and will gladly crash, while Firefox and Chrome work just fine.

If we refresh our invalid PHP page now, we'll get clickable links which open the IDE at the precise location of the error:



The process is the same for other IDEs and editors.

Xdebug with Vagrant and PhpStorm

Why stop at this, though? Many people today develop on virtual machines, making sure no part of the PHP runtime ever touches their main machine, keeping everything fast and smooth. How does Xdebug behave in those cases? Additionally, is it even possible to do break-point debugging where you step through your code and inspect each line separately when using such complex environments?

Luckily, Xdebug supports break-point-powered remote connections perfectly. We've covered the process before, so for a full gif-powered setup tutorial, please follow this guide.

Using the Profiler

As a final quick tip, let's inspect one of the often neglected features: the profiler. For that, we'll need a heavy application like Laravel.

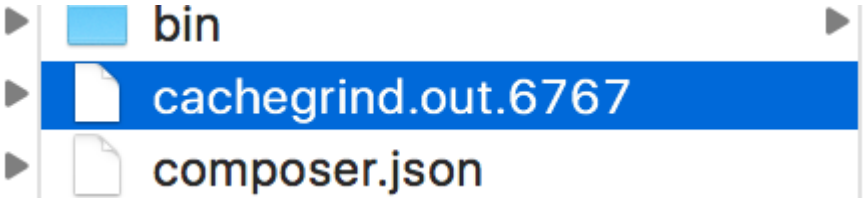
```
composer create-project --prefer-dist laravel/laravel xdebug
```

Once again, we need to edit the `20-xdebug.ini` file, and add the following:

```
xdebug.profiler_enable_trigger = 1
xdebug.profiler_output_dir = /home/vagrant/Code/
```

Note that we're not using `xdebug.profiler_enable = 1` because we don't want it to stay on 100% of the time. Instead, we'll use the trigger query param "XDEBUG_PROFILE" to selectively activate it. We're also outputting the cachegrind profile into the main shared folder of our VM so that we can inspect it with tools on the host operating system.

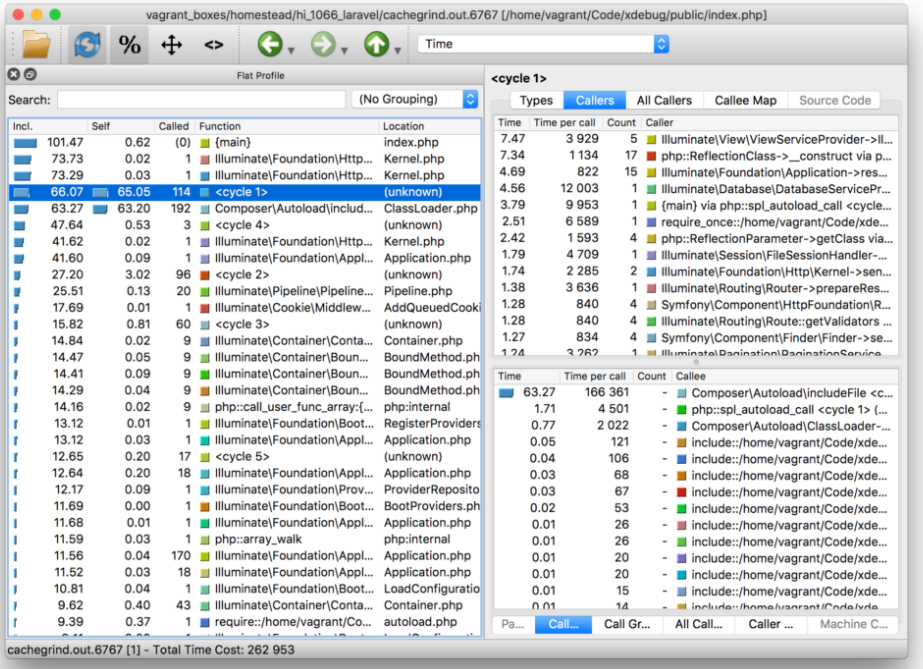
After restarting PHP, we can try it out by executing `homestead.app/?XDEBUG_PROFILE` (replace homestead.app with whichever vhost you picked, or the VM's IP). Sure enough, the file is there:



Every OS will have its own cachegrind inspector tool, and on OS X one of those is qcachegrind, easily installed via Homebrew. Refer to your OS's preferred visualizer for installation instructions. After installing it...

```
brew install qcachegrind --with-graphviz
```

... and opening the file in the viewer, we can see a nice breakdown of the execution flow:



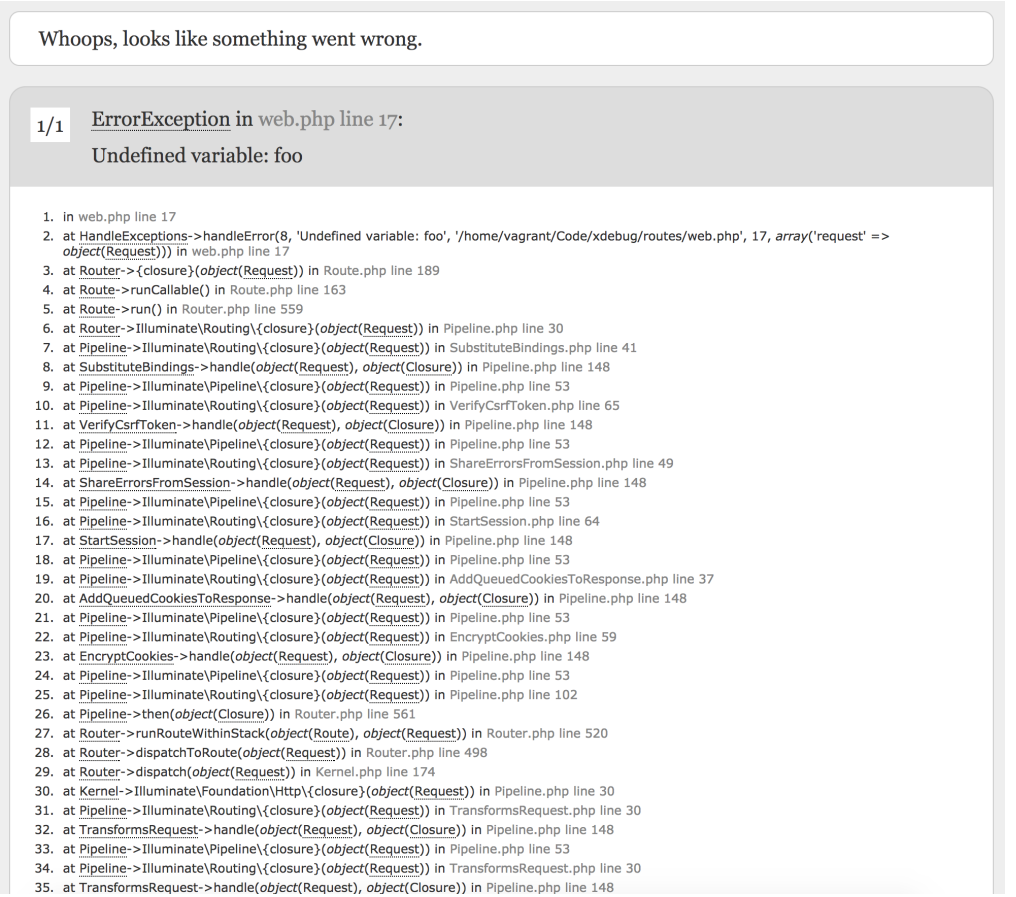
The profiler offers and immeasurable wealth of data and truly deep insight into the way your code behaves, just like Blackfire. With the profiler's local output, however, it's easier than ever to automate the continuous tracking of performance and execution complexity.

Forcing Xdebug's Render on Laravel

By default, Laravel has custom error reports and rendering set up. An error like the one we caused before with an undefined variable would, in Laravel, look like this:

```
<?php

use Illuminate\Http\Request;

Route::get('/', function(Request $request){
```

```
    echo $foo;
    return view('welcome');
});
```

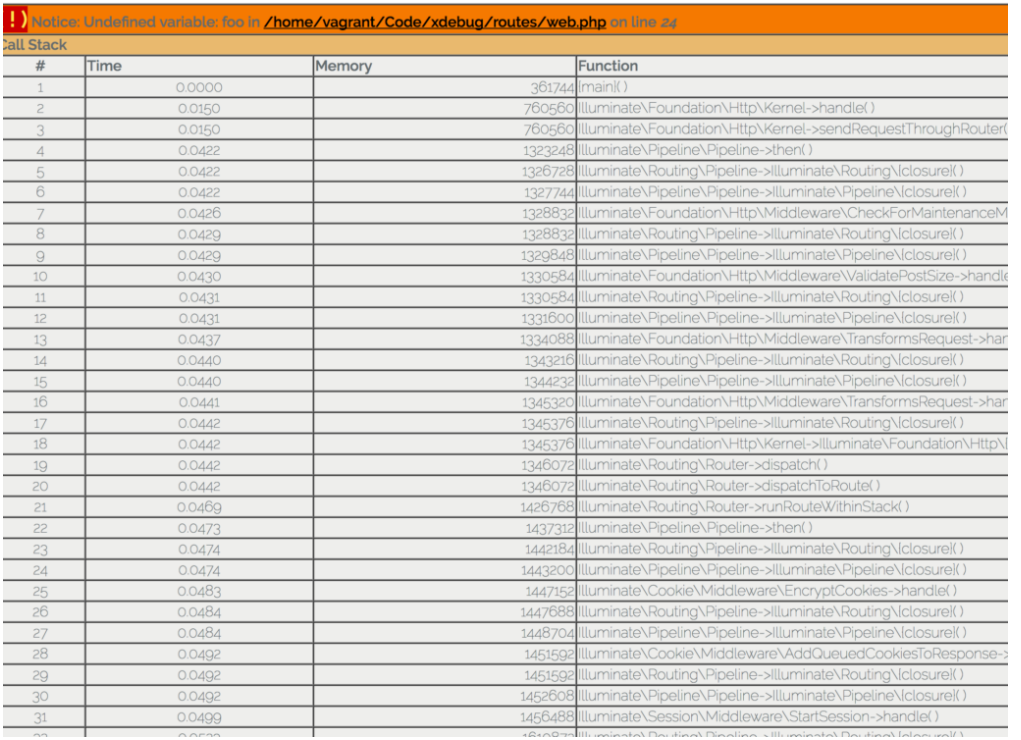


While Symfony's error screen (which is what Laravel is using here) is configured to also play nice with Xdebug's clickthrough (try it, you can now click on these files and their lines, too!), I really miss the memory output (Xdebug by default also outputs the memory usage at every point in time in the stacktrace). Let's revert this to Xdebug's screen while in development mode, so we can inspect that attribute.

```
<?php
use Illuminate\Http\Request;

Route::get('/', function(Request $request){
    ini_set('display_errors', 1);
    restore_error_handler();
    echo $foo;
    return view('welcome');
});
```

You can see here we updated our default route so that it first activates the displaying of errors (the screen we saw earlier is not a shown error per-se, but a caught exception the stack trace of which was manually extracted and rendered), and then we restore the error handler to its default value, overriding Laravel's.

After refreshing, sure enough, our old screen is back - just look at that stack trace tower and memory consumption!


Notice: Undefined variable: foo in /home/vagrant/Code/xdebug/routes/web.php on line 24

Call Stack

#	Time	Memory	Function
1	0.0000	361744	{main}( )
2	0.0150	760560	Illuminate\Foundation\Http\Kernel->handle( )
3	0.0150	760560	Illuminate\Foundation\Http\Kernel->sendRequestThroughRouter(
4	0.0422	1323248	Illuminate\Pipeline\Pipeline->then( )
5	0.0422	1326728	Illuminate\Routing\Pipeline->Illuminate\Routing\{closure}( )
6	0.0422	1327744	Illuminate\Pipeline\Pipeline->Illuminate\Pipeline\{closure}( )
7	0.0426	1328832	Illuminate\Foundation\Http\Middleware\CheckForMaintenanceM
8	0.0429	1328832	Illuminate\Routing\Pipeline->Illuminate\Routing\{closure}( )
9	0.0429	1329848	Illuminate\Pipeline\Pipeline->Illuminate\Pipeline\{closure}( )
10	0.0430	1330584	Illuminate\Foundation\Http\Middleware\ValidatePostSize->handl
11	0.0431	1330584	Illuminate\Routing\Pipeline->Illuminate\Routing\{closure}( )
12	0.0431	1331600	Illuminate\Pipeline\Pipeline->Illuminate\Pipeline\{closure}( )
13	0.0437	1334088	Illuminate\Foundation\Http\Middleware\TransformsRequest->har
14	0.0440	1343216	Illuminate\Routing\Pipeline->Illuminate\Routing\{closure}( )
15	0.0440	1344232	Illuminate\Pipeline\Pipeline->Illuminate\Pipeline\{closure}( )
16	0.0441	1345320	Illuminate\Foundation\Http\Middleware\TransformsRequest->har
17	0.0442	1345376	Illuminate\Routing\Pipeline->Illuminate\Routing\{closure}( )
18	0.0442	1345376	Illuminate\Foundation\Http\Kernel->Illuminate\Foundation\Http\
19	0.0442	1346072	Illuminate\Routing\Router->dispatch( )
20	0.0442	1346072	Illuminate\Routing\Router->dispatchToRoute( )
21	0.0469	1426768	Illuminate\Routing\Router->runRouteWithinStack( )
22	0.0473	1437312	Illuminate\Pipeline\Pipeline->then( )
23	0.0474	1442184	Illuminate\Routing\Pipeline->Illuminate\Routing\{closure}( )
24	0.0474	1443200	Illuminate\Pipeline\Pipeline->Illuminate\Pipeline\{closure}( )
25	0.0483	1447152	Illuminate\Cookie\Middleware\EncryptCookies->handle( )
26	0.0484	1447688	Illuminate\Routing\Pipeline->Illuminate\Routing\{closure}( )
27	0.0484	1448704	Illuminate\Pipeline\Pipeline->Illuminate\Pipeline\{closure}( )
28	0.0492	1451592	Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse->
29	0.0492	1451592	Illuminate\Routing\Pipeline->Illuminate\Routing\{closure}( )
30	0.0492	1452608	Illuminate\Pipeline\Pipeline->Illuminate\Pipeline\{closure}( )
31	0.0499	1456488	Illuminate\Session\Middleware\StartSession->handle( )

I encourage you to investigate further on your own - look around in the docs, play with the options, see what you can find out about your applications.

Conclusion

Xdebug is a valuable tool for any developer's toolbelt. It's a powerful extension that fully lives up to the word, extending the language we work in daily to be more verbose, more user friendly, and less mysterious when errors appear.

With 15 whole years behind it, Xdebug has set a high standard for debugging tools. I'd like to thank Derick for developing and maintaining it all this time, and I'd love it if you chose to write a tutorial or two about in-depth usage, caveats, or secret feature combinations no one's thought of before. Let's spread the word and help it thrive for another 15 years.

Happy birthday, Xdebug!

Chapter 3

Localization Demystified: Php-Intl for Everyone

by Younes Rafie

Most applications perform locale aware operations like working with texts, dates, timezones, etc. The PHP Intl extension provides a good API for accessing the widely known ICU library's functions.

Installation

The extension is installed by default on PHP 5.3 and above. You can look for it by running the following command:

```
php -m | grep 'intl'
```

If the extension is not present, you can install it manually by following the installation guide. If you're using Ubuntu, you can directly run the following commands.

```
sudo apt-get update
sudo apt-get install php5-intl
```

If you're using PHP7 on your machine, you need to add the (`ppa:ondrej/php`) PPA, update your system and install the Intl extension.

```
# Add PPA
sudo add-apt-repository ppa:ondrej/php-7.0
# Update repository index
sudo apt-get update
# install extension
sudo apt-get install php7.0-intl
```

Message Formatting

Most modern applications are built with localization in mind. Sometimes, the message is a plain string with variable placeholders, other times it's a complex pluralized string.

Simple Messages

We're going to start with a simple message containing a placeholder. Placeholders are patterns enclosed in curly braces. Here is an example:

```
var_dump(
    MessageFormatter::formatMessage(
        "en_US",
```

```
        "I have {0, number, integer} apples.",
        [ 3 ]
    )
);
```

```
// output

string(16) "I have 3 apples."
```

The arguments passed to the `MessageFormatter::formatMessage` method are:

- The message locale.
- String message.
- Placeholder data.

The `{0, number, integer}` placeholder will inject the first item of the data array as a `number - integer` (see the table below for the list of options). We can also use named arguments for placeholders. The example below will output the same result.

```
var_dump(
    MessageFormatter::formatMessage(
        "en_US",
        "I have {number_apples, number, integer} apples.",
        [ 'number_apples' => 3 ]
    )
);
```

Different languages have different numeral systems, like Arabic, indian, etc.

Western Arabic	0	1	2	3	4	5	6	7	8	9
Eastern Arabic	٠	١	٢	٣	٤	٥	٦	٧	٨	٩
Perso-Arabic variant	٠	١	٢	٣	۴	۵	۶	٧	٨	٩
Urdu variant	۰	۱	۲	۳	۴	۵	۶	۷	۸	۹

The previous example is targeting the `en_US` locale. Let's change it to `ar` to see the difference.

```
var_dump(
    MessageFormatter::formatMessage(
        "ar",
        "I have {number_apples, number, integer} apples.",
        [ 'number_apples' => 3 ]
    )
);
```

```
string(17) "I have ٣ apples."
```

We can also change it to Bengali locale (`bn`).

```
var_dump(
    MessageFormatter::formatMessage(
        "bn",
        "I have {number_apples, number, integer} apples.",
        [ 'number_apples' => 3 ]
    )
);
```

```
string(18) "I have ৩ apples."
```

So far, we've only worked with numbers. Let's take a look at other types that we can use.

```
$time = time();
var_dump( MessageFormatter::formatMessage(
    "en_US",
    "Today is {0, date, full} - {0, time}",
    array( $time )
) );
```

```
string(47) "Today is Wednesday, April 6, 2016 - 11:21:47 PM"
```

```
var_dump( MessageFormatter::formatMessage(
    "en_US",
    "duration: {0, duration}",
    array( $time )
) );
```

```
string(23) "duration: 405,551:27:58"
```

We can also spell out the passed numbers.

```
var_dump( MessageFormatter::formatMessage(
    "en_US",
    "I have {0, spellout} apples",
    array( 34 )
) );
```

```
string(25) "I have thirty-four apples"
```

It also works on different locales. Here is an example using the Arabic language.

```
var_dump( MessageFormatter::formatMessage(
    "ar",
    "{0} لدي, spellout} تفاحة",
    array( 34 )
) );
```

```
string(44) "لدي أربعة و ثلاثون تفاحة"
```

argType	argStyle
number	integer, currency, percent
date	short, medium, long, full
time	short, medium, long, full
spellout	short, medium, long, full
ordinal	
duration	

Pluralization

An important part of localizing our application is to manage plural messages to make our UI as intuitive as possible. The apples example above will do the job. Here's how messages should look like in this case.

- (`number_apples` = 0): I have no apples.
- (`number_apples` = 1): I have one apple.
- (`number_apples` > 1): I have X apples.

```
var_dump( MessageFormatter::formatMessage(
    "en_US",
 'I have {number_apples, plural, =0{no apples} =1{one apple}
↪ other{# apples}}',
    array('number_apples' => 10)
) );
```

```
// number_apples = 0
string(16) "I have no apples"

// number_apples = 1
string(16) "I have one apple"

// number_apples = 10
string(16) "I have 10 apples"
```

The syntax is really straightforward, and most pluralization packages adopt this syntax. Check the documentation for more details.

```
 {data, plural, offsetValue =value{message}...
↪ other{message}}
```

- `data`: value index.
- `plural`: argType.
- `offsetValue`: the offset value is optional(`offset:value`). It subtracts the offset from the value.
- `=value{message}`: value to test for equality, and the message between curly braces. We can repeat this part multiple times (`=0{no apples} =1{one apple} =2{two apple}`).
- `other{message}`: The default case, like in a `switch - case` statement. The `#` character may be used the inject the `data` value.

Choices

In some cases, we need to print a different message for every range. The example below does this.

```
var_dump( MessageFormatter::formatMessage(
    "en_US",
    'The value of {0,number} is {0, choice,
                                        0 # between 0 and 19 |
                                        20 # between 20 and 39 |
                                        40 # between 40 and 59 |
                                        60 # between 60 and 79 |
                                        80 # between 80 and 100 |
                                        100 < more than 100 }',
    array(60)
) );
```

```
string(38) "The value of 60 is between 60 and 79 "
```

The `argType` in this case is set to `choice`, and this is the syntax format:

```
{value, choice, choiceStyle}
```

The official definition from the ICU documentation is:

```
 choiceStyle = number separator message ('|' number separator
↪ message)*

number = normal_number | ['-']  ∞ (U+221E, infinity)
normal_number = double value (unlocalized ASCII string)

separator = less_than | less_than_or_equal
less_than = '<'
less_than_or_equal = '#' |  ≤ (U+2264)
```

Note: ICU developers discourage the use of the choice type.

Select

Sometimes we need something like the select option UI component. Profile pages use this to update the UI messages according to the user's gender, etc. Here's an example:

```
var_dump( MessageFormatter::formatMessage(
    "en_US",
    "{gender, select, ".
      "female {She has some apples} ".
      "male {He has some apples.}".
      "other {It has some apples.}".
    "}",
    array('gender' => 'female')
) );
```

```
string(19) "She has some apples"
```

The pattern is defined as follows:

```
{value, select, selectStyle}

// selectStyle
selectValue {message} (selectValue {message})*
```

The `message` argument may contain other patterns like choice and plural. The next part will explain a *complex* example where we combine multiple patterns. Check the ICU documentation for more details.

Complex Cases

So far, we've seen some simple examples like pluralization, select, etc. Some cases are more complex than others. The ICU documentation has a very good example illustrating this. We'll insert part by part to make it simpler to grasp.

```
var_dump( MessageFormatter::formatMessage(
    "en_US",
    "{gender_of_host, select, ".
      "female {She has a party} ".
      "male {He has some apples.}".
      "other {He has some apples.}".
    "}",
 array('gender_of_host' => 'female', "num_guests" => 5,
↪ 'host' => "Hanae", 'guest' => 'Younes' )
) );
```

This is the same example we used before, but instead of using a simple message, we customize it depending on the `num_guests` value (talking about pluralization here).

```
var_dump( MessageFormatter::formatMessage(
    "en_US",
    "{gender_of_host, select, ".
      "female {".
        "{num_guests, plural, offset:1 ".
          "=0 {{host} does not have a party.}".
          "=1 {{host} invites {guest} to her party.}".
 "=2 {{host} invites {guest} and one other person to her
↪ party.}".
 "other {{host} invites {guest} and # other people to her
↪ party.}}}".
      "male {He has some apples.}".
      "other {He has some apples.}}",
 array('gender_of_host' => 'female', "num_guests" => 5,
↪ 'host' => "Hanae", 'guest' => 'Younes' )
) );
```

Notice that we're using the `offset:1` to remove one guest from the `num_guests` value.

```
 string(53) "Hanae invites Younes and 4 other people to her
↪ party."
```

Here's the full snippet of this example.

```
var_dump( MessageFormatter::formatMessage(
    "en_US",
    "{gender_of_host, select, ".
      "female {".
        "{num_guests, plural, offset:1 ".
          "=0 {{host} does not have a party.}".
          "=1 {{host} invites {guest} to her party.}".
 "=2 {{host} invites {guest} and one other person to her
↪ party.}".
 "other {{host} invites {guest} and # other people to her
↪ party.}}}".
      "male {".
        "{num_guests, plural, offset:1 ".
          "=0 {{host} does not have a party.}".
          "=1 {{host} invites {guest} to his party.}".
 "=2 {{host} invites {guest} and one other person to his
↪ party.}".
 "other {{host} invites {guest} and # other people to his
↪ party.}}}".
      "other {".
        "{num_guests, plural, offset:1 ".
          "=0 {{host} does not have a party.}".
          "=1 {{host} invites {guest} to their party.}".
 "=2 {{host} invites {guest} and one other person to their
↪ party.}".
 "other {{host} invites {guest} and # other people to their
↪ party.}}}}",
 array('gender_of_host' => 'female', "num_guests" => 5,
↪ 'host' => "Hanae", 'guest' => 'Younes' )
```

```
) );
```

Change the number of guests to test all message types.

```
// num_guests = 2
 string(55) "Hanae invites Younes and one other person to her
↪ party."

// num_guests = 1
string(34) "Hanae invites Younes to her party."

// num_guests = 0
string(28) "Hanae does not have a party."
```

Message Parsing

There's not much to say about parsing messages; we use the pattern we used for formatting to extract data from an output message.

```
 $messageFormater = new MessageFormatter("en_US", 'I have {0,
↪ number}');
var_dump( $messageFormater->parse("I have 10 apples") );
```

```
array(1) {
  [0]=>
  int(10)
}
```

Check the documentation for more details about message parsing.

Chapter 4

Event Sourcing in a Pinch

by Christopher Pitt

Let's talk about Event Sourcing. Perhaps you've heard of it, but haven't found the time to attend a conference talk or read one of the older, larger books which describe it. It's one of those topics I wish I'd known about sooner, and today I'm going to describe it to you in a way that I understand it.

Most of this code can be found on GitHub. I've tested it using PHP `7.1`.

I've chosen this title for a few reasons. Firstly, I don't consider myself an expert on the topic. For that, you'd be hard pressed to find a better tutor than the authors of those books, or someone like Mathias Verraes. What I'm about to tell you is only the tip of the iceberg. A pinch of salt, if you will.

Event sourcing is also part of a larger, broader set of topics; loosely defined as Domain Driven Design. Event sourcing is one design pattern amongst many, and you'd do well to learn about the other patterns associated with DDD. In fact, it's often not a good idea to pluck just Event Sourcing out of the DDD toolbox, without understanding the benefits of the other patterns.

Still, I think it's a fascinating and fun exercise, and few people cover it well. It's especially suited for those developers who have yet to dip their toes in the pool of DDD. So, if you find yourself needing something like Event Sourcing, but don't know or understand the rest of DDD, I hope this post helps you. In a pinch.

Common Language

One of the strongest themes of Domain Driven Design is the need for a common language. When your client decides they need a new application, they are thinking about how it will affect their ice-cream sales. They're concerned about how their patrons will find their favorite flavor of ice-cream, and how that will affect foot-traffic at their ice-cream stand.

You may think in terms of website users and geolocated outlets, but those words don't necessarily mean anything to your client. Though it may take some time, initially, your communication with your client will be greatly improved if you both use the same words when talking about the same thing.

A Safety Net

You'll also find that modeling the entire system in the words your client understands gives you a bit of a safety net against scope changes. It's much easier to say; "You initially asked for customers to purchase ice-cream before the invoice is sent (shown here in code and email), but now you're asking for the invoice to be sent first..." than it is to describe the changes they're asking for in language/code only you understand.

That's not to say all your code needs to be understood by the client, or that you have to use something like Behat for your integration testing. But, at the very least, you should call entities and actions the same thing as your client does.

An added benefit of this is that future developers will be able to understand the intent of the code (and how it applies to the business process), without as much help from the client or project manager.

I'm waffling a bit, but this point will be important when we start to write code.

Storing State vs. Storing Behavior

Most of the websites I've built have had some form of CRUD (Create, Read, Update, and Delete) database functionality. These operations are intentionally generic, as they have traditionally mapped to the underlying relational database they use.

Storing State

We may even be used to using something like Eloquent:

```
$product = new Product();
$product->title = "Chocolate";
$product->cents_per_serving = 499;
$product->save();

$outlet = new Outlet();
$outlet->location = "Pismo Beach";
$outlet->save();

$outlet->products()->sync([
    $product->id => [
        "servings_in_stock" => 24,
    ],
])
```

This is enough for the most basic presentation of ice-cream information on the client's website. It's how we've been building websites for ages. But it has a significant weakness — we don't know what happened to get us here.

Let's think of some things which could influence how the data got to this point:

- When did we start selling "Chocolate"? Many Object Relation Mappers (ORM) libraries will add fields like `created_at` and `updated_at`, but those only go so far in telling us what we want to know.
- How did we get that much stock? Did we get a delivery? Did we give some away?
- What happens to our analytics when we no longer want to sell "Chocolate", or when we want to move all stock to another outlet? Do we add a boolean field (to the products' table), to indicate that the product is no longer sold, but should remain in the analytics? Or perhaps we should add a timestamp, so we know when that all happened...

Storing Behavior

The weakness is such that we only know what the data is like now. Our data is like a photo, when what we want is a video. What if we tried something different?

```
$events = [];

$events[] = new ProductInvented("Chocolate");
$events[] = new ProductPriced("Chocolate", 499);
$events[] = new OutletOpened("Pismo Beach");
 $events[] = new OutletStocked("Pismo Beach", 24,
↪ "Chocolate");

store($events);
```

This is storing the same eventual information, but each of the steps is self-contained. They describe the behavior of the customers, outlets, stock etc.

Using this approach, we have much better control over the timeline of events which have lead to the current state. We could add events for stock giveaways, or product discontinuation:

```
$events = [];
```

```
$events[] = new OutletStockGivenAway(
    "Pismo Beach", 2, "Chocolate"
);

$events[] = new OutletDiscontinuedProduct(
    "Pismo Beach", "Chocolate"
);

store($events);
```

This isn't more complex than storing state, but it is far more descriptive of the events that happen. It's also really easy for the client to understand what's going on.

When we start to store behavior (instead of the state at one point in time), we gain the ability to easily step through the events. Almost like we're traveling through time:

```
$lastWeek = Product::at("Chocolate", date("-1 WEEK"));
$yesterday = Product::at("Chocolate", date("-1 DAY"));

printf(
    "Chocolate increased, from %s to %s, in one week",
    $lastWeek->cents_per_serving,
    $yesterday->cents_per_serving
);
```

... and we could do that without any extra boolean/timestamp fields. We could come back to already-stored data, and create a new kind of report. That's so valuable!

So Which Is It?

Event Sourcing is both of these things. It's about capturing every event (which you can think of as every change in application data) as a self-contained,

repeatable thing. It's about storing these events in the same time-order they happened, so that we can at-will journey to any point in time.

It's about understanding how to interface this architecture with other systems that aren't built in the same way, which means having a way to represent just the latest application data state.

The events are append-only, which means we never delete any of them from the database. And, if we're doing things right, they describe (in their names and properties) what they mean to the business and customer they relate to.

Making Events

We're going to use classes to describe events. They're useful, simple containers we can define; and they'll help us validate the data we put in and the data we get out for each event.

Avoiding Jargon

Those experienced in Event Sourcing may be itching to hear how I describe things like aggregates. I'm intentionally avoiding jargon — in much the same way as I'd avoid differentiating between mocks, doubles, stubs, and fakes — if I were teaching someone their first bit of testing. It's the **idea** that is important, and **the idea behind Event Sourcing is recording behavior.**

Here's the abstract event that we can use to model real events:

```
abstract class Event
{
    /**
     * @var DateTimeImmutable
     */
    private $date;

    protected function __construct()
    {
        $this->date = date("Y-m-d H:i:s");
    }
```

```
    public function date(): string
    {
        return $this->date;
    }

    abstract public function payload(): array;
}
```

This is from `events.php`

It's really important (in my opinion) that event classes are simple. Using PHP 7 type hints, we can validate the data we use to define events. A handful of simple accessors will help us get the important data out again.

On top of this class, we can define the real event types we want to record:

```
final class ProductInvented extends Event
{
    /**
     * @var string
     */
    private $name;

    public function __construct(string $name)
    {
        parent::__construct();

        $this->name = $name;
    }

    public function payload(): array
    {
        return [
            "name" => $this->name,
            "date" => $this->date(),
        ];
    }
```

```
}
```

```
final class ProductPriced extends Event
{
    /**
     * @var string
     */
    private $product;

    /**
     * @var int
     */
    private $cents;

    public function __construct(string $product, int $cents)
    {
        parent::__construct();

        $this->product = $product;
        $this->cents = $cents;
    }

    public function payload(): array
    {
        return [
            "product" => $this->product,
            "cents" => $this->cents,
            "date" => $this->date(),
        ];
    }
}
```

```
final class OutletOpened extends Event
{
    /**
     * @var string
     */
```

```
    private $name;

    public function __construct(string $name)
    {
        parent::__construct();

        $this->name = $name;
    }

    public function payload(): array
    {
        return [
            "name" => $this->name,
            "date" => $this->date(),
        ];
    }
}
```

```
final class OutletStocked extends Event
{
    /**
     * @var string
     */
    private $outlet;

    /**
     * @var int
     */
    private $servings;

    /**
     * @var string
     */
    private $product;

    public function __construct(string $outlet, ↩
        int $servings, string $product)
    {
```

```
        parent::__construct();

        $this->outlet = $outlet;
        $this->servings = $servings;
        $this->product = $product;
    }

    public function payload(): array
    {
        return [
            "outlet" => $this->outlet,
            "servings" => $this->servings,
            "product" => $this->product,
            "date" => $this->date(),
        ];
    }
}
```

Notice how we've made each of these `final`? We have to fight to keep the events simple, and they wouldn't continue to be simple if another developer could come along and subclass them (for whatever reason).

I also find it interesting how we can isolate the definition, format, and accessibility of the event dates: by defining `$date` as private and requiring subclasses to access it through the `date` method. This is perhaps a tad too defensive, but it obeys the Law of Demeter in that the concrete events need not know how the date is defined or formatted, in order to use it.

With this isolation, we can change the entire system's timezone, or change to using UNIX timestamps, and we'd only need to change a single line of code.

Omitting These Classes

We could omit these classes if we're willing to sacrifice performance (and do runtime associative array checks) or type safety.

Storing Events

Let's store these events in a SQLite database. We could use an ORM for that, but perhaps this is a good opportunity to recap how PDO works.

Using PDO

The first bit of code, for connecting to any supported database through PDO, is:

```
$connection = new PDO("sqlite::memory:");

$connection->setAttribute(
    PDO::ATTR_ERRMODE,
    PDO::ERRMODE_EXCEPTION
);
```

This is from `sqlite-pdo.php`

PDO connections are typically made using a Data Source Name (DSN). Here we define the database type as `sqlite`, and the location as an in-memory database. This means the database will disappear as soon as the script finishes.

It's also a good idea to set the error-mode to throw exceptions when a SQL error occurs. That way we'll get immediate feedback on our mistakes.

SQL Ahead

If you've not done any raw SQL queries before, this next bit may be confusing. Check out this great introduction to SQL.

Next, we should create some tables to work from:

```
$statement = $connection->prepare("
    CREATE TABLE IF NOT EXISTS product (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT
```

```
    )
");

$statement->execute();
```

This is from `sqlite-pdo.php`

One of those tables is going to be where we generate and store unique product identifiers. The exact syntax of `CREATE TABLE` differs slightly between database types, and you'd typically find more columns in a table.

A great way to learn how your database creates tables is to make a table through a GUI, and then run `SHOW CREATE TABLE my_new_table`. This will generate `CREATE TABLE` syntax, in all of PDO's supported databases.

Prepared statements (using `prepare` and `execute`) are the recommended way of executing SQL queries. They are even more useful when you need to pass query parameters:

```
$statement = $connection->prepare(
    "INSERT INTO product (name) VALUES (:name)"
);

$statement->bindValue("name", "Chocolate");
$statement->execute();
```

This is from `sqlite-pdo.php`

Bound values are automatically quoted and escaped, avoiding the most common kinds of SQL injection. We can also use prepared statements to return rows:

```
$row = $connection
    ->prepare("SELECT * FROM product")
    ->execute()->fetch(PDO::FETCH_ASSOC);

$rows = $connection
```

```
    ->prepare("SELECT * FROM product")
    ->execute()->fetchAll(PDO::FETCH_ASSOC);
```

This is from `sqlite-pdo.php`

These `fetch` and `fetchAll` methods will return arrays and arrays or arrays, respectively, given that we're using the `PDO::FETCH_ASSOC` type.

Adding Helper Functions

As you can probably guess, using PDO directly can lead to a lot of needless repetition. I've found it useful to create a few helper functions:

```
function connect(string $dsn): PDO
{
    $connection = new PDO($dsn);

    $connection->setAttribute(
        PDO::ATTR_ERRMODE,
        PDO::ERRMODE_EXCEPTION
    );

    return $connection;
}

function execute(PDO $connection, string $query, ↩
    array $bindings = []): array
{
    $statement = $connection->prepare($query);

    foreach ($bindings as $key => $value) {
        $statement->bindValue($key, $value);
    }

    $result = $statement->execute();

    return [$statement, $result];
}
```

```
function rows(PDO $connection, string $query, ↩
    array $bindings = []): array
{
    $executed = execute($connection, $query, $bindings);

    /** @var PDOStatement $statement */
    $statement = $executed[0];

    return $statement->fetchAll(PDO::FETCH_ASSOC);
}

function row(PDO $connection, string $query, ↩
    array $bindings = []): array
{
    $executed = execute($connection, $query, $bindings);

    /** @var PDOStatement $statement */
    $statement = $executed[0];

    return $statement->fetch(PDO::FETCH_ASSOC);
}
```

This is from `sqlite-pdo-helpers.php`

This are much the same code as we saw before. They're a little nicer to use than the direct PDO code though:

```
$connection = connect("sqlite::memory:");

execute(
    $connection,
    "CREATE TABLE IF NOT EXISTS product ↩
        (id INTEGER PRIMARY KEY AUTOINCREMENT,name TEXT)"
);

execute(
    $connection,
```

```
    "INSERT INTO product (name) VALUES (:name)",
    ["name" => "Chocolate"]
);

$rows = rows(
    $connection,
    "SELECT * FROM product"
);

$row = row(
    $connection,
    "SELECT * FROM product WHERE name = :name",
    ["name" => "Chocolate"]
);
```

This is from `sqlite-pdo-helpers.php`

You may not like the idea of defining global functions for these things. They're like something you'd see in the dark ages of PHP. But they're so concise, and easy to use!

They're not even difficult to test:

```
$fake = new class("sqlite::memory:") extends PDO
{
    private $valid = true;

    function prepare($statement, $options = null) {
        if ($statement !== "SELECT * FROM product") {
            $this->valid = false;
        }

        return $this;
    }

    function execute() {
        return;
    }
```

```
    function fetchAll() {
        if (!$this->valid) {
            throw new Exception();
        }

        return [];
    }
};

assert(connect("sqlite::memory:") instanceof PDO);
assert(is_array(rows($fake, "SELECT * FROM product")));
```

This is from `sqlite-pdo-helpers.php`

We're not testing all the variations, of the helper functions, but you get the idea...

If you're still confused, for a more in-depth look at PDO, see this post.

Storing Events

Let's take another look at the events we want to store:

```
$events = [];

$events[] = new ProductInvented("Chocolate");
$events[] = new ProductPriced("Chocolate", 499);
$events[] = new OutletOpened("Pismo Beach");
 $events[] = new OutletStocked("Pismo Beach", 24,
↪ "Chocolate");
```

The simplest approach would be to create a database table for each of these event types:

```
execute($connection, "
    CREATE TABLE IF NOT EXISTS product (
```

```
        id INTEGER PRIMARY KEY AUTOINCREMENT
    )
");

execute($connection, "
    CREATE TABLE IF NOT EXISTS event_product_invented (
        id INT,
        name TEXT,
        date TEXT
    )
");

execute($connection, "
    CREATE TABLE IF NOT EXISTS event_product_priced (
        product INT,
        cents INT,
        date TEXT
    )
");

execute($connection, "
    CREATE TABLE IF NOT EXISTS outlet (
        id INTEGER PRIMARY KEY AUTOINCREMENT
    )
");

execute($connection, "
    CREATE TABLE IF NOT EXISTS event_outlet_opened (
        id INT,
        name TEXT,
        date TEXT
    )
");

execute($connection, "
    CREATE TABLE IF NOT EXISTS event_outlet_stocked (
        outlet INT,
        servings INT,
        product INT,
        date TEXT
```

```
    )
");
```

This is from `storing-events.php`

In addition to a table for each event, I've also added tables to store and generate product and outlet IDs. Each event table has a date field, the value of which is generated by the abstract `Event` class.

The real magic happens in the `store` and `storeOne` functions:

```
function store(PDO $connection, array $events)
{
    foreach($events as $event) {
        storeOne($connection, $event);
    }
}

function storeOne(PDO $connection, Event $event)
{
    $payload = $event->payload();

    if ($event instanceof ProductInvented) {
        inventProduct(
            $connection,
            newProductId($connection),
            $payload["name"],
            $payload["date"]
        );
    }

    if ($event instanceof ProductPriced) {
        priceProduct(
            $connection,
            productIdFromName($connection, $payload["name"]),
            $payload["cents"],
            $payload["date"]
        );
```

```
    }

    if ($event instanceof OutletOpened) {
        openOutlet(
            $connection,
            newOutletId($connection),
            $payload["name"],
            $payload["date"]
        );
    }

    if ($event instanceof OutletStocked) {
        stockOutlet(
            $connection,
            outletIdFromName(
                $connection, $payload["outlet_id"]
            ),
            $payload["servings"],
            productIdFromName(
                $connection, $payload["product_id"]
            ),
            $payload["date"]
        );
    }
}
```

This is from `storing-events.php`

The `store` function is just a convenience. PHP has no concept of typed arrays, so we could add runtime checking, or use the signature of `storeOne` to validate that we're only trying to store `Event` subclass instances.

We can get specific event data via the `payload` method. This data will differ based on the event class being stored, so we should only assume keys after we're sure which event type we're dealing with.

We're also using some product and outlet helper methods. Here's what they look like:

```
function newProductId(PDO $connection): int
{
    execute(
        $connection,
        "INSERT INTO product VALUES (null)"
    );

    return $connection->lastInsertId();
}

function inventProduct(PDO $connection, int $id, ↩
    string $name, string $date)
{
    execute(
        $connection,
        "INSERT INTO event_product_invented ↩
            (id, name, date) VALUES (:id, :name, :date)",
        ["id" => $id, "name" => $name, "date" => $date]
    );
}

 function productIdFromName(PDO $connection, string $name):
↪ int
{
    $row = row(
        $connection,
        "SELECT * FROM event_product_invented ↩
            WHERE name = :name",
        ["name" => $name]
    );

    if (!$row) {
        throw new InvalidArgumentException("Product not found");
    }

    return $row["id"];
}

function priceProduct(PDO $connection, int $product, ↩
    int $cents, string $date)
```

```
{
    execute(
        $connection,
        "INSERT INTO event_product_priced ↩
            (product, cents, date) VALUES ↩
            (:product, :cents, :date)",
        ["product" => $product, "cents" => $cents, ↩
            "date" => $date]
    );
}

function newOutletId(PDO $connection): int
{
    execute(
        $connection,
        "INSERT INTO outlet VALUES (null)"
    );

    return $connection->lastInsertId();
}

function openOutlet(PDO $connection, int $id, ↩
    string $name, string $date)
{
    execute(
        $connection,
        "INSERT INTO event_outlet_opened (id, name, date) ↩
            VALUES (:id, :name, :date)",
        ["id" => $id, "name" => $name, "date" => $date]
    );
}

 function outletIdFromName(PDO $connection, string $name):
↪ int
{
    $row = row(
        $connection,
        "SELECT * FROM event_outlet_opened ↩
            WHERE name = :name",
        ["name" => $name]
```

```
    );

    if (!$row) {
        throw new InvalidArgumentException("Outlet not found");
    }

    return $row["id"];
}

function stockOutlet(PDO $connection, int $outlet, ↩
    int $servings, int $product, string $date)
{
    execute(
        $connection,
        "INSERT INTO event_outlet_stocked ↩
            (outlet_id, servings, product_id, date) ↩
            VALUES (:outlet, :servings, :product, :date)",
        ["outlet" => $outlet, "servings" => $servings, ↩
            "product" => $product, "date" => $date]
    );
}
```

This is from `storing-events.php`

`inventProduct`, `priceProduct`, `openOutlet`, and `stockOutlet` are all pretty self-explanatory. In order to get the IDs they refer to, we need the `newProductId` and `newOutletId` functions. These insert empty rows so that unique identifiers will be generated and can be returned (using the `$connection->lastInsertId()` method).

Naming Pattern

You do not have to follow this same naming pattern. In fact, it's better to use names and patterns that you and your client agree define the core concepts of the product, as far as DDD is concerned.

We can test these using a pattern similar to:

```
store($connection, [
    new ProductInvented("Cheesecake"),
]);

$row = row(
    $connection,
    "SELECT * FROM event_product_invented WHERE name = :name",
    ["name" => "Cheesecake"]
);

assert(!is_null($row));
```

This is from `storing-events.php`

Projecting Events

As we've seen, the method of storing behavior gives us an unprecedented look at the entire history of our data. It's not very good for rendering views, though. As I mentioned, we also need a way to interface an event sourcing architecture with other systems that are not built in the same way.

That means we need to be able to tell the outside world what the more recent state of the application is, as if we were storing it like that in the database. This is often called projection, because we sort through all the events to display a final state for everyone else to see. So, projection in the sense of forecasting a future state, based on present trends.

Earlier we saw functions like:

```
Product::at("Chocolate", date("-1 WEEK"));
// → ["id" => 1, "name" => "Chocolate", ...]
```

Ideally, we'd also have these methods:

```
Product::latest();
// → [["id" => 1, "name" => "Chocolate", ...], ...]

Product::latest("Chocolate");
// → ["id" => 1, "name" => "Chocolate", ...]
```

First, we need to load all the events stored in the database:

```
function fetch(PDO $connection): array {
    $events = [];

    $tables = [
        ProductInvented::class => "event_product_invented",
        ProductPriced::class => "event_product_priced",
        OutletOpened::class => "event_outlet_opened",
        OutletStocked::class => "event_outlet_stocked",
    ];

    foreach ($tables as $type => $table) {
        $rows = rows($connection, "SELECT * FROM {$table}");

        $rows = array_map(
            function($row) use ($connection, $type) {
                return $type::from($connection, $row);
            }, $rows
        );

        $events = array_merge($events, $rows);
    }

    usort($events, function(Event $a, Event $b) {
        return strtotime($a->date()) - strtotime($b->date());
    });

    return $events;
}
```

This is from `projecting-events.php`

There's quite a bit going on here, so let's break it down:

1. We define a list of event tables to get rows from.
2. We fetch the rows for each type/table, and convert the resulting associative arrays to instances of the events.
3. We use the `date` of each event, to sort them into chronological order.

We need to add these new `from` methods to each of our events:

```
abstract class Event
{
    // ...snip

    public function withDate(string $date): self
    {
        $new = clone $this;
        $new->date = $date;

        return $new;
    }

    abstract
    public
    static
    function
    from(PDO $connection, array $data);
}
```

```
final class ProductInvented extends Event
{
    // ...snip

    public static function from(PDO $connection, array $data)
    {
        $new = new static(
            $data["name"]
```

```
        );

        return $new->withDate($data["date"]);
    }
}
```

```
final class ProductPriced extends Event
{
    // ...snip

    public static function from(PDO $connection, array $data)
    {
        $new = new static(
            productNameFromId($connection, $data["product"]),
            $data["cents"]
        );

        return $new->withDate($data["date"]);
    }
}
```

```
final class OutletOpened extends Event
{
    // ...snip

    public static function from(PDO $connection, array $data)
    {
        $new = new static(
            $data["name"]
        );

        return $new->withDate($data["date"]);
    }
}
```

```
final class OutletStocked extends Event
{
    // ...snip

    public static function from(PDO $connection, array $data)
    {
        $new = new static(
            outletNameFromId($connection, $data["outlet"]),
            $data["servings"],
            productNameFromId($connection, $data["product"])
        );

        return $new->withDate($data["date"]);
    }
}
```

This is from `events.php`

We're also using a couple of new global functions:

```
 function productNameFromId(PDO $connection, int $id): string
↪ {
    $row = row(
        $connection,
        "SELECT * FROM event_product_invented WHERE id = :id",
        ["id" => $id]
    );

    if (!$row) {
        throw new InvalidArgumentException("Product not found");
    }

    return $row["name"];
}

 function outletNameFromId(PDO $connection, int $id): string
↪ {
    $row = row(
```

```
        $connection,
        "SELECT * FROM event_outlet_opened WHERE id = :id",
        ["id" => $id]
    );

    if (!$row) {
        throw new InvalidArgumentException("Outlet not found");
    }

    return $row["name"];
}
```

This is from `projecting-events.php`

The reason we need any of these `*NameFromId` and `*IdFromName` functions is because we want to create and present the events using entity names, but we want to store them as foreign keys in the database. That's just a personal preference of mine, and you're free to define/present/store them however makes sense to you.

We can now turn a list of events into database rows, and back again:

```
$events = [];

$events[] = new ProductInvented("Chocolate");
$events[] = new ProductPriced("Chocolate", 499);
$events[] = new OutletOpened("Pismo Beach");
 $events[] = new OutletStocked("Pismo Beach", 24,
↪ "Chocolate");

store($connection, $events); // ← events stored in database
 $stored = fetch($connection); // ← events loaded from
↪ database

assert(json_encode($events) === json_encode($stored));
```

Now, how do we convert this to something usable? We need to define a few more helper functions:

```
function project(PDO $connection, array $events): array {
    $entities = [
        "products" => [],
        "outlets" => [],
    ];

    foreach ($events as $event) {
        $entities = projectOne($connection, $entities, $event);
    }

    return $entities;
}

function projectOne(PDO $connection, array $entities, ↩
    Event $event): array
{
    if ($event instanceof ProductInvented) {
        $entities = projectProductInvented(
            $connection, $entities, $event
        );
    }

    if ($event instanceof ProductPriced) {
        $entities = projectProductPriced(
            $connection, $entities, $event
        );
    }

    if ($event instanceof OutletOpened) {
        $entities = projectOutletOpened(
            $connection, $entities, $event
        );
    }

    if ($event instanceof OutletStocked) {
        $entities = projectOutletStocked(
            $connection, $entities, $event
```

```
        );
    }

    return $entities;
}
```

This is from `projecting-events.php`

This code is similar to the code we use to store events. For each type of event, we modify the array of entities. After all the events have been projected, we should have the latest state. Here's what the other projector methods look like:

```
function projectProductInvented(PDO $connection, ↩
    array $entities, ProductInvented $event): array
{
    $payload = $event->payload();

    $entities["products"][] = [
        "id" => productIdFromName($connection, $payload["name"]),
        "name" => $payload["name"],
    ];

    return $entities;
}

function projectProductPriced(PDO $connection, ↩
    array $entities, ProductPriced $event): array
{
    $payload = $event->payload();

    foreach ($entities["products"] as $i => $product) {
        if ($product["name"] === $payload["product"]) {
            $entities["products"][$i]["price"] = ↩
                $payload["cents"];
        }
    }

    return $entities;
```

```
}

function projectOutletOpened(PDO $connection, ↩
    array $entities, OutletOpened $event): array
{
    $payload = $event->payload();

    $entities["outlets"][] = [
        "id" => outletIdFromName($connection, $payload["name"]),
        "name" => $payload["name"],
        "stock" => [],
    ];

    return $entities;
}

function projectOutletStocked(PDO $connection, ↩
    array $entities, OutletStocked $event): array
{
    $payload = $event->payload();

    foreach ($entities["outlets"] as $i => $outlet) {
        if ($outlet["name"] === $payload["outlet"]) {
            foreach ($entities["products"] as $j => $product) {
                if ($product["name"] === $payload["product"]) {
                    $entities["outlets"][$i]["stock"][] = [
                        "product" => &$product,
                        "servings" => $payload["servings"],
                    ];
                }
            }
        }
    }

    return $entities;
}
```

This is from `projecting-events.php`

Each of these projection methods accepts a type of event, and sorts through the event payload to make their mark on the `$entities` array.

At this point, we can use the structure we've created to populate a website. Since our projectors accept events, we can even generate an initial projection (at the beginning of a request) and then apply any new events to them, as they happen.

As you can probably guess, this isn't the most efficient way to query a database, just to render a website. If you need the projections a lot of the time, it might be better to periodically project your events and then store the resulting structure in denormalized database tables.

That way, you can capture events (through things like API requests or form posts), and still query "normal" database tables, when displaying data in an application.

Projecting Specifically

So far, we've seen how we can describe, store, and project (to the latest state). Projecting to a specific point in time is just a matter of adjusting the projection functions, so that they apply events up to, or after, a certain timestamp.

We've covered way more than I originally planned to, so I'll leave that last bit as an exercise for you. Think of how you could model the traditional (for CMS applications) model of draft/working and published versions of content.

Summary

If you've managed to get this far; well done! It's been a long journey, but worth it I feel. Let us know what you like or don't like about this design pattern. If you want to learn more about Event Sourcing (or DDD in general), definitely check out the books linked at the start.

Code full of arrays can get ugly, fast! I highly recommend you check out Adam Wathan's book, about refactoring loop code to collections.

Chapter 5

Disco with Design Patterns: A Fresh Look at Dependency Injection

by Reza Lavaryan

Dependency Injection is all about code reusability. It's a design pattern aiming to make high-level code reusable, by separating the object creation / configuration from usage.

Consider the following code:

```
<?php

class Test {
```

```
    protected $dbh;

    public function __construct(\PDO $dbh)
    {
        $this->dbh = $dbh;
    }

}

 $dbh  = new PDO('mysql:host=localhost;dbname=test',
↪ 'username', 'password');
$test = new Test($dbh)
```

As you can see, instead of creating the PDO object inside the class, we create it outside of the class and pass it in as a dependency - via the constructor method. This way, we can use the driver of our choice, instead of having to to use the driver defined inside the class.

Our very own Alejandro Gervasio has explained the DI concept fantastically, and Fabien Potencier also covered it in a series.

There's one drawback to this pattern, though: when the number of dependencies grows, many objects need to be created/configured before being passed into the dependent objects. We can end up with a pile of boilerplate code, and a long queue of parameters in our constructor methods. Enter Dependency Injection containers!

A Dependency Injection container - or simply a DI container - is an object which knows exactly how to create a service and handle its dependencies.

In this piece, we'll demonstrate the concept further with a newcomer in this field: Disco.

For more information on dependency injection containers, see our other posts on the topic here.

As frameworks are great examples of deploying DI containers, we will finish by creating a basic HTTP-based framework with the help of Disco and some Symfony Components.

Installation

To install Disco, we use Composer as usual:

```
composer require bitexpert/disco
```

To test the code, we'll use PHP's built-in web server:

```
php -S localhost:8000 -t web
```

As a result, the application will be accessible under `http://localhost:8000` from the browser. The last parameter `-t` option defines the document root - where the `index.php` file resides.

Getting Started

Disco is a container_interop compatible DI container. Somewhat controversially, Disco is an annotation-based DI container.

`container_interop`'s Interfaces

Note that the package `container_interop` consists of a set of interfaces to standardize features of container objects. To learn more about how that works, see the tutorial in which we build our own, SitePoint Dependency Injection Container, also based on container-interop.

To add services to the container, we need to create a **configuration class**. This class should be marked with the `@Configuration` annotation:

```
<?php
/**
 * @Configuration
 */
 class Services {
    // ...
 }
```

Each container service should be defined as a **public** or **protected** method inside the configuration class. Disco calls each service a **Bean**, which originates from the Java culture.

Inside each method, we **define** how a service should be created. Each method **must** be marked with `@Bean` which implies that this a service, and `@return` annotations which notes the type of the returned object.

This is a simple example of a Disco configuration class with one "Bean":

```
<?php
/**
 * @Configuration
 */
class Configuration {

    /**
     * @Bean
     * @return SampleService
     */
    public function getSampleService()
    {
        // Instantiation
        $service  = new SampleService();

        // Configuration
        $service->setParameter('key', 'value');
        return $service;
    }
```

```
}
```

The `@Bean` annotation accepts a few configuration parameters to specify the **nature** of a service. Whether it should be a **singleton** object, **lazy loaded** (if the object is resource-hungry), or even its state **persisted during the session's lifetime** is specified by these parameters.

By default, all the services are defined as **singleton** services.

For example, the following Bean creates a singleton lazy-loaded service:

```
<?php

// ...

/**
 * @Bean({"singleton"=true, "lazy"=true})
 * @return \Acme\SampleService
 */
 public function getSampleService()
 {
     return new SampleService();
 }

// ...
```

Disco uses ProxyManager to do the lazy-loading of the services. It also uses it to inject additional behaviors (defined by the annotations) into the methods of the configuration class.

After we create the configuration class, we need to create an instance of `AnnotationBeanFactory`, passing the configuration class to it. **This will be our container.**

Finally, we register the container with `BeanFactoryRegistry`:

```
<?php

// ...

use \bitExpert\Disco\AnnotationBeanFactory;
use \bitExpert\Disco\BeanFactoryRegistry;

// ...

// Setting up the container
 $container = new AnnotationBeanFactory(Services::class,
↪ $config);
BeanFactoryRegistry::register($container);
```

How to Get a Service from the Container

Since Disco is `container/interop` compatible, we can use `get()` and `has()` methods on the container object:

```
// ...

$sampleService = $container->get('sampleService');
$sampleService->callSomeMethod();
```

Service Scope

HTTP is a stateless protocol, meaning on each request the whole application is bootstrapped and all objects are recreated. We can, however, influence the lifetime of a service by passing the proper parameters to the `@Bean` annotation. One of these parameters is `scope`. The scope can be either `request` or `session`.

If the scope is `session`, the service state will persist during the session lifetime. In other words, on subsequent HTTP requests, the last state of the object is retrieved from the session.

Let's clarify this with an example. Consider the following class:

```
<?php

class sample {

    public $counter = 0;

    public function add()
    {
        $this->counter++;
        return $this;
    }
}
```

In the above class, the value of `$counter` is incremented each time the `add()` method is called; now, let's add this to the container, with scope set to `session`:

```
// ...
/**
 * @Bean({"scope"="session"})
 * @return Sample
 */
public function getSample()
{
    return new Sample();
}
// ...
```

And if we use it like this:

```
// ...
$sample = $container->get('getSample');
$sample->add()
       ->add()
       ->add();
```

```
echo $sample->counter; // output: 3
// ...
```

In the first run, the output will be three. If we run the script again (to make another **request**), the value will be six (instead of three). This is how object state is persisted across requests.

If the scope is set to `request`, the value will be always three in subsequent HTTP requests.

Container Parameters

Containers usually accept parameters from the outside world. With Disco, we can pass the parameters into the container as an associative array like this:

```
// ...
$parameters = [

    // Database configuration
    'database' => [
        'dbms' => 'mysql',
        'host' => 'localhost',
        'user' => 'username',
        'pass' => 'password',
    ],
];

// Setting up the container
 $container = new AnnotationBeanFactory(Services::class,
↪ $parameters);
BeanFactoryRegistry::register($container);
```

To use these values inside each method of the configuration class, we use `@Parameters` and `@parameter` annotations:

```
<?php
// ...

/**
 * @Bean
 * @Parameters({
 *    @parameter({"name"= "database"})
 * })
 *
*/
public function sampleService($database = null)
{
    // ...
}
```

Disco in Action

In this section, we're going to create a basic HTTP-based framework. The framework will create a **response** based on the information received from the **request**.

To build our framework's core, we'll use some Symfony Components.

- **HTTP Kernel**.The heart of our framework. Provides the request / response basics.
- **Http Foundation**.A nice object-oriented layer around PHP's HTTP super globals.
- **Router**. According to the official website: maps an HTTP request to a set of configuration variables - more on this below.
- **Event Dispatcher**. This library provides a way to hook into different phases of a request / response lifecycle, using listeners and subscribers.

To install all these components:

```
 composer require symfony/http-foundation symfony/routing
↪ symfony/http-kernel symfony/event-dispatcher
```

As a convention, we'll keep the framework's code under the `Framework` namespace.

Let's also register a PSR-4 autoloader. To do this, we add the following namespace-to-path mapping under the `psr-4` key in `composer.json`:

```
// ...
 "autoload": {
        "psr-4": {
            "": "src/"
        }
    }
// ...
```

As a result, all namespaces will be looked for within the `src/` directory. Now, we run `composer dump-autoload` for this change to take effect.

Throughout the rest of this section, we'll write our framework's code along with code snippets to make some concepts clear.

The Kernel

The foundation of any framework is its kernel. This is where a **request** is processed into a **response**.

We're not going to create a Kernel from scratch here. Instead, we'll extend the `Kernel` class of the **HttpKernel** component we just installed.

```
<?php
// src/Framework/Kernel.php

namespace Framework;
```

```
use Symfony\Component\HttpKernel\HttpKernel;
use Symfony\Component\HttpKernel\HttpKernelInterface;

 class Kernel extends HttpKernel implements
↪ HttpKernelInterface {
}
```

Since the base implementation works just fine for us, we won't reimplement any methods, and will instead just rely on the inherited implementation.

Routing

A `Route` object contains a **path** and a **callback**, which is called (by the Controller Resolver) every time the route is matched (by the URL Matcher).

The URL matcher is a class which accepts a collection of routes (we'll discuss this shortly) and an instance of RequestContext to find the active route.

A request context object contains information about the current request.

Here's how routes are defined by using the **Routing** component:

```
<?php

// ...

use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$routes = new RouteCollection();

 $routes->add('route_alias', new Route('path/to/match',
↪ ['_controller' => function(){
    // Do something here...
}]
));
```

To create routes, we need to create an instance of `RouteCollection` (which is also a part of the `Routing` component), then **add** our routes to it.

To make the routing syntax more expressive, we'll create a route builder class around `RouteCollection`.

```
<?php
// src/Framework/RouteBuilder.php

namespace Framework;

use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

class RouteBuilder {

    protected $routes;

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

    public function get($name, $path, $controller)
    {
        return $this->add($name, $path, $controller, 'GET');
    }

    public function post($name, $path, $controller)
    {
        return $this->add($name, $path, $controller, 'POST');
    }

    public function put($name, $path, $controller)
    {
        return $this->add($name, $path, $controller, 'PUT');
    }

    public function delete($name, $path, $controller)
    {
```

```
        return $this->add($name, $path, $controller, 'DELETE');
    }

    protected function add($name, $path, $controller, $method)
    {
 $this->routes->add($name, new Route($path,
↪ ['_controller' => $controller], ['_method' =>
↪ $method]));

        return $this;
    }

}
```

This class holds an instance of `RouteCollection`. In `RouteBuilder`, for each HTTP verb, there is a method which calls `add()`. We'll keep our route definitions in the `src/routes.php` file:

```
<?php
// src/routes.php

use Symfony\Component\Routing\RouteCollection;
use Framework\RouteBuilder;

$routeBuilder = new RouteBuilder(new RouteCollection());

$routeBuilder

->get('home', '/', function() {
            return new Response('It Works!');
})

->get('welcome', '/welcome', function() {
            return new Response('Welcome!');
});
```

The Front Controller

The entry point of any modern web application is its front controller. It is a PHP file, usually named `index.php`. This is where the class autoloader is included, and the application is bootstrapped.

All the requests go through this file, and are from here dispatched to the proper controllers. Since this is the only file we're going to expose to the public, we put it inside our web root directory, keeping the rest of the code outside.

```
<?php
//web/index.php

require_once __DIR__ . '/../vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RequestStack;
 use
↪ Symfony\Component\HttpKernel\EventListener\RouterListener;
 use
↪ Symfony\Component\HttpKernel\Controller\ControllerResolver;

// Create a request object from PHP's global variables
$request = Request::createFromGlobals();

$routes = include __DIR__.'/../src/routes.php';
 $UrlMatcher = new Routing\Matcher\UrlMatcher($routes, new
↪ Routing\RequestContext());

// Event dispatcher & subscribers
$dispatcher = new EventDispatcher();
 // Add a subscriber for matching the correct route. We pass
↪ UrlMatcher to this class
 $dispatcher->addSubscriber(new
↪ RouterListener($UrlMatcher, new RequestStack()));


 $kernel = new Framework\Kernel($dispatcher, new
```

```
↪ ControllerResolver());
$response  = $kernel->handle($request);

// Sending the response
$response->send();
```

In the above code, we instantiate a `Request` object based on PHP's global variables.

```
<?php
// ...
$request = Request::createFromGlobals();
// ...
```

Next, we load the `routes.php` file into `$routes`. Detecting the right route is the responsibility of the `UrlMatcher` class, so we create it, passing the route collection along with a `RequestContext` object.

```
<?php
// ...
$routes = include __DIR__.'/../src/routes.php';
 $UrlMatcher = new Routing\Matcher\UrlMatcher($routes, new
↪ Routing\RequestContext());
// ...
```

To use the `UrlMatcher` instance, we pass it to the `RouteListener` event subscriber.

```
<?php
// ...
// Event dispatcher & subscribers
$dispatcher = new EventDispatcher();
 // Add a subscriber for matching the correct route. We pass
↪ UrlMatcher to this class
```

```
 $dispatcher->addSubscriber(new
↪ RouterListener($UrlMatcher, new RequestStack()));
// ...
```

Any time a request hits the application, the event is triggered and the respective listener is called, which in turn detects the proper route by using the `UrlMatcher` passed to it.

Finally, we instantiate the kernel, passing in the **Dispatcher** and an instance of **Controller Resolver** - via its constructor:

```
<?php
// ...

 $kernel    = new Framework\Kernel($dispatcher, new
↪ ControllerResolver());
$response  = $kernel->handle($request);

// Sending the response
$response->send();
// ...
```

Disco Time

So far we had to do plenty of instantiations (and configurations) in the front controller, from creating the request context object, the URL matcher, the event dispatcher and its subscribers, and of course the kernel itself.

It is now time to let Disco wire all these pieces together for us.

As before, we install it using Composer:

```
composer require bitexpert/Disco;
```

Then, we create the configuration class, and define the services we'll need in the front controller:

```
<?php
// src/Framework/Services.php

use bitExpert\Disco\Annotations\Bean;
use bitExpert\Disco\Annotations\Configuration;
use bitExpert\Disco\Annotations\Parameters;
use bitExpert\Disco\Annotations\Parameter;

/**
 * @Configuration
 */
class Services {

    /**
     * @Bean
     * @return \Symfony\Component\Routing\RequestContext
     */
    public function context()
    {
        return new \Symfony\Component\Routing\RequestContext();
    }

    /**
     * @Bean
     *
     * @return \Symfony\Component\Routing\Matcher\UrlMatcher
     */
    public function matcher()
    {
 return new
↪ \Symfony\Component\Routing\Matcher\
UrlMatcher($this->routeCollection(),
↪ $this->context());
    }

    /**
     * @Bean
```

```
     * @return \Symfony\Component\HttpFoundation\RequestStack
     */
    public function requestStack()
    {
        return new \Symfony\Component\HttpFoundation\
RequestStack();
    }

    /**
     * @Bean
     * @return \Symfony\Component\Routing\RouteCollection
     */
    public function routeCollection()
    {
        return new \Symfony\Component\Routing\RouteCollection();
    }

    /**
     * @Bean
     * @return \Framework\RouteBuilder
     */
    public function routeBuilder()
    {
 return new
↪ \Framework\RouteBuilder($this->routeCollection());
    }

    /**
     * @Bean
 * @return
↪ \Symfony\Component\HttpKernel\Controller\ControllerResolver
     */
    public function resolver()
    {
 return new
↪ \Symfony\Component\HttpKernel\Controller\ControllerResolver();
    }


    /**
```

```
     * @Bean
 * @return
↪ \Symfony\Component\HttpKernel\EventListener\RouterListener
     */
    protected function listenerRouter()
    {
 return new
↪ \Symfony\Component\HttpKernel\EventListener\RouterListener(
            $this->matcher(),
            $this->requestStack()
        );
    }

    /**
     * @Bean
     * @return \Symfony\Component\EventDispatcher\EventDispatcher
     */
    public function dispatcher()
    {
 $dispatcher = new
↪ \Symfony\Component\EventDispatcher\EventDispatcher();

        $dispatcher->addSubscriber($this->listenerRouter());

        return $dispatcher;
    }

    /**
     * @Bean
     * @return Kernel
     */
    public function framework()
    {
 return new Kernel($this->dispatcher(),
↪ $this->resolver());
    }

}
```

Seems like a lot of code; but in fact, it's the same code that resided in the front controller previously.

Before using the class, we need to make sure it has been autoloaded by adding it under the `files` key in our `composer.json` file:

```
// ...
 "autoload": {
        "psr-4": {
            "": "src/"
        },
        "files": [
            "src/Services.php"
        ]
    }
// ...
```

And now onto our front controller.

```
<?php

//web/index.php

require_once __DIR__ . '/../vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;

$request   = Request::createFromGlobals();

 $container = new
↪ \bitExpert\Disco\AnnotationBeanFactory(Services::class);
\bitExpert\Disco\BeanFactoryRegistry::register($container);

$routes   = include __DIR__.'/../src/routes.php';

$kernel   = $container->get('framework')
$response = $kernel->handle($request);
```

```
$response->send();
```

Now our front controller can actually breathe! All the instantiations are done by Disco when we request a service.

But How About the Configuration?

As explained earlier, we can pass in parameters as an associative array to the `AnnotationBeanFactory` class.

To manage configuration in our framework, we create two configuration files, one for **development** and one for the **production** environment.

Each file returns an associative array, which we can be loaded into a variable.

Let's keep them inside `Config` directory:

```
// Config/dev.php

return [
    'debug' => true;
];
```

And for production:

```
// Config/prod.php

return [
    'debug' => false;
];
```

To detect the environment, we'll specify the environment in **a special plain-text file**, just like we define an environment variable:

```
ENV=dev
```

To parse the file, we use **PHP dotenv**, a package which loads environment variables from a file (by default the filename is `.env`) into PHP's `$_ENV` super global. This means we can get the values by using PHP's getenv() function.

To install the package:

```
composer require vlucas/phpdotenv
```

Next, we create our `.env` file inside the `Config/` directory.

Config/.env

```
ENV=dev
```

In the front controller, we load the environment variables using PHP dotenv:

```
<?php
//web/index.php

// ...

// Loading environment variables stored .env into $_ENV
$dotenv = new Dotenv\Dotenv(__DIR__ . '/../Config');
$dotenv->load();

 // Load the proper configuration file based on the
↪ environment
 $parameters = require __DIR__ . '/../config/' .
↪ getenv('ENV') . '.php';

 $container = new
↪ \bitExpert\Disco\AnnotationBeanFactory(Services::class,
↪ $parameters);
↪ \bitExpert\Disco\BeanFactoryRegistry::register($container);
```

```
// ...
```

In the preceding code, we first specify the directory in which our `.env` file resides, then we call `load()` to load the environment variables into `$_ENV`. Finally, we use `getenv()` to get the proper configuration filename.

Creating a Container Builder

There's still one problem with the code in its current state: whenever we want to create a new application we have to instantiate `AnnotationBeanFactory` in our front controller (`index.php`). As a solution, we can create a factory which creates the container, whenever needed.

```
<?php
// src/Factory.php

namespace Framework;

class Factory {

    /**
     * Create an instance of Disco container
     *
     * @param  array $parameters
     * @return \bitExpert\Disco\AnnotationBeanFactory
     */
    public static function buildContainer($parameters = [])
    {
 $container = new
↪ \bitExpert\Disco\AnnotationBeanFactory(Services::class,
↪ $parameters);
        \bitExpert\Disco\BeanFactoryRegistry::register($container);

        return $container;
    }
```

```
}
```

This factory has a static method named `buildContainer()`, which creates and registers a Disco container.

This is how it improves our front controller:

```
<?php
//web/index.php

require_once __DIR__ . '/../vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;

// Getting the environment
$dotenv = new Dotenv\Dotenv(__DIR__ . '/../config');
$dotenv->load();

 // Load the proper configuration file based on the
↪ environment
 $parameters = require __DIR__ . '/../config/' .
↪ getenv('ENV') . '.php';

$request   = Request::createFromGlobals();
$container = Framework\Factory::buildContainer($parameters);
$routes    = include __DIR__.'/../src/routes.php';

$kernel   = $container->get('framework')
$response = $kernel->handle($request);
$response->send();
```

It looks much neater now, doesn't it?

Application Class

We can take things one step further in terms of usability, and abstract the remaining operations (in the front controller) into another class. Let's call this class `Application`:

```
<?php

namespace Framework;

use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpFoundation\Request;

class Application {

    protected $kernel;

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

    public function run()
    {
        $request = Request::createFromGlobals();

        $response = $this->kernel->handle($request);
        $response->send();
    }
}
```

`Application` is dependent on the kernel, and works as a wrapper around it. We create a method named `run()`, which populates the request object, and passes it to the kernel to get the response.

To make it even cooler, let's add this class to the container as well:

```
<?php

// src/Framework/Services.php

// ...

    /**
     * @Bean
     * @return \Framework\Application
     */
    public function application()
    {
        return new \Framework\Application($this->kernel());
    }

// ...
```

And this is the new look of our front controller:

```
<?php

require_once __DIR__ . '/../vendor/autoload.php';

// Getting the environment
$dotenv = new Dotenv\Dotenv(__DIR__ . '/../config');
$dotenv->load();

 // Load the proper configuration file based on the
↪ environment
 $parameters = require __DIR__ . '/../config/' .
↪ getenv('ENV') . '.php';

// Build a Disco container using the Factory class
$container = Framework\Factory::buildContainer($parameters);

// Including the routes
require __DIR__ . '/../src/routes.php';
```

```
// Running the application to handle the response
$app = $container->get('application')
          ->run();
```

Creating a Response Listener

We can use the framework now, but there is still room for improvement. Currently, we have to return an **instance** of `Response` in each controller, otherwise, an exception is thrown by the Kernel:

```
<?php

// ...

$routeBuilder

->get('home', '/', function() {
            return new Response('It Works!');
});

->get('welcome', '/welcome', function() {
            return new Response('Welcome!');
});

// ...
```

However, we can make it optional and allow for sending back pure strings, too. To do this, we create a special subscriber class, which automatically creates a `Response` object if the returned value is a string.

Subscribers must implement the `Symfony\Component\EventDispatcher\EventSubscriberInterface` interface. They should implement the `getSubscribedMethods()` method in which we define the events we're interested in subscribing to, and their event listeners.

In our case, we're interested in the `KernelEvents::VIEW` event. The event happens when a response is to be returned.

Here's our subscriber class:

```
<?php
// src/Framework/StringResponseListener
namespace Framework;

 use
↪ Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpFoundation\Response;
 use
↪ Symfony\Component\HttpKernel\Event\
GetResponseForControllerResultEvent;
use Symfony\Component\HttpKernel\KernelEvents;

 class StringResponseListener implements
↪ EventSubscriberInterface
{
 public function onView(GetResponseForControllerResultEvent
↪ $event)
    {
        $response = $event->getControllerResult();

        if (is_string($response)) {
            $event->setResponse(new Response($response));
        }
    }

    public static function getSubscribedEvents()
    {
        return array(KernelEvents::VIEW => 'onView');
    }
}
```

Inside the listener method `onView`, we first check if the response is a string (and not already a `Response` object), then create a response object if required.

To use the subscriber, we need to add it to the container as a protected service:

```
<?php

// ...

/**
 * @Bean
 * @return \Framework\StringResponseListener
 */
protected function ListenerStringResponse()
{
    return new \Framework\StringResponseListener();
}

// ...
```

Then, we add it to the dispatcher service:

```
<?php

// ...

/**
 * @Bean
 * @return \Symfony\Component\EventDispatcher\EventDispatcher
 */
public function dispatcher()
{
 $dispatcher = new
↪ \Symfony\Component\EventDispatcher\EventDispatcher();

    $dispatcher->addSubscriber($this->listenerRouter());
$dispatcher->addSubscriber($this->ListenerStringRespons
↪ e());
    return $dispatcher;
}
```

```
// ...
```

From now on, we can simply return a string in our controller functions:

```
<?php

// ...

$routeBuilder

->get('home', '/', function() {
            return 'It Works!';
})

->get('welcome', '/welcome', function() {
            return 'Welcome!';
});

// ...
```

The framework is ready now.

Conclusion

We created a basic HTTP-based framework with the help of Symfony Components and Disco. This is just a basic Request/Response framework, lacking any other MVC concepts like models and views, but allows for the implementation of any additional architectural patterns we may desire.

The full code is available on Github.

Disco is a newcomer to the DI-container game and if compared to the older ones, it lacks a comprehensive documentation. This piece was an attempt at providing a smooth start for those who might find this new kind of DI container interesting.

Chapter 6

A Comprehensive Guide to Using Cronjobs

by Reza Lavaryan

There are times when there's a need for running a group of tasks automatically at certain times in the future. These tasks are usually administrative, but could be anything - from making database backups to downloading emails when everyone is asleep.

Cron is a time-based job scheduler in Unix-like operating systems, which triggers certain tasks at a point in the future. The name originates from the Greek word *χρόνος* (chronos), which means time.

The most commonly used version of Cron is known as Vixie Cron, originally developed by Paul Vixie in 1987.

This piece is an in-depth walkthrough of this program, and a reboot of this ancient, but still surprisingly relevant post.

Terminology

- **Job:** a unit of work, a series of steps to do something. For example, sending an email to a group of users. In this chapter, we'll use *task*, *job*, *cron job* or *event* interchangeably.
- **Daemon:** (/ˈdiːmən/ or /ˈdeɪmən/) is a computer program which runs in the background, serving different purposes. Daemons are often started at boot time. A web server is a daemon serving HTTP requests. Cron is a daemon for running scheduled tasks.
- **Cron Job:** a cron job is a scheduled *job*, being run by Cron when it's due.
- **Webcron:** a time-based job scheduler which runs within the web server environment. It's used as an alternative to the standard Cron, often on shared web hosts that do not provide shell access.

Getting Started

This tutorial assumes you're running a Unix-based operating system like Ubuntu. If you aren't, we recommend setting up Homestead Improved - it's a 5 minute process which will save you years down the line.

If we take a look inside the `/etc` directory, we can see directories like `cron.hourly`, `cron.daily`, `cron.weekly` and `cron.monthly`, each corresponding to a certain frequency of execution. One way to schedule our tasks is to place our scripts in the proper directory. For example, to run `db_backup.php` on a daily basis, we put it inside `cron.daily`. If the folder for a given frequency is missing, we would need to create it first.

`run-parts`

This approach uses the `run-parts` script, a command which runs every executable it finds within the specified directory.

This is the simplest way to schedule a task. However, if we need more flexibility, we should use Crontab.

Crontab Files

Cron uses special configuration files called `crontab` files, which contain a list of jobs to be done. Crontab stands for **Cron Table**. Each line in the crontab file is called a cron job, which resembles a set of columns separated by a space character. Each row specifies **when** and **how often** a certain command or script should be executed.

In a crontab file, blank lines or lines starting with #, spaces or tabs will be ignored. Lines starting with # are considered comments.

Active lines in a crontab are either the declaration of an environment variable or a cron job, and comments are not allowed on the active lines.

Below is an example of a crontab file with just one entry:

```
0 0 * * *  /var/www/sites/db_backup.sh
```

The first part `0 0 * * *` is the cron expression, which specifies the frequency of execution. The above cron job will run once a day.

Users can have their own crontab files named after their username as registered in the `/etc/passwd` file. All user-level crontab files reside in Cron's spool area. These files should **not** be edited directly. Instead, we should edit them using the `crontab` command-line utility.

spool Directory Location

The spool directory varies across different distributions of Linux. On Ubuntu it's `/var/spool/cron/crontabs` while in CentOS it's `/var/spool/cron`.

To edit our **own** crontab file:

```
crontab -e
```

The above command will automatically open up the crontab file which belongs to our user. If a default system editor for the crontab hasn't been selected before, a choice will be presented listing the installed ones. We can also explicitly choose or change our desired editor for editing the crontab file:

```
export VISUAL=nano; crontab -e
```

After we save the file and exit the editor, the crontab will be checked for accuracy. If everything is set properly, the file will be saved to the spool directory.

Privileges

Each command in the crontab file is executed from the perspective of the user who owns the crontab, so if your command runs as root (sudo) you will not be able to define this crontab from your own user account unless you log in as root.

To list the installed cron jobs belonging to our own user:

```
crontab -l
```

We can also write our cron jobs in a file and send its contents to the crontab file like so:

```
crontab /path/to/the/file/containing/cronjobs.txt
```

The preceding command will overwrite the existing crontab file with `/path/to/the/file/containing/cronjobs.txt`.

To remove the crontab, we use the `-r` option:

```
crontab -r
```

Anatomy of a Crontab Entry

The anatomy of a user-level crontab entry looks like the following:

```
# ┌──────────────── min (0 - 59)
 # │ ┌────────────── hour (0 - 23)
 # │ │ ┌──────────── day of month (1 - 31)
 # │ │ │ ┌────────── month (1 - 12)
 # │ │ │ │ ┌──────── day of week (0 - 6)
 # │ │ │ │ │
 # │ │ │ │ │
 # * * * * *  command to execute
```

The first two fields specify the time (**minute** and **hour**) at which the task will run. The next two fields specify the **day of the month** and the **month**. The fifth field specifies the **day of the week**.

The command will be executed when the minute, hour, month and either **day of month** or **day of week** match the current time.

If both **day of week** and **day of month** have certain values, the event will be run when **either** field matches the current time. Consider the following expression:

```
0 0 5-20/5 Feb 2 /path/to/command
```

The preceding cron job will run once per day every five days, from 5th to 20th of February **plus** all Tuesdays of February.

Specifying Both Day of Month and Day of Week

When both **day of month** and **day of week** have certain values (not an asterisk), it will create an `OR` condition, meaning both days will be matched.

The syntax in system crontabs (`/etc/crontab`) is slightly different than user-level crontabs. The difference is that the sixth field is **not** the command, but it is the **user** we want to run the job as.

```
* * * * * testuser /path/to/command
```

It's not recommended to put system-wide cron jobs in `/etc/crontab`, as this file might be modified in future system updates. Instead, we put these cron jobs in the `/etc/cron.d` directory.

Editing Other Users' Crontab

We might need to edit other users' crontab files. To do this, we use the `-u` option as below:

```
crontab -u username -e
```

Privileges

We can only edit other users' crontab files as the root user, or as a user with administrative privileges.

Some tasks require super admin privileges, thus, they should be added to the **root** user's crontab file:

```
sudo crontab -e
```

Ownership

Please note that using `sudo` with `crontab -e` will edit the root user's crontab file. If we need to to edit another user's crontab while using `sudo`, we should use `-u` option to specify the crontab owner.

To learn more about the `crontab` command:

```
man crontab
```

Standard and Non-Standard Values

Crontab fields accept numbers as values. However, we can put other data structures in these fields, as well.

Ranges

We can pass in ranges of numbers:

```
0 6-18 1-15 * * /path/to/command
```

The above cron job will be executed from 6 am to 6 pm from 1st to 15th of each month in the year. Note that the specified range is inclusive, so 1-5 means 1,2,3,4,5.

Lists

A list is a group of values separated by commas. We can have lists as field values:

```
0 1,4,5,7 * * * /path/to/command
```

The above syntax will run the cron job at 1 am, 4 am, 5 am and 7 am every day.

Steps

Steps can be used with ranges or the asterisk character `(*)`. When they are used with ranges they specify the number of values to **skip** through the end of the range. They are defined with a `/` character after the range, followed by a number. Consider the following syntax:

```
0 6-18/2 * * * /path/to/command
```

The above cron job will be executed every two hours from 6 am to 6 pm.

When steps are used with an asterisk, they simply specify the frequency of that particular field. As an example if we set the minute to `*/5`, it simply means **every five minutes**.

We can combine lists, ranges, and steps together to have more flexible event scheduling:

```
0 0-10/5,14,15,18-23/3 1 1 * /path/to/command
```

The above event will run every five hours from midnight of January 1st to 10 am, 2 pm, 3 pm and also every three hours from 6pm to 11 pm.

Names

For the fields **month** and **day of week** we can use the first three letters of a particular day or month, like `Sat`, `sun`, `Feb`, `Sep`, etc.

```
* * * Feb,mar sat,sun /path/to/command
```

The preceding cron job will be run only on Saturdays and Sundays of February and March.

Please note that the names are **not** case-sensitive. Ranges are not allowed when using names.

Predefined Definitions

Some cron implementations may support some special strings. These strings are used instead of the first five fields, each specifying a certain frequency:

- **@yearly, @annually** Run once a year at midnight of January 1 `(0 0 1 1 *)`
- **@monthly** Run once a month, at midnight of the first day of the month `(0 0 1 * *)`
- **@weekly** Run once a week at midnight of Sunday `(0 0 * * 0)`
- **@daily** Run once a day at midnight `(0 0 * * *)`
- **@hourly** Run at the beginning of every hour `(0 * * * *)`

- **@reboot** Run once at startup

Multiple Commands in the Same Cron Job

We can run several commands in the same cron job by separating them with a semi-colon (`;`).

```
* * * * * /path/to/command-1; /path/to/command-2
```

If the running commands depend on each other, we can use double ampersand (`&&`) between them. As a result, the second command will not be executed if the first one fails.

```
* * * * * /path/to/command-1 && /path/to/command-2
```

Environment Variables

Environment variables in crontab files are in the form of `VARIABLE_NAME = VALUE` (The white spaces around the equal sign are optional). Cron does not source any startup files from the user's home directory (when it's running user-level crons). This means we should manually set any user-specific settings required by our tasks.

Cron daemon automatically sets some environmental variables when it starts. `HOME` and `LOGNAME` are set from the crontab owner's information in `/etc/passwd`. However, we can override these values in our crontab file if there's a need for this.

There are also a few more variables like `SHELL`, specifying the shell which runs the commands. It is `/bin/sh` by default. We can also set the `PATH` in which to look for programs.

```
PATH = /usr/bin;/usr/local/bin
```

Use Quote Marks

We should wrap the value in quotation marks when there's a space in the value. Please note that values are considered as ordinary strings and are **not** interpreted or parsed in any way.

Different Time Zones

Cron uses the system's time zone setting when evaluating crontab entries. This might cause problems for multiuser systems with users based in different time zones. To work around this problem, we can add an environment variable named `CRON_TZ` in our crontab file. As a result, all crontab entries will be parsed based on the specified timezone.

How Cron Interprets Crontab Files

After Cron starts, it searches its spool area to find and load crontab files into the memory. It additionally checks the `/etc/crontab` and or `/etc/cron.d` directories for system crontabs.

After loading the crontabs into memory, Cron checks the loaded crontabs on a minute-by-minute basis, running the events which are due.

In addition to this, Cron regularly (every minute) checks if the spool directory's `modtime` (modification time) has changed. If so, it checks the `modetime` of all the loaded crontabs and reloads those which have changed. That's why we don't have to restart the daemon when installing a new cron job.

Cron Permissions

We can specify which user should be able to use Cron and which user should not. There are two files which play an important role when it comes to cron permissions: `/etc/cron.allow` and `/etc/cron.deny`.

If `/etc/cron.allow` exists, then our username must be listed in this file in order to use `crontab`. If `/etc/cron.deny` exists, it shouldn't contain our username. If

neither of these files exist, then based on the site-dependent configuration parameters, either the **super user** or **all users** will be able to use `crontab` command. For example, in Ubuntu, if neither file exists, all users can use crontab by default.

We can put `ALL` in `/etc/cron.deny` file to prevent all users from using cron:

```
echo ALL > /etc/cron.deny
```

Allow and Deny

If we create an `/etc/cron.allow` file, there's no need to create a `/etc/cron.deny` file as it has the same effect as creating a `/etc/cron.deny` file with `ALL` in it.

Redirecting Output

We can redirect the output of our cron job to a file, if the command (or script) has any output:

```
 * * * * * /path/to/php /path/to/the/command >>
↪ /var/log/cron.log
```

We can redirect the standard output to dev null, to get no email (more on emails below), but still allowing the standard error to be sent as email:

```
* * * * * /path/to/php /path/to/the/command > /dev/null
```

To prevent Cron from sending any emails to us, we change the respective crontab entry as below:

```
 * * * * * /path/to/php /path/to/the/command > /dev/null
↪ 2>&1
```

This means "send both the standard output, and the error output into oblivion".

Email the Output

The output is mailed to the owner of the crontab or the email(s) specified in the `MAILTO` environment variable (if the standard output or standard error are not redirected as above).

If `MAILTO` is set to empty, no email will be sent out as the result of the cron job.

We can set several emails by separating them with commas:

```
MAILTO=admin@example.com,dev@example.com
* * * * * /path/to/command
```

Cron and PHP

We usually run our PHP command line scripts using the PHP executable.

```
php script.php
```

Alternatively, we can use shebang at the beginning of the script, and point to the PHP executable:

```
#! /usr/bin/php

<?php

// PHP code here
```

```
```

As a result, we can execute the file by calling it by name. However, we need to make sure we have the permission to execute it.

To have more robust PHP command line scripts, we can use third-party components for creating console applications like Symfony Console Component or Laravel Artisan. This article is a good start for using Symfony's Console Component.
Creating console commands using Laravel Artisan has been also covered here. If you'd rather use another command line tool for PHP, we have a comparison here.

Task Overlaps

There are times when scheduled tasks take much longer than expected. This will cause overlaps, meaning some tasks might be running at the same time. This might not cause a problem in some cases, but when they are modifying the same data in a database, we'll have a problem. We can overcome this by increasing the execution frequency of the tasks, but still it's not guaranteed that these overlaps won't happen again.

We have several options to prevent cron jobs from overlapping.

Using Flock

Flock is a nice tool to manage lock files from within shell scripts or the command line. These lock files are useful for knowing whether or not a script is running.

When used in conjunction with Cron, the respective cron jobs do not start if the lock file exists. You can install Flock using `apt-get` or `yum` depending on the Linux distribution.

```
apt-get install flock
```

Or

```
yum install flock
```

Consider the following crontab entry:

```
 * * * * * /usr/bin/flock --timeout=1 /path/to/cron.lock
↪ /usr/bin/php /path/to/scripts.php
```

In the preceding example, `flock` looks for `/path/to/cron.lock`. If the lock is acquired in one second, it will run the script, otherwise, it will fail with an exit code of 1.

Using a Locking Mechanism in the Scripts

If the cron job executes a script, we can implement a locking mechanism in the script. Consider the following PHP script:

```
<?php
 $lockfile = sys_get_temp_dir() . '/' md5(__FILE__) .
↪ '.lock';
 $pid      = file_exists($lockfile) ?
↪ trim(file_get_contents($lockfile)) : null;

if (is_null($pid) || posix_getsid($pid) === false) {

    // Do something here

    // And then create/update the lock file
    file_put_contents($lockfile, getmypid());

} else {
    exit('Another instance of the script is already running.');
}
```

In the preceding code, we keep `pid` of the current PHP process in a file, which is located in the system's `temp` directory. Each PHP script has its own lock file, which is the MD5 hash of the script's filename.

First, we check if the lock file exists, and then we get its content, which is the process ID of the last running instance of the script. Then we pass the `pid` to posix_getsid PHP function, which returns the session ID of the process. If `posix_getsid` returns `false` it means the process is not running anymore and we can safely start a new instance.

Anacron

One of the problems with Cron is that it assumes the system is running continuously (24 hours a day). This causes problems for machines which are not running all day long (like personal computers). If the system goes off during the time a task is scheduled to run, Cron will not run that task retroactively.

Anacron is not a replacement for Cron, but it has been developed to solve this problem. It runs the commands once a day, week or month but not on a minute-by-minute or hourly basis as Cron does. It is, however, guaranteed that the task will run even if the system goes off for an unanticipated period of time.

Only root or a user with administrative privileges can manage Anacron tasks. Anacron does not run in the background like a daemon, but only once, executing the tasks which are due.

Anacron uses a configuration file (just like crontab) named `anacrontabs`. This file is located in the `/etc` directory.

The content of this file looks like this:

```
# /etc/anacrontab: configuration file for anacron

# See anacron(8) and anacrontab(5) for details.

SHELL=/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
```

```
 # the maximal random delay added to the base delay of the
↪ jobs
RANDOM_DELAY=45
# the jobs will be started during the following hours only
START_HOURS_RANGE=3-22

 #period in days   delay in minutes   job-identifier
↪ command
 1       5       cron.daily              nice run-parts
↪ /etc/cron.daily
 7       25      cron.weekly             nice run-parts
↪ /etc/cron.weekly
 @monthly 45     cron.monthly            nice run-parts
↪ /etc/cron.monthly
```

In an `anacrontab` file, we can only set the frequencies with a period of `n` days, followed by the delay time in minutes. This delay time is just to make sure the tasks do not run at the same time.

The third column is a unique name, which is used to identify the task in the Anacron log files.

The fourth column is the actual command to be run.

Consider the following entry:

```
 1       5       cron.daily              nice run-parts
↪ /etc/cron.daily
```

The above tasks are run daily, 5 minutes after Anacron is run. It uses `run-parts` to execute all the scripts within `/etc/cron.daily`.

The second entry in the list above runs every 7 days (weekly), with a 25 minutes delay.

Collision Between Cron and Anacron

As you have probably noticed, Cron is also set to execute the scripts inside `/etc/cron.*` directories. This sort of possible collision with Anacron is handled differently in different flavors of Linux. In Ubuntu, Cron checks if Anacron is present in the system, and if it so, it won't execute the scripts within `/etc/cron.*` directories.

In other flavors of Linux, Cron updates the Anacron times-stamps when it runs the tasks, so Anacron won't execute them if they have been already run by Cron.

Quick Troubleshooting

Absolute Path to the commands

It's a good habit to use the absolute paths to all the executables we use in a crontab file.

```
* * * * * /usr/local/bin/php /absolute/path/to/the/command
```

Make Sure Cron Daemon Is Running

If our tasks are not running at all, first we need to make sure the Cron daemon is running:

```
ps aux | grep crond
```

The output should similar to this:

```
 root      7481  0.0  0.0 116860  1180 ?        Ss    2015
↪ 0:49 crond
```

Check `/etc/cron.allow` and `/etc/cron.deny` Files

If the cron jobs are not running, then we need to check if `/etc/cron.allow` exists. If it does, we need to make sure our username is listed in this file. If `/etc/cron.deny` exists, we need to make sure our username is not listed in this file.

If we edit a user's crontab file whereas the user does not exist in the `/etc/cron.allow` file, including the user in the `/etc/cron.allow` won't run the cron until we re-edit the crontab file.

Execute Permission

We need to make sure that the owner of the crontab has the execute permissions for all the commands and scripts in the crontab file. Otherwise, the cron will not work. Execute permissions can be added to any folder or file with `chmod +x /some/file.php`

New Line Character

Every entry in the crontab should end with a new line. This means there must be a blank line after the last crontab entry, or the last cron job will never run.

Wrapping Up

Cron is a daemon, running a list of events scheduled to take place in the future. These jobs are listed in special configuration files called crontab files. Users can have their own crontab file, if they are allowed to use Cron, based on `/etc/cron.allow` or `/etc/cron.deny files`. In addition to user-level cron jobs, Cron also loads the system-wide cron jobs which are slightly different in syntax.

Our tasks are commonly PHP scripts or command-line utilities. In systems which are not running all the time, we can use Anacron to run the events which happen in the period of `n` days.
When working with Cron, we should also be aware of the tasks overlapping each other, to prevent data loss. After a cron job is finished, the output will be sent to

the owner of the crontab and or the email(s) specified in the `MAILTO` environment variable.

Chapter 7

Event Loops in PHP

by Christopher Pitt

PHP developers are always waiting for something. Sometimes we're waiting for requests to remote services. Sometimes we're waiting for databases to return rows from a complex query. Wouldn't it be great if we could do other things during all that waiting?

If you've written some JS, you're probably familiar with callbacks and DOM events. And though we have callbacks in PHP, they don't work in quite the same way. That's thanks to a feature called the event loop.

We're going to look at how the event loop works, and how we can use the event loop in PHP.

We're going to see some interesting PHP libraries. Some would consider these not yet stable enough to use in production. Some would consider the examples presented as "better to do in more mature languages". There are good reasons to try these things. There are also good reasons to avoid these things in production. The purpose of this post is to highlight what's possible in PHP.

Where Things Go To Wait

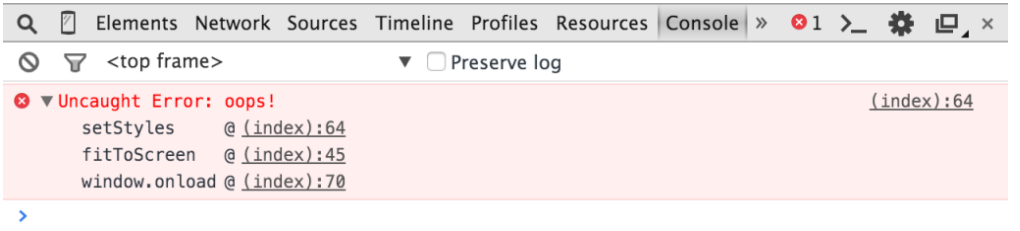
To understand event loops, let's look at how they work in the browser. Take a look at this example:

```
function fitToScreen(selector) {
    var element = document.querySelector(selector);

    var width = element.offsetWidth;
    var height = element.offsetHeight;

    var top = "-" + (height / 2) + "px";
    var left = "-" + (width / 2) + "px";

    var ratio = getRatio(width, height);

    setStyles(element, {
        "position": "absolute",
        "left": "50%",
        "top": "50%",
        "margin": top + " 0 0 " + left,
        "transform": "scale(" + ratio + ", " + ratio + ")"
    });
}

function getRatio(width, height) {
    return Math.min(
        document.body.offsetWidth / width,
        document.body.offsetHeight / height
    );
}
```

```
function setStyles(element, styles) {
    for (var key in styles) {
        if (element.style.hasOwnProperty(key)) {
            element.style[key] = styles[key];
        }
    }
}

fitToScreen(".welcome-screen");
```

This code requires no extra libraries. It will work in any browser that supports CSS scale transformations. A recent version of Chrome should be all you need. Just make sure the CSS selector matches an element in your document.

These few functions take a CSS selector and center and scale the element to fit the screen. What would happen if we threw an `Error` inside that `for` loop? We'd see something like this…



We call that list of functions a stack trace. It's what things look like inside the stack browsers use.

This is like how PHP uses a stack to store context. Browsers go a step further and provide APIs for things like DOM events and Ajax callbacks. In its natural state, JavaScript is every bit as asynchronous as PHP. That is: while both look like they can do many things at once, they are single threaded. They can only do one thing at a time.

With the browser APIs (things like setTimeout and addEventListener) we can offload parallel work to different threads. When those events happen, browsers

add callbacks to a callback queue. When the stack is next empty, browses pick the callbacks up from the callback queue and execute them.

This process of clearing the stack, and then the callback queue, is the event loop.

Life Without An Event Loop

In JS, we can run the following code:

```
setTimeout(function() {
    console.log("inside the timeout");
}, 1);

console.log("outside the timeout");
```

When we run this code, we see `outside the timeout` and then `inside the timeout` in the console. The `setTimeout` function is part of the WebAPIs that browsers give us to work with. When 1 millisecond has passed, they add the callback to the callback queue.

The second `console.log` completes before the one from inside the `setTimeout` starts. We don't have anything like `setTimeout` in standard PHP, but if we had to try and simulate it:

```
function setTimeout(callable $callback, $delay) {
    $now = microtime(true);

    while (true) {
        if (microtime(true) - $now > $delay) {
            $callback();
            return;
        }
    }
}

setTimeout(function() {
    print "inside the timeout";
```

```
}, 1);

print "outside the timeout";
```

When we run this, we see `inside the timeout` and then `outside the timeout`. That's because we have to use an infinite loop inside our `setTimeout` function to execute the callback after a delay.

It may be tempting to move the `while` loop outside of `setTimeout` and wrap all our code in it. That might make our code feel less blocking, but at some point we're always going to be blocked by that loop. At some point we're going to see how we can't do more than a single thing in a single thread at a time.

While there is nothing like `setTimeout` in standard PHP, there are some obscure ways to implement non-blocking code alongside event loops. We can use functions like `stream_select` to create non-blocking network IO. We can use C extensions like EIO to create non-blocking filesystem code. Let's take a look at libraries built on these obscure methods…

Icicle

Icicle is library of components built with the event loop in mind. Let's look at a simple example:

```
function setTimeout(callable $callback, $delay) {
    $now = microtime(true);

    while (true) {
        if (microtime(true) - $now > $delay) {
            $callback();
            return;
        }
    }
}

setTimeout(function() {
```

```
    print "inside the timeout";
}, 1);

print "outside the timeout";
```

This is with `icicleio/icicle` version `0.8.0`.

Icicle's event loop implementation is great. It has many other impressive features; like A+ promises, socket, and server implementations.

Icicle also uses generators as co-routines. Generators and co-routines are a different topic, but the code they allow is beautiful:

```
use Icicle\Coroutine;
use Icicle\Dns\Resolver\Resolver;
use Icicle\Loop;

 $coroutine = Coroutine\create(function ($query, $timeout =
↪ 1) {
    $resolver = new Resolver();

    $ips = (yield $resolver->resolve(
        $query, ["timeout" => $timeout]
    ));

    foreach ($ips as $ip) {
        print "ip: {$ip}\n";
    }
}, "sitepoint.com");

Loop\run();
```

This is with `icicleio/dns` version `0.5.0`.

Generators make it easier to write asynchronous code in a way that resembles synchronous code. When combined with promises and an event loop, they lead to great non-blocking code like this!

ReactPHP

ReactPHP has a similar event loop implementation, but without all the interesting generator stuff:

```
$loop = React\EventLoop\Factory::create();

$loop->addTimer(0.1, function () {
    print "inside timer";
});

print "outside timer";

$loop->run();
```

This is with `react/event-loop` version `0.4.1`.

ReactPHP is more mature than Icicle, and it has a larger range of components. Icicle has a way to go before it can contend with all the functionality ReactPHP offers. The developers are making good progress, though!

Conclusion

It's difficult to get out of the single-threaded mindset that we are taught to have. We just don't know the limits of code we could write if we had access to non-blocking APIs and event loops.

The PHP community needs to become aware of this kind of architecture. We need to learn and experiment with asynchronous and parallel execution. We need to pirate these concepts and best-practices from other languages who've had event loops for ages, until "how can I use the most system resources, efficiently?" is an easy question to answer with PHP.

Chapter 8

PDO – the Right Way to Access Databases in PHP

by Parham Doustdar

PDO is the acronym of PHP Data Objects. As the name implies, this extension gives you the ability to interact with your database through objects.

Why not `mysql` and `mysqli`?

The very valid question people ask when confronted by a new technology is simply, why should they upgrade? What does this new technology give them that is worth the effort of going through their entire application and converting everything to this new library, extension, or whatever?

It's a very valid concern. We've written about this to some degree before, but let's go through why we think it's worth it to upgrade.

PDO is object-oriented

Let's face it: PHP is rapidly growing, and it is moving toward becoming a better programming language. Usually, when this happens in a dynamic language, the language increases its strictness in order to allow programmers to write enterprise applications with peace of mind.

In case of PHP, better PHP means *object-oriented PHP*. This means the more you get to use objects, the better you can test your code, write reusable components, and, usually, increase your salary.

Using PDO is the first step in making the database layer of your application object-oriented and reusable. As you will see in the rest of this chapter, writing object-oriented code with PDO is much simpler than you may think.

Abstraction

Imagine that you have written a killer application using MySQL at your current workplace. All of a sudden, someone up the chain decides that you must migrate your application to use Postgres. What are you going to do?

You have to do a lot of messy replaces, like converting mysql_connect or mysqli_connect to pg_connect, not to mention all the other functions you used for running queries and fetching results. If you were using PDO, it would be very simple. Just a few parameters in the main configuration file would need changing, and you'd be done.

It allows parameter binding

Parameter binding is a feature that allows you to replace placeholders in your query with the value of a variable. It means:

- You don't have to know, at runtime, how many placeholders you will have.
- Your application will be much safer against SQL injection.

You can fetch data into objects

People who have used ORMs like Doctrine know the value of being able to represent data in your tables with objects. If you would like to have this feature, but don't want to learn an ORM, or don't want to integrate it into an already existing application, PDO will allow you to fetch the data in your table into an object.

The `mysql` extension is no longer supported!

Yes, the `mysql` extension is finally removed from PHP 7. That means if you're going to use PHP 7, you need to change all those functions to `mysqli_*` instead of `mysql_*`. This is a great time to just upgrade straight to PDO because of how much it helps you in writing maintainable, portable code with much less effort.

I hope the reasons above have convinced you to start integrating PDO into your application. Don't worry about setting it up; you may already have it on your system!

Verifying the Existence of PDO

If you are using PHP 5.5.X and above, chances are that your installation already includes PDO. To verify, simply open the terminal on Linux and Mac OS X, or the command prompt on Windows, and type the following command:

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

You can also create a php file under your webroot, and insert a `phpinfo()` statement inside it:

```
php -i | grep 'pdo'
```

Now, open this page in your browser and search for the `pdo` string.

If you have PDO and MySQL, skip the installation instructions. If you have PDO but don't have it for MySQL, you merely need to install the `mysqlnd` extension per the instructions below. However, if you don't have PDO at all, your path is longer, but not harder! Keep on reading and we'll tell you how to gear up by installing PDO and `mysqlnd`!

Installation of PDO

If you have already installed PHP from a repository through a package manager (e.g. apt, yum, pacman, and so on), installing PDO is very simple and straightforward; just run the installation command that is listed under your respective operating system and distribution below. If you haven't, though, I have also included my recommended methods for starting from scratch.

Fedora, RedHat and CentOS

First, if you don't have it already, add the Remi repository using the instructions provided on their blog. When that's done, you can easily install `php-pdo` using the following command:

```
sudo yum --enablerepo=remi,remi-php56 install php-pdo
```

Make Sure to Refer to the Desired Repository

Although having the `remi` repository enabled is required, you need to replace `remi-php56` with your desired repository in the command above.

Of course, if you don't have it already, you also need to install the `mysqlnd` extension using the following command:

```
sudo yum --enablerepo=remi,remi-php56 install php-mysqlnd
```

Debian and Ubuntu

On Ubuntu, you need to add the Ondrej repository. This link points to the PPA for 5.6, but you can find links to previous versions there as well.

On Debian, you should add the Dotdeb repository to your system.

On both of these distributions, once you've installed the `php5` metapackage, you already have PDO ready and configured. All you need to do is to simply install the `mysqlnd` extension:

```
sudo apt-get install php5-mysqlnd
```

Windows

You should try and use a Linux virtual machine for development on Windows, but in case you're not up for it, follow the instructions below.

On Windows, you usually get the full lamp stack using Wamp or Xampp. You can also just download PHP straight from windows.php.net. Obviously, if you do the latter, you will only have PHP, and not the whole stack.

In either case, if PDO isn't already activated, you just need to uncomment it in your `php.ini`. Use the facilities provided in your lamp stack to edit `php.ini`, or in case of having downloaded PHP from windows.php.net, just open the folder you chose as your installation directory and edit `php.ini`. Once you do, uncomment this line:

```
;extension=php_pdo_mysql.dll
```

Beginning with PDO: a High-level Overview

When querying your database using PDO, your workflow doesn't change much. However, there are a few habits you must learn to drop, and others that you have

to learn. Below are the steps you need to perform in your application to use PDO. We will explain each one in more detail below.

- Connecting to your database
- Optionally, preparing a statement and binding parameters
- Executing the query

Connecting to your database

To connect to your database, you need to Instantiate a new PDO object and pass it a data source name, also known as a DSN[1].

For example, here is how you can connect to a MySQL database:

```
 $connection = new
↪ PDO('mysql:host=localhost;dbname=mydb;charset=utf8', 'root',
↪ 'root');
```

The function above contains the DSN, the username, and the password. As quoted above, the DSN contains the driver name (`mysql`), and driver-specific options. For `mysql`, those options are `host` (in the `ip:port` format), `dbname`, and `charset`.

Queries

Contrary to how `mysql_query()` and `mysqli_query()` work, there are two kinds of queries in PDO: ones that return a result (e.g. `select` and `show`), and ones that don't (e.g. `insert`, `delete`, and so on). Let's look at the simpler option first: the ones that don't.

1. In general, a DSN consists of the PDO driver name, followed by a colon, followed by the PDO driver-specific connection syntax. Further information is available from http://php.net/manual/en/pdo.drivers.php

Executing queries

These queries are very simple to run. Let's look at an insert.

```
 $connection->exec('INSERT INTO users VALUES (1,
↪ "somevalue"');
```

Technically, I lied when I said that these queries don't return a result. If you change the code above to the following code, you will see that `exec()` returns the number of affected rows:

```
 $affectedRows = $connection->exec('INSERT INTO users
↪ VALUES (1, "somevalue"');
echo $affectedRows;
```

As you can probably guess, for insert statements, this value is usually one. For other statements though, this number varies.

Fetching Query Results

With `mysql_query()` or `mysqli_query()`, here is how you would run a query:

```
$result = mysql_query('SELECT * FROM users');

while($row = mysql_fetch_assoc($result)) {
    echo $row['id'] . ' ' . $row['name'];
}
```

With PDO, however, things are much more intuitive:

```
 foreach($connection->query('SELECT * FROM users') as
↪ $row) {
    echo $row['id'] . ' ' . $row['name'];
```

```
}
```

Fetching Modes: `Assoc`, `Num`, `Obj` and `class`

Just as with the `mysql` and `mysqli` extensions, you can fetch the results in different ways in `PDO`. To do this, you must pass in one of the `PDO::fetch_*` constants, explained in the help page for the fetch function. If you want to get all of your results at once, you can use the fetchAll function.

Below are a few of what we think are the most useful fetch modes.

- PDO::FETCH_ASSOC: returns an array indexed by column name. That is, in our previous example, you need to use `$row['id']` to get the `id`.
- PDO::FETCH_NUM: returns an array indexed by column number. In our previous example, we'd get the `id` column by using `$row[0]` because it's the first column.
- PDO::FETCH_OBJ: returns an anonymous object with property names that correspond to the column names returned in your result set. For example, `$row->id` would hold the value of the `id` column.
- PDO::FETCH_CLASS: returns a new instance of the requested class, mapping the columns of the result set to named properties in the class. If fetch_style includes PDO::FETCH_CLASSTYPE (e.g. PDO::FETCH_CLASS | PDO::FETCH_CLASSTYPE) then the name of the class is determined from a value of the first column. If you remember, we noted that `PDO`, at its simplest form, can map column names into classes that you define. This constant is what you would use to do that.

This List Is Not Comprehensive

This list is not complete, and we recommend checking the aforementioned help page to get all of the possible constants and combinations.

As an example, let's get our rows as associative arrays:

```
$statement = $connection->query('SELECT * FROM users');
```

```
while($row = $statement->fetch(PDO::FETCH_ASSOC)) {
    echo $row['id'] . ' ' . $row['name'];
}
```

Always Choose a Fetch Mode

We recommend always choosing a fetch mode, because fetching the results as `PDO::FETCH_BOTH` (the default) takes twice as much memory, since PHP provides access to different column values both through an associative array and a normal array.

As you might remember, above, when we were listing the advantages of `PDO`, we mentioned that there's a way to make `PDO` store the current row in a class you have previously defined. You have probably also seen the `PDO::FETCH_CLASS` constant explained above. Now, let's use it to retrieve the data from our database into instances of a `User` class. Here is our `User` class:

```
class User
{

    protected $id;
    protected $name;

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

    public function setId($id)
    {
        $this->id = $id;
    }

    public function getName()
    {
        return $this->name;
    }
```

```
    public function setName($name)
    {
        $this->name = $name;
    }

}
```

Now, we can make the same query again, this time using our `User` class, which is, in these cases, also known as `Model`, `Entity`, or a plain old PHP object (taken from Plain Old Java Object in the world of Java).

```
$statement = $connection->query('SELECT * FROM users');

 while($row = $statement->fetch(PDO::FETCH_CLASS, 'User'))
↪ {
    echo $row->getId() . ' ' . $row->getName();
}
```

Prepared Statements and Binding Parameters

To understand parameter binding and its benefits, we must first look more deeply into how PDO works. When we called `$statement->query()` above, `PDO` internally prepared a statement, and executed it, returning the resulting statement to us.

When you call `$connection->prepare()`, you are creating a prepared statement. Prepared statements are a feature of some database management systems that allow them to receive a query like a template, compile it, and execute it when they receive the value of placeholders - think of them as rendering your Blade or Twig templates.

When you later on call `$statement->execute()`, you are passing in the values for those placeholders, and telling the database management system to actually run the query. It's like calling the `render()` function of your templating engine.

To see this in action, let's create a query that returns the specified `id` from the database:

```
 $statement = $connection->prepare('Select * From users
↪ Where id = :id');
```

```
$statement = $connection->prepare('Select * From users Where id =
:id');
```

The above PHP code sends the statement, including the `:id` placeholder, to the database management system. The database management system parses and compiles that query, and based on your configuration, may even cache it for a performance boost in the future. Now, you can pass in the parameter to your database engine and tell it to execute your query:

```
$id = 5;
$statement->execute([
    ':id' => $id
]);
```

Then, you can fetch the result from the statement:

```
$results = $statement->fetchAll(PDO::FETCH_OBJ);
```

The Benefits of Parameter Binding

Now that you are more familiar with how prepared statements work, you can probably guess at their benefits.

`PDO` has taken the task of escaping and quoting the input values you receive from the user out of your hands. For example, now you don't have to write code like this:

```
 $results = mysql_query(sprintf("SELECT * FROM users WHERE
↪ name='%s'",
        mysql_real_escape_string($name)
    )
) or die(mysql_error());
```

Instead, you can say:

```
 $statement = $connection->prepare('Select * FROM users
↪ WHERE name = :name');
$results = $connection->execute([
    ':name' => $name
]);
```

If that isn't short enough for you, you can even make it shorter, by providing parameters that are not `named` - meaning that they are just numbered placeholders, rather than acting like named variables:

```
 $statement = $connection->prepare('SELECT * FROM users
↪ WHERE name = ?');
$results = $connection->execute([$name]);
```

Likewise, having a prepared statement means that you get a performance boost when running a query multiple times. Let's say that we want to retrieve a list of five random people from our `users` table:

```
 $numberOfUsers = $connection->query('SELECT COUNT(*) FROM
↪ users')->fetchColumn();
$users = [];
 $statement = $connection->prepare('SELECT * FROM users
↪ WHERE id = ? LIMIT 1');

for ($i = 1; $i <= 5; $i++) {
    $id = rand(1, $numberOfUsers);
```

```
 $users[] =
↪ $statement->execute([$id])->fetch(PDO::FETCH_OBJ);
}
```

When we first call the prepare function, we tell our DBMS to parse, compile and cache our query. Later on in our `for` loop, we only send it the values for the placeholder - nothing more. This allows the query to run and return quicker, effectively decreasing the time our application would need in order to retrieve the results from the database.

You also might have noticed that I have used a new function in the piece of code above: fetchColumn. As you can probably guess, it returns the value of one column only, and is good for getting scalar values from your query result, such as `count`, `sum`, `min`, `max`, and other functions which return only one column as their result.

Binding Values to an `IN` Clause

Something that has a lot of people stumped when they first start to learn about `PDO` is the `IN` clause. For example, imagine that we allow the user to enter a comma-separated list of names that we store in `$names`. So far, our code is:

```
$names = explode(',', $names);
```

What most people do at this point is the following:

```
 $statement = $connection->prepare('SELECT * FROM users
↪ WHERE name IN (:names)');
$statement->execute([':names' => $names]);
```

This doesn't work - you can only pass in a scalar value (like integer, string, and so on) to prepared statements! The way to do this is - you guessed it - to construct the string yourself.

```
$names = explode(',', $names);
 $placeholder = implode(',', array_fill(0, count($names),
↪ '?'));

 $statement = $connection->prepare("SELECT * FROM users
↪ WHERE name IN ($placeholder)");
$statement->execute([$names]);
```

Despite its scary appearance, line 2 is simply creating an array of question marks that has as many elements as our `names` array. It then concatenates the elements inside that array and places a `,` between them - effectively creating something like `?,?,?,?`. Since our `names` array is also an array, passing it to `execute()` works as expected - the first element is bound to the first question mark, the second is bound to the second question mark, and so on.

Providing Datatypes When Binding Parameters

The techniques we showed above for binding values to parameters are good when you are just starting out to learn `PDO`, but it's always better to specify the type of every parameter you bind. Why?

- Readability: for someone reading your code, it's easy to see what type a variable must be in order to be bound to a parameter
- Maintainability: knowing that the first placeholder in your query must be an integer allows you to catch any errors that slip out. For example, if someone passes a variable containing `test` to your function which will then use that value to search for a particular id as an integer, having a datatype allows you to quickly find the error.
- Speed: when you specify the datatype of the variable, you are telling your database management system that there's no need to cast the variable and that you're providing it the correct type. In this way, you don't have the (small) overhead that comes with casting between datatypes.

To specify the type of each variable, I personally recommend the bindValue function. Let's alter our code above to specify the type of our placeholder:

```
 $numberOfUsers = $connection->query('SELECT COUNT(*) FROM
↪ users')->fetchColumn();
$users = [];
 $statement = $connection->prepare('SELECT * FROM users
↪ WHERE id = ? LIMIT 1');

for ($i = 1; $i <= 5; $i++) {
    $id = rand(1, $numberOfUsers);
    $statement->bindValue(1, $id, PDO::PARAM_INT);
    $statement->execute();
    $users[] = $statement->fetch(PDO::FETCH_OBJ);
}
```

As you can see, the only thing that has changed is our call to `execute()`: instead of passing in the values straight to it, we have bound it first, and have specified that its type is an integer.

`bindValue()`'s Paremeter in the Above

You have probably noticed that we have specified the first parameter to `bindValue()` as `1`. If we were using a named parameter (recommended), we would pass in the name of our parameter (e.g. `:id`). However, in the case of using the `?` as a placeholder, the first argument to `bindValue()` is a number specifying which question mark you are referring to. Be careful - this is a 1-indexed position, meaning that it starts from `1`, not `0`!

Conclusion

As PHP improves, so do the programmers that use it. PDO allows you to write better code. It's agile, fast, easy to read, and a delight to work with, so why not implement it in your own project?

Chapter 9

Vagrant: The Right Way to Start with PHP

by Bruno Škvorc

I often get asked to recommend beginner resources for people new to PHP. And, it's true, we don't have many *truly* newbie friendly ones. I'd like to change that by first talking about the basics of environment configuration. In this piece, you'll learn about the very first thing you should do before starting to work with PHP (or any other language, for that matter).

We'll be re-introducing Vagrant powered development.

Want More Detail?

Note that this topic (among other best practices) is covered in much more depth in SitePoint's Jump Start PHP Environment Book.

Please take the time to read through the entire chapter—I realize it's a wall of text, but it's an important wall of text. By following the advice within, you'll be doing not only yourself one hell of a favor, but you'll be benefitting countless other developers in the future as well. The post will be mainly theory, but in the end we'll link to a quick 5-minute tutorial designed to get you up and running with Vagrant in almost no time. It's recommended you absorb the theory behind it before you do that, though.

Just in case you'd like to rush ahead and get something tangible up and running *before* getting into theory, here's the link to that tutorial.

What?

Let's start with the obvious question - what is Vagrant? To explain this, we need to explain the following 3 terms first:

- Virtual Machine
- VirtualBox
- Provisioning

Virtual Machine

In definitions as simple as I can conjure them, a Virtual Machine (VM) is an isolated part of your main computer which thinks it's a computer on its own. For example, if you have a CPU with 4 cores, 12 GB of RAM and 500 GB of hard drive space, you could turn 1 core, 4 GB or RAM and 20GB or hard drive space into a VM. That VM then thinks it's a computer with that many resources, and is completely unaware of its "parent" system - it thinks it's a computer in its own right. That allows you to have a "computer within a computer" (yes, even a new "monitor", which is essentially a window inside a window - see image below):

A Windows VM inside a Mac OS X system

This has several advantages:

- you can mess up anything you want, and nothing breaks on your main machine. Imagine accidentally downloading a virus - on your main machine, that could be catastrophic. Your entire computer would be at risk. But if you downloaded a virus inside a VM, only the VM is at risk because it has no real connection to the parent system it lives off of. Thus, the VM, when infected, can simply be destroyed and re-configured back into existence, clean as a whistle, no consequences.
- you can test out applications for other operating systems. For example, you have an Apple computer, but you really want that one specific Windows application that Apple doesn't have. Just power up a Windows VM, and run the application inside it (like in the image above)!
- you keep your main OS free of junk. By installing stuff onto your virtual machine, you avoid having to install anything on your main machine (the one on which the VM is running), keeping the main OS clean, fast, and as close to its "brand new" state as possible for a long time.

You might wonder - if I dedicate that much of my host computer to the VM (an entire CPU core, 4GB of RAM, etc), won't that:

- make my main computer slower?
- make the VM slow, because that's kind of a weak machine?

The answer to both is "yes" - but here's why this isn't a big deal. You only run the VM when you need it - when you don't, you "power it down", which is just like shutting down a physical computer. The resources (your CPU core, etc.) are then instantly freed up. The VM being slow is not a problem because it's not meant to be a main machine - you have the host for that, your main computer. So the VM is there only for a specific purpose, and for that purpose, those resources are far more than enough. If you really need a VM more powerful than the host OS, then just give the VM more resources - like if you want to play a powerful game on your Windows machine and you're on a Mac computer with 4 CPU cores, give the VM 3 cores and 70-80% of your RAM - the VM instantly becomes powerful enough to run your game!

But, how do you "make" a virtual machine? This is where software like VirtualBox comes in.

VirtualBox

VirtualBox is a program which lets you quickly and easily create virtual machines. An alternative to VirtualBox is VMware. You can (and should immediately) install VirtualBox here.

VirtualBox provides an easy to use graphical interface for configuring new virtual machines. It'll let you select the number of CPU cores, disk space, and more. To use it, you need an existing image (an installation CD, for example) of the operating system you want running on the VM you're building. For example, if you want a Windows VM as in the image above, you'll need a Windows installation DVD handy. Same for the different flavors of Linux, OS X, and so on.

Provisioning

When a new VM is created, it's bare-bones. It contains nothing but the installed operating system - no additional applications, no drivers, nothing. You still need

to configure it as if it were a brand new computer you just bought. This takes a lot of time, and people came up with different ways around it. One such way is *provisioning*, or the act of using a pre-written script to install everything for you.

With a provisioning process, you only need to create a new VM and launch the provisioner (a provisioner is a special program that takes special instructions) and everything will be taken care of automatically for you. Some popular provisioners are: Ansible, Chef, Puppet, etc - each has a special syntax in the configuration "recipe" that you need to learn. But have no fear - this, too, can be skipped. Keep reading.

Vagrant

This is where we get to Vagrant. Vagrant is another program that combines the powers of a provisioner and VirtualBox to configure a VM for you.

You can (and should immediately) install Vagrant here.

Vagrant, however, takes a different approach to VMs. Where traditional VMs have a graphical user interface (GUI) with windows, folders and whatnot, thus taking a long time to boot up and become usable once configured, Vagrant-powered VMs don't. Vagrant strips out the stuff you don't need because it's *development oriented*, meaning it helps with the creation of development friendly VMs.

Vagrant machines will have no graphical elements, no windows, no taskbars, nothing to use a mouse on. They are used exclusively through the terminal (or command line on Windows - but for the sake of simplicity, I'll refer to it as the terminal from now on). This has several advantages over standard VMs:

1. Vagrant VMs are brutally fast to boot up. It takes literally seconds to turn on a VM and start developing on it.
2. Vagrant VMs are brutally fast to use - with no graphical elements to take up valuable CPU cycles and RAM, the VM is as fast as a regular computer
3. Vagrant VMs resemble real servers. If you know how to use a Vagrant VM, you're well on your way to being able to find your way around a real server, too.
4. Vagrant VMs are very light due to their stripped out nature, so their configuration can typically be much weaker than that of regular, graphics-

powered VMs. A single CPU core and 1GB of RAM is more than enough in the vast majority of use cases when developing with PHP. That means you can not only boot up a Vagrant VM on a very weak computer, you can also boot up several and still not have to worry about running out of resources.

5. Perhaps most importantly, Vagrant VMs are destructible. If something goes wrong on your VM - you install something malicious, you remove something essential by accident, or any other calamity occurs, all you need to do to get back to the original state is run two commands: `vagrant destroy` which will destroy the VM and everything that was installed on it after the provisioning process (which happens right after booting up), and `vagrant up` which rebuilds it from scratch and re-runs the provisioning process afterwards, effectively turning back time to before you messed things up.

With Vagrant, you have a highly forgiving environment that can restore everything to its original state in minutes, saving you hours upon hours of debugging and reinstallation procedures.

Why?

So, why do this for PHP development in particular?

1. The ability to test on several versions of PHP, or PHP with different extensions installed. One VM can be running PHP 5.5, one can be running PHP 5.6, one can be running PHP 7. Test your code on each - no need to reinstall anything. Instantly be sure your code is cross-version compatible.
2. The ability to test on several servers. Test on Apache in one VM, test on Nginx in another, or on Lighttpd on yet another - same thing as above: make sure your code works on all server configurations.
3. Benchmark your code's execution speed on different combinations of servers + PHP versions. Maybe the code will execute twice as fast on Nginx + PHP 7, allowing you to optimize further and alert potential users to possible speed gains.
4. Share the same environment with other team members, avoiding the "it works on my machine" excuses. All it takes is sharing a single Vagrantfile (which contains all of the necessary configuration) and everyone has the *exact same setup as you do*.

5. Get dev/prod parity: configure your Vagrant VM to use the same software (and versions) as your production (live) server. For example, if you have Nginx and PHP 5.6.11 running on the live server, set the Vagrant VM up in the exact same way. That way, you're 100% certain your code will instantly work when you deploy it to production, meaning *no downtime* for your visitors!

These are the main but not the only reasons.

But why not XAMPP? XAMPP is a pre-built package of PHP, Apache, MySQL (and Perl, for the three people in the world who need it) that makes a working PHP environment just one click away. Surely this is better than Vagrant, no? I mean, a single click versus learning about terminal, Git cloning, virtual machines, hosts, etc…? Well actually, it's much worse, for the following reasons:

1. With XAMPP, you absorb *zero* server-config know-how, staying 100% clueless about terminal, manual software installations, SSH usage, and everything else you'll one day desperately need to deploy a real application.
2. With XAMPP, you're never on the most recent version of the software. It being a pre-configured stack of software, updating an individual part takes time and effort so it's usually not done unless a major version change is involved. As such, you're always operating on something at least a little bit outdated.
3. XAMPP forces you to use Apache. Not that Nginx is the alpha and omega of server software, but being able to at least test on it would be highly beneficial. With XAMPP and similar packages, you have no option to do this.
4. XAMPP forces you to use MySQL. Same as above, being able to switch databases at will is a great perk of VM-based development, because it lets you not only learn new technologies, but also use those that fit the use case. For example, you won't be building a social network with MySQL - you'll use a graph database - but with packages like XAMPP, you can kiss that option goodbye unless you get into additional shenanigans of installing it on your machine, which brings along a host of new problems.
5. XAMPP installs on your host OS, meaning it pollutes your main system's space. Every time your computer boots up, it'll be a little bit slower because of this because the software will load whether or not you're planning to do some development that day. With VMs, you only power them on when you need them.

6. XAMPP is version locked - you can't switch out a version of PHP for another, or a version of MySQL for another. All you can do is use what you're given, and while this may be fine for someone who is 100% new to PHP, it's harmful in the long run because it gives a false sense of safety and certainty.
7. XAMPP is OS-specific. If you use Windows and install XAMPP, you have to put up with the various problems PHP has on Windows. Code that works on Windows might not work on Linux, and vice versa. Since the vast, vast majority of PHP sites are running on Linux servers, developing on a Linux VM (powered by Vagrant) makes sense.

There are many more reasons not to use XAMPP (and similar packages like MAMP, WAMP, etc), but these are the main ones.

How?

So how does one power up a Vagrant box?

The first way, which involves a bit of experimentation and downloading of copious amounts of data is going to Hashicorp's Vagrant Box list here, finding one you like, and executing the command you can find in the box's details. For example, to power up a 64bit Ubuntu 14.04 VM, you run: `vagrant init ubuntu/trusty64` in a folder of your choice after you installed Vagrant, as per instructions. This will download the box into your local Vagrant copy, keeping it for future use (you only have to download once) so future VMs based off of this one are set up faster.

Note that the Hashicorp (which, by the way, is the company behind Vagrant) boxes don't have to be bare-bones VMs. Some come with software pre-installed, making everything that much faster. For example, the laravel/homestead box comes with the newest PHP, MySQL, Nginx, PostgreSQL, etc pre-installed, so you can get to work almost immediately (more on that in the next section).

Another way is grabbing someone's pre-configured Vagrant box from Github. The boxes from the list in the link above are decent enough but don't have everything you might want installed or configured. For example, the homestead box does come with PHP and Nginx, but if you boot it up you won't have a server configured, and you won't be able to visit your site in a browser. To get this, you

need a provisioner, and that's where Vagrantfiles come into play. When you fetch someone's Vagrantfile off of Github, you get the configuration, too - everything gets set up for you. That brings us into HI.

Hi!

HI (short for Homestead Improved) is a version of laravel/homestead. We use this box at SitePoint extensively to bootstrap new projects and tutorials quickly, so that all readers have the same development environment to work with. Why a version and not the original homestead you may wonder? Because the original requires you to have PHP installed on your host machine (the one on which you'll boot up your VM) and I'm a big supporter of cross-platform development in that you don't need to change *anything* on your host OS when switching machines. By using Homestead Improved, you get an environment ready for absolutely any operating system with almost zero effort.

The gif above where I boot up a VM in 25 seconds - that's a HI VM, one I use for a specific project.

I recommend you go through this quick tip to get it up and running quickly. The first run might take a little longer, due to the box having to download, but subsequent runs should be as fast as the one in my gif above.

Please do this now - if at any point you get stuck, please let me know and I'll come running to help you out; I really want everyone to transition to Vagrant-driven-development as soon as possible.

Conclusion

By using HI (and Vagrant in general), you're paving the way for your own cross-platform development experience and keeping your host OS clean and isolated from all your development efforts.

Below you'll find a list of other useful resources to supercharge your new Vagrant powers:

- SitePoint Vagrant posts - many tutorials on lots of different aspects of developing with Vagrant, some explaining the links below, some going beyond that and diving into manually provisioning a box or even creating your own, and so on.
- StackOverflow Vagrant Tag for questions and answers about Vagrant, if you run into problems setting it up
- PuPHPet - a way to graphically configure the provisioning of a new Vagrant box to your needs - select a server, a version of PHP, a database, and much more. Uses the Puppet provisioner. Knowledge of Puppet not required.
- Phansible - same as PuPHPet but uses the Ansible provisioner. Knowledge of Ansible not required.
- Vaprobash a set of Bash scripts you can download (no provisioner - raw terminal commands in various files that just get executed) as an alternative to the above two. Requires a bit more manual work, but usually results in less bloated VMs due to *finetuneability*.
- 5 ways to get started with Vagrant - lists the above resources, plus some others.