Design Patterns

with PHP & Laravel

Kelt Dockins

Design Patterns with PHP and Laravel

Kelt Dockins

This book is for sale at http://leanpub.com/larasign

This version was published on 2015-06-29

Tweet This Book!

Please help Kelt Dockins by spreading the word about this book on Twitter!

The suggested tweet for this book is:

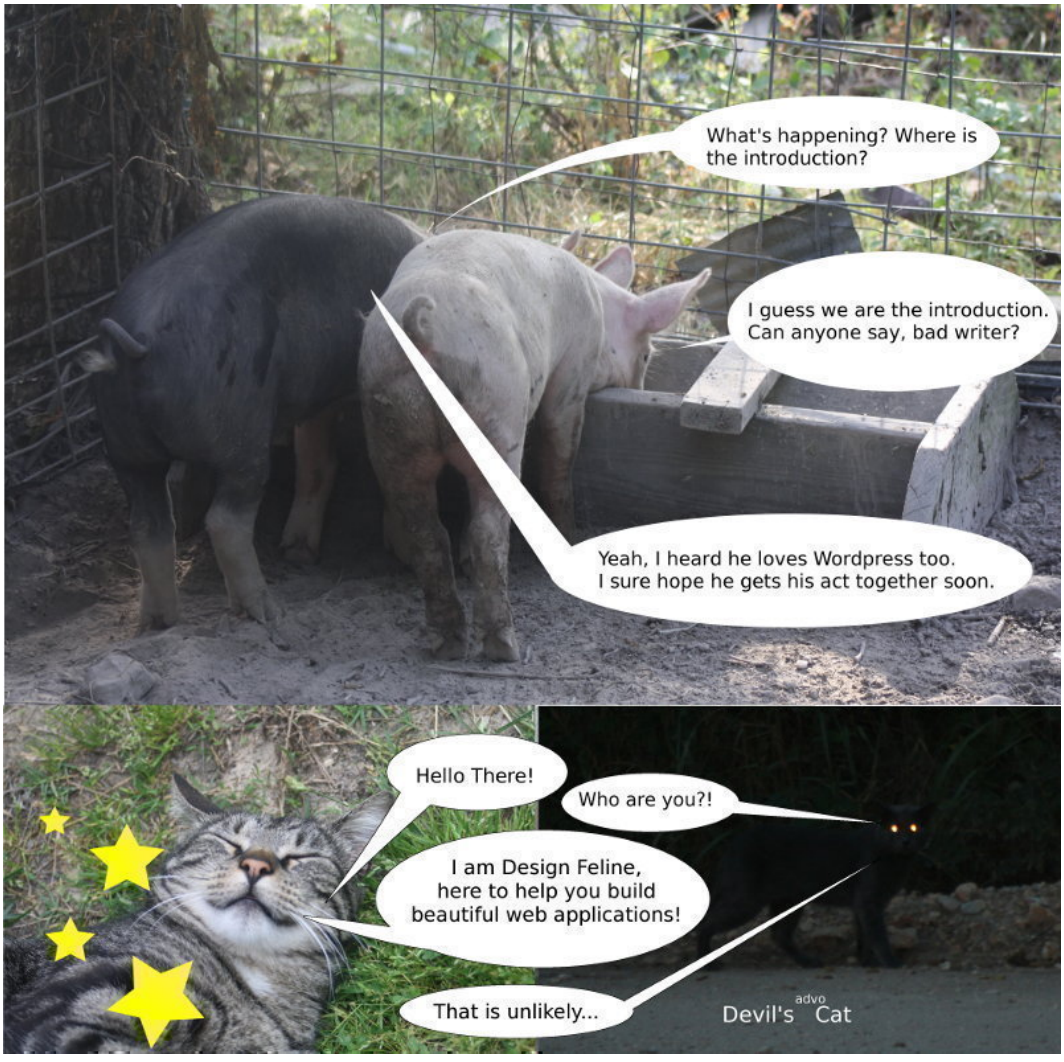
I just bought design patterns with @laravelphp by @kdocki at https://leanpub.com/larasign #larasign

The suggested hashtag for this book is #larasign.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#larasign

Contents



Introduction

Hello there. I see you've meet a few characters that will be in this book. Don't worry, there will be plenty more where that came from!

Please allow me to introduce myself. I'm Kelt. Together we are going to explore design patterns together. Not only that but we will learn more about php and Laravel. If you're looking to improve your knowledge on any of these subjects then hopefully this book will help you. Also be forewarned, there is a ton of really bad humor in

this book. Here is an example.

My pappy once told me, two days ago, that all great books have at least three things. He asked if I knew what those three things were. I replied,

1. words
2. sentences
3. paragraphs

Wrong! as he slapped me with a trout. *You fail!* he barked! He corrected,

1. cuss **words**
2. death **sentences**
3. **para**phernalia, sex, drugs and alcohol

"But that's more than thre…", I questioned him, but I was reprimanded with yet another *trout slap*. "Okay," I said. I quickly shut my yapper. **Reader beware**, this book has all those things. *Don't let your kids read this* unless they are over thirty and living in your basement. In that case tell them to read this book and go get a job.

Disclaimer: **I don't do drugs**. This book might make you think otherwise but honestly, I don't need drugs to act stupid. Yes, really.

Who is this book for

Anyone with a sawbuck can be the proud owner of this book. That's the only requirement. But it will help a lot if you already know at least a little bit about php or Laravel. If you work on web apps and you've ever found yourself cussing at your past self from 6 months ago then you'll probably enjoy this book.

You will not be interested in this book if:

1. You are a Zombie who eats cats
2. You don't have 10 dollars
3. You'd rather use assembly language than php
4. You invented the internet
5. You can bench press 624lbs, exactly
6. You hate reading really long lists
7. You think cows should never be given a typewriter
8. You think David Hassellhoff is cute
9. You were born yesterday
10. You run marathons…
11. … backwards
12. You've never jumped on a bed
13. You like to spit on pigs
14. You didn't see Watchmen movie only because of the changes made to the story
15. You would use php-snow[1] for every project, ever
16. You think Ents are dumb… in fact if …
17. You hate Lord of the Rings, stop now and I'll give you your money back. HATER.
18. You pass gas in elevators and smile wildly at other people at the same time
19. … that last one only counts on elevators in buildings over 50 stories high

[1] https://code.google.com/p/php-snow/

20. You work for the FBI or CIA - you are cool but please don't read my book
21. You have had unusual thoughts about Smurfette, shame on you and shame on me
22. You count sheep to stay awake
23. You can't read
24. You still blog about TV series Full House
25. You troll books (don't ask me how)
26. You hold deeconversations with Nick Jr. Face[2]
27. You believe aliens don't exist
28. You work for the FBI, gonna list twice just in case you didn't catch it the first time
29. You walk like a penguin and evilly plot against Batman
30. You were born the day before yesterday
31. You didn't watch the Hunger Games because you've already read the book
32. You think Zelda is a dude in a green outfit
33. You will code in Rails until the day you die (good for you, good for you)
34. You are not a Zombie but still eat cats
35. You like to wear dresses in winter and you name is Pat
36. You can ignore the previous line about dresses (I have a friend… who does this)
37. You can find your entire life story in a Dr. Seuss book
38. You go to the library, find people there and tell them spoilers to great novels
39. You don't quite understand peaches
40. You've never seen Back to the Future
41. You are in a Josie and the Pussycats cover band
42. You think Michael Bay should direct every movie, ever
43. You are Michael Bay
44. You visited the moon and didn't bring me back a moon rock (jerk)
45. You can do one hundred consecutive push ups (you are too bad-ass to be reading this book)
46. You were born tomorrow
47. You think Chewbacca is chewing tabacco

[2] https://www.youtube.com/watch?v=LaXIL8QMDzU

Did you really read this entire list? Awesome! I want to talk about gloves. Design patterns are like gloves. If it fits, wear 'em. On the flipside, don't try to wear gloves that don't fit. Learning **when** to wear your design gloves is important. It is my hope that with the aid of this book you'll understand when to leave the gloves off or put the gloves on. We are almost ready to get started with Laravel basics but first let's go over the structure of this book.

Layout of the book

Our first chapter provides us with some laravel basics to get started coding. The next chapter we will cover elementary OO concepts. From there on out, we start learning different patterns and applying them in the context of Laravel and php.

I created a branch for every pattern in a git repository. You can view the git repository at https://github.com/kdocki/larasign[3]. At the beginning of every chapter will be git command to checkout that chapter's relevant code samples. You will need to clone this repository down if you want to follow along with code examples.

```
$> git clone git@github.com:kdocki/larasign.git
```

This book organizes the patterns in a similar fashion as the Gang of Four book. The GoF patterns book came out 20 years ago in 1994 and the patterns are still being seen and talked about even in 2014. To me - that is awesome. As you learn some of these patterns we will also be using the php framework Laravel and hopefully pick up little bits and pieces of the framework along the way. You will see that the Laravel MVC framework let's us write quality code. In this book we will cover these patterns.

Creational

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Simple Singleton
- Simple Factory

[3] https://github.com/kdocki/larasign

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Laravel basics

If you're going to be doing php development, you should download Laravel. Symfony and Laravel are the most popular and best frameworks for php in the world. To get Laravel on your machine you can follow the instructions on the quick start page[4]. You will need php5 with the curl & mcrypt extensions enabled. You know how to install those right? Awesome. But just in case, on ubuntu:

Install php and a few dependencies

```
> sudo apt-get install php5 php5-curl php5-mcrypt
```

Now create a new laravel application inside the folder named designpatterns. As we build out various applications in the book I will make a git branch for each chapter so you git Jedi's can trace along.

The next thing we will do is look at composer. Laravel is built off of twenty-odd-something composer packages - composer is the cat's me-ow.

[4] http://laravel.com/docs/quick

Meow

What is Composer?

Composer is **the** dependency management tool for php. It allows you to list the packages your application depends upon to function correctly. A file in the root of the project named `composer.json` allows for plenty of configuration options, so let's brush over some of those.

Composer does several neat things like

- dependency management with packages
- PSR and custom file based autoloading
- compiler optimization to help code run faster

- custom hooks into life-cycle events, e.g. - application installed, updated or first created.
- stability checking

With your favorite text editor, open up the `composer.json` file inside of the project root. Note that through out this book file names will all be relative to the project root. Just to be clear here, when I say **project root**, that means directly inside the `designpatterns` folder we created, e.g. - `app/models/User.php` is actually the path `/home/kelt/book/designpatterns/app/models/User.php` on my machine.

Meta Information

In the first part of the composer manifest, we see basic meta information.

composer.json

```
"name": "laravel/laravel",
"description": "The Laravel Framework.",
"keywords": ["framework", "laravel"],
"license": "MIT",
```

All this above information is used by a website called Packagist[5] which catalogs packages out there in the wild. As a standard practice, if you create packages to host on packagist you'll probably want the `name` the same as the github repository for that package.

Dependency management

Next we see a require block. Here is where package dependency management comes into play. Currently we are only requiring the laravel framework which is made up of many other packages; however, as time goes on we will add additional packages.

[5] http://packagist.org

composer.json

```
"require": {
        "laravel/framework": "4.1.*"
},
```

Seems pretty straight forward, right? One gotcha here though is that you might see a `~4.1` or `>=1.0,<1.1 | >=1.2`. Visiting this getcomposer.org page[6] explains the different rules for versions.

Name	Example	Description
Exact version	`1.0.2`	You can specify the exact version of a package.
Range	`>=1.0`	By using comparison operators you can specify ranges of valid
	`>=1.0,<2.0`	versions. Valid operators are `>`, `>=`, `<`, `<=`, `!=`.
	`>=1.0,<1.1 | >=1.2`	You can define multiple ranges, separated by a comma, which will be treated as a **logical AND**. A pipe symbol `|` will be treated as a **logical OR**. AND has higher precedence than OR.
Wildcard	`1.0.*`	You can specify a pattern with a `*` wildcard. `1.0.*` is the equivalent of `>=1.0,<1.1`.
Tilde Operator	`~1.2`	Very useful for projects that follow semantic versioning. `~1.2` is equivalent to `>=1.2,<2.0`.

Although it isn't shown here, you could add a mapping for development only packages by using `require-dev`. Some good candidates for development only packages are **phpunit**, **phpspec** and **clockwork**.

[6] https://getcomposer.org/doc/01-basic-usage.md#package-versions

Autoloading

Eariler I mentioned that composer comes with an autoloader and even optimizes the php to run faster. It knows how to do this because of the `autoload` section.

composer.json

```
"autoload": {
        "classmap": [
                "app/commands",
                "app/controllers",
                "app/models",
                "app/database/migrations",
                "app/database/seeds",
                "app/tests/TestCase.php"
        ]
},
```

One downside of using `classmap` is that anytime we add files to a directory then we need to run `composer dumpautoload` to regenerate the class mappings inside of the file `vendor/composer/autoload_classmap.php`.

Wouldn't it suck if every time we added a new file in `app/models` we had to go run `composer dumpautoload`. Thankfully, this issue is solved for us by a ClassLoader that works in conjunction with composer.

app/start/global.php

```
14 ClassLoader::addDirectories(array(
15
16         app_path().'/commands',
17         app_path().'/controllers',
18         app_path().'/models',
19         app_path().'/database/seeds',
20
21 ));
```

You can also use PSR autoloading. If you've never heard of PSR before then you can take a moment and visit this article[7]. Basically it deals with standardizing the folder structure, namespace and class names of our php.

We are going to follow in the footsteps of Java and create a `src/` directory in the root of our Laravel application. This folder will allow us to focus strictly on a design pattern without interfering with Laravel specific nomenclature.

composer.json

```
"autoload": {
        "classmap": [ ... omitted ... ],
        "psr-4" : {
                "": "src/"
        }
}
```

Now our application will look inside of the `src` folder and autoload any files from that directory for us. Pretty nifty right?

Lifecycle Hooks/Scripts

Below are a list of scripts we execute after running `composer install` or `composer update` or `composer create-project` (respectively).

[7] http://petermoulding.com/php/psr

composer.json

```
"scripts": {
        "post-install-cmd": [
                "php artisan clear-compiled",
                "php artisan optimize"
        ],
        "post-update-cmd": [
                "php artisan clear-compiled",
                "php artisan optimize"
        ],
        "post-create-project-cmd": [
                "php artisan key:generate"
        ]
},
```

We can tap into certain events of composer if we want to run custom commands here. I use these hooks to automatically run things like migrations anytime composer install is executed on the server. When we deploy to production servers we just follow this simple two-step process.

1. `git pull`
2. `composer install`

We don't have to remember to run migrations or clean assets or whatever else because we do that anytime after composer install finishes running.

What is the difference between `composer install` and `composer update`?

Running `composer update` will do two things.

1. update all required packages to the latest version matched.
2. update composer.lock with exact versions of dependencies.

Running `composer install` will install the dependencies listed in `composer.lock`. If the lock file does not exist then this command becomes identical to `composer update` as it will create the composer.lock file for us after downloading dependencies.

Why do we do this?

Let's say we ran composer on **machine 1** and later on **machine 2**. We need to ensure packages are exactly the same on both machines. If we run `composer update` the versions of packages could very well differ machine to machine. This can cause problems. Imagine if a particular package we required as a dependency changes a feature. Suddenly **machine 2** is throwing a big fat **500 Internal Server Error** while **machine 1** is still working fine. We should always want to avoid that kind of behavior.

Ideally we want our `production`, `staging` and various `local` environments to be as similar as possible.

We could fix this problem by removing `vendor` folder from `.gitignore` and committing **EVERYTHING** but there is a better way. The lock file will already have the specific github commit hashes which should not change and we can use this to our advantage by following the basic principle of..

Only run `composer update` on your local development box. Never on production.

Stability

We come the end of our `composer.json` file.

composer.json

```
"config": {
        "preferred-install": "dist"
},
"minimum-stability": "stable"
```

Composer can fetch your dependencies through source code or a distributed zip file. This config option is telling composer to use prefer distribution files over source code. You can read more about config options here[8].

The `minimum-stability` flag is used to keep other packages from inherently installing unstable versions of other packages into your application.

- You **require** package A.
- Package A **requires** package B@dev
- Composer will squawk about it

The squawking happens because our stability is set to `stable` but a sub package is dependent on an less stable version. In this case, package A depends on a development version of package B.

How would we fix this? In this scenario, in order to install package A we need to explicitly add package B@dev to the `require` array. Another way to fix this, for the more brave developers only, change the minimum-stability to one of the following: `dev`, `alpha`, `beta`, `RC`, or `stable`.

Running Composer

You can download composer via the website http://getcomposer.org/[9]. Once installed, verify your installation by running

[8] https://getcomposer.org/doc/04-schema.md#config
[9] http://getcomposer.org/

```
> composer -v
```

There are a lot of commands you can run with composer. One example is instead of editing your `composer.json` with a text editor, you can run the composer command to require dependencies.

```
> composer require
```

Another good composer command to use is `validate`. Despite all my JavaScript training, I still manage to leave trailing commas which is invalid json, so this is a good practice to validate your composer.json file after changes.

Setting Up Your Environment Variable

When you get an error in Laravel, if the debug mode is not enabled then we will see a generic error page that simply says, "Whoops, something went wrong." If we turn on debug mode in `app/config/app.confg` we get a stack trace that is very helpful to debug our application when things go wrong.

Only when developing locally do we want debug mode enabled and thankfully Laravel makes it super easy to have separate config files for different environments. So let's create a file called `app/config/local/app.php` and inside of it turn on debug. It's okay to erase all the other settings, as Laravel is smart enough to merge the local settings in `app/config/local/app.php` with the global configuration in `app/config/app.php`.

We can do the same thing for database.php and all the other config files. For example, when you are working on a team of developers, if you wanted to use a database username and password on your development machine without changing the app/config/app.php for everyone else on your team this would be perfect place to use the local database config instead. Verify that `/app/config/local` is added to the .gitignore file so your local folder doesn't get shared with everyone else on the team.

When you need to merge an array between two different environment configs[10] use `append_config`. This merges two array configurations together instead of overriding

[10] http://laravel.com/docs/4.2/configuration

the original array. This is useful for adding a few service providers in your app configuration for your local environment. Another not so commonly known feature is that you can override production values by creating a production folder. This might be handy to let your system admin know.

Now that you know how to take advantage of environments in Laravel, how do we actually do it? To turn on this awesome feature, we only need to tell Laravel the environment we are running on.

Out of the box, Laravel expects you to use machine names but personally I've found this to be inflexible at times, especially when working on a team, each member with various machine names. It becomes easier just to use an system-wide environment variable called `LARAVEL_ENV`. If you know your machine name, you can skip this step, otherwise go ahead and change your bootstrap start file to the following below:

bootstrap/start.php

```
1  $env = $app->detectEnvironment(function() {
2
3          return getenv('LARAVEL_ENV') ?: 'local';
4
5  });
```

Now you should set up your environment variable and use the value *local*. On my ubuntu machine I just add a line to my bashrc file which executes anytime I login.

/home/kelt/.bashrc

```
LARAVEL_ENV="local"
```

You can always checkout what environment you are currently in by using `App::environment()`. This brings us to our next tool in Laravel which is the die and dump method - `dd(...)`. So open up the routes.php file and add the following line at the end of the file:

app/routes.php

```
dd(App::environment());
```

If we run php artisan, we should see a die and dump of our current environment.

Testing out our environment setups

```
> export LARAVEL_ENV="fooly cooly"; php artisan
> string(11) "fooly cooly"
```

Make sure to remove the die and dump from your routes.php file. Feel free to experiment here with different configuration files and environments. Die and dump is very useful for quickly debugging in Laravel. It is not a replacement for good tests or Xdebug though. Now that we've covered some basics in Laravel let's continue onto SOLID principles.