DEPLOYING PHP APPLICATIONS

BY NIKLAS MODESS

Deploying PHP Applications

Niklas Modess

Tweet This Book!

Please help Niklas Modess by spreading the word about this book on Twitter!

The suggested hashtag for this book is #deployphpapps.

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=#deployphpapps

Contents

Introduction

Background

The PHP community is in an interesting phase right now and it has for a long time lagged behind on the more traditional software development practices. Practices that have been more or less considered as given in other communities. Continuous integration, package management, dependency injection and adopting object oriented programming to its extent to name a few. But as of PHP 5.3 there are no more excuses to why you can't write modern and clean code. The community have responded to this and a great number of people have stepped forward in supporting this through teaching and building tools for accomplishing these practices.

Yet we seem to forget deployment in this and I feel it's time to bring it to the table for discussion. There are a lot of great tools and services that aid you out there but few resources are available on how to build, maintain and optimize your deployment process. In this book I hope to shed some light on what is important and how it can be achieved.

Deploying is not just about pushing changes to a production server but an important part of the software development process[1]. You need to think of every step and every application is unique and need to be dealt with according to that. When working in a team is when you most of all need a great deployment process and the team should work in certain ways to allow for the process to be beneficial for all parties involved.

Every software application has a life cycle that the deployment process supports and is necessary for. You want to be able to maintain the application, roll out new features and do bug fixes without constant pain and headaches while doing it. You should be able to manage your branches, push your code, run the appropriate tests, migrate your database and deploy the changes in a fast and confident manner. With a good deployment process and a work flow that enables this it will be a breeze. If you can do a fast and easy rollback too, your confidence in pushing code will benefit a lot.

In a more and more agile software development world being able to deploy is important. Release cycles are getting shorter and shorter and some organisations even push it to the limit with continuous delivery[2]. In an environment with short release cycles, the importance of the deployment process intensifies. Not being able to deploy or rollback fast enough could end up slowing you down your entire development.

[1]http://en.wikipedia.org/wiki/Software_development_process

[2]http://en.wikipedia.org/wiki/Continuous_delivery

Who it's for

You are already familiar with PHP and you're not afraid of the command line. But you could also be a manager of a software development team that deal with deployment on a regular basis.

Legacy is not only a code issue but a process issue as well. Are you looking to streamline your deployment process or you want to scrap your current one and start of fresh? Then this book is for you.

Outside its scope

I consider server provisioning an almost crucial part of deploying your application but it's too big of a topic for this book. It could without a doubt be a book on its own (and perhaps will be?). I also don't want this book to focus on any framework or tool but keep the content and name broad instead of something like *Deploying PHP application to Amazon Web Services with Chef*.

The tools and commands used will be outside the scope unless it is a deployment tool (then it will have a dedicated section). I will make examples with Git, Composer, Grunt, PHPUnit and various other tools and if you want to learn more about the tools there are a vast amount of books, screencasts and blog posts to find, Google is your friend.

Assumptions

I know it's not nice to make assumptions about people or software. But I'm still going to do it to some extent in this book. I will make them when I approach examples but I'll not judge you or your application in any way.

Where you deploy to

You will need a hosting environment that you are some what in control of. If you are not able to install software or run commands it will limit you in what you can achieve. Whether it is a hosted server, co-location server or a VPS does not matter but to use everything in this book you need some control over it in terms of installing software, changing configuration, etc.

Git

The base of some of the topics discussed will use Git as version control. Why? Because I think it enables work flows that is best suitable for a good deployment process. There is a chapter on the topic as Git about version control and branching strategy for a good deployment process that could've been named *Git version control*. But I'll leave the name without Git in it since there are perhaps a lot in there that you can apply to other version control systems as well. Other than Git I have worked with Subversion and Perforce but when I found Git and started incorporating it in my work flow I have never looked back.

Both ends

There will also be an assumption about your application that it is not only a backend application. If your application is a REST API for example with no frontend what so ever it will not matter though. I will give some general examples on how to manage builds for your frontend as well but all the commands used will be arbitrary.

About the author

I have been developing PHP applications for over 10 years now. During this time I've developed and deployed a great variety of applications. The scale of these applications have been from a few hundred users to over 250 million users. Currently I'm working as a kind of free agent consultant focused on web architecture while spending the rest of my time on my own projects.

I'm one of two co-organizers for the meetup group Laravel Stockholm[3] trying to get some nice people together discussing Laravel and PHP concepts in general. I contribute to open source projects as much as possible. Most that is developing and maintaining Git Pretty Stats[4] which is a self hosted tool for statistics and graphs of Git repositories.

Oh, by the way I'm from Stockholm, Sweden. So I'm writing a book in my second language and I would appreciate all the help I can get when it comes to spelling and grammar. If you find anything, please create an issue in this repository[5] or fork it, fix it and send me a pull request. Thank you!

Code samples

In the repository on github[6], you can find code samples structured by chapter. Any substantial amount of code used in the book is available there for reference and use.

Thanks to

My friend and talented designer extraordinaire **Joakim Unge** for the awesome cover image. Look at it, it's a fucking rocket ship! You can find his portfolio at www.ashbagraphics.com[7].

My sister and **Jenny Modess** the talented copywriter, for proof reading from a non-technical perspective. Keeping my spelling, grammar and storytelling in check!

[3]http://www.meetup.com/Laravel-Stockholm/
[4]https://github.com/modess/git-pretty-stats
[5]https://github.com/modess/deploying-php-applications
[6]https://github.com/modess/deploying-php-applications
[7]http://www.ashbagraphics.com

1. Automate, automate, automate

I want to get this off the bat right away since one of the most crucial ingredients in your deployment process should be to **automate everything** in order to **minimize human errors**. If your deployment process involves manual steps, you're going to have a bad time and shit will hit the inevitable fan. Nobody is perfect. People can, and will, forget if they need to remember.

An automated deployment process will boost the confidence for the person deploying. Knowing that everything will be taken care of is a major contributor to feeling safe during a deploy. Of course other unexpected issues can arise but with a flexible process with logging and notifications you can relax and figure out what went wrong.

1.1 One button to rule them all

You should have **one** button to push or **one** command to run, to deploy your application. If you don't, something is wrong and you should automate all steps. Perhaps the only manual intervention I would consider okay is *"Are you sure? This will push stuff to production. [N/y]"*. This might be a bold statement but I truly believe in it.

Even if you're the only developer on a project and you're deploying the application each time I would say it's a bad practice having manual steps. When it comes to teams it becomes a lot worse since if not all team members are familiar with the deployment steps they won't be able to deploy. Team members come and go and whenever a new team member arrive they will need to learn how to deploy in the correct manner. Sure there can be documentation for it. But whenever someone is familiar enough with it they will most likely start to deploy without it.

1.2 Example of a manual step

Let's take an example where you have a revision number of your assets in code. I would say this is a rather common practice (unfortunately) and I've seen it on many occasions. This revision number handles cache busting so browsers do not keep serving old assets after a deploy.

This definition is a constant in a class somewhere for managing static assets.

```
class Assets
{
    const REVISION = 14;

    // [...]
}
```

Then it’s applied to the static assets.

```
<link rel="stylesheet" type="text/css" href="style.css?v=<?=Assets::REVISION?>">
```

This is truly bad manual steps you could have. It is easy to forget since it requires a code change and if it’s forgotten it might break the users’ experience serving cached and out dated assets. A manual step where you would have to run a command in connection to the deploy would be better since it would be easier to remember. When you are already have the command line in front of you will be more prone to remember it.

Since it requires a code change another issue will likely occur. That is when you remember the step in the middle of the deploy just before pushing changes to production. You stop yourself before pressing the button screaming “Shit, the assets revision number!”. Now you have to deal with changing the code, committing it and push the code through a new deploy. In a perfect world you would’ve already merged a release branch and tagged it (discussed in chapter 2), so how would you approach that now? Reset your repository, remove the tag, commit and create the release branch again? Or would you just commit and push, knowing that it will break traceability[8] for this specific deploy? This is a problem that goes away with automation.

1.3 Time is always a factor

You might be prone to say that you don’t have time automating all the trivial steps or commands. If you spend one hour automating a command that takes one second to run you would have to run that command 3601 times before you have saved time on it. Yes, I did the math. While this is true I would say that it isn’t the whole truth.

We need to take into account the time spent on dealing with issues arising when forgetting the step or command. Time spent on cleaning up. Can you measure bad user experience when your application breaks or behaves incorrect? Most likely not. In the previous example you can’t tell how much time that you will spend on correcting that error and neither can you tell the impact on the users. If the problem is not caught in time it could persist for a long time.

[8]http://en.wikipedia.org/wiki/Traceability#Software_development

2. Goals

I really hope you have decided that you want to improve your deployment process. Well, that is probably why you're reading this book at this moment. You should at this point probably establish a few things. Ask yourself the questions of where you currently are, and where you would like to be. These are essential questions in reaching goals, between here and there you should probably set a few milestones as well since sensing that you achieve something is always important. This is starting to sound like a self help book, but I'm of course talking about where your deployment process is and where you would like it to be.

We have a few things we need to go through before we get to the actual list of goals. Understanding the background of the various parts will help you better grok the big picture.

2.1 How it usually begins

Take a deep breath and imagine a peaceful lake in your mind. No just kidding, stop with that. This is still not a self help book.

Since the deployment process is just like any other process in the software development cycle it will mature in certain ways. It usually starts at a fairly similar place. This place has seen a lot of developers and it will continue to see a lot of other developers pass by. What I want is to show people what kind of place that is and why you should try avoiding to go there. The main reason why that place is bad is because it's **at the end** of everything. I realize that I just said that the beginning is at the end, but let me explain what I mean before you call me an idiot.

An application usually starts with an idea. Then some wireframes (either mental or actual ones). Then perhaps some design and of course some code. Then the first beta is ready for release. Wow that was fast, but anyway. Success, get the minimum viable product out there! This is where people stop and think "Oh right, we need some kind of hosting", someone with a credit cards pays for the hosting and a developer is tasked with deploying the code to it. Of course that's not a problem for the developer, but usually not much thought will be put into this part. No deployment tools will be used, no automation is put into place. I'm not saying this is the case for all project, but it's a fairly good estimation for most projects.

Do you see the issue at hand here? **Deployment is in most cases an after thought**. It's something many feel they just have to do and most wouldn't get out of bed earlier on a Monday because they are so eager to set up a sexy deployment process. Usually this will be a "fix things as you go" kind of process with quick and dirty fixes, duct taping that shit if it's necessary. This is something I usually refer to as *duct tape deployment*, and yes you can quote me on that. This is more than often a one man or one woman show and if someone else would like to deploy they wouldn't be able to. They would first have to ask the "one man/woman deploy army" on how it's done, what the passwords are, which paths are used, where the database is and everything else that comes with it. Efficient? No. Sustainable? No.

What we need to start with is making the deployment an important part in our applications lives. It should be there for our application as a supporting and nurturing parent. Would you let your kid go to the first day of school without you being there? Of course not. The school system in the United States is very fond of the *No Child Left Behind Act* and I would like to propose a *No Application Left Behind Act* where no applications is left behind because of a bad deployment process. It might be a bad analogy, since they have a really crappy education system in the United Stated. But I hope I got my point across anyway.

2.2 Maturity

During your applications lifetime the deployment process will (hopefully and most likely) mature. That is matures is usually a good thing and somewhere a long the line someone will probably realize that steps need to be taken to improve the it. In which ways it will mature is usually according to the following but maybe not in this particular order:

- Documentation
- Automation
- Verification
- Notification
- Tests
- Tools
- Monitoring

Documentation. This is usually what happens when the transition is made from one person deploying to multiple people deploying. It will be a natural step since the deployment process will most likely have manual steps so you can't deploy without a documentation. Somewhere a document or something similar will end up describing the different steps that needs to be done in order to deploy.

Automation. Multiple people are not deploying but stuff breaks because of inconsistency. Even if all manual steps are documented well, unexpected things will occur that falls outside the scope of the documentation. In these cases it will end up on the one woman/man deploy army since they will have the answers. The response and solution to this is automation. It should be a humble effort to in the end automate everything in the process.

Verification. The process have now been automated enough to make anyone deploy in a consistent manner. But still it can break in numerous ways and the automation will be altered to handle errors so we can verify that the steps are completed without errors. The deploy can be aborted if any automation fails and display an appropriate message to the person deploying on what went wrong.

Notification. With verification in place making it notify the right people when it fails should be trivial. Having a deploy fail silently could be catastrophic and deploying the wrong changes could also be very troublesome. A notification on what was deployed (branch, commit, etc) and that it did so successfully can be just as useful as for a failing one.

Tools. It seldom happen that a deployment process will use tools from the start. With a stable process that works in most cases but is difficult to extend, tools will most likely be put in place. Adding commands to run with a deployment tool is usually a lot easier than changing the scripted automation and will leave the scripts obsolete and replace them. It will often also replace the documentation part since it will more or less document itself, if done correctly.

Monitoring. This is a step few will actually reach and is definitely more of a *nice to have* than a *need to have* maturity. It's about monitoring how the application and it environments responds to a deploy. The monitoring could monitor CPU usage, memory usage or I/O-operations. If a deploy increases memory usage by 30% you might have a big problem on your hands. So the goal here is to monitor and notify on big deviations for certain key metrics.

2.3 Agility

The world around us as software developers is getting more and more agile. I'm talking about the world of agile software development[9], not the regular outside world where people drink lattes and worry about their mortgages. More and more companies are jumping on the agile bandwagon and for a good reason. There are some practices in this that are great, such as behaviour-driven development (BDD), test-driven development (TDD), pair programming, continuous integration, sprints, user stories and cross-functional teams just to name a few.

> Agile: "Characterized by quickness, lightness, and ease of movement; nimble."

This sums it up pretty well because you want your software development cycle to be fast, easy and adaptable where you work in small increments and respond quickly to feedback through iteration. At the beginning of 2001, a bunch of very smart people got together to talk about lightweight development methods and what they ultimately came up with was the publication Manifesto for Agile Software Development[10], this is the main part of it:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> **Individuals and interactions** over Processes and tools
>
> **Working software** over Comprehensive documentation

[9] http://en.wikipedia.org/wiki/Agile_software_development
[10] http://agilemanifesto.org/

> **Customer collaboration** over Contract negotiation
>
> **Responding to change** over Following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.
>
> *Kent Beck, James Grenning, Robert C. Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Arie van Bennekum, Andrew Hunt, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, Jon Kern, Dave Thomas, Martin Fowler, Brian Marick Â© 2001, the above authors. This declaration may be freely copied in any form, but only in its entirety through this notice.*

This is really exciting stuff and if you take a look at what is valued in the items on the left it's people, flexibility and speed. Flexibility and speed of the actual development and people is both about the people building the software and the people using the software.

Now you might say that I have been talking about the deployment **process** a lot and the manifesto clearly states that it favours individuals and interactions over processes and tools. Yes, this is true; however it also say that there is value in the items on the right and I consider the deployment process to be essential in supporting a true agile development process. The core in agility for software development is iteration, an ability to adapt fast and respond to feedback from people. How will you be able to do this efficiently without a good deployment process? If you iterate over a feature and spend two hours on it and then have to spend two hours on deploying it, is that really agile? I would argue no. Even if the deploy takes “only” 30 minutes you need confidence in it and if there are many manual steps or you can't do a quick and easy rollback, what is agile about it then?

2.4 Plan for a marathon

What could be better in all this than doing a comparison with a marathon? In an agile world you work in sprints and if you plan your deployment process for the current sprint your plan will be very short sighted and situational. If you expect your application to live longer than for a few sprints you need a plan for its future. So plan for a marathon, not a short sprint.

Making estimations is just as hard whether you make it for your development time or your applications future. This is especially true in the agile world where adapting fast to feedback could change the direction of that future in a heartbeat. But we should at least try to do this. Try to make some predictions where your application could end up. Coming up with a few scenarios that could occur will not harm you.

Say you deploy your application to a VPS with limited resources and no real options to scale it fast. If you predict there is a possibility your application would get featured on Hacker News, Reddit, Product Hunt or Slashdot and the traffic goes through the roof, now what? Do you have a deployment process that is good enough for quickly moving your application to a new hosting provider? Does your application and deployment process support running on multiple nodes? Just to be clear, I’m not saying you should make your deployment process deal with all the possible scenarios since that would be extremely time inefficient. What I’m saying is that you should keep those scenarios in the back of your head while you plan and implement with a “if X happens, I can change this easily to fit it”-mentality. Your deployment process should be agile too and try to make it so you can change things fast and easy.

2.5 Release cycles

You can write how much code you like, but if you’re not shipping any of it you’re not contributing with any value. You and your team should have a goal of how often you want to release code. Whether it be once every month or four times a day, as long as you have a goal and a plan for it. If you do not have a goal for it you can’t reach something. So set a goal if you do not have one and try to get there. Knowing your current and your wanted release cycle is important for planning your deployment process.

By “shipping code” I refer to deploying it, from finishing a feature in your local environment to getting it out to the production server(s).

There are a few ways in which we ship code and the different models of it can be summed up to:

- I’m done, ship it.
- The new version is done, ship it.
- X amount of time has passed, ship it.
- I pushed it, ship it.

This is of course simplified and there are many variants of these out there in the wild. But let’s discuss these individually to see what they’re about.

I’m done, ship it. We could call this *feature deploy* or *shotgun deploy* depending on how we feel about it. This is a typical model for when there is one or few developers for an application. It’s an ad-hoc type of deploy and when you have finished and tested your feature you deploy it. I think this is a very underestimated type of deploy if you can make sure that you do not push changes someone else have made. Why should we wait for a ritualistic deploy? If you version control with a good branching strategy these types of ad-hoc deploys is definitely possible. For larger teams and applications this is something I wouldn’t recommend though.

The new version is done, ship it. This would be when your software has reached a new version, either minor or major. The point here is that the deploy payload will probably be large and the development has been undergoing for a considerable amount of time. I would avoid this model like the plague when it comes to web applications. One of the great advantages of web applications is the ability to change the application quickly without having to go through build steps and knowing your users have updated their applications. On the web you ship your code and then your user doesn't have a choice. Why even deploy only after a minor version is completed? If you are doing this, one of the reasons could be that the deployment process is to complex. Fix it and stop pushing like this.

X amount of time has passed, ship it. If your team is working in sprints this is probably how you deploy. Whenever a sprint is complete you deploy. The amount of time between deploys can vary depending on how long sprints you have of course and the main take here is that deploys happen on a recurring time, for example every other week on Mondays. I do not mind this model for deployment at all if you have reasonable sprint durations. If you have three month sprints it could be bad and should take a look at why you're doing this. But if you have sprints of perhaps 1-2 weeks, go for it.

I pushed it, ship it. Also known as *continuos deployment*. Ah, the unicorn of deploys. Here everything is deployed when one or more commits are pushed to a location. This is done through (at least should be) a very complex system of automation with testing and monitoring. The topic of how you can accomplish this is not an easy one and I will not try to explain it here.

One important point to make here is that being able to continuously deploy is living the dream. Very few will have time, knowledge and patience to set up a complete process of continuos deploys. However it's something that we can strive for even if you do not want to continuos deploy. Building a culture, environment and process where it's possible will benefit you tremendously. Having the correct tools in place for automation, monitoring, etc will never be a bad thing. If you could have it all and just flip off the switch that deploys everything automatically, you should have a sense of bliss and serenity. You've done it.

2.6 Technical debt and rot

Technical debt and software rot is something well known and talked about in the developer community, but the concepts can be applied to more than code. We should take care of our deployment just as much as our code. When an ad-hoc deployment process is put in place we immediately start hoarding technical debt and when it starts to rot we have **shipping rot**. The problem with this kind of debt and rot is that it's more than usually exponential and self enforcing. Have you heard of the broken window theory[11]? The TL;DR version is: when people see that a building has a broken window, they stop caring less about maintaining it and into the spiral it goes. This applies perfectly to software development where legacy code stays legacy and bad code breeds more bad code. The same goes for our deployment process: a bad process will stay bad or get worse. If you can stop that first window from breaking or quickly repair it, everyone else will care for that it stays that way. Having goals set is a great tool for quickly dealing with that broken window and if you at some point realize that there are many broken windows, you should perhaps fix them all in one big swoop.

[11] http://en.wikipedia.org/wiki/Broken_windows_theory

2.7 The list

We can condense the goals in to a bullet list since everybody likes a clear and concise list, right? When reading the list, try to reflect on where you currently are at the different goals.

- Automated
- Responsive
- Atomic
- Reversible
- Simple
- Fast
- Agnostic

Automated. Have you read the first chapter? If not, do it. Not having an automated process is the root of all evil.

Responsive. A good deployment process responds to what is happening. If an error occurs somewhere it should abort and notify you or the appropriate people somehow. Having steps fail silently in a chain can be very destructive for your application.

Atomic. Nothing in your deploy should be able to break your application and you should comply with the concept of completing a build before serving it to any user. The transition between the previous and the new build should be as close to instant as possible.

Simple. Everyone should understand it and use it. Everyone should feel confident and comfortable with it. This is true for deploying, making changes or extending the process.

Reversible. Because it sounds better than "rollbackable". Being able to roll back is just as important as deploying your changes and this reverse process should also follow the list of goals.

Fast. You want it to be fast because speed is key in being able to deploy often. Fast rollbacks combined with fast deploys is another key which spells confidence. If something breaks you can easily go back and fix things without that stress knowing production is in a broken state.

Agnostic. Building a deployment process that is dependent on its environment could be devastating. Being able to only deploy to Amazon Web Services for example will be great until you want to switch provider. The application should happily be deployed anywhere and it should also be agnostic about who or what is deploying.

3. Environments

Having multiple environments for your application is essential for quality. Each environment serve a certain purpose and represents a state of your application in some way. When I talk about an environment I mean in which your application lives, that could be separate servers or virtual machines. Having multiple environments running on one machine is not an issue if they are separated but they must be it in some way to prevent them from interfering with eachother. If two of your environments are sharing the same database or cache, you could end up with unexpected results and even create bugs that didn't exist in the first place.

3.1 Repeatable

Being told you're a "broken record" isn't fun but for your environments it's the best thing you could say. The most important thing to strive for with your environments is that they always have the same setup where operating system and software should be of the same version and have the same configuration. And in a best case scenario they are **repeatable through automation**.

Many times have I encountered unexpected behaviour because of environments differing from each other. Some operating systems, CentOS for example, treat file names case sensitive while others like Ubuntu treat them case insensitive. This could end up with files not being read into your application. Another good example is how JSON is encoded/decoded; some environments will encode `0` to `"0"`. If your application is expecting an integer and will get a string then things can go wrong.

Knowing that your application will behave the same in your various environments is great because then you do not have to think about it. *Removing uncertainty* is important for gaining confidence in your deployment process.

Server provisioning

I said before that this topic is outside the scope of this book but I still want to touch briefly on it. If you want to achieve repeatability in your environments to the point where all your environments are all the same: you need it. Server provisioning is automating your environments. Say you're firing up a new virtual instance or installing a new server, what are your next steps? Often they are:

- Install software
- Set configuration values for the software
- Set up a service or application
- Configure the service or application

- Serve the service or application

If you do this manually you will inevitably loose control of how to repeat it. Making sure that correct values are set in `php.ini` across all servers will never be maintainable if done in a manual fashion. The more environments you have, the tougher it will become. Need to update the file size upload limit on 78 server instances? Have fun. Perhaps get an intern.

There are a number of tools for server provisioning. The current most popular ones are Chef[12], Puppet[13] and Ansible[14]. The latter have gained a lot of traction lately. I suggest you read up on them and take a look at what problems they try to solve and how they solve them. They can help you with taking your deployment process to the next level.

3.2 Configuration / environment variables

An important aspect of environments is configuring your application for the environment it's running in. Separation of code and configuration is a part of the twelve-factor app[15] and it explains the importance of this separation as:

> Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires strict separation of config from code. Config varies substantially across deploys, code does not.

The distinction between code and configuration is important since your application should be able to run without depending on configuration in your repository. They state a simple but powerful question to this, which is *can you at any given time open source your application without compromising any credentials?* If you could in theory do this you have a strict separation of code and configuration and I'll show you how to do this with a simple and great tool called *PHP dotenv*.

The goal is to supply your application with configuration that is outside of the repository and this could be any important hostnames or credentials for database connections, cache connections or third party services such as *S3*, *IronMQ* or *MailChimp*.

PHP dotenv

This tool is probably all you need for your configuration requirements, it's simple yet powerful. It does not require you to edit any virtual hosts in Nginx or Apache nor add or modify any PHP configuration values. All you do is create a file with variables in a key/value manner and it populates the global variables `$_ENV` and `$_SERVER` for you and also makes them accessible via the `getenv()`[16] function.

Installation is done through Composer:

[12]https://www.getchef.com/
[13]http://puppetlabs.com/
[14]http://www.ansible.com/
[15]http://12factor.net/config
[16]http://php.net/manual/en/function.getenv.php

```
composer require vlucas/phpdotenv
```

Then you need to bootstrap the configuration loading. In this example we pass `__DIR__` to it which tries to locate the configuration file in the same directory as the executed file, but this could be replaced with any directory you want to store your configuration file in:

```
$dotenv = new Dotenv\Dotenv(__DIR__);
$dotenv->load();
```

After this we create a file called `.env` in the directory we supplied and for the sake of an example we add our database configurations to it:

```
DATABASE_HOST=localhost
DATABASE_USERNAME=user
DATABASE_PASSWORD=password
```

We can now access these configuration values in our application and all three of these return *localhost*:

```
$databaseHost = getenv('DATABASE_HOST');
$databaseHost = $_ENV['DATABASE_HOST'];
$databaseHost = $_SERVER['DATABASE_HOST'];
```

I recommend using `getenv()` or even better writing a wrapper function that allows you to pass a default value if the configuration value isn't set (inspired by LaravelÂ´s env()[17] function).

```
function env($key, $default = null)
{
    $value = getenv($key);

    if ($value === false) {
        return value($default);
    }

    switch (strtolower($value)) {
        case 'true':
        case '(true)':
            return true;
        case 'false':
```

[17]https://github.com/laravel/framework/blob/a1dc78820d2dbf207dbdf0f7075f17f7021c4ee8/src/Illuminate/Foundation/helpers.php#L626

```
        case '(false)':
            return false;
        case 'empty':
        case '(empty)':
            return '';
        case 'null':
        case '(null)':
            return;
    }

    return $value;
}
```

This allows you to fetch a configuration value or get a default value

```
$databaseHost = env('DATABASE_HOST', 'localhost');
```

Make sure your configuration file is ignored in your repository so it doesn't get committed. Remember the part where you should be able to, at least in theory, open source your application without compromising any credentials.

Example configuration file

For reference and easy setup create an example configuration file that you can copy and replace the values from. Create `.env.example` that contains all necessary keys you need for your application.

3.3 Local environment

Having a local development environment is something you should strive for. With local I mean on your actual computer, not on a centralized server somewhere you access through a VPN service or similar. Being able to work on your code anywhere without an internet connection is nice.

I always run my local development environment in a virtual machine with Vagrant[18]. This allows me to go haywire with everything and if I fuck up the environment badly enough I just destroy the virtual machine and provision it from scratch. It's great for testing things in your environment without having to worry about breaking something on your computer or for someone else. Want to test your application in a different PHP version for example? Install, test, reset.

The main point is that you want to be able to work on your application whenever. With a local environment you can work on a flight across the atlantic and push/pull changes as soon as you get

[18]http://www.vagrantup.com

a connection to the internet. If you end up in a situation where you are dependent on an internet connection to work you'll curse yourself when you don't have one.

Having your local environment repeatable is the number one tool for introducing new team members to your code. Imagine you start at a new job and all you do it `git clone <insert vagrant repository here>` and then a `vagrant up` and you're all set. Sounds like a dream doesn't it? Usually time is invested in writing documentation for setting up a local environment and that documentation is more than often outdated and no one really wants to take responsibility for keeping it updated.

3.4 Development environment

The name of this one can be somewhat misleading. No development work should be done here but the name comes from the branching strategy in chapter 6 on Version Control. This is a common ground for features and bug fixes where all code that should be shared and tested by others will end up here. The first step after finishing something in your local environment will most often be to merge it to your main branch and push it to the development environment.

It's a great place for early testing by yourself and your team. Getting an extra set of eyes on something early on can be a great quality tool. Code reviews should be performed before something is merged and pushed to this environment since it's a very effective tool for quality, knowledge sharing and stomping out some obvious errors. All developers will have certain domain knowledge and someone else could perhaps tell right away if your code won't work with a specific part of the domain.

3.5 Staging environment

Your staging environment is very important and it should to the fullest extent possible duplicate your production environment. This is where you do your final testing of your code before you push it to production and serve your application to the users.

With duplicate to the fullest extent I do not mean that it should connect to the production database for example. Having a database that gets updated with a production dump every 24 hours is good enough. Does your application have multiple database instances and perhaps connect to a cache cluster? The staging environment will need that as well. Is the application running on multiple nodes behind a load balancer? Set up your staging environment in the same say. Remember, **as close to the production environment as possible**.

Being able to test against production data is sometimes necessary but be extremely careful when doing this for obvious reasons. Make sure you do not test features that for example could end up e-mailing users. Your staging environment should perhaps only allow outgoing e-mail to certain addresses that belongs to people in your team or in your company. Here is a simple example that logs the e-mail if it's not in production:

```php
function sendEmail ($to, $from, $subject, $message) {
        $productionEnvironment = (getenv('ENVIRONMENT') === 'production');
        $internalEmail = (strstr($to, '@yourapplication.com') !== false);

        if (!$productionEnvironemnt && !$internalEmail) {
                $log = sprintf(
                        'Sending "%s" e-mail to %s from %s',
                        $subject,
                        $to,
                        $from
                );
                error_log($log);

                return true;
        }

        $headers = 'From: ' . $from . "\r\n" .
                'Reply-To: ' . $from;

        return mail($to, $subject, $message, $headers);
}
```

3.6 Production environment

This one is quiet obvious and it's the server(s) serving your application to your end users. Whether it's one server or multiple ones behind load balancer; this is the endpoint for your users.

It will contain an as stable as possible code base with code that is extensively tested and as bug free as possible. What is to be said here is that **it's for this environment we need a great deployment process**. All the effort we put in to the process will reflect on the quality and stability of your production environment.

3.7 Testing environment(s)

Having one or multiple test environments is not obligatory but it is something that can bring a lot of benefits for your application. Depending on the size of your application it will make testing a lot easier. If you are able to set up test environment for specific branches for example, you can easily give access to testers for feature branches.

There are a few things that are essential for having success with test environments: speed and repeatability. If you can without major effort set up a test environment and check out a certain branch, your testing possibilities will increase dramatically. When I say speed I would consider 30 minutes or less a reasonable amount of time. Longer than that and you can start to experience too big discrepancies in development speed and test environment speed. Also If you're not able to update your test environment with the latest changes it will hardly be worth your time. The loop here is to get quick feedback on your code and react to it. Fix the code, push to the test environment again and then get new feedback.

3.8 Simplify server access

In the process of managing multiple servers you end up wanting to make the access to the them faster and easier. First of all creating host entries for your various servers will remove the need of looking up IP addresses. For some servers you won't need it since they will be publicly accessible through domains such as your production environment. But even for your production environment you might have servers for databases, load balancers, caches, etc and they might not be accessible through a public domain. But you will still need to access them on a regular basis for maintenance and debugging so create host entries for them.

The second thing is setting up and maintaining your SSH configuration file. In this you can manage hosts, hostnames, keys, ports and everything else you need to simplify your access to the servers. Compare these commands and you'll realize that you can gain a lot by this:

```
ssh -i ~/.ssh/myapp-staging.key staging@123.123.1.9 -p 2223

# instead when using configuration file:

ssh myapp-staging
```

All we need for this is to create ~/`.ssh/config` and add our configuration for our environments there. For the previous example it could look like this, with host entries added and expanded for covering multiple environments:

```
Host myapp-staging
    HostName staging.myapp.com
    Port 2223
    User staging
    IdentityFile ~/.ssh/myapp-staging.key

Host myapp-staging-db
    HostName staging-db.myapp.com
    Port 2223
    User staging
    IdentityFile ~/.ssh/myapp-staging.key

```

```
Host myapp-production
    HostName myapp.com
    Port 2224
    User production
    IdentityFile ~/.ssh/myapp-production.key

Host myapp-production-db
    HostName db.myapp.com
    Port 2224
    User production
    IdentityFile ~/.ssh/myapp-production.key
```

As you can see, multiple hosts can be defined and you can create one for each of your servers that you manage often. This will save you lots of time in accessing the servers. There are even more cool stuff you can do with it such as specifying wild card hosts and more. You can do some further reading on the man page for ssh_config[19].

[19]https://www.freebsd.org/cgi/man.cgi?query=ssh_config&sektion=5

4. Atomic deploys

Once upon a time there was a team of developers working on a service for sending e-mail campaigns. Their goal was that the system would allow users to manage subscribers and e-mail templates, so that the user could create e-mail campaigns and send them to their subscribers. The service got its first early adopters and began delivering successful campaigns with happy users. The focus for the team was feature development, adding features that was requested by customers or the ones they felt needed for the service.

The team had a simple infrastructure in place, and they had a plan for when the service (hopefully) had to scale up. They divided it in two logical components. The first component was the one presented to end users, a kind of a mishmash of CMS and CRM. This web component was responsible for creating the e-mails and sending them over to the mail server. The second component, the mail server, had a single responsibility of receiving pre-configured e-mails and sending them out. The components were deployed on two separate cloud instances and the plan was to scale horizontally if the service gained enough traction. When it came to deployment the lead developer would access the instances, pull down the latest changes and complete the steps necessary for a deploy. This was manual labor.

The service continued to grow, more and more users signed up for it and started using it as their main e-mail marketing channel. Not only did more users sign up but also users with a lot larger subscriber lists migrated over from other similar services. To import subscribers the user had to copy and paste their list(s) into a textarea with one e-mail address on each row. This was insufficient for many users and the most requested feature became an improved importer, being able to upload files with tens, or even hundreds, of thousands subscribers to be imported. The team got to work and once they had a completed feature they rolled it out to the public and immediately notified users about their grand new feature. It became a frequently used feature, especially for users migrating from other services.

A few weeks went by and suddenly the team started receiving e-mails and phone calls from annoyed users who had used their import feature. The users repeatedly claimed that when they had uploaded their file, they had to wait for a long time before anything happened and all of a sudden the import was canceled for no apparent reason. The team started a thorough investigation and they dove deep into the code and logs trying to reproduce the problem. They set disproportionate time and memory limits for code execution, but to no avail. Instead they tried looking at patterns for when in time this happened, could it be an issue when the load is high? They looked at different monitored values for their servers but couldn't find anything. But what they did find was that the e-mails and phone calls they received with complaints often happened every other week on Mondays. And usually continued to around Wednesday the same week.

It now occurred to the team that the pattern followed their release cycle. They worked in two week sprints and deployed every other Monday. It was these Monday's that the problem occurred. Upon this realisation it became quite obvious what caused the issue. When deploying their application they always restarted the *php-fpm* service just in case. When the service restarted, all users who was running an import (a long running PHP process) would get their import canceled because all PHP processes were killed. It now occurred to the team that their deploy process wasn't *atomic*.

4.1 What is atomicity?

What is an atomic deploy? It's all about hiding things from the rest of the world. Let's take a look at one definition:

> In concurrent programming, an operation (or set of operations) is atomic, linearizable, indivisible or uninterruptible if it appears to the rest of the system to occur instantaneously.[20]

There are two parts here that is interesting. First of all, atomicity is achieved when one or more operations **appear** to happen instantaneously, and also that a synonym for it is **uninterruptible**. Some call this *Zero Downtime Deployment*, like Envoyer[21] does. Let's go back to the teams issue. When deploying, restarting the *php-fpm* service seemed instantaneous to the person deploying, it was a matter of milliseconds. But in this case it proved to interrupt other parts of the system which broke the atomicity of the deploy.

What we see here is that when an application reaches a critical mass of users and/or a certain complexity, you need to deploy without any interruptions to the service. Striving for your deployment to be fast and responsive just won't cut it; it needs to be atomic. In the previous example, it's not a matter of speed, but of reliability. Even a nanosecond interruption in the *php-fpm* service will result in breaking behaviour for the end user.

Perhaps you could ask yourself the question, or at least argue for, that it isn't that big of a deal if the user notices or something unintended happens. I could go along with that to a certain extent. However if you're able to solve it, why not keep your users as happy as possible? Interruptions or unintended behaviour could also cause data corruption that you would have to deal with. If we take a look what the other side of the fence might look like, I would say it's quite bad. That would encompass scheduling, notifying users in advance and take your application offline while deploying. In the long run this will most likely not work out. The obvious example is when you need to deploy an urgent hotfix or when the service scales up.

4.2 Achieving atomicity

Atomic deploy is when you switch between the previously deployed version and the new one as quick as possible. I'm talking milliseconds or even less, anything slower than that can't be considered

[20]http://en.wikipedia.org/wiki/Linearizability
[21]https://envoyer.io/

atomic. Doing this without your users even noticing is key. Remember the part about making it **appear** instantaneous to the rest of the system? Your application does not need to be very complex or be deployed on an advanced infrastructure, it's enough that somewhere in your process the different parts can come out of sync from each other. Dependencies on packages could get out of sync with your code if the code is updated before dependencies. You also never want to interrupt any current running processes as shown by the example.

There is a few things that usually need to be in order for achieving atomic deploys and I will go through them below. The next chapter will cover deploy tools and there I'll show you how to do this with the various tools.

4.3 Concept of builds

Even though we as PHP developers seldom had to consider the concept of builds of our application it's getting more frequent and we must internalize this for achieving atomic deploys. If we can not create a separate build while deploying, it's not possible. Or actually we could do that with a smart infrastructure, shutting down traffic to servers that are in "deploy mode". But for the sake of argument and simplicity we won't go down that route. There is a simple way of doing this on one server; all we need to do is create a build in a folder that is not being served to our users. Then we can do the old switcheroo on the currently deployed build and the *to be* deployed build without the user noticing.

Having an appropriate folder structure for allowing this is simple. I will later in this chapter propose a structure, but there are other components to this that I want to discuss prior to it. For now just consider a folder with your application with all dependencies and configuration complete as a build. I would consider a build complete if you can put it anywhere on your server and serve it to your users.

When switching builds you should also save a number of older builds, perhaps the five or ten previous builds. This allows for quick rollbacks and can be crucial if a deploy needs to be reversed. How many you should save is impossible to answer so just go with a number you feel comfortable with, it also depends on your release cycle. If you deploy once a week or once a day you can probably without any discomfort save your five or ten last builds. If you're deploying continuously on every commit then you should probably save a lot more than five.

4.4 File persisted data (shared data)

When switching out a previously deployed build for the new one you must assure that no file persisted data is lost. One example of this is if your application stores session data or user uploaded files inside your application folder. If you switch out the old build for a clean new one, users might be logged out or data could be lost. And this is never an ideal scenario.

On the other hand there could be file persisted data that you do not care if it gets lost, maybe you would even prefer that it does. If your application has a cache for rendered templates you'd probably prefer if that cache is wiped so your underlying logic and presented views won't get out of sync. In these cases just make sure that your new build replaces or wipes these folders.

What is important here is to identify the files and/or folders that needs to remain persistent between builds. I refer to these files or folders as **shared data** and I will show how to deal with this in the proposed folder structure.

4.5 Symbolic folder links

Although it's not really necessary, but to grok symbolic folder links (*symlinks*) is strongly advised. What we want to do with symlinks is also possible by just copying or moving directories. But to make things atomic we should leverage symlinks since they are pretty much instant.

A symlink looks like a folder or a file that appear to exist in a location. But the symlink is a reference to a file or a folder in another location. It allows us to instantly switch out which folder or file a symlink is pointing to. When we deploy and want to switch out the old build, we just update a symlink to point to our new build instead. Likewise we will do this for shared data. In that way we make the switch extremely fast and can make sure that our shared data is there when we need it and is stored in one location only.

For Linux users, it's the `ln`[22] command. It's possible on Windows but it's complicated and I suggest doing a Google search for it.

4.6 Proposed folder structure

The proposed folder structure needs to reside inside a root directory somewhere. Where that is doesn't really matter, it just needs to contain the following structure. How you develop your application is not of importance and this example assumes that the folder structure is on the server serving the application.

The names of the folders inside the root are arbitrary, name them as you please. My examples that follow will use this structure and it looks like this:

[22] http://www.unix.com/man-page/posix/1posix/ln/

```
├── builds/
|   ├── 20141227-095321/
|   ├── 20141229-151010/
|   ├── 20150114-160247/
|   ├── 20150129-083519/
|   └── 20150129-142832/
├── latest/ <- symlink to builds/20150129-142832/
├── repository/
|   ├── composer.lock
|   ├── composer.json
|   ├── index.php
|   └── sessions/ <- symlink to shared/sessions/
└── shared/
    └── sessions/
```

builds/ - this folder contains the X number of builds that I discussed earlier. Perhaps the current one deployed and the four previous ones. I suggest naming all builds with a timestamp with date and time down to seconds, you never want a build to overwrite a previous build by accident.

latest/ - symlink to the current build that is deployed. Your web server will use this folder as its document root for your site. Unless there is a subfolder that should be served, depending on your set up and framework.

repository/ - the folder where the repository resides and this is what we will make builds from.

shared/ - any shared data that needs to be persisted between builds will reside here. It can be both files and folders.

4.7 Pseudo deploy process

The process for making an atomic deploy is then straight forward:

- Update the code in *repository/*, probably through pulling the latest changes from your remote.
- Update dependencies and configuration, this is also done in *repository/*.
- Make a copy of *repository/* to a folder in *builds/*.
- Update the symlink *latest/* to point to the new build that was copied.
- Remove excessive builds in *builds/*.

Need to perform a rollback? Update the symlink *latest/* to point to the previous build you want deployed.

5. Tools

So far we have covered theory only but now it is time to get more into the technical aspects of the deployment process.

In chapter 2 about Goals we talked about the maturity a deployment process usually goes through. Almost at the bottom (or top, depending on how you see it) of the list was a bullet on **tools**. The beauty is that it will replace or extend steps taken in the previous parts of the maturity process. This is the best place to start if you're setting up a sustainable deployment process for your application. If your current process are in the previous steps of maturity; replacing them with a tool is nothing major in most cases and something you gain a lot from.

What a tool provides for us is a way of gathering related parts of our deploy in one place. The core principle of a tool is **automation** which we've already covered in both chapter 1 and 2. It also provide a way of making your process readable and you will achieve a kind of self documentation for your process by using any of the tools. It could be considered a manifest for it.

Most tools do an excellent job in separating your environments. They make it easy to make every deploy to any environment repeatable. Perhaps you don't want to minify your Javascript and CSS files when you deploy to a testing environment? This makes debugging easier. But when you deploy to the staging or production environment you should do it. The tools will help you run the commands you want, when you want and where you want. Some of them also make sure that every person deploying have the correct endpoints to the environments for the deploys.

I will below cover a few of the common tools. They all have their pros and cons as with everything when it comes to technology. I won't tell you if one tool is better than another because it's a matter of taste and what works best for you and your application. With each tool I'll provide a cookbook on how to automate builds and also how to automate atomic builds. These will serve the purpose of how to get started with them and provides a good boilerplate set up for extending them to a full fledged deploy tool suitable for your application.

5.1 Git hooks

When it comes to simplicity nothing beats git hooks in my opinion. But this comes with a big **if** though. That **if** is if you don't want to do anything fancy with it. When you dive deeper and find out how the hooks work internally it can get a bit messy and scary. For simple deploys and builds I think it's great.

They require you to use Git as your version control system. But triggering scripts on pushes to remotes is a very good tool to use. They can do much more than that and there is currently **17 different** ones to chose from. What they do is that they hook into different places where Git performs operations. They are divided in two groups, those who runs client side and those who runs server side.

My basis for using git hooks here is to achieve automation for builds. But there are other things that can be accomplished with hooks as well. Enforcing policies is one of them. By using the right hooks, you can reject pushes that contains things that break a policy. What kind of policy that might be is up to you, but it could be how commit messages are formatted or who made the push or commits.

Here are the available hooks, on which side they are on, when they trigger and example of usages. The examples are just there to give you a general idea of what can be accomplished with it.

Hook	Side	Trigger	Example of usage
pre-commit	client	Before a commit is done	Checking code style or linting
prepare-commit-msg	client	After the default commit message is created	Changing the default message. Could be providing a template for commit messages
commit-msg	client	After a commit message is provided	Validating the commit message. If this does not pass the commit will be aborted
post-commit	client	After a commit is committed	Notifications
applypatch-msg	client	Before a patch proposed commit message is applied	Validate the message just as with *commit-msg*
pre-applypatch	client	After a patch is applied, but before the commit is created	Verify the contents of the patch by running test suites for example
post-applypatch	client	After a patch commit is created	Notifications
pre-rebase	client	Before a rebase operation	Denying rebasing of pushed commits

Hook	Side	Trigger	Example of usage
post-checkout	client	After a checkout operation is done	Verify the application state. Running a `composer dump-autoload` here is a good example.
post-merge	client	After a merge operation is done	Restore data in the working tree that git can't track
post-rewrite	client	When a command that replaces commits runs	See *post-merge* and *post-checkout*
pre-push	client	During a push. After the remote refs are updated but before any objects have been transferred	Validate a set of refs
pre-auto-gc	client	Before garbage collection	Notify that it's done or abort it if the timing is bad
pre-receive	server	Before receiving anything from a push	Access control
post-receive	server	After receiving a push	Notifications
update	server	Before receiving anything from a push, once per pushed branch	Same as *pre-receive* but branch specifically
post-update	server	After refs have been updated from a push	Builds

Making them run

All hooks are located in `.git/hooks` in your checked out repository. This means that you can have a different set of hooks that run depending on where your repository is checked out. A hook for the production environment will most likely differ to some degree for the same hook in your development environment.

The hooks should have a filename that corresponds to the hook name. This is how you would set up *pre-commit*:

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

We have to make them executable, this is important or otherwise they won't run. They will actually silently fail if not executable, you will not get any kind of notice on this. We'll get into more on what you can put in your hooks later.

Take note that when a hook runs, its working directory will be `.git`. So if you need to do anything in another folder, you need to use the *cd* command there first or specify absolute or relative paths. This is also important if you manually want to run a hook. Then you need to be in the `.git` directory and run a hook with `bash hooks/post-update` for example.

Version control your hooks

If you want your hooks to be version controlled and deployable, track them in your repository and create symbolic links to them.

You could put a `hooks/post-update-staging` in your repository for dealing with a push to your staging environment. Then on your staging server, create a symbolic link to the hook with: `ln -s hooks/post-update-staging .git/hooks/post-update`.

Language agnostic

One of the best features about git hooks is that they are language agnostic. If your shell can run the language, your hook can use that language.

If you want to use Bash in your hooks, you can do that:

```bash
#!/bin/bash
echo "Hello World!"
```

Or if you prefer Python, use that:

```python
#!/bin/python
print "Hello world"
```

Or why not PHP? You're probably familiar with this language.

```php
#!/bin/php
echo "Hello world";
```

And so on. This makes the hooks extremely versatile since you can write them in the language you are most comfortable in. However, you can also choose to write them to be able to run anywhere. So using Bash, for example, makes it run pretty much anywhere (except for Windows, but you can probably get that to work also).

I will use Bash in my examples. I prefer it since I know it will run almost anywhere.

Environment variables

Some of the hooks will have access to environment variables. But these are specific for each hook though, and in some cases the same variable can differ in settings from hook to hook. They can also be different depending on which version of git you're running. This is a bit inconsistent and can be confusing sometimes. But don't worry, if you're aware of them you can work around them pretty well. All environment variables can be found here[23] in the official documentation. Please note that not all variables are available everywhere.

Let's take the example of the environment variable `GIT_DIR`. This one is usually set to `.git`. This is true for the hooks *pre-commit*, *prepare-commit-msg*, *commit-msg*, *post-checkout* and *post-commit* for example. But then when you get to the hooks like *pre-receive*, *update*, *post-receive* and *post-update*, it's all of a sudden set to `.`. To me this is a very strange behaviour, but I assume there is a reason behind it, or at least I hope so. To make things even more complicated, it's not even set when running hooks like *applypatch-msg*, *post-applypatch* and *pre-applypatch*.

Knowing the environment variables and how they behave is not essential. It's sufficient to know that they are there, how you can work with them and try to achieve consistency. One thing I usually do is when I work the hooks *pre-receive* and *post-update* is that I just unset it to achieve consistency. This can be done by a simple

```
unset GIT_DIR
```

Now if I make it work with the environment variable unset, I know that it should behave as intended across different environments.

Automated builds

Now I want to show you an example of how we can set up a remote (an environment we can deploy to) to receive pushes from your local environment and proceed to make an automated build. It's actually really simple. We will leverage the **pre-receive** and **post-update** hooks for this.

Set up the environments

On your local environment, the application will be located in

~/dev/app

And on your remote it will be located in

/var/www/app.git

The paths are arbitrary, I will just use these in my examples.

First of all we will set up the remote. This is done by logging in on your server through a shell and executing the following

[23] http://git-scm.com/book/en/v2/Git-Internals-Environment-Variables

```
mkdir -p /var/www/app.git
cd /var/www/app.git
git init
```

Now you have an empty repository in place for your application on your remote.

Then we create a repository in our local environment and add the remote there so we can push to it.

```
mkdir -p ~/dev/app
cd ~/dev/app
git init
git remote add production <user>@<yourhost>:/var/www/app.git/
```

Voila! That's all we had to do. This could of course be done with an existing repository but then we just skip creating the directory and initializing a repository. Instead we only run the last command inside our current repository. Note that the remote is named *production*, if we we're setting up a staging or development environment we would name it accordingly. It could also be the name of the host or any other name that you like.

If we commit something now in our local environment we can push it to our remote by running

git push production master

Set up the hooks

We're now able to push commits to our remote server. But it does not do us any good in terms of automation yet. So, now we will set up the hooks necessary to start automating builds when we push. **All of this is done on the remote, not in your local environment**.

First of all we create the hook for *post-receive*. It's responsibility will be to move `HEAD` pointer to the latest commit. Since a push will result in pushing all the refs, but the pointers won't be updated.

```
cd /var/www/app.git
touch .git/hooks/post-receive
chmod +x .git/hooks/post-receive
```

Now it exists and is executable. Let's add all the commands that will update the repository.

```
#!/bin/bash
cd ..
unset GIT_DIR

env -i git reset --hard
git checkout -f
```

As I mentioned earlier, it changes the current working directory and unsets the `GIT_DIR` environment variable for consistency. Then it does a hard reset of the repository and forces a checkout of the latest commit. This ensures that our other hook will have the correct state to work with.

Now we need to make the actual build. This will be handled by a *post-update* hook since that runs when all refs have been updated. First we create it and make it executable, just as we did with the *post-receive* hook.

```
cd /var/www/app.git
touch .git/hooks/post-update
chmod +x .git/hooks/post-update
```

Now we can add all the commands we need to run. These commands will be arbitrary and you will have to add all the build steps necessary for your own application. I will just add a simple `composer update` for updating dependencies.

```
#!/bin/bash
cd ..
unset GIT_DIR

composer update
```

And there we have it!

Atomic deploys

Making atomic deploys with git hooks is not a very complicated matter actually. We just need a good template to work with and add some scripting logic to it. Take a look at the previous chapter on atomic deploys since we'll used the proposed folder structure in this example.

The hooks

There is a few steps we will go over here to make this all glue together. We'll start by creating a *post-receive* hook again:

```
cd /var/www/app.git/repository
touch .git/hooks/post-receive
chmod +x .git/hooks/post-receive
```

And adding the necessary commands for updating the repository on a push:

```
#!/bin/bash
cd ..
unset GIT_DIR

env -i git reset --hard
git checkout -f
```

And we create the *post-update* hook:

```
cd /var/www/app.git/repository
touch .git/hooks/post-update
chmod +x .git/hooks/post-update
```

And add a simple start to it:

```
#!/bin/bash
cd ..
unset GIT_DIR
```

So all of this is exactly what we did in the previous example, the only difference is that we created the hooks in `/var/www/app.git/repository` instead of `/var/www/app.git`. This also means that we have a different folder to push to, and the command for adding the remote in your local environment is slightly different

```
git remote add production <user>@<yourhost>:/var/www/app.git/repository/
```

If you want to change a current remote instead of adding it you can do this in `.git/config` on your local environment. Just use your text editor of choice and open it, then find this section and change the line accordingly:

```
[remote "production"]
~~url = <user>@<yourhost>:/var/www/app.git/~~
url = <user>@<yourhost>:/var/www/app.git/repository
fetch = +refs/heads/*:refs/remotes/production/*
```

Now we can push to the correct location and trigger the hooks when we push. So let's start taking the steps of making the deploy atomic. All of this goes into the *post-update* hook.

We need to be aware of the different paths that we have set up so we can perform things based on that. So we create a few variables for dealing with that. Then we also save the date in a variable for creating folder names.

```
# Variables
NOW=$(date +"%Y%m%d-%H%M%S")
APPLICATION_DIR="`pwd`"
ROOT_DIR="`cd .. && pwd`"
BUILDS_DIR="$ROOT_DIR/builds"
BUILD_LOCATION="$BUILDS_DIR/$NOW"
SHARED_DIR="$ROOT_DIR/shared"
LATEST_SYMLINK="$ROOT_DIR/latest"
```

After all this, we can add the command for updating our dependencies. Any command can be added here that you need for building your application, but we will stick with this simple example for now:

```
# Update dependencies
composer update  --no-dev
```

Now we should remove any shared folders inside our application and replace them with symbolic links to the *shared* folder. Let's assume this is a Laravel application and we need to keep `app/storage` persistent through builds.

```
# Add symbolic links for shared data
rm -rf app/storage
ln -s $SHARED_DIR/storage app/storage
```

Now we have our complete build that we should copy to `builds/` and create or change the `latest/` symbolic link to point to it.

```
# Copy build
cp -rp $APPLICATION_DIR $BUILD_LOCATION

# Symbolic link from build to latest
ln -nsf $BUILD_LOCATION $LATEST_SYMLINK
```

And finally we do some clean up, you only want to keep your latest 5 builds.

```
# Remove older builds, keep 5
cd $BUILDS_DIR
rm -rf $(ls -1t | tail -n +6)
```

Congratulations, you now have a fully functional atomic deploy!

I will end by showing you the complete *post-update* hook here and what it looks like:

```
#!/bin/bash
cd ..
unset GIT_DIR

# Variables
NOW=$(date +"%Y%m%d-%H%M%S")
APPLICATION_DIR="`pwd`"
ROOT_DIR="`cd .. && pwd`"
BUILDS_DIR="$ROOT_DIR/builds"
BUILD_LOCATION="$BUILDS_DIR/$NOW"
SHARED_DIR="$ROOT_DIR/shared"
LATEST_SYMLINK="$ROOT_DIR/latest"

# Update dependencies
composer update  --no-dev

# Add symbolic links for shared data
rm -rf app/storage
ln -s $SHARED_DIR/storage app/storage

# Copy build
cp -rp $APPLICATION_DIR $BUILD_LOCATION

# Symbolic link from build to latest
ln -nsf $BUILD_LOCATION $LATEST_SYMLINK

```

```
# Remove older builds, keep 5
cd $BUILDS_DIR
rm -rf $(ls -1t | tail -n +6)
```

More or fewer than 5 builds?

We are using `rm -rf $(ls -1t | tail -n +6)` in this example. This will save the 5 latest builds, but perhaps you want another number. Notice **+6** in the command, change that to *(N+1)* builds. So if you want to save 10 builds, you change **+6** in the command to **+11**.

It's a simple example but it serves well as a boilerplate set up for atomic deploys.

Output and logs

If you have tried any of the examples, you noticed a wonderful thing. The remote will send back the output of the running hook to you when doing a push. This allows for you to follow along with exactly what is happening in the push.

What I tend to prefer is to save all the output to a log file as well, having a deploy log is great for debugging purposes. This is how you in a very easy way can add logging for your hooks, just add it to the top of your hook:

```
NOW=$(date +"%Y%m%d-%H%M%S")
deploy_log="/var/logs/deploy/deploy-$NOW.log"
touch $deploy_log
exec > >(tee $deploy_log)
exec 2>&1
```

Just make sure that `/var/logs/deploy` exists and you have write permissions to it. Then it will create a log file with a date in the name for each time the hook runs.

Pros and cons

The good parts are:

- Simplicity (mostly)
- Integrates with git
- Automate things in your local environment
- Language agnostic
- Arbitrary commands

The less good parts are:

- Need to be created on each remote
- Pushing asynchronously is tricky
- Can be very complicated when scratching below the surface

5.2 Phing

This tool has been around for a very long time, according to their changelog the initial pre-release was in October 2002. Which is an eon in internet time. You can find it on its website[24]. It's based on Apache's Ant[25] tool which is a build tool built in Java. You can probably already tell that since it's very old and based on a Java tool that it's not cutting edge. The strength is that it has been around for a very long time and with that comes stability and maturity. Actually the "preferred method" of installation is through PEAR according to their official documentation, and that's not a good sign. But it can be installed through Composer as well, which it should be and the documentation cover this.

Installation

I would suggest that you install it through Composer. You can do this by either running (replace composer with how you access your composer binary)

composer require "phing/phing:2.*"

or adding

"phing/phing": "2.*"

to your required dependencies in `composer.json` and running `composer update`. Sweet and simple as that. You now have your `phing` binary in the directory `vendor/bin/`, which I suggest you to always have in your path.

However there might be reasons for you not being able to install it through Composer. In this very unlikely event you can download the Phar archive[26] and put it in the root of your repository. You can then use this to run Phing with `php phing-latest.phar`.

Either way is fine, but just make sure you stay away from the PEAR package.

Build files

It uses build files that are written in XML which is a format developers have been shying away from for a while. It does not make the tool better or worse, it's just the way it's done. This is because Ant uses XML build files and therefore Phing does it that way too. If you do not specify a build file when running Phing, it will try to find the file `build.xml`. Because of this it's always the easiest to name your build file that, unless you have very advanced needs for multiple build files or you can't name it that for some reason.

A valid build file must contain a few elements. First of all it must contain only one root element called `<project>` and everything in your build file goes inside this element. Then it must contain

[24]http://www.phing.info/
[25]http://ant.apache.org/
[26]http://www.phing.info/get/phing-latest.phar

several *type* elements such as `<property>`, `<fileset>`, `<patternset>`, etc. And it also must contain one or more `<target>` elements.

Start by creating the file `build.xml` and adding a appropriate XML header:

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

Project

All build files **must** contain this element and there can only be one. And it must be the root for your build file. It accepts a few attributes you can put on it:

Required	Attribute	Description
Yes	default	This is not optional and must exist. This is where you specify the default *target* to run when no targets are explicitly provided.
No	name	The name of your project, name it what you want.
No	basedir	The base directory of your project. Use "." for using the current directory. When it doubt, use "." as your base directory.
No	description	The description of your project, describe it how you want

So now let's add a *project* element to your build file

```xml
<project name="my-app" basedir="." default="dist">
    <!-- The rest of your deploy stuff will go in here -->
</project>
```

Targets

There can be one or many target elements, and they can depend on each other. This is great for preparing certain scenarios. Example: Let's say that you want to dump your database and move the dump somewhere, then you would make a target for dumping the database and a target for moving the dump that depends on the target for dumping.

```xml
<project name="my-app" basedir="." default="dist">
    <target name="dist" depends="move-database-dump">
        <!-- Dump database -->
    </target>

    <target name="dump-database">
        <!-- Dump database -->
    </target>

    <target name="move-database-dump" depends="dump-database">
        <!-- Move the database dump -->
    </target>
</project>
```

If you would run `phing` now nothing would really happen since we only have comments in the targets, but you should get a output showing that the build file was used and the targets where executed in the right order. Notice that we receive a waning that our target has no tasks or dependencies though:

```
Buildfile: /path/to/my-app/build.xml
Warning: target 'dump-database' has no tasks or dependencies

my-app > dump-database:

my-app > move-database-dump:

my-app > dist:

BUILD FINISHED

Total time: 0.1998 seconds
```

There are a few attributes that can be assigned to *target* elements too:

Required	**Attribute**	**Description**
Yes	name	The name of the target
No	depends	Specify target dependencies in a comma-separated fashion
No	if	A *Property* that must be set for the target to be executed
No	unless	A *Property* that must not be set for the target to be executed

Tasks

This is where it gets interesting. The official documentation have this to say on what a *task* is:

> A *task* is a piece of PHP code that can be executed

Which is really cool. This means that we can define tasks in our build that is executing PHP code. Let's stop for a second here and take a step back to look at what this offers us. First of all this allows us to **integrate our application with our deploy**. We could write build tasks in our application under a certain namespace, with this we get a deep integration where we could achieve really cool stuff. Whether it be busting caches through our actual application or building static assets with your favorite Composer package for that. Second of all, it allows us to **write tests for our build**. We could write tests that cover the entire build!

I digress a bit from the tool at hand, so let's go back to tasks. The way tasks are defined differs a bit from the other elements. Where others have a fixed element name (<*project* /> and <*target* />), tasks are a bit more dynamic. The element will instead be defined based on the name of the task. If we have a task that will clear our application cache, it might perhaps be defined as `<clear-application-cache />`. Phing provides you with a good amount of core tasks you can used and then you are free to create your own ones. Let's create a simple greet-task using the *AdhocTaskdefTask* provided by Phing for creating our own tasks inside the build file.

When defining a custom task, we need to create a class that extends `Task`. We can then define attributes to be passed in and create setters for them. In the example below we will use a *recipient* attribute to tell our task who we want to greet:

```
<target name="greet">
    <adhoc-task name="greet"><![CDATA[
        class Greet extends Task {
            private $recipient;

            function setRecipient($recipient) {
                $this->recipient = $recipient;
            }

            function main() {
                $this->log("Hello " . $this->recipient . "!");
            }
        }
    ]]></adhoc-task>

    <greet recipient="World"/>
</target>
```

We can now execute this with `phing greet` and we should see a proper output for it:

```
my-app > greet:

[greet] Hello World!

BUILD FINISHED

Total time: 0.0451 seconds
```

CDATA in adhoc-tasks

We should use *<![CDATA[…]]>* so we don’t have to quote entities within the task.

Properties

We have to option to define property elements in our build file which later can be referenced, you could see them as variables. These can be defined and we can tell them through an attribute that we want the option to be able to override them. If we say that we’re allowed to override a property, then we can specify the property value through the command line when running Phing.

If we go back to our example of greeting someone in the build, we can get a good demonstration of properties at work. First we define a property of whom we want to greet, this is done inside the *project* element.

```
<property name="recipient" value="World" override="true" />
```

So we say that by default we want to greet “World”, but this also give us the option to override this if we want to. Now we need to change our *task* to use this property. When referencing properties within the build file you use `${property-name}`. So in our case we have to reference `${recipient}`.

```
<greet recipient="${recipient}"/>
```

Now we can call the target and optionally change whom we want to greet. To override properties we use the `-D` option for the command line runner:

```
$ phing greet

my-app > greet:

[greet] Hello World!

---

$ phing greet -Drecipient=Niklas

my-app > greet:

    [greet] Hello Niklas!
```

Automated builds

Again, let's take a real world example of a simple application that install its dependencies with Composer. One down side to Phing is that you have to execute the command on the server in order for it to run the build. However, if you follow the examples for git hooks you can easily make a git hook run Phing when you push to your remote.

We will assume that we have a super simple application that looks like this:

```
├── composer.json
├── composer.lock
├── index.php
└── vendor/
```

What do we need to achieve in deploying this application? It's as simple as running a `composer update`. But for the sake of our example we want to remove the `vendor/` folder on each deploy to do a clean install of dependencies. This allows us to use the things we learned previously.

Now we need a build file for Phing. We start creating `build.xml` and declaring the basic parts of it.

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="app" basedir="." default="dist">
    <!-- The rest of will go in here -->
</project>
```

This is the basic XML declaration, a project element and a default target to run. We do not have that target yet so we can't run it yet. We said we wanted to do a clean install of our dependencies on each deploy, so let's create a target for preparing that state.

```
<target name="prepare" description="Delete dependencies">
    <delete dir="vendor"/>
</target>
```

And let's also create the default target *dist* that depends on the *prepare* target.

```
<target name="dist" depends="prepare" />
```

If you now run `phing` you should see in the output that it runs *prepare* first and then *dist*, resulting in that the vendor folder is removed (if it existed).

What we now need to do is have a target that handles installing the dependencies through composer. We run composer through the exec task, you can find more on this in the Phing's documentation on ExecTask[27]. This can be used to execute any arbitrary command.

```
<target name="composer-install" description="Install dependencies">
    <exec executable="composer" checkreturn="true">
        <arg line="install"/>
    </exec>
</target>
```

And we also update the *dist* target to depend on this.

```
<target name="dist" depends="prepare,composer-install" />
```

If we now run `phing` it will execute the *dist* target. That target have two dependencies, the *prepare* target and the *composer-install* target. They will run one after another, reassuring that the vendor folder is removed before it will run `composer install`.

Atomic builds

Making the deploy process atomic is a great way of explaining the parts of Phing. In this we rely on our process being in certain states before continuing with other steps. Again we'll use the proposed folder structure in this example.

And again we'll start by creating our basic `build.xml` file

[27]http://www.phing.info/docs/stable/hlhtml/index.html#ExecTask

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project name="app" basedir="." default="dist">
    <!-- The rest of will go in here -->
</project>
```

We can think of targets as certain states we need our process to be in before proceeding. The steps we can identify for the atomic build is:

- Create build
- Copy build
- Link shared data
- Update link to latest build
- Clean old builds

Each and one of these will have a target and we want them to be done in a synchronous manner because they always depend on the previous targets' state. Creating a build in this context will be just updating the dependencies in our *repository/* folder.

```xml
        <target name="create-build" description="Create build of the application">
                <exec executable="composer" dir="repository" checkreturn="true">
                        <arg line="update"/>
                </exec>
        </target>
```

After that we want to copy our build to the *builds* folder. We'll use the convention of named our build with the current timestamp for easier reference, so we'll create a property called *TSTAMP* that we can use to copy and reference our build from. Phing has a TstampTask[28] we'll use.

```xml
        <target name="copy-build" description="Copy current build">
                <tstamp>
                        <format property="TSTAMP" pattern="%G%m%d-%H%M%S" />
                </tstamp>

                <copy todir="builds/${TSTAMP}">
                  <fileset dir="repository"/>
                </copy>
        </target>
```

So now we want to link our shared data in the build. Again we assume that we store persisted data in `app/storage` inside the application, and have a `shared/storage` folder for that. Lucky for us, there's a task in Phing called SymlinkTask[29] that we can use.

[28]https://www.phing.info/docs/guide/trunk/TstampTask.html
[29]https://www.phing.info/docs/guide/trunk/SymlinkTask.html

```
        <target name="link-shared-data" description="Link shared data">
                <symlink target="${project.basedir}/shared/storage" link="builds/${TSTAMP}/app\
/storage" overwrite="true" />
        </target>
```

This leaves us with a complete build of the application with shared data prepared. All we need to do now is to serve it to the users. That's just a matter of updating the symlink *latest* that the web server has as document root.

```
        <target name="link-build" description="Link new build">
                <symlink target="${project.basedir}/builds/${TSTAMP}/app/storage" link="latest\
" overwrite="true" />
        </target>
```

We only have a final step to complete now. To not save infinite number of builds, it's time for some cleanup. We'll use the same bash command as we did for the git hook atomic deploy and put it inside Phing.

```
        <target name="clean-old-builds">
                <exec dir="builds" command="rm -rf $(ls -1t | tail -n +6)" />
        </target>
```

More or fewer than 5 builds?

We are using `rm -rf $(ls -1t | tail -n +6)` in this example. This will save the 5 latest builds, but perhaps you want another number. Notice **+6** in the command, change that to *(N+1)* builds. So if you want to save 10 builds, you change **+6** in the command to **+11**.

All steps, as targets, necessary are in place to make an atomic build now. We just have to put them all together in order. The default target in our build file is *dist*, so we create that and append all targets as dependencies to it. Here is the build file in its entirety.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="app" basedir="." default="dist">
        <target name="dist" depends="create-build,copy-build,link-shared-data,link-buil\
d,clean-old-builds" />

        <target name="create-build" description="Create build of the application">
                <exec executable="composer" dir="repository" checkreturn="true">
                        <arg line="update"/>
                </exec>
        </target>

        <target name="copy-build" description="Copy current build">
                <tstamp>
                        <format property="TSTAMP" pattern="%G%m%d-%H%M%S" />
                </tstamp>

                <copy todir="builds/${TSTAMP}">
                  <fileset dir="repository"/>
                </copy>
        </target>

        <target name="link-shared-data" description="Link shared data">
                <symlink target="${project.basedir}/shared/storage" link="builds/${TSTAMP}/app\
/storage" overwrite="true" />
        </target>

        <target name="link-build" description="Link new build">
                <symlink target="${project.basedir}/builds/${TSTAMP}/app/storage" link="latest\
" overwrite="true" />
        </target>

        <target name="clean-old-builds">
                <exec dir="builds" command="rm -rf $(ls -1t | tail -n +6)" />
        </target>
</project>
```

All that is left is to run the Phing binary.

Pros and cons

The good parts are:

- Tested and stable

- Abundance of predefined tasks
- Create your own tasks in PHP
- Dependencies between targets

The less good parts are:

- Written in Java
- Build files are written in XML

5.3 Capistrano

In their own words, Capistrano is a *Remote multi-server automation tool* and it is written in Ruby. That it is written in Ruby can be somewhat of an annoyance if you're not familiar with it, but with its great documentation and your Google skills you shouldn't have a hard time using it at all.

It started out as a tool for deploying Ruby applications, but has since then grown and can be used for deploying applications written in pretty much any language. Today they have official PHP specific plugins for Composer, Laravel and Symfony. Another great feature about it is that it will always perform atomic deploys. My suggested folder structure for atomic deploys and their folder structure[30] is almost identical on the server side. It also provides a good structure for your deploy tasks and configuration in your repository.

Web server document root

Since Capistrano uses **current** as a symbolic link to the current deploy, we need our web server to use that as the document root.

How it works is that you add your environments and servers in the configuration files for Capistrano, then you add tasks that should be performed on these different environments and server. After that you can easily execute the tasks remotely over safe and secure SSH connections. This was you can manage your deploy process from your own familiar command line on your computer.

You could also use it as a basic way of performing tasks on your servers that are not deploy related, since you can execute arbitrary commands over SSH on multiple servers at the same time. Want to get the memory usage on all your database production servers? Add a task to do this and capture the output from all your servers in one go. However this is not a book on infrastructure orchestration or provisioning, so we'll focus on deploying your PHP application with it.

At the time of writing this, Capistrano is at version 3.

Installation

It's bundled as a Ruby gem which means you need a Ruby environment to run it (version 1.9.3 or later). How you install Ruby is up to you and varies for different operating systems; I suggest starting a Ruby's web page[31] for instructions on how to install it on your system. After you have a Ruby environment, installation is as easy as `gem install capistrano`.

Then you can initialize it in your application

[30] http://capistranorb.com/documentation/getting-started/structure/

[31] https://www.ruby-lang.org/en/

```
cd ~/dev/app
cap install
```

And you will now have a structure like this in your application

```
├── Capfile
├── config
│   ├── deploy
│   │   ├── production.rb
│   │   └── staging.rb
│   └── deploy.rb
└── lib
    └── capistrano
            └── tasks
```

As you can see, by default it sets up a structure you could easily extend. It adds staging and production as environments since they are the most common ones, but nothing will stop you from setting up more environments if needed. A bunch of boilerplate configuration is provided for you in the configuration files. Most of it is commented out but it gives you a good overview of what is possible and how you can achieve certain things. Some parts are non-optional and we will go through them to make you application deployable.

Global configuration

You can find the global configuration in `config/deploy.rb` and this is where you want to start. For now we set the application name and the remote repository.

```
~~set :application, 'my_app_name'~~
~~set :repo_url, 'git@example.com:me/my_repo.git'~~
set :application, 'my-app'
set :repo_url, 'https://github.com/modess/deploy-test-application.git'
```

Capistrano now knows what our application is called and where to locate the remote repository. The remote repository is where is will pull code from when deploying. We'll also change the path of where to deploy it, and we'll stick to our regular path for examples: `/var/www/app`.

```
~~# set :deploy_to, '/var/www/my_app_name'~~
# set :deploy_to, '/var/www/app'
```

This is all we'll do for our global configuration now.

Separate your application and deploy process

Noticed that we specify a remote repository for our deploys? This means we can have separate repositories for our application and our deploy process. Separating concerns is usually a good thing, keeping things nice and tidy.

Server configuration

Now it needs to know where it should deploy to, and that is a server. We will not care for a staging environment at this point so we'll only focus on the production environment. In the file `config/deploy/production.rb` we can configure that.

```
~~# server 'example.com', user: 'deploy', roles: %w{app db web}, my_property: :my_\~~
~~value~~
server 'my-app.com', user: 'niklas', roles: %w{app}
```

This is a very simple setup, we've specified that our production server is *my-app.com*, I want to deploy as the user *niklas* and the role for this server is *app*. You would change the user you want to deploy as and don't mind the role part for now.

We can actually deploy the application now by running `cap production deploy`, however the application will not work since we haven't installed our dependencies.

Installing dependencies

A plugin for Composer is provided as an official plugin for Capistrano. It's also provided as a gem, so first we install it on our system with

```
gem install capistrano-composer
```

And then we need to add a simple line in `Capfile` for requiring it

```
require 'capistrano/composer'
```

You know what the best part of this is? Capistrano will now always run a Composer install for our deploys. If you run `cap production deploy` now you should be able to see the command and its flags in the output:

```
[...]
INFO [c8e8e7be] Running /usr/bin/env composer install --no-dev --prefer-dist --n\
o-interaction --quiet --optimize-autoloader as niklas@my-app.com
[...]
```

File permissions

Either your application is working now or it is crashing due to the fact that it's not allowed to write to the `storage/` directory in the application. We start by installing the file permission plugin for Capistrano on our system

gem install capistrano-file-permissions

We then require it in our `Capfile` and use it to set proper permissions

```
require 'capistrano/file-permissions'
```

And finally we add the path, user and running the file permissions in our deploy workflow. This we do in our global configuration file `config/deploy.rb`:

```
set :file_permissions_paths, ["storage"]
before "deploy:updated", "deploy:set_permissions:acl"
```

It will now set the correct file permission for our storage folder on each deploy.

Shared data

We still have one thing we need to fix for our deploy, and that is shared data. Every time we deploy we will get a new life span for the application, since we save the first run to a file in our application located at `storage/first_deploy`. We need this to persist between our deploys. Luckily for us, Capistrano provides a way for shared folders and files.

We move back to our global configuration file `config/deploy.rb` and find the line where `linked_dirs` is commented out and change it to an appropriate value for our application.

```
~~# set :linked_dirs, fetch(:linked_dirs, []).push('log', 'tmp/pids', 'tmp/cache', \~~
~~ 'tmp/sockets', 'vendor/bundle', 'public/system')~~
set :linked_dirs, fetch(:linked_dirs, []).push('storage')
```

Now we can deploy as many times as we want, but the application life span is never changed due to the fact that our storage folder is persisted between all deploys.

Pros and cons

The good parts are:

- Atomic deploys by design
- Structural overview of your deploy
- Separate your deploy process from your code
- Plugins

The less good parts are:

- Ruby based, so you need an environment for that

5.4 Rocketeer

If you want a tool that is tightly integrated with PHP, fast, modern and easy to use, this could be the tool you're looking for. It has taken many proven concepts from predecessors and made it into a tool worthy of modern development and deployment. It is coded in PHP but it can manage deploying any type of application. It provides a powerful set of features that is probably enough for many applications, but it is also based on a modular approach that enables you to swap, extend or hack parts to fit your particular need if necessary.

It has a very interesting philosophy while being heavy influenced and dependant on the Laravel framework, it has many `illuminate/*` dependencies which is core Laravel packages. Through this it's able to leverage a lot of power for configuration, events, remote execution, cli among other things. There is also a focus on readability and simplicity for the API it provides.

Just as Capistrano this tools treats deploys atomically and has an almost identical folder structure at the remote deploy locations. While we have to mimic this functionality manually with git hooks or Phing, we don't have to worry about it here. Other than it allows us to manage things in multiples; multiple servers, multiserver connections and multiple stages per server.

Installation

There are two good ways of installing the tool, either downloading the *phar* or installing it through Composer. I'm always a fan of using Composer so I would recommend installing it that way:

```
composer require anahkiasen/rocketeer --dev
```

You will then have the rocketeer binary at. `vendor/bin/rocketeer`. The other way is downloading the *phar* and using that instead:

```
wget http://rocketeer.autopergamene.eu/versions/rocketeer.phar
chmod +x rocketeer.phar
```

When running commands, I will simply refer to the binary as `rocketeer`, but you will probably either run `php vendor/bin/rocketeer` or `php rocketeer.phar` depending on how you installed Rocketeer. With the binary in place we can setup our project by running the command

```
rocketeer ignite
```

You will be asked a bunch of questions for the setup process, they are all straightforward and you always have the option to edit them later on. After this, Rocketeer will generate all the files necessary for it to work under the `.rocketeer` folder.

```
└── myapp/
    └── .rocketeer/
        ├── config.php
        ├── hooks.php
        ├── paths.php
        ├── remote.php
        ├── scm.php
        ├── stages.php
        └── strategies.php
```

Configuring

We will configure our application just the way we set up and deployed using Capistrano. We'll start by configuring where our application should reside and be deployed to on our server. We find this in `remote.php` where we make our changes.

```
// The root directory where your applications will be deployed
// This path *needs* to start at the root, ie. start with a /
~~'root_directory' => '/home/www/',~~
'root_directory' => '/var/www/',

// The folder the application will be cloned in
// Leave empty to use `application_name` as your folder name
~~'app_directory'  => '',~~
'app_directory'  => 'myapp',
```

In our previous examples we've saved five previous deployed versions of our application as well. The default setting in Rocketeer is to save four, so let's change that also:

```
// The number of releases to keep at all times
~~'keep_releases'  => 4,~~
'keep_releases'  => 5,
```

One last thing to do in this file and that is making sure our shared storage directory is persisted between deploys.

```
// A list of folders/file to be shared between releases
// Use this to list folders that need to keep their state, like
// user uploaded data, file-based databases, etc.
'shared'        => [
~~    'storage/logs',~~
~~    'storage/sessions',~~
    **'storage',**
],
```

And let us update the writable directories/files so we don't receive errors for trying to set permissions to things that don't exist

```
// Permissions$
////////////////////////////////////////////////////////////////////

'permissions' => [

    // The folders and files to set as web writable
    'files'    => [
~~        'app/database/production.sqlite',~~
        'storage',
~~        'public',~~
    ],

// [...]
```

Now we can deploy our application, just run:

```
rocketeer deploy
```

Our application is now deployed, but there are some more concepts that Rocketeer provides that we should go through so we won't really settle here.

Tasks

For the example application, we actually have a full fledged deploy process. But there is a core concept in Rocketeer I want to discuss anyway because it's central to how the tool work, this is **tasks**. It's a term they've chosen to use and they are used for executing steps inside your deploy process. They can be either one line commands such as `composer install`, or they can be closures that allows for you to write an inline style of task, or they can be separate classes. Most of the predefined features you're presented with is tasks written for the core.

This allows us to easily extend or write new functionality to our deploy process that we can hook into at the points we want in the process, tailoring it for our applications needs. If you're more interested in how the event system works and what hooks are available, read more about it in the events documentation[32]. And if you're more interested in tasks and what you are able to do with them you should read the tasks documentation[33].

Unfortunately as I said before is that Rocketeer has managed all our needs for the deploy. So what we could do is create a simple task that runs just after our deploy and display a one liner of the latest commit made to the repository. We'll do this by adding a closure that hooks into `deploy.after` and just runs a git commands, catches the output and prints it to our console. We do this in `.rocketeer/hooks.php`.

```
    // Tasks to execute after the core Rocketeer Tasks
    'after'  => [
        'setup'   => [],
~~        'deploy'  => [],~~
        'deploy'  => [
          function($task) {
            $task->command->info('Latest commit for the repository:');
            $task->runForCurrentRelease('git --no-pager log --decorate=short --p\
retty=oneline --abbrev-commit -n1');
          }
        ],
        'cleanup' => [],
    ],
```

Now if we run `rocketeer deploy` we see something similar in the output just before the cleanup process starts:

```
Repository is currently at:
$ cd /var/www/myapp/releases/20150717121012
$ git --no-pager log --decorate=short --pretty=oneline --abbrev-commit -n1
[niklas@my-app.com] (production) 551936f (grafted, HEAD, origin/master, origin/H\
EAD, master) sexier frontend and with persistant storage
```

Pros and cons

The good parts are pretty similar to Capistrano, but with a few added benefits:

- Atomic deploys by design

[32]http://rocketeer.autopergamene.eu/#/docs/docs/II-Concepts/Events
[33]http://rocketeer.autopergamene.eu/#/docs/docs/II-Concepts/Tasks

- Structural overview of your deploy
- Separate your deploy process from your code
- Modular
- Plugins

The less good parts are:

- No obvious ones really if you're a PHP developer

6. Version control

This is not a book about version control, neither will this chapter go into how and why you should version control. This chapter will discuss a workflow that is suitable for optimizing your deployment process. You do this through a *branching strategy*.

As said before, Git will be used as a reference in this chapter. But the concepts can probably be applied to most version control systems if you're able to create branches. Why Git is so beneficial in a branching strategy is because of its low overhead for those operations. Branching and merging is mostly fast and simple. Conflicts will arise in all systems.

The overhead in Git when you branch is minimal, we're talking around **4 kilobytes**! And creating a branch takes a few milliseconds of your life. I'm pretty sure no other major version control system can contest that. If you are interested in how this is possible, I'll provide some resources for further learning on Git.

And why is version control important for your deployment process? You might ask. It comes down to being able to get code to its proper places. With a good version control system and a workflow, you can choose what you push code to the different environments. And for this you use a branching strategy. I will propose one in this chapter and I'm not saying that's the one you have to use. There are many strategies out there and you should try to find one that fits you and your team.

The term branching strategy is just a fancy way of explaining when, how and where you branch from and merge to. Think of it as a schematic for your code flow between branches and environments.

6.1 Git-flow

I'm proposing the git-flow branching strategy that is widely used and have a proven record. It probably is the most used branching strategy for Git. It started with a guy named Vincent Driessen who published a blog article called A successful Git branching model[34]. He wanted to share his workflow he had introduced for both private and work related projects. Maybe he wasn't the first, but his article is used as the main reference for git-flow. It has since then been praised and adopted by many people who work with software development. Many companies and open source projects have introduced it as their branching strategy.

There is nothing magical about it though, no unicorns or any of that shit. It's just a branching **strategy**. You can work according to it just using the git binary, or you can use a git extension[35] for it. But all in all it's a way of which branches you have, how you name them and how you merge them.

[34]http://nvie.com/posts/a-successful-git-branching-model/

[35]https://github.com/nvie/gitflow

Some tools have adopted it too. One of the most popular Git clients, SourceTree[36] by Atlassian, have support for it out of the box. You don't even have to install git-flow on your system to work with it in the application. It can convert a current repository to support the workflow and then it helps you with all the steps. I recommend you use a Git client. It makes things a lot simpler and you get a better overview of your repository. Only when I need to do more advanced operations like going through the reflog or such, I will resort to the command line.

6.2 Main branches

There are two branches that has to exists in order for git-flow to work. These are your main branches and serves two different purposes. They are called *master* and *develop*. When converting a repository for git-flow you will end up with these branches. You can name them whatever you want, but I will refer to them as their original names.

Master branch

This is the branch you have in your production environment. It should always be stable and deployable. I'm trying to come up with some other things to say about it, but there is nothing more to it actually. The code here is the face of your application towards its users.

Develop branch

Your develop branch is where all deployable code should be merged to. This is a branch for testing your code before it gets merged into *master*. Once code is in the *develop* it can be pushed to different environments so it can be tested thoroughly. Perhaps you have a QA-person/team that can test your features with manual and/or automated regression tests. But it should also pass some sort of automated testing (continuos integration) like Jenkins or Travis. Our your fellow developer colleagues could test it in a shared environment.

The code here should be stable enough to be merged in to master at any point. But of course bugs will be spotted and dealt with and that's the whole point of this branch. You want your code as tested as possible so you can merge it into master and deploy it into production. You want to be able to do this with as little uncertainty as possible.

6.3 Feature branches

All features you are working on should have its own branch. These branches are prefixed with *feature/*. If for example you're working on a OAuth implementation, it be called *feature/oauth-login*.

[36]http://www.sourcetreeapp.com/

How you name the branches after the prefix is completely up to you though. If you use a service for issue tracking it can be good to have the issue number in the name as well for traceability.

They all start out from develop, and up there as they are done. So a feature branch will branch out from develop and when the feature is complete it goes back there. This is true for all feature branches.

6.4 Release branches

Before a deploy you will create a release branch. It branches off from *develop* and will get a *release/* prefix. Depending on how and what you deploy, the name will differ. If you're working in sprints it could be the sprint number, eg. *release/sprint-39*.

When you have your release branch it can be pushed to the different environments and tested. One of the most important places to do this is in your staging environment. You want to make sure it works in an as close to production environment as possible. Because when it has been tested it will be deployed.

So you have a release branch you want to deploy, then what? Then it will be merged into **both** develop and master. It will also be tagged with an appropriate tag, such as a version number. This ensures that all your deploys correspond to a tag. It's an important concept when you want to roll back, because you then want to roll back to the previously deployed tag.

6.5 Hotfix branches

These are the branches you hope to never use. But you will. Why? Because these are the branches you use when something went wrong and needs to be fixed asap. If you discover some nasty bug in production that requires a quick response, a hotfix branch is your go to guy. Can you guess the prefix for it? Yes, it is *hotfix/*.

When you create one it will branch off from *master*. Since you might already have new changes in *develop*, you do not want those to end up in production yet. You will then fix your bug in this branch. If you have time this should be tested as much as possible too, but sometimes you have to deploy it ten minutes ago.

They are treated just as release branches when merging. Shove it into *develop* and *master*, create a tag and off it goes to production.

6.6 Labs branches

This is outside the scope of git-flow and is only a personal preference I have. Often I end up with things I want to play around with that is not tied to actual deliverables. Then I want to separate my playground from my work stuff, so I prefix them with *labs/*. Often these branches will be played around with and then thrown away. If they do end up being complete I merge them in to *develop* like a regular feature branch.

7. Dependencies

The day has long passed since dependencies in a PHP application was not managed by one or more package managers. Composer made it easy for people to share their packages with the world and let other people to use them, and also contribute to them. It has in a way brought the members of the PHP community closer together. One might of course ask the question on why dependencies is even brought up in a book about deploying applications. The answer is that managing, updating and installing dependencies in a consistent way so they stay synchronized between all environments can sometimes be tricky.

Any modern application have (or at least should have) some dependencies. This does not have to be PHP specific, usually it's a mix of dependencies from one or more dependency management tools. The most popular being Composer, Node Package Manager (NPM) and Bower. I probably don't have to tell you about Composer, it's the de-facto standard for PHP packages your application depend on. NPM is a versatile tool that can be used to install dependencies for tools and both frontend and backend packages. Since *node.js* can be used for both server and client side code, its package manager reflects this. And it's often used for build tools and their dependencies, such as *grunt* or *gulp*. Then there is also Bower which is great at handling dependencies for the frontend, pretty much any assets you can think of can be managed with it.

In this chapter I will not describe how to use these tools; instead I'm going to discuss what they have in common. What they have in common is the way we specify versions for our dependencies, why that is and how we effectively can manage our dependencies when understanding this. They all follow a standard called **semantic versioning**.

7.1 Semantic Versioning

One day Tom Preston-Werner, one of the co-founders of github, decided it was time for a proposal on how software versioning should be done. The result was Semantic Versioning[37] and it has since been widely adopted by developers and tools. Understanding the principles of semantic versioning will make your life managing dependencies easier. The ongoing development is of course available on github[38].

Specification

The foundation is extremely simple, a version consists of three parts. The *MAJOR*, *MINOR* and *PATCH* parts, and they **must** exists on all versions. The format of this is `X.Y.Z` in the order I

[37]http://semver.org/

[38]https://github.com/mojombo/semver

previously stated, this is how I will continue to mention them. `X` for a major version, `Y` for a minor version and `Z` for a patch. If a version is missing any of these parts, it can not be considered to adhere to semantic versioning. I will sometimes write certain parts in lower case, when I do this it's to highlight something of importance for one of the other parts which will be in upper case.

For a quick example: a major release could be `1.0.0`, when a minor version is released it will become `1.1.0`, and if it is patched it will become `1.1.1`. The version can also be suffixed with `-dev`, `-patch`, `-alpha` or `-beta`, such as `1.0.0-RC1`, but I will not go deeper in how the version constraints deal with these edge cases.

The list of how versioning must be applied to comply with semantic versioning is long, but the interesting parts are:

- Once a version has been released, it must not change under any circumstances.
- Major version zero (`0.y.z`) is for initial development only, **anything can change at any time**.
- Major version one (`1.0.0`) must declare a *public API*, this can be in form of explicit documentation or by the code itself.
- Once a public API has been declared, all non-backwards compatible changes must increase the major version (`1.y.z` would become `2.0.0`).
- Minor version (`x.Y.z`) must be incremented if new backwards compatible changes have been made to the public API or if marks anything as deprecated in it. There are also some cases when you may increase it.
- Patch version (`x.y.Z`) must be incremented only if backwards compatible bug fixes are introduced.

Okay, a lot of variables here, but it serves us great purpose to understand this. With this in mind we can define versions for our dependencies with understanding and predicting their update behaviours. Since semantic versioning supports wildcards and constraints, we can more easily predict the way our dependencies will be updated. It can also aid us in selecting dependencies. If we need a package that will be used in a crucial part of the system, we should almost never use a package that does not have a `1.0.0` release; since no public API has been defined and anything can change at any time. This includes breaking changes that could break your application.

Let's say you want to use the popular HTTP client package Guzzle. You can now surely tell the difference between including version `5.2.*`, `5.*.*` and `*` of this package. The last example is a terrible idea, since it will install *all* new versions of the package, including when `6.0.0` will be released. Since that would be a major version, which by definition includes non-backwards compatible changes, it will most likely break your application. Perhaps `5.*.*` will work because it's supposed to include only backwards compatible changes, but could you live with peace of mind hoping that the maintainer of the package won't introduce breaking changes?

Remember, package maintainers are people too and are not fault free. They do all their hard work in trying to help you with their code; but shit happens and you should probably cut them some slack when it happens. The number of possible versions your version constraint accepts will probably correlate to the possibility of your application receiving breaking changes. This results in a balancing act; stability on one side and potential security flaws and bugs on the other side. Finding that balance is not an easy formula; just as many other things it depends on a number of variables.

Version constraints

We now have the background for the versioning of the dependencies we might use; but there are some special operators we can use when defining our version constraints.

If you've ever used Composer, npm or Bower, you have seen package definitions in the corresponding file the package manager uses. It can look like this:

```
"some-package": "1.0.0"
```

This is the most simple example of defining a version constraint to a package, whatever you do it will always install version `1.0.0` of this package. If you would like to include all new patch versions for this major version, you could change it to:

```
"some-package": "1.0.*"
```

If the package maintainer is doing a good job, this should never be an issue and I always prefer this. Since it could fix bugs or security exploits you haven't encountered yet, and by definition is should never include any breaking changes. Trust the maintainer even more? Change it to:

```
"some-package": "1.*"
```

There is hardly any merit in taking this approach. What you're accepting now is added functionality and deprecations to the public API, but no non-backwards compatible changes. If you've built your code depending on `1.1.0` for example, you won't automatically leverage changes in `1.2.0` because it's added functionality. So why risk it? Perhaps one reason might be possible performance improvements, but it's probably still not work the risk. I would argue that even dependencies that are for development only should not take this approach, because it could break a build step or test suite somewhere and stop your deploy or development process. Upgrading a minor or major version should always be an active effort from a developer, making sure things are working.

Ranges

Version constraints can also be specified using ranges, the valid ones are

- greater than: >
- greater than or equal to: >=

- less than: `<`
- less than or equal to: `<=`
- not equal to: `!=`
- between: -

How these work should come pretty easy for a developer, but they also support logical *AND* and logical *OR*. A space or a comma will be treated as a logical *AND*. So `>1.0 <1.1` or `>1.0,<1.1` is the same expression and can be pronounced *greater than 1.0 and less than 1.1*. To make a logical *OR* you use `||`, that could look like `<1.0 || >1.1` which can be pronounced *less than 1.0 or greater than 1.1‘.

Logical operators precedence

A logical *AND* will always have precedence over logical *OR*.

You can also make a range by using a hyphen, with it you can specify a range between two versions such as `1.0 - 2.0`; which should be self explanatory.

Tilde

This is one of two special version constraint operators you will encounter. A tilde translated into english could be: *install this version and upgrade the packages lowest version constraint specified, but never anything higher (and also never a major version)*. Let’s look at some examples.

Will upgrade to:	**2.0.2**	**2.0.3**	**2.0.4**	**2.1.0**
“some-package”: “∼2.0.1”	Yes	Yes	Yes	No

The lowest constraint specified here is the patch version. It will never install any version lower than `2.0.1` (such as `2.0.0`) and it will upgrade to any `2.0.Z` version. But it will **not** upgrade the minor version.

Another example of the tilde operator:

Will upgrade to:	**2.1.1**	**2.1.2**	**2.2.0**	**2.2.1**	**3.0.0**
“some-package”: “∼2.1”	Yes	Yes	Yes	Yes	No

This time our lowest constraint specified is minor version. It will never install any version lower than `2.1.0` (such as `2.0.0` or `2.0.8`) and it will upgrade to any `2.Y.Z` version. But it will **not** upgrade the major version.

And the final example:

Will upgrade to:	**2.0.1**	**2.0.2**	**2.1.0**	**2.1.1**	**3.0.0**
"some-package": "~2"	Yes	Yes	Yes	Yes	No

Notice something familiar with this? It behaves the same as ~`2.1`, the difference is that it can install lower versions than `2.1.0`.

Caret

This is the other kind of special operator for version constraint, the caret `^`. It behaves slightly different than the tilde operator does, being more or less conservative in certain scenarios. We'll run through examples for this as well:

Will upgrade to:	**2.0.2**	**2.0.3**	**2.0.4**	**2.1.0**	**2.2.1**	**3.0.0**
"some-package": "^2.0.1"	Yes	Yes	Yes	Yes	Yes	No

This is a very liberal constraint. You tell it to install `2.0.1` and then upgrade to **any higher version that is not a major version**.

On we go:

Will upgrade to:	**2.1.1**	**2.1.2**	**2.2.0**	**2.2.1**	**3.0.0**
"some-package": "^2.1"	Yes	Yes	Yes	Yes	No

This is the exact same as ~`2.1`, but there is a big difference when dealing with packages that doesn't have a major version yet. Let's do the same thing but with `^0.2`:

Will upgrade to:	**0.2.1**	**0.2.2**	**0.2.3**	**0.3.0**
"some-package": "^0.2"	Yes	Yes	Yes	No

If we had used ~`0.2` here instead, it would have allowed anything lower than `1.0.0`. But using the caret on packages without a major release (no public API defined), we can handle these more conservatively. This can be very effective when we want to depend on packages without a major release and want to protect ourself against breaking changes as much as possible.

Using `^2` is the same as ~`2` and `2.*`.

Game of balance

Dealing with version constraints is always a game of balance and the same rules will not apply everywhere. Your applications critical parts should probably never depend on a package with a wide range, but instead be narrow.

Let's take a real world example of this that happened to one of the most popular PHP frameworks, Laravel. A security exploit was discovered in version `4.1.25` that needed an urgent fix. The security exploit allowed for hijacking of "remember me"-cookies used in authentication; this allowed the attacker to remain logged in as another user for a very long time. A patch was written and `4.1.26` was released. What was less known to the community was that this patch introduced a non-backwards compatible change. Anyone who had `4.*` or `4.1.*` for example that updated their dependencies would receive the change which broke their entire application. This caused quite an outrage and people were extremely upset, however the intent of the patch was good and it needed a change to the database schema. The people who had `4.1.25` specified as their version constraint could sit back and watch. They could then pull down the latest version in their development environment, update their code and database schema to align with the patch. When they were done they could easily deploy their application when it was working as intended with the patch. This was also true for people using Composer's lock file (more on that later in the chapter).

This is a great example since it was a dependency that many applications relied on, the ultimate core dependency (if you do not count Laravels dependencies). This dependency were mission critical for all those applications and people lived in good faith of that a patch version should never introduce a breaking change. But it happens, and sometimes must happen. The alternative would be to leave the framework unpatched unless upgrading to a major version, since that is what semantic versioning constitutes. That is not really an option, and suddenly releasing a new major version just for this little security fix would be very troublesome for the maintainer and confusing for the users.

You will have less mission critical parts of your system where you can be more liberal in version constraints. If you have filterable lists in your application that allows your users to export them to spreadsheets; your dependency for generating Excel files probably won't be mission critical. Then you can open up your constraints a bit to allow for bug, security and performance fixes without you having to worry about it.

7.2 Version control

The most frequent question I hear when it comes to dependencies is "should I version control them?" or "should they be committed to the repository?". Short answer: **no**, **never**. Some people will argue that you should version control them with arguments such as you know they will work with the current code or that Github might be down when you need to install or upgrade them. If you know how to manage your dependencies with version constraints, the first issue will never arise. The second issue could arise, and I've been there a few times myself. However it happens so seldom that I consider it to be a problem that can be ignored. If it occurs at the exact same moment you need to deploy an urgent hotfix, well that sucks. But perhaps you then could instead do a rollback to your previous build?

Having dependencies in your version control gets really messy. For one they increase the size of your repository, most often they will be a larger part of your repository than your application. Second they will be part of commits and pull requests, increasing the entropy in both. Pull request should be short and sweet, having an entire package in there will only bloat it. If you instead just change one line in a package managers dependency file, it will be abundantly clear which package and what version of it you're using.

The only exception is when you version control **entire builds** of your application. This could be common when deploying Docker containers or such. You could make it a part of your normal deploy process as well, if you want to save every build of your application. But a git tag for all your build points should suffice.

7.3 Development dependencies

Almost any dependency management tool separates packages in (at least) regular dependencies and development dependencies. This allows for a good separation in your workflow when dealing with dependencies in certain environments. Take *phpunit* for example, this is a package you want to have in your local environment if you need to run unit tests for your PHP code. Perhaps you do continuous integration, then you need it there too to be able to run your test suite. But once your application is deployed and served to your end users, there is no need for this package to be installed. You would never run a test suite in your production environment.

It's because of these scenarios that the dependency management tool separate packages in this manner. Having all your dependencies installed in your production environment will slow down the install/upgrade process for them and it could potentially slow down your application as well. Make sure you separate them appropriately; there is one really simple question when installing a package, and that is "Will I ever need this in production?". If the answer is no, always put it amongst your development dependencies. And never install those dependencies in your production environment.

The default behaviour for the tools is always to install development dependencies, so make sure you turn that behaviour off in your production environments. Here are the flags to use for the different tools when installing or updating without installing the development dependencies:

Tool	**Flag**
composer	–no-dev
npm	–production
bower	–production

7.4 The composer.lock file

Whenever you do an install or update of your dependencies through Composer, a `composer.lock` file will be generated along side your `composer.json` file. There have been some discussion on the lock file in the community. Usually it's people making an effort to inform others on what the lock file provides and how it makes Composer behave when running *install* or *update*. So should you commit it to your repository and version control it? The answer is in most cases **yes**. Since this book is about *applications*, the answer should be yes. If you're instead developing a component or a library, you should turn to the interwebs for further advice.

So what does this file actually achieve? Except from generating merge conflicts in your repository?

It registers exact versions of the installed dependencies. If you install version `1.2.3` of a package, commit the lock file and push your changes; when your co-coder pulls down your changes to the lock file and runs `composer install`, she will get version `1.2.3` even if the version constraint is `1.2.*` and `1.2.7` is the latest patch version.

So you should get into the habit of committing the lock file to any repository for your applications. And also try to use `composer install` instead of `composer update` unless you willingly want to update packages. One added benefit to having your lock file version controlled is also that installing dependencies will be faster since Composer does not have to do version discovery (see next section).

I recommend you open a lock file in one of your projects and take a look. It's a json representation of your packages and quite readable to humans.

Updating all dependencies

When you run `composer update`, all dependencies will be updated. Beware that this could affect parts of the code you're not currently working on. If you want to only update a specific package, use `composer update vendor/package` instead.

7.5 Composer and github rate limit

You should keep in mind that Composer uses the Github API for version discovery and downloading dependencies. The API enforces a rate limit which means that you're only allowed a certain amount of requests in a certain amount of time. That could easily get chewed up when installing or updating dependencies through Composer. The big issue with this is that in an automated deploy process it will often halt the install/update of dependencies and ask for your login credentials for github. Logged in users does not have the same rate limit as a guest user and it allows for you to install everything without any problems. But you do not want this prompt in your process because it will stop everything. However there are things you can do to prevent it.

Committing the `composer.lock` file previously mentioned is a good practice, because it tells Composer exactly what version to install. That removes version discovery from the process. If you have `2.0.*` as a version constraint without a the lock file committed, Composer has to query the API for finding the suitable version to install. The lock file will remove the need for it.

The other thing you can do is to generate an OAuth token[39] for your Github account and add it to `~/.composer/config.json` in this format:

[39]https://help.github.com/articles/creating-an-access-token-for-command-line-use/

```json
{
    "github-oauth": {
        "github.com": "your_github_oauth_token"
    }
}
```

Composer will then use that OAuth token when interacting with the Github API and the strict limits will be lifted.

Entering credentials when prompted

If you end up doing a manual install or update and get prompted for credentials, entering them will generate an OAuth token that will be entered in `~/.composer/config.json` for future use.

8. Database migrations

Almost all applications that need to persist data will end up using a database of some sort and in most cases MySQL or PostgreSQL. With this comes a database schema (or structure if you'd like) of tables and columns that is necessary for the application to store data it uses for certain tasks such as state or statistics. The process of dealing with changes to the database schema has often been dealt with manually where in best cases some sort of documentation or sequenced SQL files ending up in a repository for reference.

But now we have entered a new era of PHP application development and the tools for dealing with database changes are now modern and often built into frameworks. The most used term is *migrations* and in this chapter I will use the Phinx[40] as the weapon of choice for handling migrations.

The benefits are many and the two major ones are that you get versions of your database schema where each step either create, delete or manipulate tables or columns in the previous schema. You will get, if you don't manually change it, a chronological list of files that contains your database changes. Adding a new migration file along with your feature or fix you can make sure that the database gets updated or performs a rollback along with the code automatically, once you add this to your deploy process. Which migration files to run next or to rollback is stored in the database itself. There is also an added benefit of getting developers up and running when starting to develop on your application, you do not need to dump your database and import it or keep track of the structure in a separate file. Clone the repository and run the migrations and they should have a fully functional database to work with.

8.1 Installation & configuration

As always I recommend using Composer to install Phinx

```
composer require robmorgan/phinx
```

This will install all dependencies and the Phinx binary at `vendor/bin/phinx`. So let's bootstrap a configuration file:

```
php vendor/bin/phinx init
```

You can now find `phinx.yml` in the root directory of your project. Take a look inside it and set up your database credentials.

[40] https://phinx.org/

8.2 Migration files

Generating migration files is really easy, you run the command `php vendor/bin/phing create NameOfYourMigration` and it will generate a file named `YYYYMMDDHHMMSS_name_of_your_migration.php` in the `migrations/` folder unless you change this in the configuration file.

All migration will end up in files that extends `Phinx\Migration\AbstractMigration` and will initially consist of a `change()` method only. This is a behavior that was introduced in version *0.2.0* of Sphinx and I do not like it. The reason being that it introduces magic in our migrations by trying to guess how to revert changes and leaves you with no option of modifying that, also it only works with a certain set of operations. I prefer the old fashioned way of using `up()` and `down()` since it enables us to be flexible and I will in the next section continue my reasoning on this. So I always start with replacing the contents of my newly generated migration file with:

```
public function up()
{
    // Up changes goes here
}

public function down()
{
    // Down changes goes here
}
```

These two methods is where you put your intended changes to the database. `up()` will be the method for when you migrate and the `down()` method is used for reverting the changes you made in the `up()` method, this is to provide consistency in migrating and performing rollbacks. So in a case of creating a table for users when migrating, you would drop that table when performing a rollback.

We can now start to adding our changes to the migration file. Let's create that user table I've been going on and on about, it would look like this.

```
    public function up()
    {
        $table = $this->table('users');
        $table
            ->addColumn('email', 'string', ['null' => false])
            ->addColumn('password', 'string', ['null' => false])
            ->addIndex('email', ['unique' => true])
            ->save();
    }

    public function down()
    {
        $this->dropTable('users');
    }
}
```

A simple table with names `users` than contains string columns for e-mail and password that can't be null. It also adds a unique index for the e-mail column since our database should never be able to store multiple users with the same e-mail. When we revert this change we simply drop the table.

The valid column types that you can pass to `addColumn()` are:

- string
- text
- integer
- biginteger
- float
- decimal
- datetime
- timestamp
- time
- date
- binary
- boolean
- enum (MySQL only)
- set (MySQL only)
- json (Postgres only)
- uuid (Postgres only)

Along side these you can apply options to the column, such as we did with `null`, and they can be found in the Phinx documentation here[41].

At this point we realize we want to keep track of when each user was created, so we need a column for that in the user table. We first generate a new migration file:

```
php vendor/bin/phinx create AddCreatedAtToUsersTable
```

We can now select our `users` table again and apply the changes we want and also provide a revert where we simply drop the column.

[41]http://docs.phinx.org/en/latest/migrations.html#valid-column-options

```
    public function up()
    {
        $table = $this->table('users');
        $table
            ->addColumn('created_at', 'datetime')
            ->save();
    }

    public function down()
    {
        $table = $this->table('users');
        $table
            ->removeColumn('created_at')
            ->update();
    }
```

8.3 Possible data loss on rollbacks

Something important to bear in mind when reverting changes is that you do not want to accidentally lose vital information. Say you implement a feature for users to pay for a subscription of some sort in your application and you create a column for storing a timestamp when their subscription expires. If you end up having to rollback this feature after a couple of days and your migration files just simply delete this column you could end up in a situation where you don't know how long a user should have their subscription.

To prevent this you can in your migration scripts either rename columns or tables instead of deleting them or you can copy the data to a temporary table or column. When the rollback is performed you can then manage the data manually since it will be outside of your automated changes to the table or column. More than often I do not encourage doing things manually but this is one occurrence where it really can not be dealt with in an automated way.

8.4 Batches

Your application will most likely not be deployed feature by feature, most likely it will be deployed release by release where each release contains one or more features. So a release could consist of more than one migration file, even a single feature could contain more than one. When you deploy and your deploy process automatically applies these changed it will take all the migration files that wasn't included in your latest release and run them against your database. All and well until you want to do a rollback when your latest deploy had multiple migration files because now it will only take the latest migration file and revert the changes in those. The chain of automation is now broken and we need a way of tracking the batches of migration files.

This example illustrates the problem:

```
├── Release #1
| 20150817080530_create_users_table.php
| 20150818081646_add_created_at_to_users_table.php
├── Release #2
| 20150831053347_add_username_to_users_table.php
| 20150831053404_add_unique_username_index_on_users_table.php
└──
```

After deploying our two releases we perform a rollback, our database will then be at the state of *20150831053347_add_username_to_users_table.php* since only the latest migration will be reverted. This puts our application and database out of sync unless someone manually performs another rollback on the database (this could also mean performing multiple rollbacks, one for each change). So we need a way of tracking our migrations as batches so we can pinpoint where we want our rollback to revert to. Unfortunately this is not something that Phinx provides for us and they have a reason[42] for it.

I have not found any good tools for dealing with these situations outside of frameworks. *Laravel* for example solves this by tracking the batch number inside its migrations database table and performs migrations and rollbacks based on that. What you need is a persistent storage for the latest migration version of each batch which can be in a file, in a database or in a cache layer. If you put it in a cache layer you need to make sure your cache layer is persistent and not only an in-memory cache. I've created a very simple class for dealing with persisting the batches to a file, nothing fancy and very prone to errors but its only purpose is to show how it can be dealt with. For example it can't accept any parameters to be passed to Phinx, it doesn't deal with any errors that might occur but instead it will only perform the necessary actions if the Phinx binary responds with an exit status of 0, which in Linux language means that everything is okay. You can take a look at the example, but *I urge you never to use it in production.*

[42]https://github.com/robmorgan/phinx/issues/99

9. Running tests

We can (almost) all agree on that tests are good for software and they should be written to some extent. How large your test suite is and how specific it is depends on your application and your teams culture of writing tests. *There are no rules* when it comes to testing, only subjective ideas and thoughts. Some advocate for 100% code coverage and this can come from trying to force developers to write tests, or that someone is a neat freak, or that someone just finds it a nice and even number. I would say that 100% code coverage is rarely a good approach since it provides a lot of overhead both for your test suite and also in developer time. Finding the sweet-spot in your test suite is usually not a percentage number but instead of finding the right code to test and have good tests in place for it. Setters and getters are usually unnecessary pieces of code to test for example, but a class that deals with credit card payments should be thoroughly tested.

This will not be a chapter on how, why and when to write tests, there are plenty of resources out there that are extensive and written by people with far better testing experience and knowledge. Instead this will be a chapter on the different types of tests and how they can or should fit into your deploy process. There are so many types of tests you can write that it would be impossible for me to cover all of them and instead I will cover the ones most commonly used in a PHP context. Testing is under constant debate without a clearly defined terminology with an array of terms being thrown around. I will not try to define types of tests but instead give a general term for them and explain the thought behind them since it's good to understand them, so you can leverage that instead of a label. Unfortunately though I must use a label when writing about them.

What I do want to get across is that tests should always be part of your deploy process. Tests are perfect for automation and the full test suite should at some point run before code ends up in the production environment.

9.1 Unit testing

The most basic of tests are unit tests and I do not believe there is another term used for these types of tests. A *unit* can be defined as a small, as small as possible in fact, piece of code that can be isolated and subjected to an automated test. They are fast, sharp and precise like a frickin' laser beam. Also no tests should depend on the state of another test but each one should run on a clean slate. The most common tool used for unit testing in the PHP world is PHPUnit and I doubt there will be a successor to it soon.

Unit tests are all about finding bugs as soon as possible. It's a kind of first line defense against errors in your code and with a TDD[43] workflow, where you write tests first, you can often reduce the

[43]https://en.wikipedia.org/wiki/Test-driven_development

amount of possible bugs down the road. Write a test, watch it fail, write code, check if code passes through tests, rinse and repeat. The TDD world is not without opponents of course and there was a lot of talk going on in the developer community when David Heinemeier Hansson release his blog post on "TDD is dead. Long live testing."[44] back in 2014. Take a look at both camps and see what you find.

9.2 Acceptance testing

In the agile software world we work with specifications and use cases. These will often be transformed into a specification of behavior, or business logic that a certain feature should comply with. Acceptance testing is used if you use behavior driven development (BDD) with tools such as Behat[45], PHPSpec[46], etc. In some cases these tests will use Selenium or Mink to run automated tests against a browser which makes it questionable if it's an acceptance test at that point, in other cases these tests will be quite fast to run. I argue that in cases of automated tests against a browser it's an end-to-end test.

When performing an acceptance test you're trying to answer the question if the feature was built correctly according to a specification. A test could run against a user story that is converted to the Cucumber language, such as this user registration feature:

```
Feature: User registration

  Scenario: Registration form submission
    Given I am a guest user
    And I enter the email "foo@bar.com"
    And I enter the password "abc123"
    When I submit the registration form
    Then I should have a user registered with the email "foo@bar.com"
```

This type of testing is great since it's a very readable format that can be shared between developers and non-developers so they can agree on the specification before the developer implements it. It removes language barriers and translation layers, and allows for better communication. If you're more interested in BDD I suggest you take a look at the talk Taking back BDD[47] by Konstantin Kudryashov.

9.3 End-to-end testing

These automated tests run against an emulated browser which often makes them slow to run, but they can ensure software quality against an entire system (such as a web page). They could perhaps

[44]http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html
[45]http://behat.org/
[46]http://www.phpspec.net/
[47]https://skillsmatter.com/skillscasts/6240-taking-back-bdd

make sure users are able to get through the entire process of registering, logging in or making a credit card payment for example. The test runner needs to start a browser and click through the process while waiting for the browser like a regular user which can be very time consuming.

A common tool used for this in PHP is Mink[48] which has support for a various number of drivers that can emulate web browsers. The different drivers have different modes of operation and in that way have different speed as well. Some are just headless browsers that are quite fast but lack support for Javascript, would you want to test a registration form that sends an XHR-request for example you're out of luck with them. But then there are some that can start full feature web browsers such as Chrome, Firefox or Internet Explorer, but these are a lot slower since a browser needs to be started and the tests will have to use it as a regular user would.

9.4 Manual testing

Unless your doing continuous deployment you will do manual testing somewhere, whether it is only in your local environment by yourself or by a quality assurance (QA) team/person. You pretty much sit like a monkey and click on and type in various things to make sure it doesn't break, you're probably trying your best to make it break. That was maybe harsh on QA-people, it takes skill and experience to perform good manual tests and trying to find the edge cases where a feature can break.

You should try to apply *hallway usability testing* whenever possible which is defined by Joel Spolsky in the classic The Joel Test: 12 Steps to Better Code[49] as:

> A hallway usability test is where you grab the next person that passes by in the hallway and force them to try to use the code you just wrote. If you do this to five people, you will learn 95% of what there is to learn about usability problems in your code.

With this technique you will find a lot of usability issues and bugs you can fix before deploying. Do this as early as possible in your chain of environments to make sure you catch stuff you then have time to fix. Doing this on a complete release branch pushed to staging could end up delaying the deploy or lead to a *we'll fix it later* issue.

9.5 What, where and when

To fit the tests into your deploy process is determining what should run where and when. If you have a large application with lots of tests that takes a long time to run, running your test suite on each commit is probably going to annoy the hell out of every developer. You want your test suite to be as fast as possible of course, but some tests such as end-to-end tests will probably be slow.

[48]http://mink.behat.org/
[49]http://www.joelonsoftware.com/articles/fog0000000043.html

You need to find a good balance here, ultimately we want to find any errors before our code gets deployed in production.

In a best case scenario you have a continuous integration environment, such as TravisCI, Jenkins or CodeShip to name a few. It can be responsible of picking up changes and running through your entire test suite once you push to a repository. If you do not have this luxury I would recommend running any time consuming tests as early as possible just after a developers local environment. Developers should be able to run unit and acceptance tests in their local environment without any issue since they should be fast. But other than that it's good to run them in your development or staging environment once a developer pushes changes to the main repository. Having a set up where running the entire test suite in your development environment that pushes all the changes to your staging environment on a successful test run is to strive for. Most important is to **automate running tests**.

9.6 Failing tests

So how would you deal with a failing test? Ignoring failing, incomplete or skipped tests will lead to code rot in your test suite and I advice you to never let this happen. In accordance to the broken window theory it will likely end up with no new tests being written, old tests not getting fixed properly and eventually you will have to scrap and rewrite your test suite. Going back to Joel Spolsky's The Joel Test: 12 Steps to Better Code[50], number five on the list is *Do you fix bugs before writing new code?*. This one can not really be applied here but I suggest a new bullet to the list: *Do you fix tests before making a deploy to production?*.

It's important to **stop** your deploy process once a test fails, make sure the right people gets **notified** and that it's managed as a prioritized task to fix the code or the test to make it pass. A deploy with any kind of failing tests should never be allowed to complete in your production environment. This is true even if you know why a test is failing and know that nothing is broken. Fix it straight away and start the deploy process again. It will promote a good culture and a good test suite for your application that is not subject to rot.

[50]http://www.joelonsoftware.com/articles/fog0000000043.html

10. Logs and notifications

In the list in the goals chapter we discussed that we want our deploys to be *responsive*. We want it to be aware of what is happening and respond to that in certain ways, for example when there an error.

Since we're always dealing with running certain commands in our deploy process, through tools or running arbitrary commands, we will have console output. Output can be verbose sometimes and that is more than often a good thing and something we can use to our advantage. Capturing output to log files or some other storage can be crucial in troubleshooting deploys that goes sideways. A silent failure in your deploy can take a lot of time and resources in debugging and fixing. If you capture output and store it somewhere where you or others can find it will make it easier.

But we don't just want to just capture the output. We also want to know when something goes the way it should or when something goes wrong. We can notify people about this through various channels. Getting a notification when a deploy is successful or when an error occurs and it is aborted gives peace of mind. A good example can be that after a successful deploy you get a notification somehow saying that "Deploy successful: [11c287] Fixed bug #215", here you can easily see that the deploy was successful and that the latest commit deployed is *11c287*. The same applies (even more so) for "Deploy failed: could not migrate database". In that event you hopefully have a log somewhere where you can go and find out exactly what made the database migration fail, maybe it could not connect to the database or it tried to perform an operation that was not supported.

10.1 Saving logs

There are two kinds of deploys and they need to be handled according to how they operate. The first kind is running a tool on your local computer that sends commands to your server(s) for deploy, this is true for tools such as Capistrano and Rocketeer. Then there is the other kind which runs on the server side under certain conditions, for example when using Git hooks or Phing. So a deploy could in a sense happen on the server side or through the client side.

When performing a deploy through the client side you most likely won't have to save your logs, since you will get the full output in your terminal. Exporting this to some central log storage would probably be a complex and unnecessary procedure. You will most likely get a clear idea of what happened through the output of your tool. But when a deploy runs server side it becomes important to save the log(s). Where you save them is not really important as long as everybody can find them easily, a timestamped log file on each separate server is probably sufficient.

When running a bash script, like demonstrated with Git hooks in the tools chapter, you can easily capture all output to a timestamped log file using this:

```
#!/bin/bash

NOW=$(date +"%Y%m%d-%H%M%S")
deploy_log="/var/logs/deploy/deploy-$NOW.log"
touch $deploy_log
exec > >(tee $deploy_log)
exec 2>&1

# The rest of your deploy commands...
```

Any commands that is executed after that will be saved in that log file.

If you don't have a bash script and you instead send a remote command to your server over SSH for example, there's a simple trick to achieve this also. Say you have a server running Jenkins that you deploy through, it runs all the tests and then send a remote command to your server(s) telling it to make a Phing build. You could then send the command:

```
phing build > /var/logs/deploy/deploy-`date +"%Y%m%d-%H%M%S"`.log 2>&1
```

This is possible with any kind of command you can run on your server to capture all output into a log file.

10.2 Types of notification

There are a few ways of notifying people, whether that is when things run smoothly or when things go awry. Which type of notification you'll use will likely depend on the current technology your company or teams use for communication. Here are a few examples of channels you can use for notifications.

E-mail

The most conventional way of sending out notifications is with a good old fashioned e-mail. There are a few upsides to sending notifications through e-mails, one being that you aren't limited to a certain number of characters or lines. You can append full log output in your e-mails for example. One other upside is that everyone has an e-mail address, creating lists and sending notifications to those is a great way of ensuring that people receive and read them. Just make sure that you keep your lists up to date, adding new and removing old people as they come and go.

Tool	**Plugin**
Git hooks	mail[51], sendmail[52]
Phing	MailTask[53]
Capistrano	gofullstack/capistrano-notifier[54], pboling/capistrano_mailer[55]
Rocketeer	Write custom task[56]

[51]http://linux.die.net/man/1/mail
[52]http://www.sendmail.org/~ca/email/man/sendmail.html
[53]https://www.phing.info/docs/guide/trunk/MailTask.html
[54]https://github.com/gofullstack/capistrano-notifier
[55]https://github.com/pboling/capistrano_mailer
[56]http://rocketeer.autopergamene.eu/#/docs/docs/II-Concepts/Tasks

Slack

Slack is the rising star of group communication for companies and their teams, for good reasons. Having a channel for your deploys can be a good way of the right people getting notified of relevant information. They have an API which makes it really to send notifications, and there are many plugins that has been built to use it. If a plugin does not exists for your deploy tool you can always use cURL or something similar for sending requests to it.

Tool	Plugin
Git hooks	API[57]
Phing	API[58]
Capistrano	j-mcnally/capistrano-slack[59], phallstrom/slackistrano[60]
Rocketeer	rocketeers/rocketeer-slack[61]

[57]https://api.slack.com/
[58]https://api.slack.com/
[59]https://github.com/j-mcnally/capistrano-slack
[60]https://github.com/phallstrom/slackistrano
[61]https://github.com/rocketeers/rocketeer-slack

HipChat

Still a popular alternative even though Slack seems to be taking more and more of the market. It is also a great tool for communication for companies and teams. Many plugins exists here as well since they have an API for sending notifications, which also always leave you the option of sending a cURL request or similar for interacting with it.

Tool	**Plugin**
Git hooks	API[62]
Phing	rcrowe/phing-hipchat[63]
Capistrano	hipchat/hipchat-rb[64], restorando/capistrano-hipchat[65]
Rocketeer	rocketeers/rocketeer-hipchat[66]

[62]https://www.hipchat.com/docs/apiv2
[63]https://github.com/rcrowe/phing-hipchat
[64]https://github.com/hipchat/hipchat-rb
[65]https://github.com/restorando/capistrano-hipchat
[66]https://github.com/rocketeers/rocketeer-hipchat

IRC

The technology that never seems to go out of fashion. It's commonly used and have an impressive track record, it has been around since 1988! The amount of characters you can include in your messages might be a bit limiting though, full log outputs is not to consider here. But a "Deploy successful: [11c287] Fixed bug #215" is often enough as a message.

Tool	Plugin
Git hooks	Send message to IRC channel from bash[67]
Phing	Send message to IRC channel from bash[68]
Capistrano	ursm/capistrano-notification[69], linyows/capistrano-ikachan[70]
Rocketeer	Mochaka/rocketeer-irc[71]

[67]http://serverfault.com/questions/183157/send-message-to-irc-channel-from-bash
[68]http://serverfault.com/questions/183157/send-message-to-irc-channel-from-bash
[69]https://github.com/ursm/capistrano-notification
[70]https://github.com/linyows/capistrano-ikachan
[71]https://github.com/Mochaka/rocketeer-irc

10.3 Useful git commands

If you're following the git-flow branching model in the version control chapter, there are some useful git commands you can use. You can leverage the git binary and capture output which you then can include in your notifications. Of course you can capture output for any CLI command that you use to include in your notifications, git is just one example and these are a handful of useful ones.

Current branch

Displaying the current branch can be good to make sure that the right branch is deployed.

```
git rev-parse --abbrev-ref HEAD
```

Commit at HEAD

It's never a bad idea to show where your HEAD pointer is. This tells you exactly where your deploy is.

```
git --no-pager log --abbrev-commit --pretty=oneline -1
```

Latest tag

Including your latest tag in a notification is great since it will tell which version that was just deployed.

```
# For current branch
git describe --abbrev=0 --tags

# Across all branches
git describe --tags $(git rev-list --tags --max-count=1)
```

Commits between latest tag and previous tag

This command allows you to generate a simple changelog with all commits that was deployed. Excellent for appending as complementary information to your notification.

```
git --no-pager log --abbrev-commit --pretty=oneline $(git rev-list --tags --max-\
count=1)...$(git describe --abbrev=0 $(git describe --tags $(git rev-list --tags\
 --max-count=1)^))
```

Conclusion

Thank you so much for purchasing this book and reading it through to the end. This has been a very inspiring and rewarding experience for me, and I hope that you have enjoyed the book and learned things. If you did enjoy it I would appreciate if you spread the word about it. Tweet it, recommend it to friends, share the link, or perhaps talk about it at a meetup you are organizing, speaking at or just attending.

To my best ability I've tried to mix theory with practice. This kind of book would get long and boring if no applicable technical aspect were covered. That is why I covered deploy tools and tried to include code in different chapters, to balance it all out.

I do not know when I will consider my work done on this book. I will continue to revise and add to it from time to time. Deploy tools is something I will continue adding for example. We all know how fast technology moves and I think it is important to try and keep up with it in this book. If you find anything that is outdated or something you find should be added to this book, don't hesitate to contact me.

Thanks again!