Introduction

Before we start talking about technical concepts, let's answer one question: what is DevOps? The answer is more complicated than you might think. A long time ago there was no DevOps but development and operations. Developer teams built the software, and operations teams deployed/operated it.

But it wasn't the best model, because these were two different teams and they always fought each other. It was mostly the developers' fault, in my opinion. A long time ago developers did all kinds of weird stuff. Just to name a few:

- Not using package managers but installing dependencies manually and copying vendor folders to servers. I'm guilty.
- Not using migrations but running SQL scripts manually on the server. I'm guilty.
- Hard-coding secrets and configs into source code. I'm guilty.
- Not using .env files properly (or not at all). I'm guilty.
- Maintaining multiple code bases to deploy multiple tenants. I'm guilty.

In the old days, people were trying to run PHP code such as this:

```
exec("php-cli http://localhost/diplomski/program/defender/tester.php");
```

This example comes from a Stackoverflow question. Just imagine the operations team when they realized that a PHP script trying to access another file on localhost:80. Which is probably where the frontend was running. Does `tester.php` need to be publicly accessible? Maybe, I don't know.

PHP devs were wondering why not just copy the `vendor` folder to the server. They hardcoded port numbers and access keys into the code.

Operations teams were pissed at developers because they applied so many "hacks" and bad practices. Developers were pissed at operations teams because they couldn't run the software smoothly.

And then 12factors came along by Heroku (maybe around 2012) and it showed us the light at the end of the tunnel. It stated things that sound very obvious nowadays. Just to name a few:

- One codebase tracked in version control, many deploys. aka be a reasonable human being.
- Explicitly declare and isolate dependencies. aka use composer.
- Store config in the environment. aka use `.env` files and use them correctly.

So the IT industry said: "Stop this madness! You'll learn to collaborate with each other. We need cross-functional teams that can write, build, deploy, release, and maintain software. We need teams that have all the necessary skill set to do so."

So there was this new thing called "DevOps." The goal was to merge developer and operations-specific knowledge together. And then Docker came out in 2013 and it streamlined the whole process. And then Kubernetes came out in 2014 and solved tons of production issues.

So DevOps is not just a set of cool technologies but a mindset or a culture. To goal is to have automated processes and make the deployment/release process as smooth as possible.

A few key concepts of modern DevOps:

- **Automation**. automating repetitive tasks, such as software builds, testing, deployments, and infrastructure provisioning, to reduce errors, improve efficiency, and increase consistency.
- **Continuous integration and continuous delivery** (CI/CD). Implementing CI/CD pipelines to enable frequent code integrations, automated testing, and rapid, reliable software releases.
- **Infrastructure as Code**. Managing infrastructure resources using version-controlled code, allowing for reproducibility, scalability, and consistency across different environments.
- **Monitoring and feedback**. Implementing monitoring and observability practices to gather feedback on application performance, user experience, and operational metrics, enabling quick detection and resolution of issues.

And now let's make you a decent DevOps guy in PHP/Laravel projects.

Building a pipeline

The project files are located in the `1-fundamentals` and `2-fundamentals-static-content` folders.

In this chapter, we're going to implement a basic but pretty useful pipeline. All pipelines should do at least these things:

- Analyze the code
- Test the code
- Deploy it to a server

So we're gonna do these steps. For now, without docker, just a standard Laravel app deployed to a VPS with shell scripts and GitHub magic. In this book, I'll use GitHub actions since this is one of the most popular CI/CD platforms (in my audience, at least). The concepts I'll talk about are the same no matter if you're using GitLab, Jenkins, or some other tool.

In GitHub, we have a few concepts such as:

- Workflows
- Events/triggers
- Jobs
- Steps

These names and concepts are the main differences across platforms. Let's start at the bottom:

- Step: it's a script (a step) that we want to run. For example, `php artisan test` is gonna be a step inside our pipeline.
- Job: it's a set of steps. For example, we can write a job called `Test` that runs the tests and then copies the generated code coverage report to a specific location. There are two steps in this job.
- Workflow: it's a set of jobs associated with some events. For example, "when someone pushes to the main branch, I want this workflow to analyze, test the code, and then deploy it to a server." Here, the event is "pushing to the main branch" and "analyze", "test", and "deploy" can be jobs. So basically, a workflow is an entire pipeline. We can have more than one workflow.
- Events and trigger: as I said earlier an event is something that "happens" and it will trigger a workflow. Events can be things such as pushing a commit, opening a PR, opening a new issue, closing a PR, and so on.

When we write a pipeline we basically write pretty simple shell scripts and commands (such as php artisan test). These scripts will run on a server. This server is called a runner. It's called a runner because, well, it runs our scripts. GitHub (and GitLab as well) provides free runners. They are of course limited in resources but perfect for smaller projects. And of course, we can snip up our own runners.

These runners will not "remember" any state. So when you trigger a pipeline it'll run on an "empty" server. Means there's no:

- php
- MySQL

- The runner does not know anything about your application (it does not have the code or the required composer packages)

So each pipeline starts with some setup code. These are (more or less) the same things that you use to do when setting up a new project on your local machine. I mean, obviously, you have php, MySQL, nginx, or docker but you still have to:

- Checkout the code
- Create a .env file
- Install composer packages
- Generate an app key

In a pipeline we have to do the exact same things, so let's start!

First, we create a new file: `.github/workflows/workflow.yml` This contains the entire workflow for now.

The first part looks like this:

```yaml
name: First pipeline

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

This defines when the workflow should run. When someone pushes commits to the main branch or opens a PR to the main branch. Later, I'll talk about other branches and staging environments. Right now we're going with this config:

```yaml
on:
  push:
    branches: [ "main" ]
```

Then we can define the jobs:

```yaml
jobs:
  tests:
    runs-on: ubuntu-latest
```

Each job can run on a specific OS. They can even run on multiple operating systems. For most PHP applications it should be some Linux distro since production servers also use Linux (and/or docker so developers also use Linux).

Now let's write some steps:

```
steps:
  - uses: shivammathur/setup-php@15c43e89cdef867065b0213be354c2841860869e
    with:
      php-version: '8.1'
```

I said earlier that a step is basically a shell script, right? I lied to you (partially). Because in GitHub a step can also be an "action." Yes, it's a bit confusing because the whole CI/CD part of GitHub is called "actions" but they also use the term "action" to describe this: `shivammathur/setup-php` If you google this term you'll find a GitHub repository. This is a predefined, reusable step. Setting up PHP on a runner happens very often. For this reason, GitHub offers a vast library of redefined scripts or steps and they call them actions. So that explains this line `uses: shivammathur/setup-php`

We can configure these actions with our own settings. That's exactly what this line does:

```
with:
  php-version: '8.1'
```

Here, we instruct `setup-php` action to use PHP 8.1 You can think about this as a config file when you use a package in Laravel.

All right, so now we have PHP8.1 installed on the runner. That was easy! For now, let's forget about MySQL, and let's continue.

The next step is to checkout the application's code to the runner. That's another step:

```
- uses: actions/checkout@v3
```

Once again, running `cd /some/directory/ && git clone my/awesome-repo` happens so often that there's an action for it. We don't need any configuration. It'll checkout the current repository to the runner.

Now, we have the repo on the runner. The next step is to create a `.env` file:

```
- name: Copy .env
  run: cp .env.ci .env
```

This step does not use a predefined action. It's just a simple shell script. We just copy `.env.ci` to `.env` Why don't we copy `.env.example` you may ask. Because we need valid values for the CI. For example, we need a valid database name, a queue connection (usually redis), and so on. We're going to talk about these later when we need them.

So here's the whole pipeline so far:

```
name: First pipeline

on:
  push:
    branches: [ "main" ]

jobs:
  tests:
    runs-on: ubuntu-latest

    steps:
      - uses: shivammathur/setup-php@15c43e89cdef867065b0213be354c2841860869e
        with:
          php-version: '8.1'
      - uses: actions/checkout@v3
      - name: Copy .env
        run: cp .env.ci .env
```

Now that we have a proper `.env` we can install composer dependencies.

```
- name: Install Dependencies
  run: composer install -q --no-ansi --no-interaction --no-scripts --no-
progress
```

Since we are in a pipeline we need to use some options:

- `-q` is for "quiet" so composer will not output messages
- `--no-ansi` disables ANSI outputs
- `--no-interaction` prevents composer from asking us questions
- `--no-scripts` prevents composer from running script defined in `composer.json`
- `--no-progress` disables progress bar since we don't really need it and it can mess with some terminals

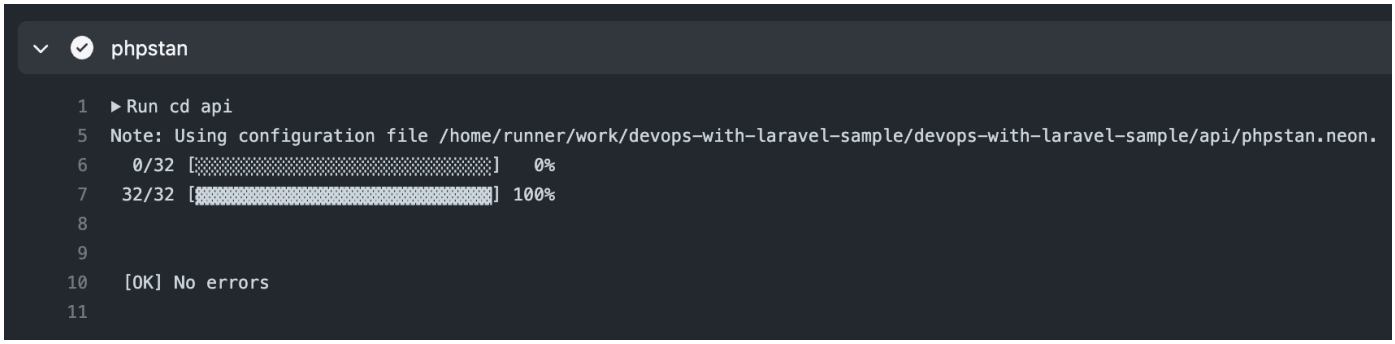
At this point, we have every dependency so we can start running some Laravel-specific commands:

```
- name: Generate key
  run: php artisan key:generate
- name: Directory Permissions
  run: chmod -R 777 storage bootstrap/cache
```

And now we're ready to run the important steps:

```
- name: phpstan
  run: ./vendor/bin/phpstan analyse --memory-limit=1G
```

This step will run phpstan which gives an output such as this if everything's fine:



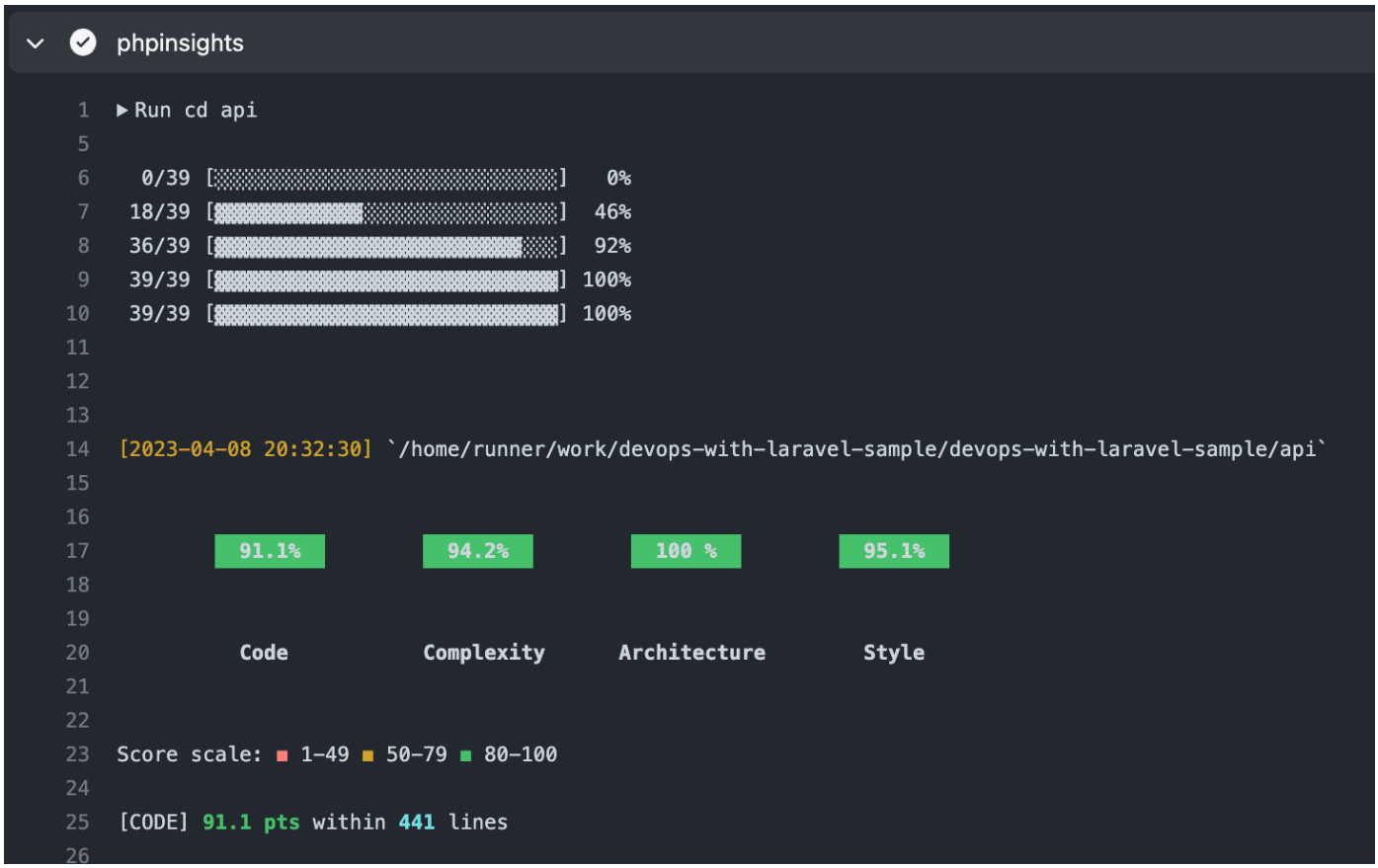
If it finds some errors it'll return `1` and the pipeline will fail. It's true for every step. If the output is not `0` the step (and by default, the whole pipeline) will fail. I'm not going to get into how phpstan works but you can see the config in the `phpstan.neon` file.

The other analysis tool I like to use is `phpinsights`:

```
- name: phpinsights
  run: |
    php artisan insights --no-interaction \
    --min-quality=90 --min-complexity=90 \
    --min-architecture=90 --min-style=90 \
    --ansi --format=github-action
```

This is how we can run multiline commands in GitHub actions. We need to use the `|` character and then we can separate the lines with the `\` sign.

phpinsights scores your code in four different categories on a scale from 0-100. Here we can tell phpinsights: "if the quality score is lower than 90 please make my pipeline fail." It gives an output like this:



Once again, if the return code is not zero the step (and the whole pipeline) fails.

After these steps, we can be sure that the code is up to our standards. Next, we can run the tests. But before we do so, we need to set up a MySQL instance since the tests require a database.

Fortunately, in GitHub it's pretty easy to do so. Each job can define some services such as a database. These services will spin up docker containers. Don't worry! It's managed by GitHub we don't need to write dockerfiles just yet. So to set up a MySQL for the tests all we need to do is this:

```
jobs:
  tests:
    runs-on: ubuntu-latest

    services:
      mysql:
        image: mysql:8.0.21
        env:
          MYSQL_DATABASE: posts-test
          MYSQL_ROOT_PASSWORD: root
        ports:
          - 3306:3306
        options: --health-cmd="mysqladmin ping"
```

At the beginning of the workflow file where we defined the job, we can add the MySQL service. It has the following options:

- `image: mysql:8.0.21` will pull the 8.0.21 MySQL image from Docker Hub
- `env` will set some environment variables to the running container. It basically creates a database with the name `posts-test` and sets up a `root` user with the password `root`.
- `ports` will bind the port `3306` to the running container so we can access it from the tests.
- `--health-cmd="mysqladmin ping"` will run a `ping` command and if it does not get an answer it will wait and send it again. If MySQL isn't responding it will fail after 3 attempts. This is necessary because we don't just need to start the container but we need to actually wait for MySQL to be healthy before we move forward in the pipeline. Otherwise, the tests would fail because they cannot connect to the database.

So we spin up a MySQL server with a database called `posts-test`. But why exactly `posts-test`? On my local machine, I always have a database called `X-test` where `X` is the current project's name. So I set up my `phpunit.xml` this way:

```
<php>
  <env name="DB_DATABASE" value="posts-test"/>
  <env name="APP_ENV" value="testing"/>
  ...
</php>
```

When you run `php artisan test` or `./vendor/bin/phpunit` it will parse the `phpunit.xml` and override values in your environment variables. So this way I have a `phpunit.xml` that defines the test database's name and I can use the same name locally and in CI as well. It's an easy setup. This way I don't have to set up a dedicated `mysql-testing` connection in the `config/database.php` file or something like that. One important thing though: `phpunit.xml` has precedence over the `.env` file. Just keep it in mind.

Now MySQL is ready and finally we can run the tests:

```
- name: Run tests
  run: php artisan test
```

This is the easiest step of all, we just need to run `php artisan test`

Congratulations! You have a basic but effective pipeline. It will analyze the code and run the tests:

```
name: First pipeline

on:
  push:
```

```yaml
    branches: [ "main" ]

jobs:
  tests:
    runs-on: ubuntu-latest

    services:
      mysql:
        image: mysql:8.0.21
        env:
          MYSQL_DATABASE: posts-test
          MYSQL_ROOT_PASSWORD: root
        ports:
          - 3306:3306
        options: --health-cmd="mysqladmin ping"

    steps:
      - uses: shivammathur/setup-php@15c43e89cdef867065b0213be354c2841860869e
        with:
          php-version: '8.1'
      - uses: actions/checkout@v3
      - name: Copy .env
        run: cp .env.ci .env
      - name: Install Dependencies
        run: |
          composer install -q --no-ansi \
          --no-interaction --no-scripts --no-progress
      - name: Generate key
        run: php artisan key:generate
      - name: Directory Permissions
        run: chmod -R 777 storage bootstrap/cache
      - name: phpstan
        run: ./vendor/bin/phpstan analyse --memory-limit=1G
      - name: phpinsights
        run: |
          php artisan insights --no-interaction \
          --min-quality=90 --min-complexity=90 \
```

```
            --min-architecture=90 --min-style=90 \
            --ansi --format=github-action
      - name: Run tests
        run: php artisan test
```

One more thing I didn't tell you. If you check out the pipeline in the sample application you see slightly different steps. Something like this:

```
- name: Generate key
  run: |
    cd api
    php artisan key:generate
```

There's an extra `cd api` before every command. The reason is that the sample project is a "mono-repo" meaning there are three folders in the root:

- api
- frontend
- deployment

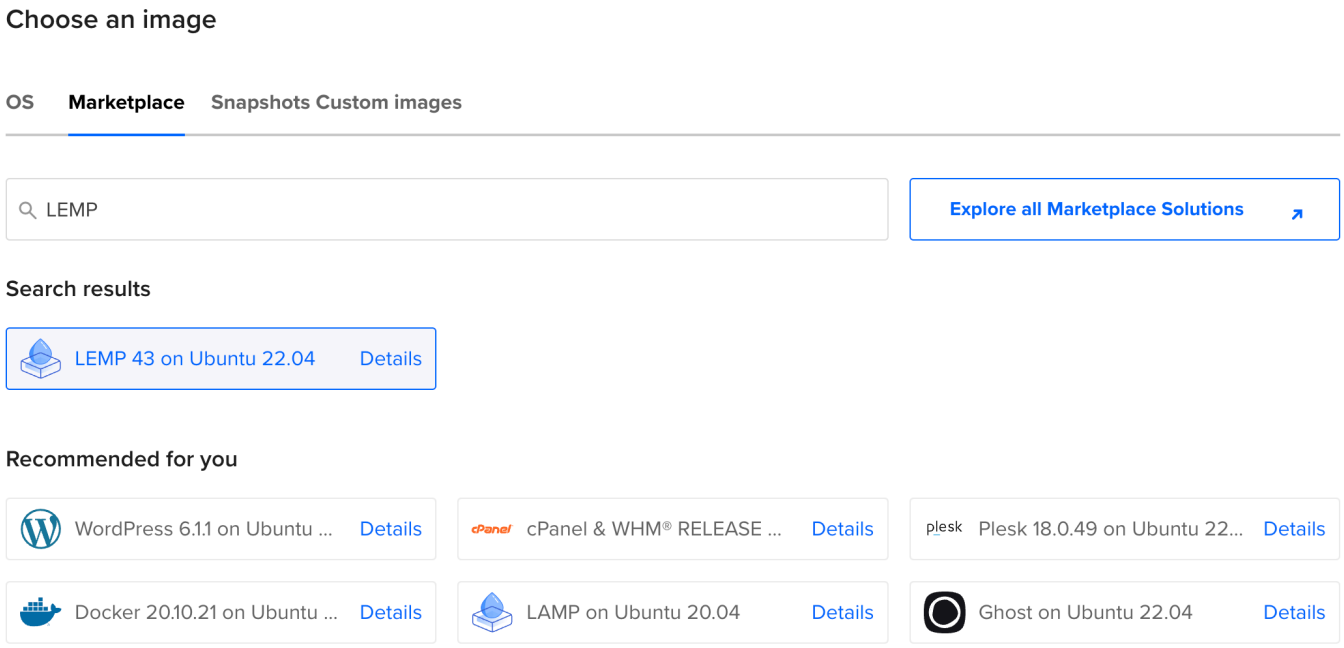
Both the FE and the API live in the same repository but in two different folders. This is why every step starts with a `cd api` command. We need to go inside the api directory where the actual Laravel project lives.

Later, when dockerizing the application we're gonna come back to the pipeline and make it more "advanced." It'll have more jobs, parallel execution, dependencies, etc. But for now, this pipeline does a great job. It's also fast and pretty simple. Which is important.

Now we can move on to deploying the project to a server. To serve an API and a FE we need a web server. So first, let me go through the basics of nginx.

nginx

If you want to try out the things explained here, the easiest way is to rent a $6/month droplet on DigitalOcean with everything pre-installed. Just choose the "LEMP" (it stands for Linux, nginx, MySQL, php) image from their marketplace:



You can destroy the droplet at any time so it costs only a few cents to try out things.

I did the same thing for this chapter. This machine comes with a standard nginx installation so the config file is located inside `/etc/nginx/nginx.conf` That's the file I'm going to edit. The contents of the sample website live inside the `/var/www/html/demo` folder. I use `systemctl` to reload the nginx config:

```
systemctl reload nginx
```

Serving static content

I won't go into too much detail about the basics since nginx is a well-documented piece of software but I try to give a good intro.

First, let's start with serving static content. Let's assume a pretty simple scenario where we have only three files:

- index.html
- style.css
- logo.png

Each of these files lives inside the `/var/www/html/demo` folder. There are no subfolders and no PHP files.

```
# `events` is not important right now..
events {}

http {
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;
  include mime.types;

  server {
    listen 80;
    server_name 138.68.81.14;
    root /var/www/html/demo;
  }
}
```

This is not a production-ready config! It's only for demo purposes.

In an nginx config, there are two important terms: context and directive.

Context is similar to a "scope" in a programming language. `http`, `server`, `location`, and `events` are the contexts in this config. They define a scope where we can configure scope-related things. `http` is global to the whole server. So if we have 10 sites on this server each will log to the `/var/log/nginx/access.log` file which is obviously not good, but it's okay for now.

Another context is `server` which refers to one specific virtual server or site. In this case, the site is `http://138.68.81.14`. Inside the `server` context we can have a `location` context (but we don't have it right now) which refers to specific URLs on this site.

So with contexts, we can describe the "hierarchy" of things:

```
http {
  # Top-level. Applies to every site on your machine.

  server {
    # Virtual server or site-level. Applies to one specific site.

    location {
      # URL-level. Applies to one specific route.
    }
  }
}
```

Inside the contexts we can write **directives**. It's similar to a function invocation or a value assignment in PHP. `listen 80;` is a directive, for example. Now let's see what they do line-by-line.

```
access_log /var/log/nginx/access.log;
error_log /var/log/nginx/error.log;
```

nginx will log every incoming request to the `access.log` file in a format like this: `172.105.93.231 - - [09/Apr/2023:19:57:02 +0000] "GET / HTTP/1.1" 200 4490 "-" "Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko"`

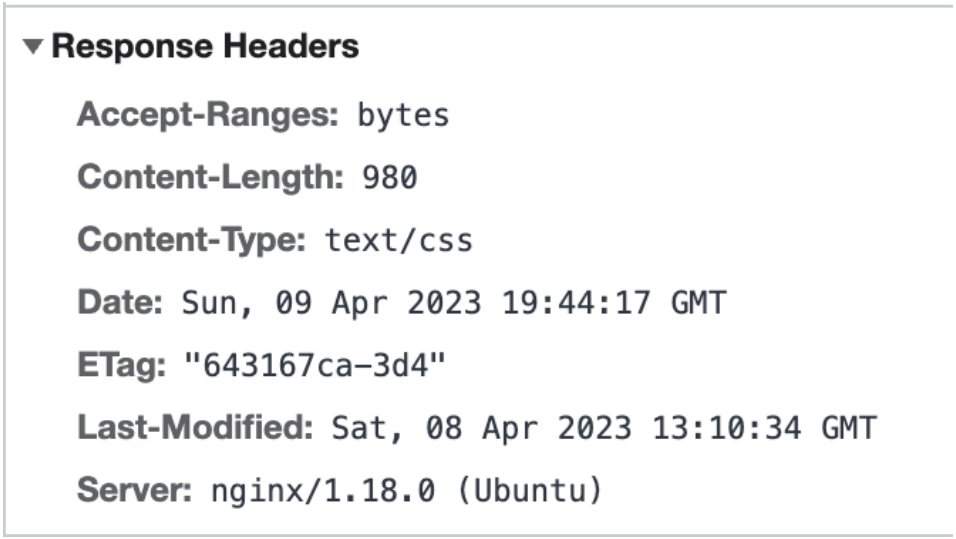
When something goes wrong if logs it to the `error.log` file. One important thing though, a 404 or 500 response is not considered as error. The `error.log` file contains only nginx-specific errors, for example, if it cannot be started because the config is invalid.

```
include mime.types;
```

Do you remember the good old `include()` function from PHP? The nginx `include` directive does the same thing. It loads another file. `mime.types` is a file located in the same directory as this config file (which is `/etc/nginx`). The content of this file looks like this:

```
types {
  text/html  html htm shtml;
  text/css  css;
  text/xml  xml;
  #  ...
}
```

As you can see it contains common mime types and file extensions. If we don't include these types nginx will send every response with the `Content-Type: text/plain` header and the browser will not load CSS and javascript properly. With this directive, if I send a request for a CSS file nginx sends a response such as:



By the way, I didn't write `mimes.type` it comes with nginx by default.

Next up, we have the server-related configs:

```
listen 80;
server_name 138.68.81.14;
```

This configuration uses HTTP without SSL so it listens on port 80. The `server_name` usually is a domain name, but right I only have an IP address so I use that.

```
root /var/www/html/demo;
```

The `root` directive defines the root folder of the given site. Every filename after this directive will refer to this path. So if you write `index.html` it means `/var/www/html/demo/index.html`

By default, if a request such as this: `GET http://138.68.81.14` comes in nginx will look for an `index.html` inside the root folder. Which, if you remember, exists so it can return to the client.

When the browser parser the HTML and sends a request for the `style.css` the request looks like this: `http://138.68.81.14/style.css` which also exists since it lives inside the root folder.

That's it! This is the bare minimum nginx configuration to serve static content. Once again, it's not production-ready and it's not optimized at all.

nginx doesn't know anything about PHP. If I add an `index.php` to the root folder and try to request it, I get the following response:

```
root@testing:/var/www/html/demo# curl http://127.0.0.1/index.php
<?php


echo datetime();
root@testing:/var/www/html/demo# curl -I http://127.0.0.1/index.php
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Sun, 09 Apr 2023 20:52:51 GMT
Content-Type: text/plain
Content-Length: 24
Last-Modified: Sun, 09 Apr 2023 20:51:14 GMT
Connection: keep-alive
ETag: "64332542-18"
Accept-Ranges: bytes
```

So it returns the content of the file as plain text. Let's fix this!

CGI, FastCGI, php-fpm

As I said, nginx doesn't know how to run and interpret PHP scripts. And it's not only true for PHP. It doesn't know what to do with a Ruby or Perl script either. So we need something that connects the web server with our PHP scripts. This is what CGI does.

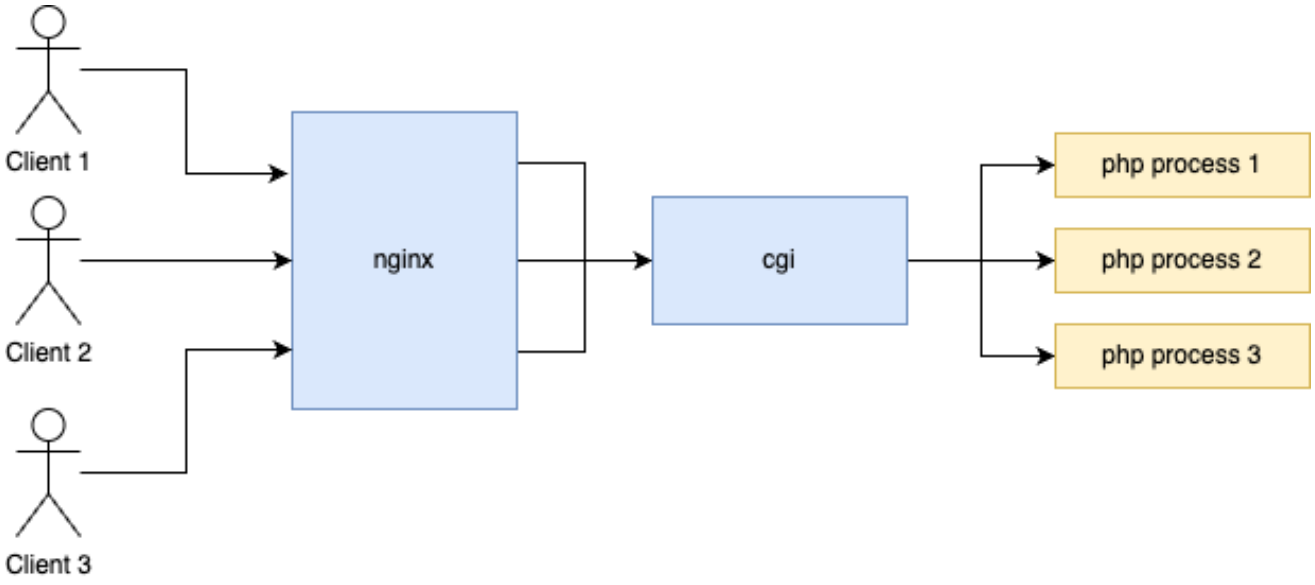
CGI

CGI stands for Common Gateway Interface. As the name suggests, it's not a package or library. No, it's an interface, a protocol. The original specification defines CGI like this:

> *The Common Gateway Interface (CGI) is a simple interface for running external programs, software or gateways under an information server in a platform-independent manner.* - CGI specification

CGI gives us a unified way to run scripts from web servers to generate dynamic content. It's platform and language-independent so the script can be written in PHP, python, or anything. It can even be a C++ or Delphi program it doesn't need to be a classic "scripting" language. It can be implemented in any language that supports network sockets.

CGI uses a "one-process-per-request" model. It means that when a request comes into the web browser it creates a new process to execute the php script:

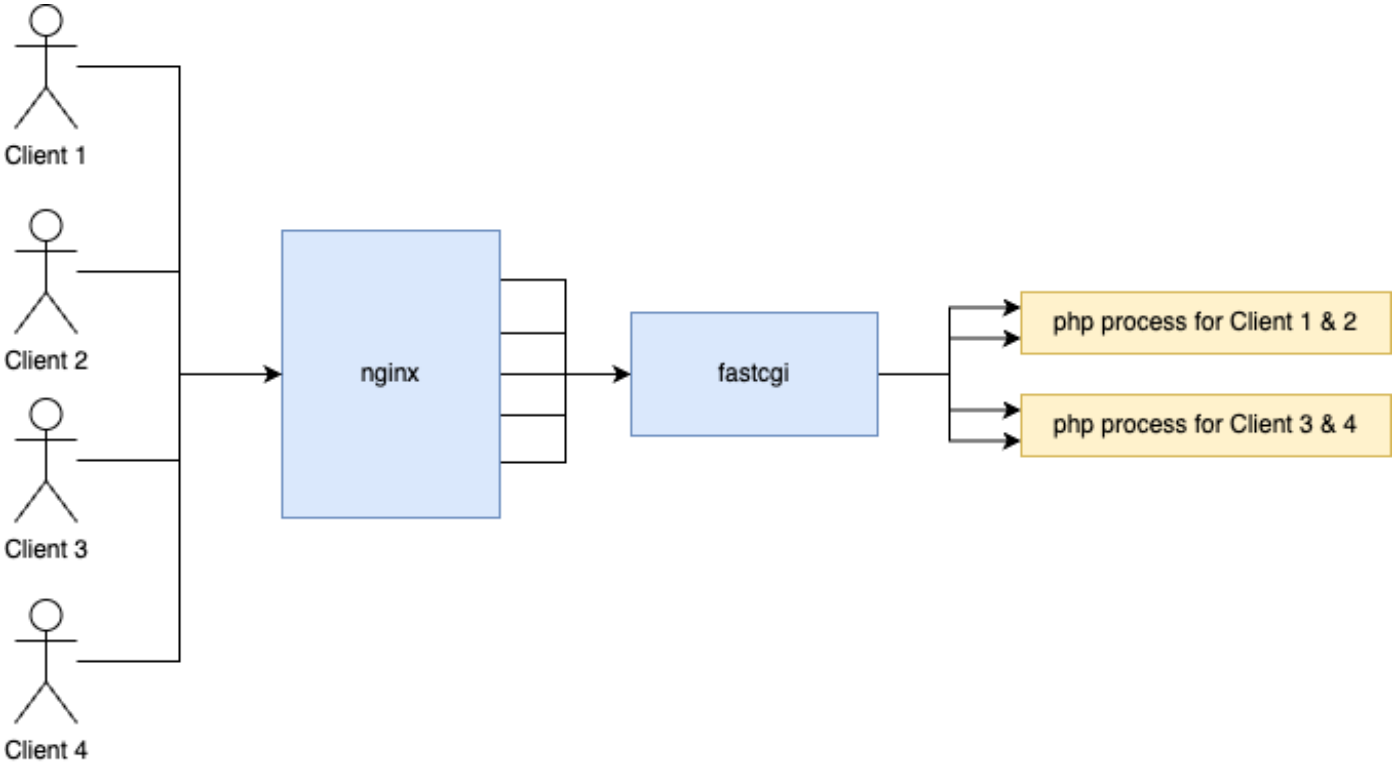


If 1000 requests come in it creates 1000 processes and so on. The main advantage of this model is that it's pretty simple but the disadvantage is that it's pretty resource intensive and hard to scale when there's a high traffic. The cost of creating and destroying processes is quite high. The CPU also needs to switch context pretty often which becomes a costly task when the load is big on the server.

FastCGI

FastCGI is also a protocol. It's built on top of CGI and as its name suggests it's faster. Meaning it can handle more load for us. FastCGI does this by dropping the "one-process-per-request" model. Instead, it has persistent processes which can handle multiple requests in its lifetime so it reduces the CPU overhead of creating/destroying processes and switching between them. FastCGI achieves this by using a technique called multiplexing.

It looks something like that:



FastCGI can be implemented over unix sockets or TCP. We're going to use both of them later.

php-fpm

fpm stands for FastCGI Process Manager. php-fpm is not a protocol or an interface but an actual executable program. A Linux package. This is the component that implements the FastCGI protocol and connects nginx with our Laravel application.

It runs as a separate process on the server and we can instruct nginx to pass every PHP request to php-fpm which will run the Laravel app and return the HTML or JSON response to nginx.

It's a process manager so it's more than just a running program that can accept requests from nginx. It actually has a master process and many worker processes. When nginx sends a request to it the master process accepts it and forwards it to one of the worker processes. The master process is basically a load balancer that distributes the work across the workers. If something goes wrong with one of the workers (for example, exceeding max execution time or memory limit) the master process can kill and restart these processes. It can also scale up and down worker processes as the traffic increases or decreases. php-fpm also helps us avoid memory leaks since it will terminate and respawn the worker process after a fixed number of requests.

By the way, this master-worker process architecture is pretty similar to how nginx works (we'll see more about that later).

nginx and PHP

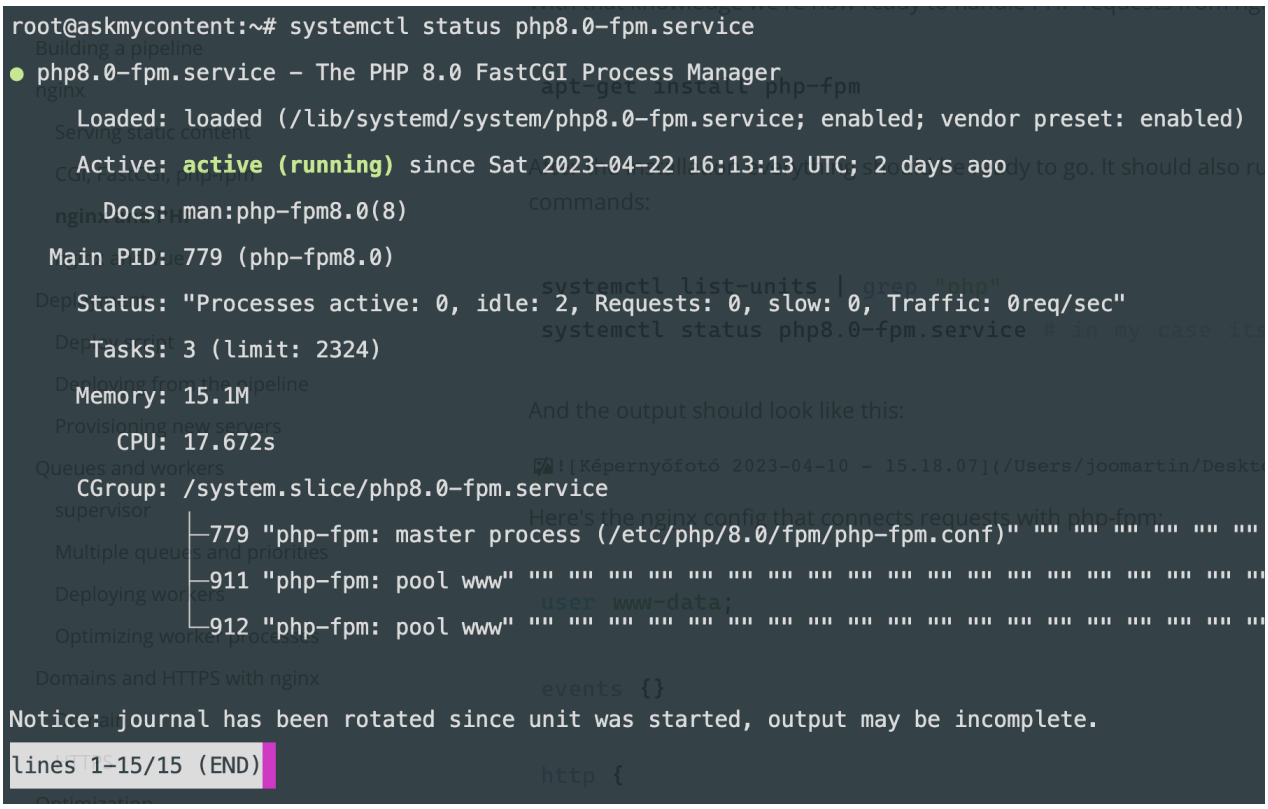
With that knowledge, we're now ready to handle PHP requests from nginx! First, let's install php-fpm:

```
apt-get install php-fpm
```

After the installation, everything should be ready to go. It should also run as a systemd service which you can check by running these commands:

```
systemctl list-units | grep "php"
systemctl status php8.0-fpm.service # in my case its php8.0
```

And the output should look like this:



Here's the nginx config that connects requests with php-fpm:

```
user www-data;

events {}

http {
  include mime.types;
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;

  server {
    listen 80;
    server_name 138.68.80.16;

    root /var/www/html/demo;

    index index.php index.html;

    location / {
      try_files $uri $uri/ =404;
    }

    location ~\.php {
      include fastcgi.conf;
      fastcgi_pass unix:/run/php/php-fpm.sock;
    }
  }
}
```

Most of it should be familiar but there are some new directives. Now we have PHP files in our project so it's a good practice to add `index.php` as the index file:

```
index index.php index.html;
```

If it cannot be found nginx will default to an index.html.

Next, we have this location scope:

```
location / {
  try_files $uri $uri/ =404;
}
```

`try_files` is an exceptionally great name because it literally tries to load the given files in order. But what is `$uri` or `=404`

`$uri` is a variable given to us by nginx. It contains the normalized URI from the URL. Here are a few examples:

- mysite.com -> /
- mysite.com/index.html -> /index.html
- mysite.com/posts/3 -> /posts/3
- mysite.com/posts?sort_by=publish_at -> /posts

So if the request contains a specific filename nginx tries to load it. This is what the first part does:

```
try_files $uri
```

If the request is `mysite.com/about.html` then it returns the contents of `about.html`.

What if the request contains a folder name? I know it's not that popular nowadays (or in Laravel) but nginx was published a long time ago. The second parameter of `try_files` makes it possible to request a specific folder:

```
try_files $uri $uri/
```

For example, if the request is `mysite.com/articles` and we need to return the `index.html` from the articles folder the `$uri/` makes it possible. This is what happens:

- nginx tries to find a file called `articles` in the root but it's not found
- Because of the `/` in the second parameter `$uri/` it looks for a **folder** named articles. Which exists.
- Since in the `index` directive we specified that the index file should be `index.php` or `index.html` it loads the `index.html` under the `articles` folder.

The third parameter is the fallback value. If neither a file nor a folder cannot be found nginx will return a 404 response:

```
try_files $uri $uri/ =404;
```

One important thing. Nginx locations have priorities. So if a request matches two locations it will hit the more specific one. Let's take the current example:

```
location / {}
location ~\.php {}
```

The first location should essentially match every request since all of them starts with a trailing / However, if a request such as `/phpinfo.php` comes in, the second location will be evaluated since it's more specific to the current request.

So the first location block takes care of static content.

And the second one handles requests for PHP files. Remember, Laravel and user-friendly URLs are not involved just yet. For now, a PHP request means something like `mysite.com/phpinfo.php` with a `.php` in the URL.

To catch these requests we need a location such as this:

```
location ~\.php {}
```

As you can see it's a regex since we want to match any PHP files:

- `~` just means it's a regex and it's case-sensitive (`~*` is used for case-insensitive regexes)
- `\.` is just the escaped version of the `.` symbol

So this will match any PHP file.

Inside the location there's an include:

```
include fastcgi.conf;
```

As we already discussed nginx comes with some predefined configurations that we can use. This file basically defines some basic environment variables for php-fpm. Things like these:

```
fastcgi_param   SCRIPT_FILENAME    $document_root$fastcgi_script_name;
fastcgi_param   QUERY_STRING       $query_string;
fastcgi_param   REQUEST_METHOD     $request_method;
```

php-fpm needs information about the request method, query string, the file that's being executed, and so on.

And the last line is where the magic happens:

```
fastcgi_pass unix:/run/php/php-fpm.sock;
```

This instructs nginx to pass the request to php-fpm via a Unix socket. If you remember, FastCGI can be used via Unix sockets or TCP connections. Here we're using the earlier one. They provide a way to pass binary data between processes. This is exactly what happens here.

Here's a command to locate the php-fpm socket's location:

```
find / -name *fpm.sock
```

It finds any file named `*fpm.sock` inside the `/` folder (everywhere on the server).

So this is the whole nginx configuration to pass requests to php-fpm:

```
location ~\.php {
  include fastcgi.conf;
  fastcgi_pass unix:/run/php/php-fpm.sock;
}
```

Later, we'll do the same inside docker containers and with Laravel. We'll also talk about how to optimize nginx and php-fpm.

nginx and Vue

When you run `npm run build` it builds to whole frontend into static HTML, CSS, and javascript files that can be served as simple static files. After the browser loads the HTML it sends requests to the API. Since this is the case serving a Vue application doesn't require as much config as serving a PHP API.

However, in the "Serving static content" I showed you a pretty basic config for demo purposes so here's a better one:

```
server {
  listen 80;
  server_name 138.68.80.16;
  root /var/www/html/posts/frontend/dist;
  index index.html;

  location / {
    try_files $uri $uri/ /index.html;
  }
}
```

As you can see, the `dist` folder is the root. This is where the build command generates its output. The frontend config needs only one location where we try to load:

- The exact file requested. Such as https://example.com/favicon.ico
- An index.html file from a folder. Such as https://example.com/my-folder
- Finally, we go to the index.html file. Such as https://example.com/

This is still not an optimized config but it works pretty fine. We're gonna talk about optimization in a dedicated chapter.

Combined nginx config

The configurations in the last two chapters only work if you have two domains, for example:

- myapp.com
- and api.myapp.com

Which is pretty common. We haven't talked about domains yet, but this would look like this:

```
user www-data;

events {}

http {
  include mime.types;
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;

  # api
  server {
    listen 80;
    server_name api.myapp.com;

    root /var/www/html/demo;

    index index.php index.html;

    location / {
      try_files $uri $uri/ =404;
    }

    location ~\.php {
      include fastcgi.conf;
      fastcgi_pass unix:/run/php/php-fpm.sock;
    }
  }

  server {
    listen 80;
```

```
    server_name myapp.ccom;
    root /var/www/html/posts/frontend/dist;
    index index.html;

    location / {
      try_files $uri $uri/ /index.html;
    }
  }
}
```

But if you don't want to use a subdomain for the API but a URI:

- myapp.com
- myapp.com/api

Then the config should look like this:

```
server {
  server_name myapp.com www.myapp.com;
  listen 80;
  index index.html index.php;

  location / {
    root /var/www/html/posts/frontend/dist;
    try_files $uri $uri/ /index.html;
    gzip_static on;
  }

  location ~\.php {
    root /var/www/html/posts/api/public;
    try_files $uri =404;
    include /etc/nginx/fastcgi.conf;
    fastcgi_pass unix:/run/php/php8.1-fpm.sock;
    fastcgi_index index.php;
    fastcgi_param PATH_INFO $fastcgi_path_info;
  }

  location /api {
    root /var/www/html/posts/api/public;
```

```
    try_files $uri $uri/ /index.php?$query_string;
  }
}
```

This config file has only one `server` block since you have only one domain and it has three locations:

- `/` serves the homepage (frontend)
- `/api` serves API requests and `try_files` will send requests to
- `~\.php` which communicates with FPM

As you can see, each location defines its own `root` folder. We're going to use this approach. One domain and the frontend sends requests to the backend such as `GET /api/posts`. Later, we're going to use a **dedicated reverse proxy** to accomplish the same results but in a cleaner way.

Now that we know the basics, let's write a deploy script!

Deployment

Deploy script

In this chapter, I'm going to write a basic but fully functional bash script that deploys our application. The project is going to live inside the `/var/www/html/posts` folder and the nginx config inside `/etc/nginx`

`deploy.sh`:

```
#!/bin/bash

set -e

MYSQL_PASSWORD=$1

PROJECT_DIR="/var/www/html/posts"
```

The `set -e` command in a bash script enables the shell's `errexit` option, which causes the script to exit immediately if any command exits with a non-zero status code (i.e. if it fails). I highly recommend starting your scripts with this option. Otherwise, if something fails the script will continue. There's also an `x` flag that enables `xtrace` which causes the shell to print each command before it is executed. It can be useful for debugging, however, it's a security risk because the script may print sensitive information to the output. Which might be stored on the filesystem.

The next line

```
MYSQL_PASSWORD=$1
```

sets a simple variable. It's not an environment variable so it can only be used in the current script. It reads the second parameter of the command which is the database password. A few pages later I'm gonna explain why we need it. So actually, the `deploy.sh` needs to be executed such as this:

```
./deploy.sh hL81sP4t@9%c
```

The last line

```
PROJECT_DIR="/var/www/html/posts"
```

sets another variable. It defines the path of the deployment so we can use absolute paths throughout the script.

The next part looks like this:

```
# make dir if not exists (first deploy)
mkdir -p $PROJECT_DIR

cd $PROJECT_DIR

git config --global --add safe.directory $PROJECT_DIR

# the project has not been cloned yet (first deploy)
if [ ! -d $PROJECT_DIR"/.git" ]; then
  GIT_SSH_COMMAND='ssh -i /home/id_rsa -o IdentitiesOnly=yes' git clone
git@github.com:mmartinjoo/devops-with-laravel-sample.git .
else
  GIT_SSH_COMMAND='ssh -i /home/id_rsa -o IdentitiesOnly=yes' git pull
fi
```

The `mkdir -p` command creates a directory and any necessary parent directories. If the folder already exists it does nothing.

This expression `[ ! -d $PROJECT_DIR"/.git" ];` checks if a `.git` folder exists in the project directory and the `!` operator negates the result. So it becomes true if the `.git` directory does not exist. This means this is the first deployment and we need to clone the project from GitHub:

```
GIT_SSH_COMMAND='ssh -i /home/id_rsa -o IdentitiesOnly=yes' git clone
git@github.com:mmartinjoo/devops-with-laravel-sample.git .
```

Using the `GIT_SSH_COMMAND` variable we can override how git tries to resolve ssh keys. In this case, I specify the exact location of an SSH key in the `/home` directory. Later I'll show you how that file gets there. But for now, the important part is that git needs an SSH key to communicate with GitHub and this key needs to be on the server.

If the `.git` directory already exists the script runs a `git pull` so the project is updated to the latest version.

So far this is what the script looks like:

```
#!/bin/bash

set -e

MYSQL_PASSWORD=$1

PROJECT_DIR="/var/www/html/posts"

# make dir if not exists (first deploy)
mkdir -p $PROJECT_DIR

cd $PROJECT_DIR

git config --global --add safe.directory $PROJECT_DIR

# the project has not been cloned yet (first deploy)
if [ ! -d $PROJECT_DIR"/.git" ]; then
  GIT_SSH_COMMAND='ssh -i /home/id_rsa -o IdentitiesOnly=yes' git clone
git@github.com:mmartinjoo/devops-with-laravel-sample.git .
else
  GIT_SSH_COMMAND='ssh -i /home/id_rsa -o IdentitiesOnly=yes' git pull
fi
```

We have the source code ready on the server. The next step is to build the frontend:

```
cd $PROJECT_DIR"/frontend"
npm install
npm run build
```

With these three commands, the frontend is built and ready. Every static file can be found inside the `dist` folder. Later, we're going to serve it via nginx.

To get the API ready we need a few steps:

```
composer install --no-interaction --optimize-autoloader --no-dev

# initialize .env if does not exist (first deploy)
if [ ! -f $PROJECT_DIR"/api/.env" ]; then
    cp .env.example .env
    sed -i "/DB_PASSWORD/c\DB_PASSWORD=$MYSQL_PASSWORD"
$PROJECT_DIR"/api/.env"
    sed -i '/QUEUE_CONNECTION/c\QUEUE_CONNECTION=database'
$PROJECT_DIR"/api/.env"
    php artisan key:generate
fi

chown -R www-data:www-data $PROJECT_DIR
```

First, composer packages are being installed. Please notice the `--no-dev` flag. It means that packages in the `require-dev` key will not be installed. They are only required in a development environment.

Next up, we have this line: `if [ ! -f $PROJECT_DIR"/api/.env" ];` It's pretty similar to the previous one but it checks the existence of a single file instead of a directory. If this is the first deployment `.env` will not exist yet, so we copy the example file.

The next line is the reason why we need the database password as an argument:

```
sed -i "/DB_PASSWORD/c\DB_PASSWORD=$MYSQL_PASSWORD" $PROJECT_DIR"/api/.env"
```

This command will write it to the `.env` file so the project can connect to MySQL. Later, I'm gonna show you where the password comes from, but don't worry, we don't need to pass it manually or anything like that. Now let's focus on the `sed` command. To put it simply: it replaces all occurrences of `DB_PASSWORD` to `DB_PASSWORD=foo` The `-i` flag modifies the file in place, and the `c` command of `sed` replaces the entire line containing `DB_PASSWORD`. By the way, `sed` is a stream editor and is most commonly used for performing string manipulation such as replacing.

We also set the `QUEUE_CONNECTION` to `database`. Later, I'm gonna use Redis but for now, MySQL is perfect.

Remember the

```
user www-data;
```

line in the nginx config? This means that nginx runs everything as `www-data`. For these reasons, I make `www-data` the owner of the project directory. The `-R` flag makes the command recursive.

Finally, it's time for Laravel-specific commands:

```
php artisan storage:link
php artisan optimize:clear

php artisan down

php artisan migrate --force
php artisan config:cache
php artisan route:cache
php artisan view:cache

php artisan up
```

You probably know most of these commands but let's go through them real quick:

- `php artisan storage:link` creates a symbolic link in the `public` folder that points to the `/storage/app/public` folder. So these files are accessible from the web.
- `php artisan optimize:clear` clears three things: config, route, and view caches. We're deploying a new version of our application. It probably has new configs or routes. This is why we need to clear the old values from the cache.
- `php artisan down` puts the application into maintenance mode so it's not available.
- `php artisan migrate --force` migrates the database without hesitation.
- The `:cache` commands will cache the new values from the new files.
- `php artisan up` ends the maintenance mode.

It's important to cache the configs, routes, and views on every deployment since it makes the performance of the application much better. For example, if you forget to run `config:cache` Laravel will read the `.env` file every time you call something like that: `config('app.my_config')` Assuming that the config file reads `MY_CONFIG` using the `env()` function.

So we're actually done with every application-specific code:

```
cd $PROJECT_DIR"/frontend"
npm install
npm run build

cd $PROJECT_DIR"/api"

composer install --no-interaction --optimize-autoloader --no-dev

# initialize .env if does not exist (first deploy)
if [ ! -f $PROJECT_DIR"/api/.env" ]; then
    cp .env.example .env
    sed -i "/DB_PASSWORD/c\DB_PASSWORD=$MYSQL_PASSWORD"
$PROJECT_DIR"/api/.env"
    sed -i '/QUEUE_CONNECTION/c\QUEUE_CONNECTION=database'
$PROJECT_DIR"/api/.env"
    php artisan key:generate
fi

chown -R www-data:www-data $PROJECT_DIR

php artisan storage:link
php artisan optimize:clear

php artisan down

php artisan migrate --force
php artisan config:cache
php artisan route:cache
php artisan view:cache

php artisan up
```

Next we need to update nginx:

```
sudo cp $PROJECT_DIR"/deployment/config/nginx.conf" /etc/nginx/nginx.conf
# test the config so if it's not valid we don't try to reload it
sudo nginx -t
sudo systemctl reload nginx
```

As you can see, it's quite easy.

It copies the config file stored in the repository into `/etc/nginx/nginx.conf` This config file contains both the FE and the API just as we discussed in the previous chapter. It's a server with a single application running on it, there are no multiple apps or multi-tenants so we don't need multiple configs and `sites-available` etc. One config file that contains the FE and the API.

`nginx -t` will test the config file and it fails if it's not valid. Since we started the script with `set -e` this command will stop the whole script if something's not right. This means we never try to load an invalid config into nginx and it won't crash.

If the test was successful the `systemctl reload nginx` command will reload nginx with the new config. This command is zero downtime.

Congratulations! You just wrote a fully functional deploy script. We're gonna talk about worker processes in a separate chapter.

Deploying from a pipeline

Now that we have seen the entire deploy script let's use it from the pipeline! But it needs a few things, first.

An ssh key. If you remember I referenced a file called `/home/id_rsa` in the deploy script to run git commands. It's the pipeline's responsibility to put the file there. But where does it come from and who's ssh key is it?

The ssh key needs to be created first, so it's your key. There are different types of keys but the most popular is RSA. This is how you can create a new key:

```
ssh-keygen -t rsa
```

An RSA SSH key is a pair of keys that consists of a private and a public key. The private key is kept secret and is used to decrypt messages that are encrypted with the corresponding public key. The public key, on the other hand, is shared with others and is used to encrypt messages that can only be decrypted using the corresponding private key.

The private key is the `id_rsa` file and the public is the `id_rsa.pub`.

Next, we need to authorize this key on the actual server. Each server defines a set of public keys that can ssh into the machine. These public keys are stored in the `$HOME/.ssh/authorized_keys` file where `$HOME` refers to the home directory of your user. Inside this file there are public keys:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABgQDaScpakJmTRm73RKaEmTB56tydaoHcnREq7VefTOXv5cpBgR
dZ75fLSo5cO1EpXonmVyx5nZzxyMnmewSJ2KDGlsGeocEHDZYLC7Im/BShYtixm4hYStJAXyG11O/k
Axukis+e36VhJTkpxmJpwCSyb2qCsHl2farHcX/Enn5C2DaHlJ6z10XobUj8Bo7NOiOQuCsm8oqMlb
WyscYTalxuV4Uu36bEs9SZ7EPGq/n2zuyYns+c2un00pW4TpKqLXTdUfHs7xriei8XZr5RSa1XnNbZ
uAFecq8wIMWawphv5YQ3wzb22vaeaKdWWPVm5jH+PlvgnZGJMgX3y/VE6+O55UJ5VlKaesZeCiIN2I
A8GS4HoUpnltHMzLUHbD8T2cZyTC+xFGxZS3ND+d7ctZcMQ5qOOF68GlFT40lRR6G4TgJ6YVorftlv
GW10N7a+tBEDxq3D8prZ2t8lJDvMdM+PebRnfM1Bhz8O0QjtrYDPasZcfev+ljRDNTKNc5nAQyk=
joomartin@Joo-MacBook-Air.local
```

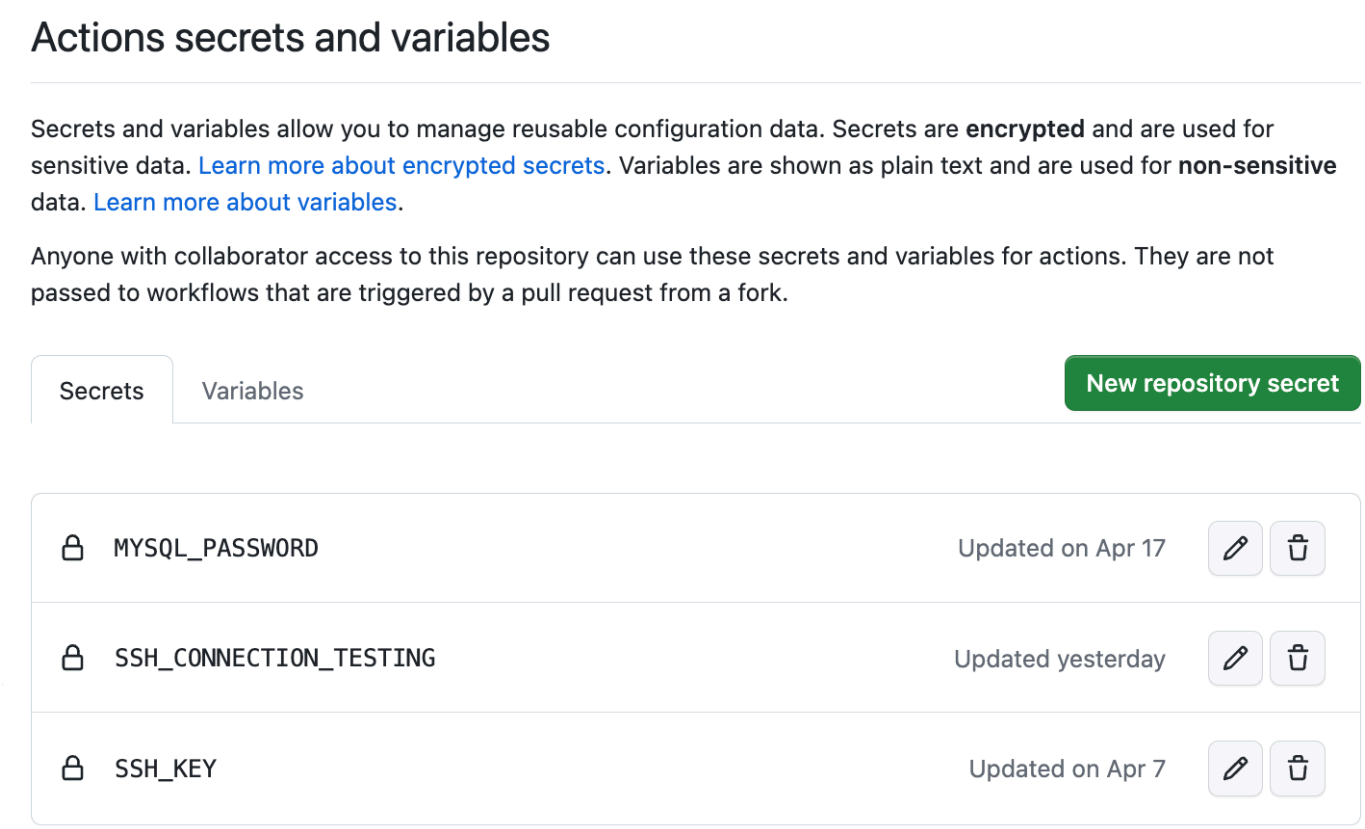
You can just put your public key into this file. After that you need to restart the `ssh` service:

```
systemctl restart ssh
```

So now you have an SSH key on your local machine which is authorized on a server so you can do this:

```
ssh user@1.2.3.4
```

Great! The next step is to use it from the pipeline. Which is not your machine. The pipeline runs on a runner server owned by GitHub. Fortunately, GitHub (and also GitLab) has "Action secrets." This is basically a secret store where we can store sensitive information such as an SSH key.



You can find this under the `Settings/Secrets and Variables/Actions` nav. As you can see, I created a new secret called `SSH_KEY` which contains the **private** key. This is why it has to be treated as a secret. Secrets are not printed to the output when the pipeline is running and their value cannot be seen from the UI.

Secrets can be used from the pipeline such as this:

```
${{ secrets.SSH_KEY }}
```

So now, we have access to a private key that can access a server. In the pipeline, after running the tests, we can copy this value to a file:

```
- name: Run tests
  run: |
    cd api
    php artisan test
- name: Copy SSH key
  run: |
    echo "${{ secrets.SSH_KEY }}" >> ./id_rsa
    chmod 600 id_rsa
```

SSH keys need to have strict permissions otherwise Linux will complain. So we just copy the secret's value to a file. Remember this file lives on the runner server. It will be destroyed after the pipeline is finished. Which is only a few minutes.

After that, the key can be used such as this:

```
- name: Run deploy script
  run: |
    scp -C -o StrictHostKeyChecking=no -i ./id_rsa ./deployment/bin/deploy.sh
${{ secrets.SSH_CONNECTION_STAGING }}:/home/martin/deploy.sh
```

Later in the next chapter, I'm going to introduce the `provision_server` script which can be used to prepare a new server for deployment. I'm going to create a new user called `martin`. This is why I'm copying the file to `/home/martin/deploy.sh`.

We're using `scp` which stands for Secure Copy Protocol. It can copy files to a remote server using ssh. It's a longer command so let's simplify it for a minute:

```
scp ./deployment/bin/deploy.sh ${{ secrets.SSH_CONNECTION_STAGING
}}:/home/martin/deploy.sh

# generic form
scp source_file user@1.2.3.4:/target_file
```

As you can see, before the target file we need to specify the exact SSH connection to connect to the remote server. This is the `user@1.2.3.4` part. But instead of `user@1.2.3.4` I use another secret from GitHub:

```
${{ secrets.SSH_CONNECTION_STAGING }}
```

This variable contains a value such as this: `user@1.2.3.4` where 1.2.3.4 is the IP address of my server and the user is the username.

Another important thing about this command is the `-i` option:

```
scp -i ./id_rsa
```

This is the ssh key we created one step earlier. So scp can actually access the server (where we authorized this exact keys a few steps earlier).

Now only two remaining `scp` options need some explanation:

```
scp -C -o StrictHostKeyChecking=no
```

`-C` enables compression during the transfer, which can improve transfer speed.

`-o StrictHostKeyChecking=no` disables strict host key checking, which means that `scp` will not prompt you to confirm the authenticity of the remote host's SSH key. This option can be useful for automated scripts, however, it also increases the risk of man-in-the-middle attacks.

So with this command, we copied `deploy.sh` to the remote server.

Next, we also need to copy the SSH key. If you remember, the `deploy.sh` script needs an SSH key to run `git` commands. So copy it:

```
scp -C -o StrictHostKeyChecking=no -i ./id_rsa ./id_rsa ${{
secrets.SSH_CONNECTION_STAGING }}:/home/martin/.ssh/id_rsa
```

This is basically the same command but copies another file. `scp` can have some problems copying file permissions so it's a good idea to make the previously copied `deploy.sh` executable:

```
ssh -tt -o StrictHostKeyChecking=no -i ./id_rsa ${{
secrets.SSH_CONNECTION_STAGING }} "chown martin:martin /home/martin/deploy.sh
&& chmod +x /home/martin/deploy.sh"
```

Since here we don't want to copy files but run `chmod` it's a simple ssh command instead of scp. And finally, everything is ready to run the script:

```
ssh -tt -o StrictHostKeyChecking=no -i ./id_rsa ${{
secrets.SSH_CONNECTION_STAGING }} "/home/martin/deploy.sh ${{
secrets.MYSQL_PASSWORD }}"
```

It runs the deploy script on the remote server with two parameters:

- API URL to the frontend
- MySQL password

Both of these values come from GitHub secrets. Here's the full script:

```
- name: Run tests
  run: |
    cd api
    php artisan test
- name: Copy SSH key
  run: |
    echo "${{ secrets.SSH_KEY }}" >> ./id_rsa
    chmod 600 id_rsa
- name: Run deploy script
  run: |
    scp -C -o StrictHostKeyChecking=no -i ./id_rsa ./deployment/bin/deploy.sh
${{ secrets.SSH_CONNECTION_TESTING }}:/home/martin/deploy.sh
    scp -C -o StrictHostKeyChecking=no -i ./id_rsa ./id_rsa ${{
secrets.SSH_CONNECTION_TESTING }}:/home/martin/.ssh/id_rsa
    ssh -tt -o StrictHostKeyChecking=no -i ./id_rsa ${{
secrets.SSH_CONNECTION_TESTING }} "chown martin:martin /home/martin/deploy.sh
&& chmod +x /home/martin/deploy.sh"
    ssh -tt -o StrictHostKeyChecking=no -i ./id_rsa ${{
secrets.SSH_CONNECTION_TESTING }} "/home/martin/deploy.sh ${{
secrets.MYSQL_PASSWORD }}"
```

I used `secrets.SSH` to make the commands a bit shorter but in reality, it's still `secrets.SSH_CONNECTION_STAGING`.

I know it was a lot to take in, so here's a quick summary:

- Create an SSH key locally that is authorized on the server
- Copy it to a GitHub secret

- Write it to a file on the runner
- Copy the deploy script to the actual server
- Copy the SSH key to the actual server
- Make the deploy script executable on the actual server
- Run the deploy script with parameters coming from GitHub secrets

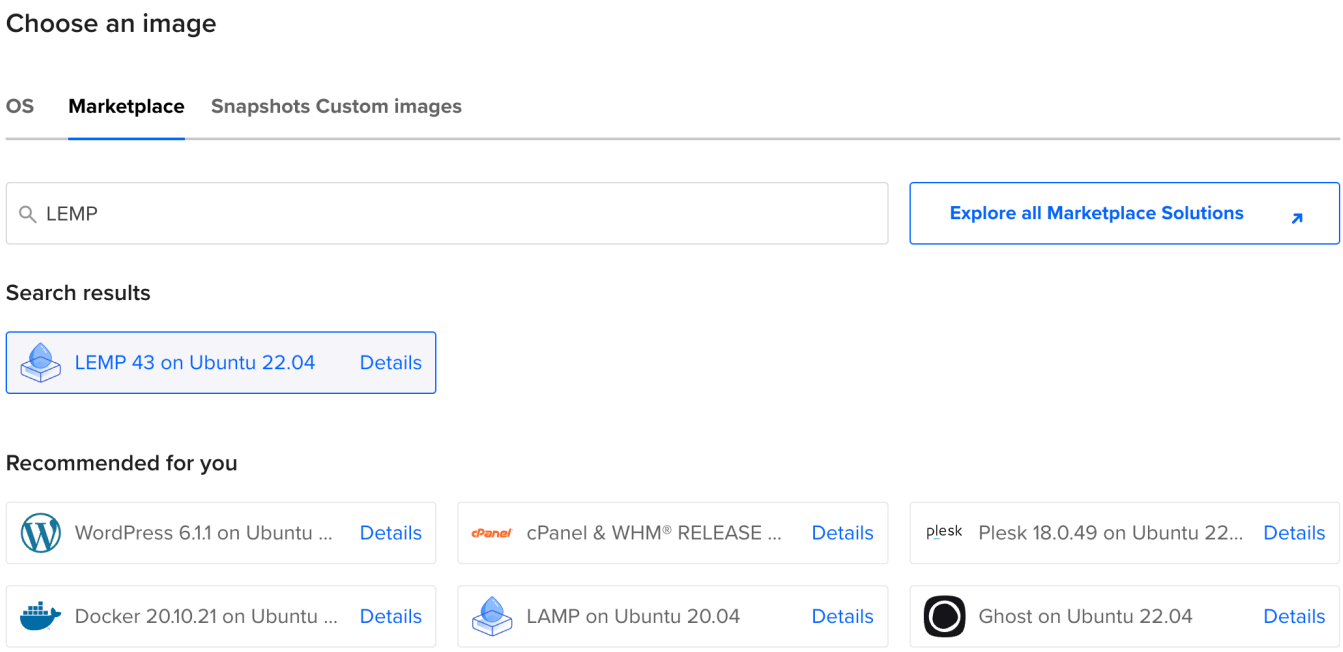
Provisioning new servers

The deploy script assumes that there's a server with PHP, MySQL, and all kinds of other stuff installed on it. Which is perfectly fine. However, it's a good practice to automate as much as possible from your server-related workflow. So even if you have only one server it's good to have a script that can provision a new one at any given moment.

In this chapter, I'll show you the script that prepares a new server for this particular project. It's important to note that the first step (actually creating the virtual machine) is still manual. Later, in the GitFlow chapter, I'm gonna automate the whole thing. The purpose of this script right now: set up a server from my local machine.

So as the first step, create a new server with the LEMP stack installed on it. Yes, I'm cheating here. I think every cloud provider gives you these pre-installed VMs, so use them! This way we don't need to install nginx and MySQL so the script will be simpler.

In DigitalOcean this is called images and marketplace. When you create a new server choose the "LEMP" image:

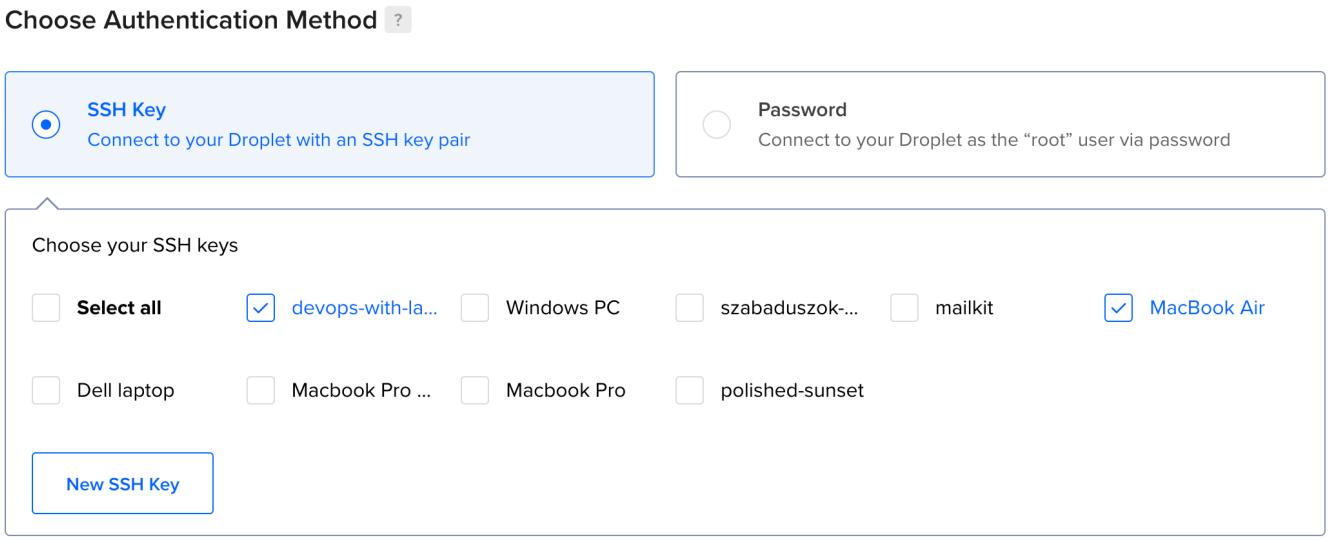


It has the following components pre-installed:

- nginx
- MySQL
- PHP8
- Certbot
- Fail2ban, Postfix

This image comes with PHP8.0 so we need to upgrade it to PHP8.1

Also, don't forget to authorize your SSH key on the new server. You can do it on the UI:



The first part of the script is pretty similar to the deploy script:

```
#!/bin/bash

set -e

MYSQL_PASSWORD=$1

PROJECT_DIR="/var/www/html/posts"

mkdir -p $PROJECT_DIR
chown -R www-data:www-data $PROJECT_DIR
cd $PROJECT_DIR
```

I'm gonna talk about the `MYSQL_PASSWORD` variable later. Other than that it creates the project directory.

The next part should also be familiar:

```
if [ ! -d $PROJECT_DIR"/.git" ]; then
  GIT_SSH_COMMAND='ssh -i ~/.ssh/id_rsa -o IdentitiesOnly=yes' git clone
git@github.com:mmartinjoo/devops-with-laravel-sample.git .
  cp $PROJECT_DIR"/api/.env.example" $PROJECT_DIR"/api/.env"
  sed -i "/DB_PASSWORD/c\DB_PASSWORD=$MYSQL_PASSWORD" $PROJECT_DIR"/api/.env"
  sed -i '/QUEUE_CONNECTION/c\QUEUE_CONNECTION=database'
$PROJECT_DIR"/api/.env"
fi
```

This part checks if the `.git` folder already exists (so the project has been checked out). Now you might be asking: why do we need to check it, if this script is supposed to run on a brand new server that knows nothing about our project? In fact, we do not need to check it. However, in my opinion, it's a good practice to make these scripts idempotent. Meaning it doesn't matter how many times we run it. It always does the same thing without side effects. Later, the script will upgrade a bunch of PHP modules to PHP8.1, enable ports on the firewall, set up a crontab schedule, and a few other things. Because of these tasks, we might run it multiple times on a server.

Earlier I mentioned that this script will run from my local machine. This is the reason why I'm using the `~/.ssh/id_rsa` SSH key. It's my SSH key in my home folder which is authorized on any server I create (because I set it in the first step).

```
# node & npm
rm -f /usr/bin/node
rm -f /usr/bin/npm
rm -f /usr/bin/npx
```

In the next steps, I remove the binaries of `node`, `npm`, and `npx`. The reason is that I'll install specific versions in the next steps:

```
cd /usr/lib
wget https://nodejs.org/dist/v14.21.3/node-v14.21.3-linux-x64.tar.xz
tar xf node-v14.21.3-linux-x64.tar.xz
rm node-v14.21.3-linux-x64.tar.xz
mv ./node-v14.21.3-linux-x64/bin/node /usr/bin/node
ln -s /usr/lib/node-v14.21.3-linux-x64/lib/node_modules/npm/bin/npm-cli.js
/usr/bin/npm
ln -s /usr/lib/node-v14.21.3-linux-x64/lib/node_modules/npx/bin/npx-cli.js
/usr/bin/npx
```

I still use Vue2 so I install node 14 with npm 6 (which is included in node). Here are the actual steps:

- Download node in the `/usr/lib` folder
- Unzip it
- Remove the zip file
- Move the binary to `/usr/bin/node` so it's available from everywhere
- Make a symbolic link to npm and npx in `/usr/bin`. `npm` comes as a js file so this is why I'm making a symlink.

The next is to upgrade PHP to 8.1:

```
add-apt-repository ppa:ondrej/php -y
```

Ubuntu (and other distros as well) use a similar ecosystem as PHP with composer packages. This command adds the `ondrej/php` repository to our list of sources that the OS uses to find and install new packages. When you run `add-apt-repository`, it adds a new file to the `/etc/apt/sources.list.d/` directory. If you list the contents you can see the new repo:

```
root@staging:/etc/apt/sources.list.d# ls -la
total 16
drwxr-xr-x 2 root root 4096 Feb  7 13:44 .
drwxr-xr-x 8 root root 4096 Oct 14  2022 ..
-rw-r--r-- 1 root root  129 Apr 17 17:52 droplet-agent.list
-rw-r--r-- 1 root root  140 Apr 17 17:52 ondrej-ubuntu-php-jammy.list
root@staging:/etc/apt/sources.list.d#
```

If you look inside the file it contains a URL:

```
root@staging:/etc/apt/sources.list.d# cat ondrej-ubuntu-php-jammy.list
deb https://ppa.launchpadcontent.net/ondrej/php/ubuntu/ jammy main
# deb-src https://ppa.launchpadcontent.net/ondrej/php/ubuntu/ jammy main
root@staging:/etc/apt/sources.list.d#
```

That's the repository where we can download PHP8.1 from.

After you add a repository you need to run `apt update` to let Linux does its things:

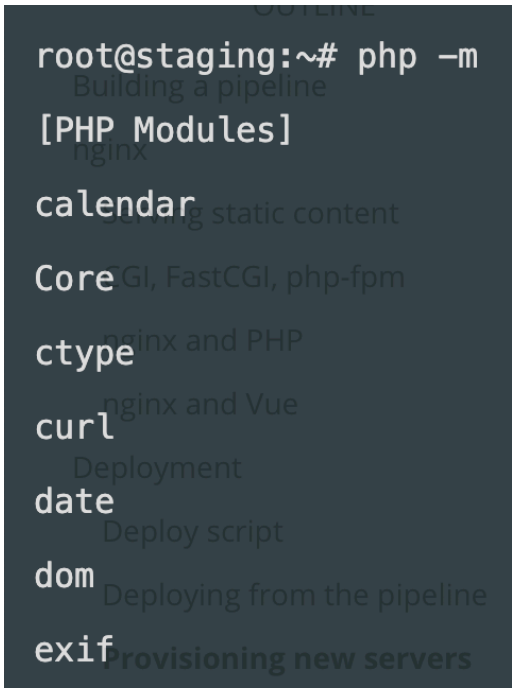
```
apt update -y
```

Remember, it's a script, so we don't want questions and interactions. The `-y` will say "yes" to every question.

Next up, we can upgrade PHP and its modules:

```
apt install php8.1-common php8.1-cli -y
apt install php8.1-dom -y
apt install php8.1-gd -y
apt install php8.1-zip -y
apt install php8.1-curl -y
apt install php8.1-mysql -y
apt install php8.1-sqlite3 -y
apt install php8.1-mbstring -y
apt install php8.1-fpm -y
```

These are the required modules to run the sample project. In your case it might be a bit different. If you're not sure what kind of modules are being used, you can run `php -m` on your local machine or an existing server:



Just make sure you included `php8.1-fpm`.

After that, the script installs a few other useful packages:

```
apt install net-tools -y
apt install supervisor -y
```

`net-tools` includes commands such as `netstat` or `ifconfig`. `supervisor` is used to run, supervise, and scale worker processes. Don't worry! We discuss everything in a dedicated chapter.

The server also needs `composer`:

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') ===
'e21205b207c3ff031906575712edab6f13eb0b361f2085f1f1237b7126d785e826a450292b6cf
d1d64d92e6563bbde02') { echo 'Installer verified'; } else { echo 'Installer
corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
mv composer.phar /usr/bin/composer
```

These are the official installation instructions.

The project (and the deploy script) needs a MySQL database. On the server we have MySQL installed but we can also prepare the database itself:

```
mysql -uroot -p$MYSQL_PASSWORD <
$PROJECT_DIR"/deployment/config/mysql/create_database.sql" || echo "Database
already exists"
mysql -uroot -p$MYSQL_PASSWORD <
$PROJECT_DIR"/deployment/config/mysql/set_native_password.sql"
```

This is the reason why the script needs the MySQL password as the first parameter. With the `mysql` command line tool you can run queries using the `<` so this line:

```
mysql -uroot -pfoo < create_database.sql
```

will run the queries in the file. The content of the file is pretty straightforward:

```
CREATE DATABASE posts;
```

It simply creates a new DB called `posts`. After the command, you see a `||` operator:

```
mysql -uroot -pfoo < create_database.sql || echo "Database already exists"
```

It's a pretty naive "try-catch" in bash. If the database already exists then MySQL throws an error, the script returns with 1 and the whole provision script fails. Earlier, I talked about idempotency. If we let the script fail because the DB already exists it's not a good script. The `||` or double pipe is used for conditional execution. It allows the first command to fail and the second one only runs in this case. So it's similar to a try-catch block:

```
try {
  createDatabase();
} catch (DatabaseExistsException $ex) {
  echo "Database already exists";
}
```

After the database is created, the second script will the authentication method to `mysql_native_password`.

Usually, every Laravel project has some kind of scheduling logic so we need to set up `crontab`:

```
echo "* * * * * cd $PROJECT_DIR && php artisan schedule:run >> /dev/null
2>&1" >> cron_tmp
crontab cron_tmp
rm cron_tmp
```

The first command write this line to a file called `cron_tmp`:

```
* * * * * cd /var/www/html/posts && php artisan schedule:run >> /dev/null
2>&1
```

The `>>` operator appends the output of the `echo` command to the file. I do this, because `crontab` can load schedules from a file by doing this:

```
crontab cron_tmp
```

After that the script removes the temp file.

The next line copies a config file:

```
cp $PROJECT_DIR"/deployment/config/supervisor/logrotate"
/etc/logrotate.d/supervisor
```

`lograte` is installed by default and you can avoid huge log files by using it. It'll rotate log files based on your config which looks like this:

```
/var/log/supervisor/*.log {
  rotate 12
  weekly
  missingok
  notifempty
  compress
  delaycompress
}
```

`supervisor` creates log files in the `/var/log/supervisor` folder so we match these. The other configs mean:

- `rotate 12`: The rotated log files will be kept for 12 weeks before being deleted.
- `weekly`: The rotation will occur once a week.
- `missingok`: If a log file is missing, no error message will be displayed.
- `notifempty`: If a log file is empty, no rotation will occur.
- `compress`: The rotated log files will be compressed using gzip.
- `delaycompress`: The compression of the rotated log files will be delayed until the next rotation.

With this config, you'll have smaller log files and you can also save disk space. By the way, nginx, mysql, php-fpm logs are rotated by default. If that's not the case for you, just copy this config and replace `supervisor`. You can check the content of the `/etc/logrotate.d` folder:

```
root@staging:/etc/logrotate.d# ls
alternatives  apt      btmp     dpkg      mysql-server  php8.0-fpm  rsyslog     ubuntu-advantage-tools  unattended-upgrades
apport        bootlog  certbot  fail2ban  nginx         php8.1-fpm  supervisor  ufw                     wtmp
root@staging:/etc/logrotate.d#
```

The next step is to create a new user. It's never a good idea to do everything as `root` so you should create a new user and use it to SSH into your server:

```
useradd -G www-data,root -u 1000 -d /home/martin martin
mkdir -p /home/martin/.ssh
touch /home/martin/.ssh/authorized_keys
chown -R martin:martin /home/martin
chown -R martin:martin /var/www/html
chmod 700 /home/martin/.ssh
chmod 644 /home/martin/.ssh/authorized_keys
```

It creates a new user in the `www-data` and `root` groups sets up a home folder and set ownership of the project and the folder. The last two lines are important because we need to copy a public key to the `authorized_keys` file to be able to SSH into the server with this user.

The next line copies a public SSH key to the `authorized_keys` file:

```
echo "$SSH_KEY" >> /home/martin/.ssh/authorized_keys
```

`$SSH_KEY` is a variable that comes from the second argument of the script:

```
#!/bin/bash

set -ex

MYSQL_PASSWORD=$1
SSH_KEY=$2
```

In the `run_provision_server_from_local.sh` script you can see I pass my SSH key to the script like this:

```
PUBLIC_SSH_KEY=$(cat $HOME/.ssh/id_ed25519.pub)
 ...

ssh -tt -o StrictHostKeyChecking=no -i $HOME/.ssh/id_ed25519
$SSH_USER@$SERVER_IP "chmod +x ./provision_server.sh && ./provision_server.sh
$MYSQL_PASSWORD \"$PUBLIC_SSH_KEY\""
```

The next line makes sure when you run a command such as `sudo <command>` with the new user, Linux won't require a password:

```
echo "martin ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers.d/martin
```

And finally, I want to make sure everything went as expected:

```
php -v
node -v
npm -v
```

To quickly summarize, now we have a script that sets up the following things on a new server:

- The project folder and the git repo
- Firewall
- node
- npm
- npx
- php8.1
- supervisor
- composer
- A new MySQL database
- crontab
- log rotation

That's a good starting point. I wrote another script to run this smoothly from my local machine. You can try it out in the `run_provision_server_from_local.sh` file.

Queues and workers

supervisor

Right now we have a working deploy script and we can set up new servers. However, we haven't talked about queues and workers but they're very important.

The most important thing is that worker processes need to run all the time even if something goes wrong. Otherwise, they'd be unreliable. For this reason, we cannot just run `php artisan queue:work` on a production server as we do on a local machine. We need a program that supervises the worker process, restarts them if they fail, and potentially scales the number of processes.

The program we'll use is called `supervisor`. It's a process manager that runs in the background (daemon) and manages other processes such as `queue:work`.

First, let's review the configuration:

```
[program:worker]
command=php /var/www/html/posts/api/artisan queue:work --tries=3 --verbose --
timeout=30 --sleep=3
```

We can define many "programs" such as `queue:work`. Each has a block in a file called `supervisord.conf`. Every program has a `command` option which defines the command that needs to be run. In this case, it's the `queue:work` but with the full `artisan` path.

As I said, it can scale up processes:

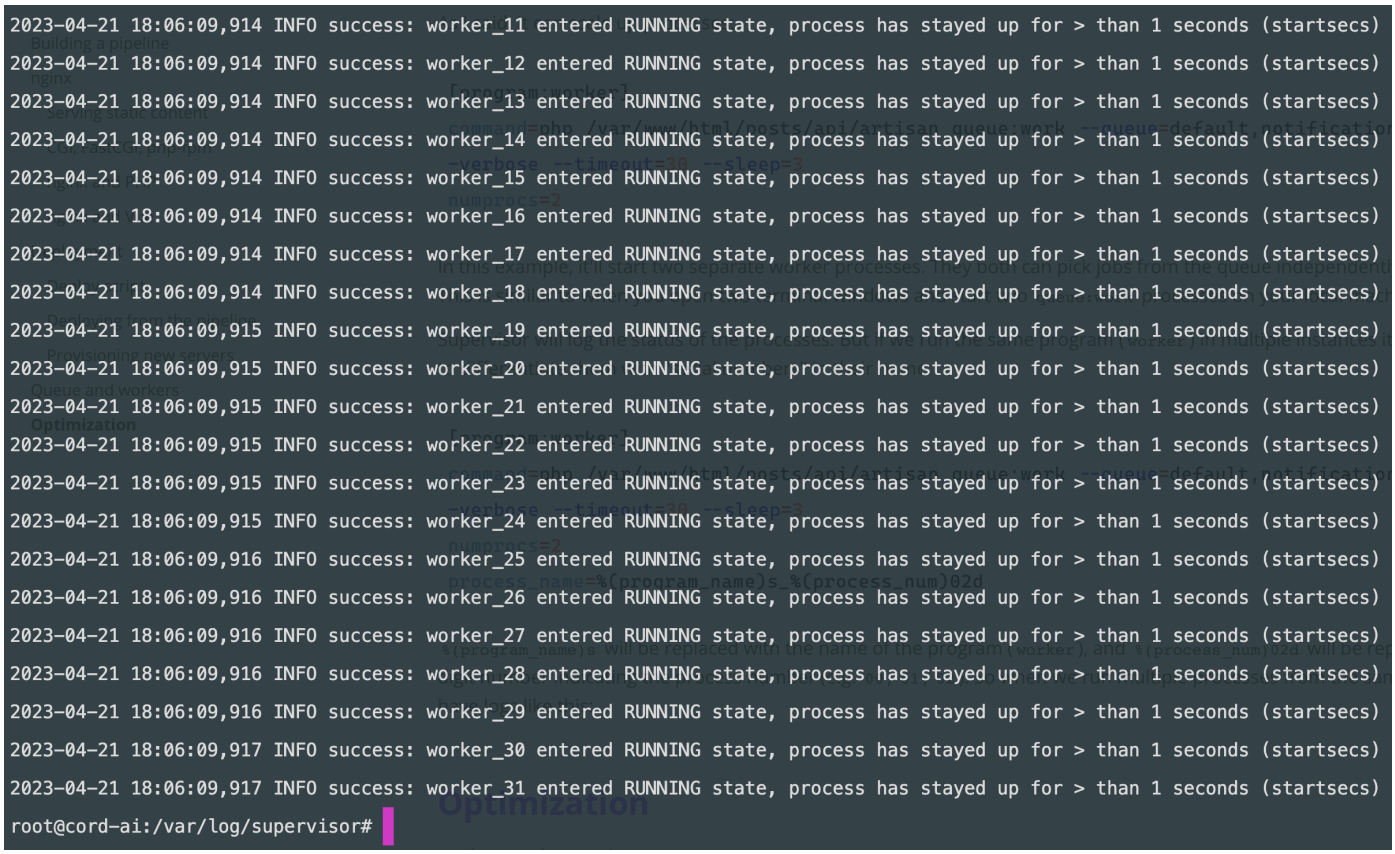
```
[program:worker]
command=php /var/www/html/posts/api/artisan queue:work --
queue=default,notification --tries=3 --verbose --timeout=30 --sleep=3
numprocs=2
```

In this example, it'll start two separate worker processes. They both can pick up jobs from the queue independently from each other. This is similar to when you open two terminal windows and start two `queue:work` processes on your local machine.

Supervisor will log the status of the processes. But if we run the same program (`worker`) in multiple instances it's a good practice to differentiate them with "serial numbers" in their name:

```
[program:worker]
command=php /var/www/html/posts/api/artisan queue:work --
queue=default,notification --tries=3 --verbose --timeout=30 --sleep=3
numprocs=2
process_name=%(program_name)s_%(process_num)02d
```

`%(program_name)s` will be replaced with the name of the program (`worker`), and `%(process_num)02d` will be replaced with a two-digit number indicating the process number (e.g. `00`, `01`, `02`). So when we run multiple processes from the same command we'll have logs like this:



Next, we can configure how `supervisor` is supposed to start or restart the processes:

```
[program:worker]
command=php /var/www/html/posts/api/artisan queue:work --
queue=default,notification --tries=3 --verbose --timeout=30 --sleep=3
numprocs=2
process_name=%(program_name)s_%(process_num)02d
autostart=true
autorestart=true
```

`autostart=true` tells `supervisor` to start the program automatically when it starts up. So when we start `supervisor` (for example when deploying a new version) it'll automatically start the workers.

`autorestart=true` tells `supervisor` to automatically restart the program if it crashes or exits. Worker processes usually take care of long-running heavy tasks, often communicating with 3rd party services. It's not uncommon that they crash for some reason. By setting `autorestart=true` we can be sure that they are **always** running.

```
[program:worker]
command=php /var/www/html/posts/api/artisan queue:work --
queue=default,notification --tries=3 --verbose --timeout=30 --sleep=3
numprocs=2
process_name=%(program_name)s_%(process_num)02d
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
```

`stopasgroup` and `killasgroup` basically mean: stop/kill all *subprocesses* as well when the parent process (queue:work) stops/dies.

As I said, errors happen fairly often in queue workers, so it's a good practice to think about them:

```
[program:worker]
command=php /var/www/html/posts/api/artisan queue:work --
queue=default,notification --tries=3 --verbose --timeout=30 --sleep=3
numprocs=2
process_name=%(program_name)s_%(process_num)02d
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
redirect_stderr=true
stdout_logfile=/var/log/supervisor/worker.log
```

`redirect_stderr=true` tells `supervisor` to redirect standard error output to the same place as standard output. We treat errors and info messages the same way.

`stdout_logfile=/var/log/supervisor/worker.log` tells `supervisor` where to write standard output for each process. Since we redirect `stderr` to `stdout` we'll have one log file for every message:

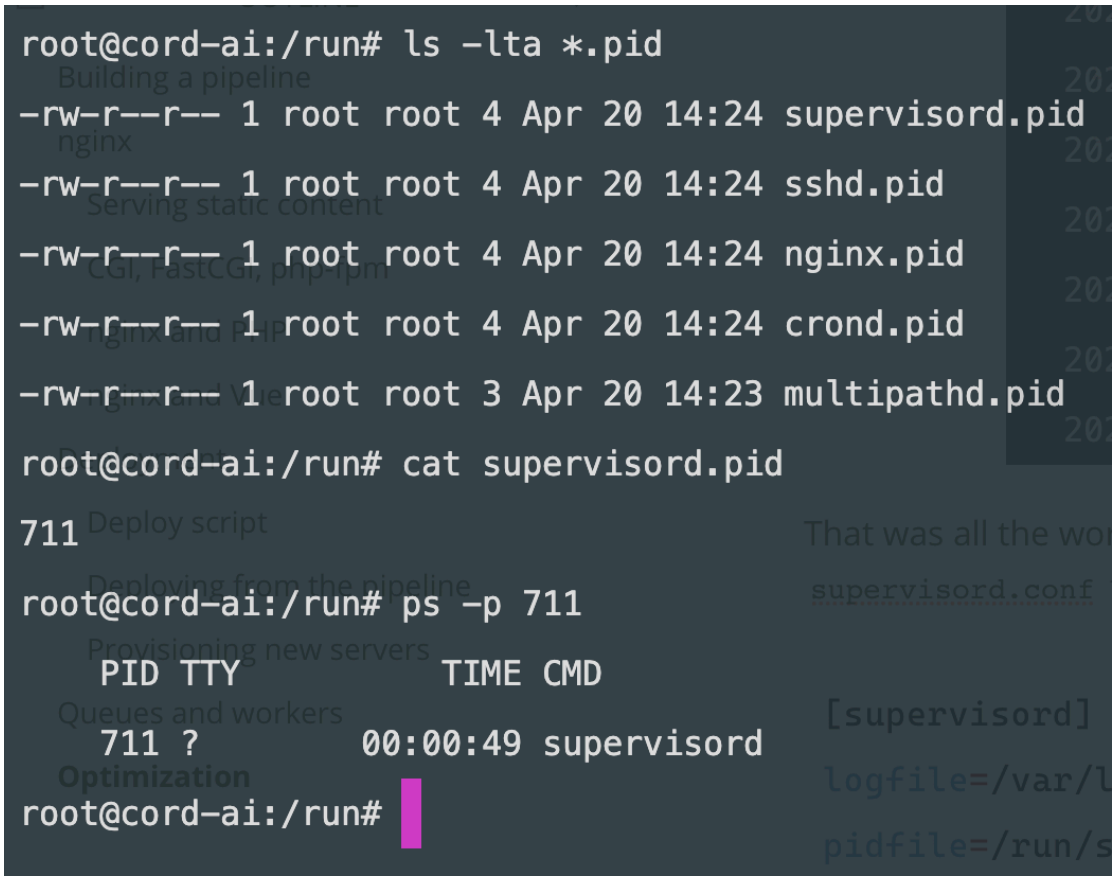
```
root@cord-ai:/var/log/supervisor# cat worker-default.log
  2023-04-20 15:06:46 App\Jobs\GenerateCodeJob 13 ..................... RUNNING
  2023-04-20 15:08:16 App\Jobs\GenerateCodeJob 13 ..................... RUNNING
  2023-04-20 15:08:16 App\Jobs\GenerateCodeJob 13 ............... 12.70ms FAIL
  2023-04-20 15:12:09 App\Jobs\GenerateCodeJob 13 ..... 322,867.10ms FAIL
  2023-04-20 15:12:33 App\Jobs\GenerateCodeJob 14 ..................... RUNNING
  2023-04-20 15:13:11 App\Jobs\GenerateCodeJob 14 ..... 37,901.77ms DONE
  2023-04-20 15:25:33 App\Jobs\GenerateCodeJob 15 ..................... RUNNING
  2023-04-20 15:26:23 App\Jobs\GenerateCodeJob 15 ..... 50,185.93ms DONE
  2023-04-20 15:27:44 App\Jobs\GenerateCodeJob 16 ..................... RUNNING
  2023-04-20 15:27:59 App\Jobs\GenerateCodeJob 17 ..................... RUNNING
  2023-04-20 15:28:29 App\Jobs\GenerateCodeJob 16 ..... 45,123.56ms DONE
  2023-04-20 15:28:53 App\Jobs\GenerateCodeJob 17 ..... 54,324.66ms DONE
```

That was all the worker-specific configuration we need but `supervisor` itself also needs some config in the same `supervisord.conf` file:

```
[supervisord]
logfile=/var/log/supervisor/supervisord.log
pidfile=/run/supervisord.pid
```

`logfile=/var/log/supervisor/supervisord.log` tells `supervisor` where to write its own log messages. This log file contains information about the different programs and processes it manages. You can see the screenshot above.

`pidfile=/run/supervisord.pid` tells `supervisor` where to write its own process ID (PID) file. These files are usually located in the `run` directory:



By the way, PID files on Linux are similar to a MySQL or Redis database for us, web devs.

They are files that contain the process ID (PID) of a running program. They are usually created by daemons or other long-running processes to help manage the process.

When a daemon or other program starts up, it will create a PID file to store its own PID. This allows other programs (such as monitoring tools or control scripts) to easily find and manage the daemon. For example, a control script might read the PID file to determine if the daemon is running, and then send a signal to that PID to stop or restart the daemon.

And finally, we have one more config:

```
[supervisorctl]
serverurl=unix:///run/supervisor.sock
```

This section sets some options for the `supervisorctl` command-line tool. `supervisorctl` is used to control Supervisor. With this tool, we can list the status of processes, reload the config, or restart processes easily. For example:

```
supervisorctl status
```

Returns a list such as this:

```
root@cord-ai:/run# supervisorctl status
workers:worker_00                RUNNING   pid 64311, uptime 0:09:59
workers:worker_01                RUNNING   pid 64312, uptime 0:09:59
workers:worker_02                RUNNING   pid 64313, uptime 0:09:59
workers:worker_03                RUNNING   pid 64314, uptime 0:09:59
workers:worker_04                RUNNING   pid 64315, uptime 0:09:59
workers:worker_05                RUNNING   pid 64316, uptime 0:09:59
workers:worker_06                RUNNING   pid 64317, uptime 0:09:59
workers:worker_07                RUNNING   pid 64318, uptime 0:09:59
workers:worker_08                RUNNING   pid 64319, uptime 0:09:59
```

Later, we're gonna use it in the deploy script the reload the config and restart every process.

And finally, the `serverurl=unix:///run/supervisor.sock` config tells `supervisorctl` to connect to `supervisor` using a Unix socket. We've already used a Unix socket when we connected nginx to php-fpm. This is the same here. `supervisorctl` is "just" a command-line tool that provides better interaction with `supervisor` and its processes. It needs a way to send requests to `supervisor`.

Multiple queues and priorities

Before we move on to actually running the workers let's talk about having multiple queues and priorities. First of all, multiple queues do not mean multiple Redis or MySQL instances.

- connection: this is what Redis or MySQL is in Laravel-land. Your app connects to Redis so it's a connection.
- queue: inside Redis, we can have multiple queues with different names.

For example, if you're building an e-commerce site, the app connects to one Redis instance but you can have at least three queues:

- payments
- notifications
- default

Since payments are the most important jobs it's probably a good idea to separate them and handle them with priority. The same can be true for notifications as well (obviously not as important as payments but probably more important than a lot of other things). And for every other task, you have a queue called default. These queues live inside the same Redis instance (the same connection) but under different keys (please don't quote me on that).

So let's say we have payments, notifications, and the default queue. Now, how many workers do we need? What queues should they be processing? How do we prioritize them?

A good idea can be to have dedicated workers for each queue, right? Something like that:

```
[program:payments-worker]
command=php artisan queue:work --queue=payments --tries=3 --verbose --
timeout=30 --sleep=3
numprocs=4

[program:notifications-worker]
command=php artisan queue:work --queue=notifications --tries=3 --verbose --
timeout=30 --sleep=3
numprocs=2

[program:default-worker]
command=php artisan queue:work --queue=default --tries=3 --verbose --
timeout=30 --sleep=3
numprocs=2
```

And then when you dispatch jobs you can do this:

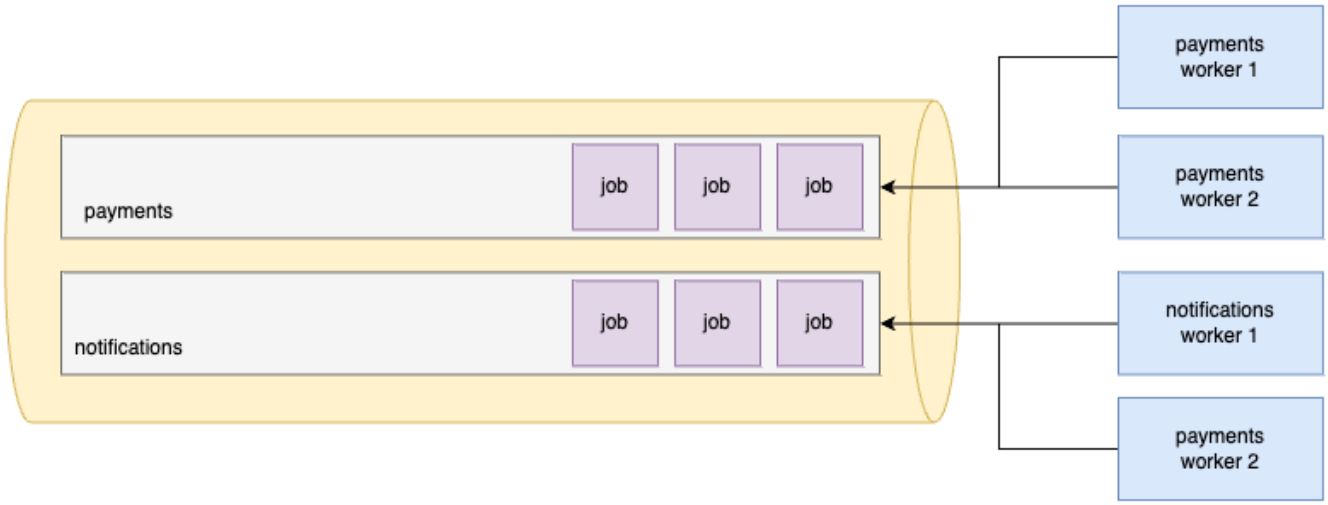
```php
ProcessPaymentJob::dispatch()->onQueue('payments');

$user->notify(
  (new OrderCompletedNotification($order))->onQueue('notifications');
);

// default queue
CustomersExport::dispatch();
```

Alternatively, you can define a `$queue` in the job itself:

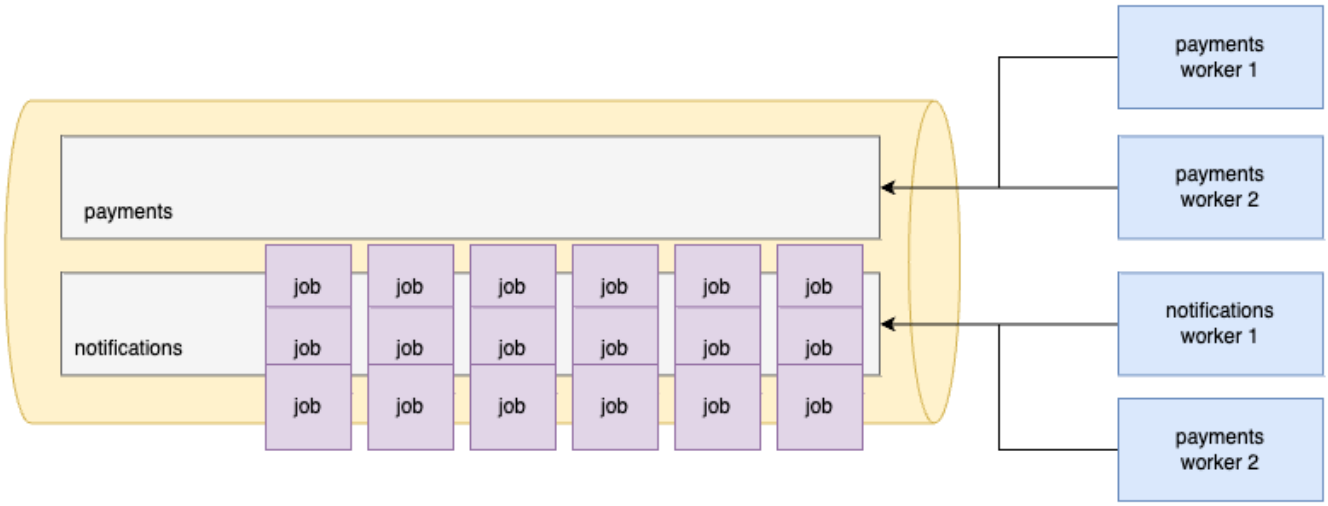
```php
class ProcessPayment implements ShouldQueue
{
  use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

  public $queue = 'payments';
}
```

By defining the queue in the job you can be 100% sure that it'll always run in the given queue so it's a safer option in my opinion.

And now, the following happens:



So the two (in fact three because there's also the default) are being queued at the same time by dedicated workers. Which is great, but what if something like that happens?



There are so many jobs in the notifications queue but none in the payments. If that happens we just waste all the payments worker processes since they have nothing to do. But this command doesn't let them to processes anything else:

```
php artisan queue:work --queue=payments
```

This means they can **only** touch the payments queue and nothing else.

Because of that problem, I don't recommend you have dedicated workers for only one queue. Instead, prioritize them!

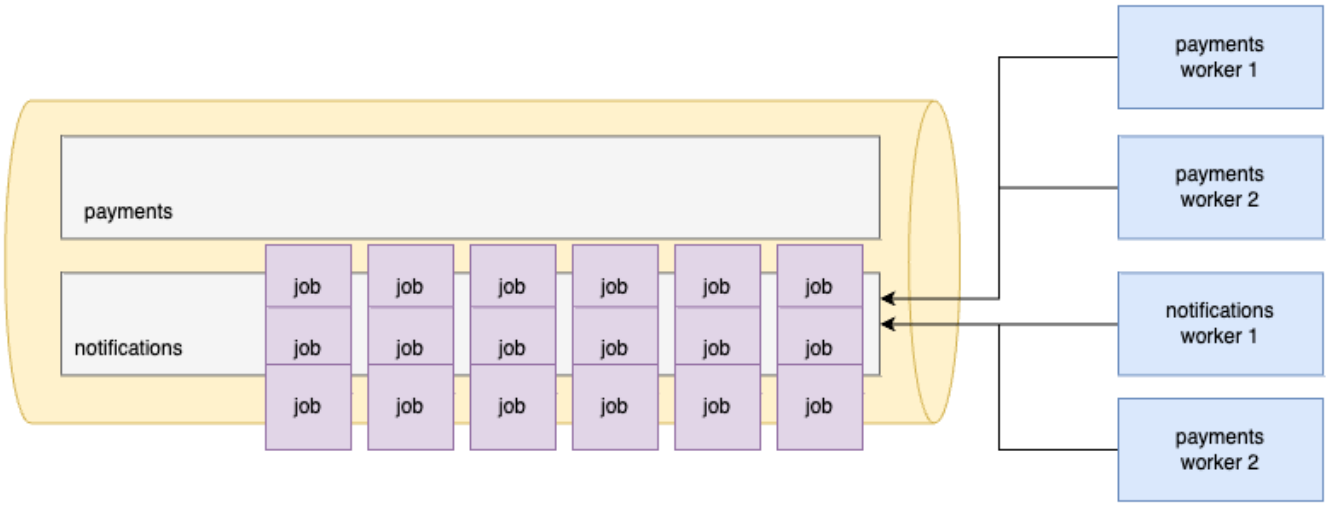
We can do this:

```
php artisan queue:work --queue=payments,notifications
```

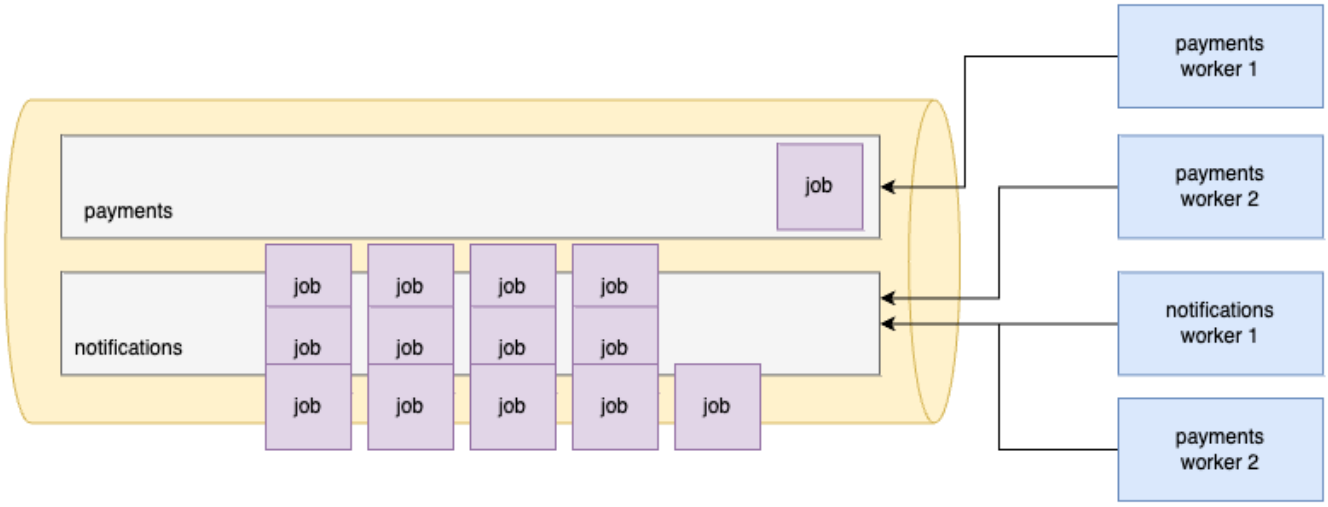
The command means that if there are jobs in the payments queue, these workers can **only** process them. However, if the payments queue is empty, they can pick up jobs from the notifications queue as well. And we can do the same for the notifications workers:

```
php artisan queue:work --queue=notifications,payments
```

Now if the payments queue is empty this will happen:



Now payments workers also pick up jobs from the notifications so we don't waste precious worker processes. But of course, if there are payments job they prioritize them over notifications:



In this example, only one payment job came in so one worker is enough to process it. All of this is managed by Laravel!

And of course, we can prioritize three queues as well:

```
# payment workers
php artisan queue:work --queue=payments,notifications,default

# notification workers
php artisan queue:work --queue=notifications,payments,default

# other workers
php artisan queue:work --queue=default,payments,notifications
```

With this setup, we cover almost any scenario:

- If there are a lot of payment jobs possibly three workers (more than three processes, of course) will process them.
- If there isn't any important job (payment or notification) there are a lot of workers available for default jobs.
- And we also cover the notifications queue.

And now let's deploy them!

Deploying workers

The `supervisor` config file lives under the `/deployment/config/supervisor` folder in the sample project. When deploying the project we only need to do three things:

- Copy the config file
- Update the config
- Restart all worker processes

By restarting all worker processes I mean we need to restart all programs (such as worker-notifications and worker-default) one by one. To make this step a bit easier `supervisor` let us declare program groups:

```
[group:workers]
programs=default-worker,notifications-worker
```

(In the sample project I don't have payment-related code this is why it only has two programs). We can put programs into a group so when interacting with `supervisorctl` we can treat them as one program. These are the three steps at the end of the deploy script:

```
cp $PROJECT_DIR"/deployment/config/supervisord.conf"
/etc/supervisor/conf.d/supervisord.conf

# update the config
supervisorctl update

# restart workers (notice the : at the end. it refers to the process group)
supervisorctl restart workers:
```

Please notice the `:` symbol at the end of `workers:` It means we refer to the **group** called `workers`. With this one command, we can restart multiple programs. That's it! Now `supervisor` will start the worker processes and log everything inside `/var/log/supervisor`.

I already showed you the installation of `supervisor` in the provision script, but here it is one more time:

```
apt install supervisor -y
```

Optimizing worker processes

Number of worker processes

That's a tricky question, but a good rule of thumb: run one process for each CPU core.

But of course, it depends on several factors, such as the amount of traffic your application receives, the amount of work each job requires, and the resources available on your server.

As a general rule of thumb, you should start with one worker process per CPU core on your server. For example, if your server has 4 CPU cores, you might start with 4 worker processes and monitor the performance of your application. If you find that the worker processes are frequently idle or that there are jobs waiting in the queue for too long, you might consider adding more worker processes.

It's also worth noting that running too many worker processes can actually decrease performance, as each process requires its own memory and CPU resources. You should monitor the resource usage of your worker processes and adjust the number as needed to maintain optimal performance.

However, there are situations when you can run more processes than the number of CPUs. It's a rare case, but if your jobs don't do much work on your machine you can run more processes. For example, I have a project where every job sends API requests and then returns the results. These kinds of jobs are not resource-heavy at all since they do not run much work on the actual CPU or disk. But usually, jobs are resource-heavy processes so don't overdo it.

Memory and CPU considerations

Queued jobs can cause some memory leaks. Unfortunately, I don't know the exact reasons but not everything is detected by PHP's garbage collector. As time goes on, and your worker processes more jobs it uses more and more memory.

Fortunately, the solution is simple:

```
php artisan queue:work --max-jobs=1000 --max-time=3600
```

`--max-jobs` tells Laravel that this worker can only process 1000 jobs. After it reaches the limit it'll be shut down. Then memory will be freed up and `supervisor` restarts the worker.

`--max-time` tells Laravel that this worker can only live for an hour. After it reaches the limit it'll be shut down. Then memory will be freed up and `supervisor` restarts the worker.

These two options can save us some serious trouble.

Often times we run workers and nginx on the same server. This means that they use the same CPU and memory. Now, imagine what happens if there are 5000 users in your application and you need to send a notification to everyone. 5000 jobs will be pushed onto the queue and workers start processing them like there's no tomorrow. Sending notifications it's too resource-heavy, but if you're using database notifications as well, it means at least 5000 queries. Let's say the notification contains a link to your and users start to come to your site. nginx has few resources to use since your workers eat up your server.

One simple solution to give a higher `nice` value to your workers:

```
nice -n 10 php artisan queue:work
```

These values can go from 0-19 and a higher value means a lower priority to the CPU. This means that your server will prioritize nginx or php-fpm processes over your worker processes if there's a high load.

Another option is to use the `rest` flag:

```
php artisan queue:work --rest=1
```

This means the worker will wait for 1 second after it finishes with a job. So your CPU has an opportunity to server nginx or fpm processes.

So this is what the final command looks like:

```
nice -n 10 php /var/www/html/posts/api/artisan queue:work --
queue=notifications,default --tries=3 --verbose --timeout=30 --sleep=3 --
rest=1 --max-jobs=1000 --max-time=3600
```

I never knew about `nice` or `rest` before reading Mohamed Said's amazing book Laravel Queues in Action.

Domains and HTTPS with nginx

Now the project and the settings are almost production ready. We only need three extra things:

- A domain
- HTTPS
- Some nginx and php-fpm optimization

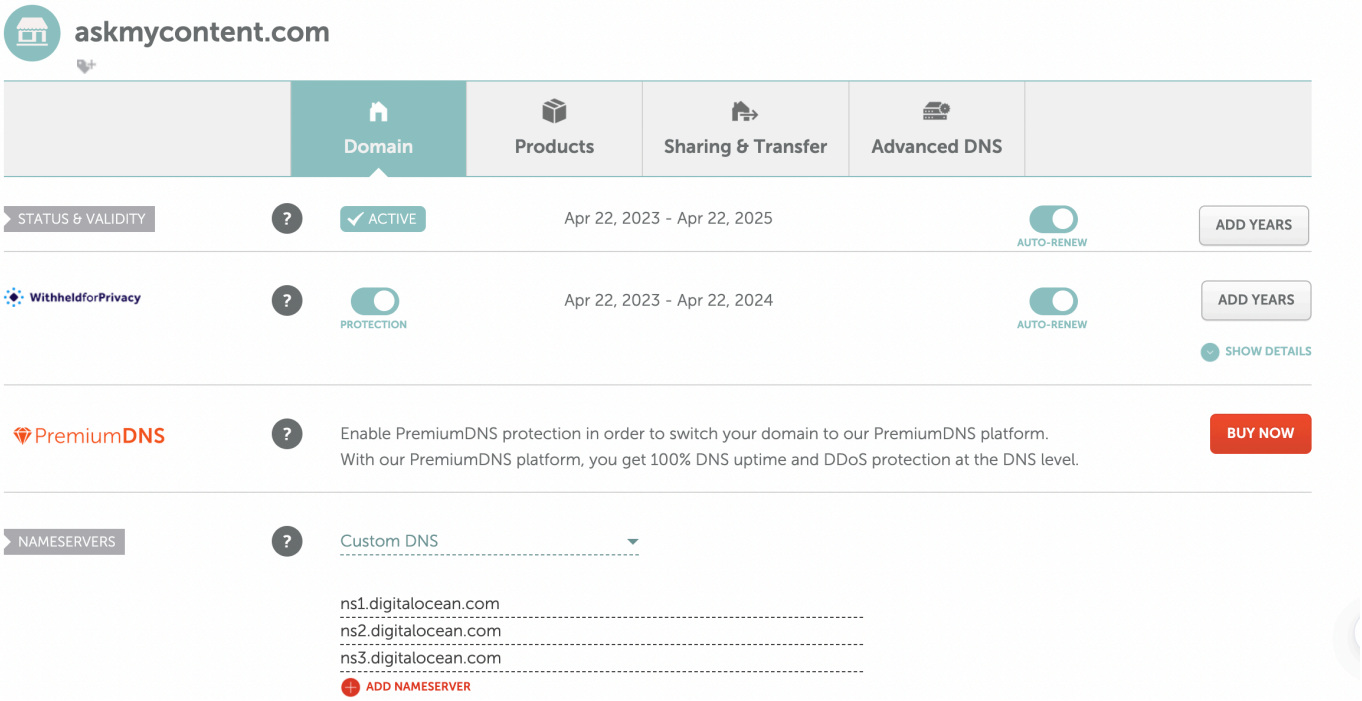
We're gonna take care of the domain and HTTPS in this chapter.

Domain

Setting up a domain is easy. You only need to do two things:

- Buy it
- Map the domain to your IP address

You can buy a domain on a lot of sites, I use Namecheap. On the domain's settings page, you can configure a nameserver. I use DigitalOcean's nameservers so I put `ns1.digitalocean.com`, `ns2.digitalocean.com`, and `ns3.digitalocean.com` in these



This means DigitalOcean will handle the domain name resolution. If you're not sure how DNS works, here's a pretty brief summary.

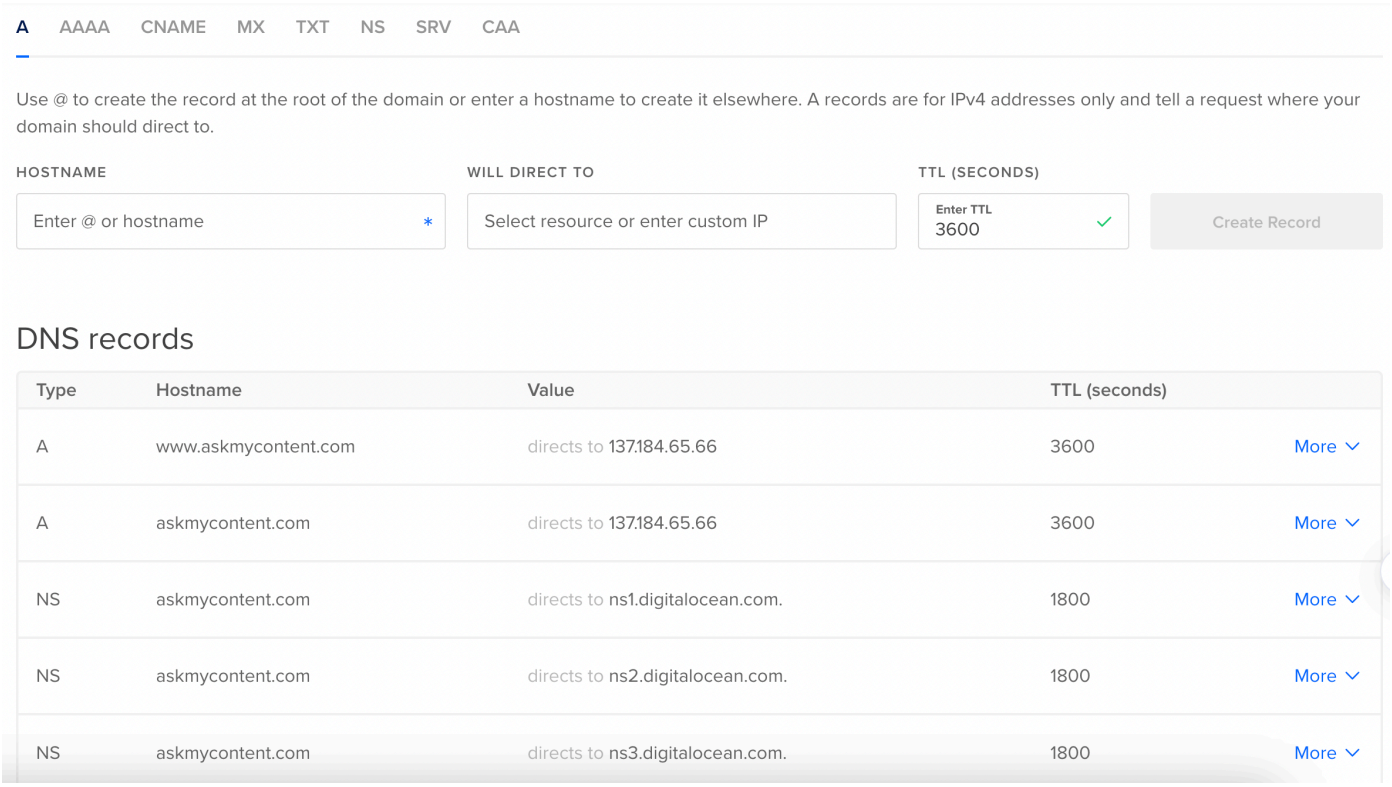
A nameserver is a server that stores DNS (Domain Name System) records for a domain. When a user requests a domain name, their computer sends a query to a DNS resolver, which in turn sends a query to the appropriate nameserver to resolve the domain name. The nameserver then responds with the IP address associated with the domain name, allowing the user's computer to connect to the web server hosting the website associated with the domain name.

So basically when users go to askmycontent.com, they first hit `ns1.digitalocean.com` which then says: "Hey! The IP address of askmycontent.com is 137.184.65.66 go there."

But we're still missing something. How does the nameserver know the IP address? We need to configure it as well. Since DigitalOcean handles the name resolution, this is a setting we need to do on their interface.

Basically, only one config is required which is a DNS "A" record. It's a type of DNS record that maps a domain name to an IP address. The "A" stands for "address". So the A record will tell the nameserver that askmycontent.com actually means 137.184.65.66

The setting is quite straightforward:



There are actually two records because www.askmycontent.com requires a separate one.

HTTPS

HTTPS works by using SSL/TLS to encrypt data sent between a web server and a web browser, ensuring that it cannot be intercepted or modified by third parties. Here's a more detailed explanation:

- The client (web browser) initiates a secure connection to the server by sending a request to the server.
- The server responds by sending its SSL/TLS certificate to the client, which contains the server's public key and other information.
- The client verifies the authenticity of the certificate by checking that it has been issued by a trusted Certificate Authority (CA) and that it has not been revoked.
- The client generates a symmetric key and encrypts it using the server's public key, which is then sent to the server.
- The server decrypts the symmetric key using its private key and sends an acknowledgment to the client.
- The client and server use the symmetric key to encrypt and decrypt all data sent between them, ensuring that it cannot be intercepted or modified by third parties.

You don't actually need to remember all these steps but it's good to have a basic understanding of the big picture.

As you can see, there has to be a certificate issued by a trusted Certificate Authority (CA). This is where Let's Encrypt comes into play. It's a totally free CA that can issue certificates and we can use them on our websites.

There's also another tool called certbot. It's a free, open-source tool that automates the process of obtaining and renewing SSL/TLS certificates from Let's Encrypt. It works well with nginx.

Let's see how certbot works:

```
certbot --nginx -d askmycontent.com -d www.askmycontent.com
```

This command requires a `server` block with the `server_name askmycontent.com www.askmycontent.com;` directive.

It asks some basic questions (such as your e-mail address) and it's all done. If everything went well you should see something like this:

```
Congratulations! You have successfully enabled https://askmycontent.com and
https://www.askmycontent.com

----------------------------------------------------------------------------
-------
IMPORTANT NOTES:

Congratulations! Your certificate and chain have been saved at:
/etc/letsencrypt/live/example.com/fullchain.pem
Your key file has been saved at:
/etc/letsencrypt/live/example.com/privkey.pem
Your cert will expire on 2023-07-21.
```

It creates the two files inside the `/etc/letsencrypt/live/<mydomain>` folder.

If you have your nginx config in the `/etc/nginx/sites-available/domain-name.com` folder then everything is done! Certbot adds a few directives to your nginx config. But in this project I store my config inside `/etc/nginx/nginx.conf` and I also want to add the HTTPS-related directives to the file inside the repository so I can deploy it.

Here's the modified `server` directive:

```
listen 80;
listen 443 ssl;
server_name askmycontent.com www.askmycontent.com;
root /var/www/html/askmycontent/public;
index index.php index.html;

# RSA certificate
ssl_certificate /etc/letsencrypt/live/askmycontent.com/fullchain.pem; #
managed by Certbot
ssl_certificate_key /etc/letsencrypt/live/askmycontent.com/privkey.pem; #
managed by Certbot

include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot

# Redirect non-https traffic to https
if ($scheme != "https") {
```

```
    return 301 https://$host$request_uri;
} # managed by Certbot
```

I think the new directives are self-explanatory for the most part:

- `listen 443` is the default port for HTTPS connections
- `ssl_certificate` and `ssl_certificate_key` tell nginx where to find the certificate files
- The `return 301` directive at the end simply redirects `HTTP` requests to their `HTTPS` counterparts

That's it! With these few steps, you can set up any domain with an HTTPS connection.

Optimization

nginx worker processes and connections

It's finally time to optimize nginx and fpm. Let's start with the low-hanging fruits.

```
user www-data;
worker_processes auto;

events {
    worker_connections 1024;
}
```

As you might guess `worker_processes` controls how many workers can nginx spin up. It's a good practice to set this value equal to the number of CPUs available in your server. The `auto` does this automatically.

`worker_connections` controls how many connections a worker process can open. In Linux, everything is treated as a file, and network sockets are no exception. When a client establishes a connection to a server, a socket file descriptor is created to represent the connection. This file descriptor can be used to read and write data to the connection, just like any other file descriptor.

So this setting controls how many file descriptors a worker process can use. It's important to note that a file descriptor is not a file. It is simply an integer value that represents an open file or resource in the operating system. This can include files on disk, network sockets, pipes, and other types of resources. When a network socket (an IP address and a port) is opened, a file descriptor is created to represent the open socket, but no file is created on disk.

So how can you figure out the right number? Run this on your server:

```
ulimit -n
```

It gives you a number which means how many file descriptors a process can open in Linux. It's usually 1024.

So to summarize: if you have four cores in your CPU and you set `worker_connections` to 1024 (based on ulimit) it means that nginx can handle 4096 connections (users) at the same time. If the 4097th user comes in, the connection will be pushed into a queue and it gets served after nginx has some room to breathe.

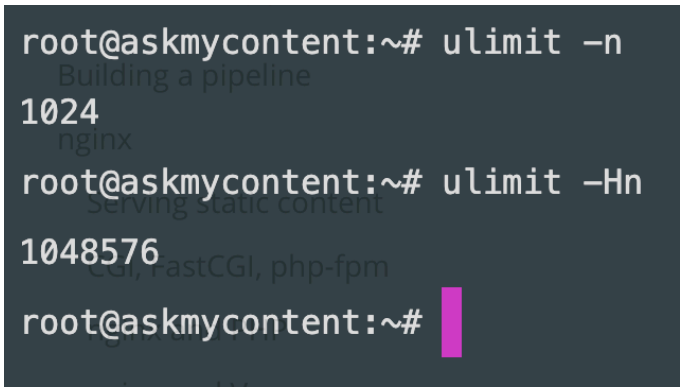
Disclaimer: If you don't have high-traffic spikes and performance problems leave this setting as it is. worker_connections === ulimit -n

However, this means that we limited ourselves to 4k concurrent users even before they hit our API. But wasn't nginx designed to handle 10k+ concurrent connections? Yes, it was. 20 years ago. So we can probably do better.

If you run the following command:

```
ulimit -Hn
```

You get the **real** connection limit for a process. This command gives you the hard limit while `-n` returns the soft limit. In my case, the difference is, let's say, quite dramatic:



Soft limit means, it's the default, but it can be overwritten. So how can we hack it?

```
user www-data;
worker_processes auto;
worker_rlimit_nofile 2048;

events {
    worker_connections 2048;
}
```

By default, the value of `worker_rlimit_nofile` is set to the system's maximum number of file descriptors, or in other words the value `ulimit -n` returns with. However, we can change it because it's only a "soft" limit.

It's important to note that setting `worker_rlimit_nofile` too high can lead to performance issues and even crashes if the server doesn't have enough resources to handle the load. Only change this if you have real spikes and problems.

fpm processes

php-fpm also comes with a number of configuration that can affect the performance of our servers. These are the most important ones:

- `pm.max_children`: This directive sets the maximum number of fpm child processes that can be started. This is similar to `worker_processes` in nginx.
- `pm.start_servers`: This directive sets the number of fpm child processes that should be started when the fpm service is first started.
- `pm.min_spare_servers`: This directive sets the minimum number of idle fpm child processes that should be kept running to handle incoming requests.
- `pm.max_spare_servers`: This is the maximum number of idle fpm child processes.
- `pm.max_requests`: This directive sets the maximum number of requests that an fpm child process can handle before it is terminated and replaced with a new child process. This is similar to the `--max-jobs` option of the `queue:work` command.

So we can set `max_children` to the number of CPUs, right? Actually, nope.

The number of php-fpm processes is often calculated **based on memory** rather than CPU because PHP processes are typically memory-bound rather than CPU-bound.

When a PHP script is executed, it loads into memory and requires a certain amount of memory to run. The more PHP processes that are running simultaneously, the more memory will be consumed by the server. If too many PHP processes are started, the server may run out of memory and begin to swap, which can lead to performance issues.

TL;DR: if you don't have some obvious performance issue in your code php usually consumes more memory than CPU.

So we need a few pieces of information to figure out the correct number for the `max_children` config:

- How much memory does your server have?
- How much memory does a php-fpm process consume on average?
- How much memory does your server need just to stay alive?

Here's a command that will give you the average memory used by fpm processes:

```
ps -ylC php-fpm8.1 --sort:rss
```

- `ps` is a command used to display information about running processes.
- `-y` tells `ps` to display the process ID (PID) and the process's controlling terminal.
- `-l` instructs `ps` to display additional information about the process, including the process's state, the amount of CPU time it has used, and the command that started the process.
- `-C php-fpm8.1` tells `ps` to only display information about processes with the name `php-fpm8.1`.
- `--sort:rss`: will sort the output based on the amount of resident set size (RSS) used by each process.

What the hell is the resident set size? It's a memory utilization metric that refers to the amount of physical memory currently being used by a process. It includes the amount of memory that is allocated to the process and **cannot be shared** with other processes. This includes the process's executable code, data, and stack space, as well as any memory-mapped files or shared libraries that the process is using.

It's called "resident" for a reason. It shows the amount of memory that cannot be used by other processes. For example, when you run `memory_get_peak_usage()` in PHP it only returns the memory used by the PHP script. On the other hand, RSS measures the total memory usage of the entire process.

The command will spam your terminal with an output such as this:

```
root@askmycontent:~# ps -ylC php-fpm8.1 --sort:rss
S   UID     PID    PPID  C PRI  NI   RSS    SZ WCHAN  TTY          TIME CMD
S     0   13789       1  0  80   0 25088 62620 ep_pol ?        00:00:18 php-fpm8.1
S    33   18313   13789  0  80   0 42440 81729 skb_wa ?        00:00:08 php-fpm8.1
S    33   18317   13789  0  80   0 43156 81750 skb_wa ?        00:00:13 php-fpm8.1
S    33   13791   13789  0  80   0 43984 81745 skb_wa ?        00:00:13 php-fpm8.1
root@askmycontent:~#
```

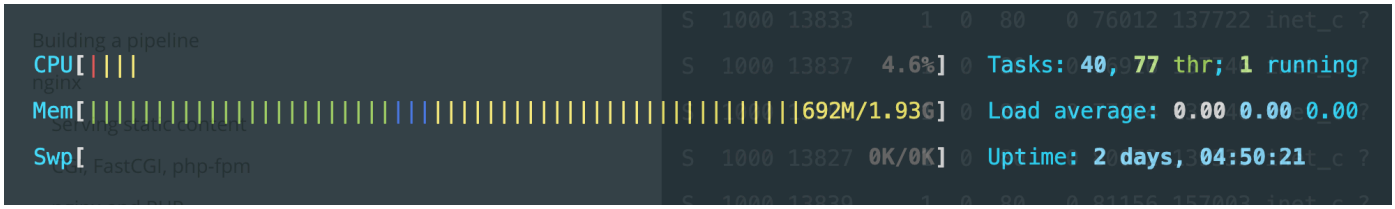
The RSS column shows the memory usage. From 25Mb 43MB in this case. The first line (which has significantly lower memory usage) is usually the master process. We can take that out of the equation and say the average memory used by a php-fpm worker process is 43MB.

However, here are some numbers from a production (older) app:

```
S  1000 13848     1  0  80   0 74408 137609 inet_c ?        00:00:05 php-fpm7.3
S  1000 13841     1  0  80   0 74460 137601 inet_c ?        00:00:06 php-fpm7.3
S  1000 13838     1  0  80   0 74768 137729 inet_c ?        00:00:08 php-fpm7.3
S  1000 13862     1  0  80   0 75000 139088 inet_c ?        00:00:01 php-fpm7.3
S  1000 13846     1  0  80   0 75640 137616 inet_c ?        00:00:04 php-fpm7.3
S  1000 13833     1  0  80   0 76012 137722 inet_c ?        00:00:10 php-fpm7.3
S  1000 13837     1  0  80   0 76916 137746 inet_c ?        00:00:10 php-fpm7.3
S  1000 13806     1  0  80   0 77484 137746 inet_c ?        00:00:16 php-fpm7.3
S  1000 13827     1  0  80   0 80072 139535 inet_c ?        00:00:12 php-fpm7.3
S  1000 13839     1  0  80   0 81156 157003 inet_c ?        00:00:07 php-fpm7.3
S  1000 13840     1  0  80   0 81652 156791 inet_c ?        00:00:07 php-fpm7.3
S  1000 13849     1  0  80   0 81900 156991 inet_c ?        00:00:06 php-fpm7.3
S  1000 13836     1  0  80   0 85164 158663 inet_c ?        00:00:08 php-fpm7.3
```

Yes, these are 130MB+ numbers.

The next question is how much memory does your server need just to stay alive? This can be determined using `htop`:



As you can see from the load average, right now nothing is happening on this server but it uses ~700MB of RAM. This memory is used by Linux, PHP, MySQL, Redis, and all the system components installed on the machine.

So the answers are:

- This server has 2GB of RAM
- It needs 700MB to survive
- On average an fpm process uses 43MB of RAM

This means there is 1.3GB of RAM left to use. So we can spin up 1300/30=30 fpm processes.

It's a good practice to decrease the available RAM by at least 10% as a kind of "safety margin". So let's calculate with 1.17GB of RAM: 1170/37=28.

So on this particular server, I can probably run 25-30 fpm processes.

Here's how we can determine the other values:

Config	General	This example
pm.max_children	As shown above	28
pm.start_servers	~25% of max_children	7
pm.min_spare_servers	~25% of max_children	7
pm.max_spare_servers	~75% of max_children	21

To be completely honest, I'm not sure how these values are calculated but they are the "standard" settings. You can search these configs on the web and you probably run into an article suggesting similar numbers. By the way, there's also a calculator here.

To configure these values we need to edit the `/etc/php/8.1/fpm/pool.d/www.conf` file:

```
pm.max_children = 28
pm.start_servers = 7
pm.min_spare_servers = 7
pm.max_spare_servers = 21
```

I added this file to the example repo inside the `deployment/config/php-fpm` directory. I also changed the deploy script and included these two lines:

```
cp $PROJECT_DIR"/deployment/config/php-fpm/www.conf"
/etc/php/8.1/fpm/pool.d/www.conf
systemctl restart php8.1-fpm.service
```

If you remember I used `systemctl reload` for nginx but now I'm using `systemctl restart`. First of all, here's the difference between the two:

- `reload`: reloads the config without stopping and starting the service. It does not cause downtime.
- `restart`: stops and starts the service, effectively restarting it. It terminates all active connections and processes associated with the service and starts it again with a fresh state. It does cause downtime.

Changing the number of children processes requires a full `restart` since fpm needs to kill and spawn processes. This is also true for nginx as well! So if you change the number of worker processes you need to restart it.

opcache

php opcache is a built-in php extension that provides byte-code caching to php. When php scripts are executed, they are first compiled into byte-code, which can be executed by the php engine. By caching the byte-code in memory, opcache eliminates the need to recompile PHP scripts on every request, which can significantly improve the performance of PHP applications.

I'd like to eliminate a misconception: opcache caches the php code itself. It does not cache the result of the php script. So it's not a problem if you code uses a database or other data sources.

Byte-code is a low-level representation of your php code that can be executed by a virtual machine, such as the Zend Engine. It's similar to assembly, for example:

```php
function increment(int $x): int
{
    return $x + 1;
}

$incremented = increment($_GET['value']);

echo $incremented;
```

Complies to something like this:

```
filename:       /path/to/file.php
function name:  (null)
number of ops:  10
compiled vars:  !0 = $x, !1 = $incremented
line     #  op              fetch          ext  return  operands
---------------------------------------------------------------------
   3     0  ZEND_ARRAY_ELEM                      !0, 'value'
         1  FETCH_DIM_R                        $0, $_GET
         2  SEND_VAL                           $0
         3  DO_ICALL                           $2
         4  ASSIGN                             !1, $3
   5     5  SEND_VAR                           !1
         6  DO_ICALL                           'echo'
         7  POP
         8  RETURN                             1
```

```
7         9   RETURN                                  null
```

Dynamic content such as values of variables or results of SQL queries are not cached.

TL;DR: opcache is cool and you should use it.

This is a basic opcahce config:

```
[opcache]
opcache.enable=1
opcache.memory_consumption=256
opcache.interned_strings_buffer=16
opcache.validate_timestamps=0
opcache.revalidate_freq=60
```

- `opcache.enable` enables opcode.
- `opcache.memory_consumption` specifies the amount of memory allocated to the cache. In this case, it's 256 MB.
- `opcache.interned_strings_buffer` specifies the amount of memory reserved for storing interned strings, which can help reduce memory usage. Interned strings are string literals that are stored in a single location in memory. For example, `$x = "hello world"` is a string literal. The memory address of a literal is stored in a hash table, and subsequent references to the same string can be resolved by looking up the memory address in the hash table. 16 means 16MBs. By the way, this behavior is (like lots of other things in php) mimicked after Java.
- `opcache.validate_timestamps` is set to 0, which disables checking for file modifications. This can improve performance in production environments where code changes are infrequent. When `opcache.validate_timestamps` is set to `1` (or `On`), opcache will check the timestamp of php source files every time a request comes into the web server. If the timestamp of a file has changed since it was last cached, opcache will invalidate the cached version of the file and recompile it, ensuring that the cached version is up-to-date.
- `opcache.revalidate_freq` determines how often the module should check the timestamp of a cached file to see if it has been modified. Since `validate_timestamps` is set to `0` we need some way to revalidate cache. That's how we do it. opcache will check the php files every 60 seconds. If something has changed it'll recompile the script.

We need to place this config into a file located in the `/etc/php/8.1/fpm/conf.d` directory. in this folder, you can place any number of `*.ini` files and they will be loaded in the main `ini` file:

Each configuration file in the `conf.d` directory corresponds to a specific PHP extension or your configuration setting. For example, the `10-pdo.ini` file enables the PDO extension, while the `20-mysqli.ini` enables the MySQLi extension.

By default, fpm includes all the configuration files in the `conf.d` directory that have a `.ini` extension.

So I created a `php.ini` in the sample repo under the `deployment/config/php-fpm` folder. In the deploy script, I copy the file into `/etc/php/8.1/fpm/conf.d`:

```
cp $PROJECT_DIR"/deployment/config/php-fpm/www.conf"
/etc/php/8.1/fpm/pool.d/www.conf
cp $PROJECT_DIR"/deployment/config/php-fpm/php.conf"
/etc/php/8.1/fpm/conf.d/php.ini
systemctl restart php8.1-fpm.service
```

The reason I used the name `php.ini` is because we can also define php-related settings:

```
error_reporting = E_ALL
log_errors = On
error_log = /var/log/php-error.log
memory_limit=64
max_execution_time=30
```

1. `error_reporting = E_ALL`: determines which types of errors are reported by PHP. In this case, it is set to report all errors and warnings.
2. `log_errors = On`: php will write errors to a log file instead of displaying them in the browser. This can be useful for debugging or troubleshooting issues.
3. `error_log = /var/log/php-error.log`: we log everything into the `/var/log/ directory.
4. `memory_limit = 64`: don't forget that this setting applies on a per-request basis. So every request has 64MB of RAM. This means if you have some poorly-written code that actually uses 64MB of RAM it only takes 20 users to use 1.2GB of RAM.
5. `max_execution_time = 30`: this also applies to every request. If you have a long-running process process it in a queue.

gzip

Using compression is a pretty low-hanging fruit when it comes to optimization. It means that nginx will compress the response using gzip before sending it to the client.

The configuration is quite simple:

```
http {
  gzip on;
  gzip_comp_level 4;
  gzip_types text/css application/javascript image/jpeg image/png;
}
```

Let's discuss `gzip_comp_level`.The gzip compression algorithm works by compressing data using a sliding window approach, where the size of the sliding window determines the amount of data that can be compressed at once. The compression level determines the size of this sliding window and therefore affects the balance between compression ratio and CPU usage.

When using gzip it's important to note that the responses will be smaller, but CPU usage will be a bit higher since the CPU will actually compress the content. In practice, a compression level of 4 is often used as a good balance between compression ratio and CPU usage.

One more important thing: we do not cache HTML responses. There are two main reasons:

- HTML content is far smaller than images of scripts.
- If the server compressed HTML content the browser would decompress it. Which takes time. So the initial loading of the page can be worse with compression.

Because of these reasons HTML is usually not compressed.

HTTP2

HTTP2 is the second major version of the HTTP protocol used to transfer data over the internet. It was released in 2015 as an upgrade to the widely used HTTP1.1 protocol. HTTP2 offers a significant improvement in performance over its predecessor by introducing several new features, such as multiplexing, server push, and header compression. These features help reduce latency, improve page load times, and enhance the overall user experience. HTTP2 is becoming increasingly popular and is widely supported by modern web browsers and servers.

Multiplexing is a feature that allows multiple requests and responses to be sent and received simultaneously over a single connection. In HTTP/1.1, each request had to wait for the previous request to complete before it could be sent, which led to a lot of latency and slower page load times.

Does it ring any bell? The same happened with CGI and FastCGI. CGI uses a "one-process-per-request" model which turned out to be a bit resource-heavy and slow so FastCGI introduced multiplexing at the process level. HTTP2 does something very similar but with TCP connections.

Here's how multiplexing works in a nutshell:

- Requests and responses are broken down into smaller units called "frames"
- Each frame is assigned a unique identifier
- These frames can be sent over the same connection, which allows multiple requests to be processed at the same time.

This reduces latency and improves page load times, especially for websites that have a lot of resources to load. Such as images, scripts, and javascript files.

Header compression is exactly what it sounds like. In HTTP1, headers were sent as plain text, which could be quite verbose, especially for requests and responses with a lot of headers. In HTTP2 headers are compressed using a technique called HPACK, which reduces their size significantly.

Server push is an awesome feature. With server push, the server can "predict" which resources the client will need and proactively push them to the client. For example, if the browser requires CSS and a PNG, the server can push those resources to the client as soon as the main HTML file is requested. So the client doesn't need to request them one by one.

For example, when a client requests an `index.html` that looks similar to this:

```
<html>
<head>
  <title>Hi mom!</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <img src="me.png">
</body>
</html>
```

An HTTP2 response looks like this:

```
Response
--------
Server: mysite.com
HTTP/2 200 OK
Content-Type: text/html
Link: </style.css>; rel=preload; as=style, </me.png>; rel=preload; as=image
```

The `Link` header is the server push-related part. The `rel=preload` attribute tells the browser to preload these resources.

Now let's see how we can enable HTTP2 with nginx:

```
listen 443 ssl http2;
```

Yes, that's it! We only need to add the `http2` to the listen directive. It's important to note that HTTP2 only works with HTTPS.

By the way, HTTP2 was introduced in 2015 so it's time to use it!

TLS1.3

First, go to this TLS checker and check which version your site uses. If it's not TLS1.3 you should upgrade.

I'm not the right person to explain the difference between the two versions, but here's the TL;DR:

- TLS 1.3 has better performance.
- TLS 1.3 has better security

Enabling 1.3 is super simple:

```
ssl_protocols TLSv1.3;
```

Now your site uses the latest TLS version.

nginx cache

There are different types of caching mechanisms in nginx. We're gonna discover three of them:

- Static content
- FastCGI
- Proxy

Caching static content

It's probably the most common and easiest one. The idea is that nginx will instruct the browser to store static content such as images or CSS files so that when an HTML page tries to load them it won't make a request to the server.

Caching static content with nginx can significantly improve the performance of a web application by reducing the number of requests to the server and decreasing the load time of pages.

nginx provides several ways to cache static content such as JavaScript, CSS, and images. One way is to use the `expires` directive to set a time interval for the cached content to be considered fresh.

This is a basic config:

```
location ~* \.(css|js|png|jpg|gif|ico)$ {
    access_log off;
    add_header Cache-Control public;
    add_header Vary Accept-Encoding;
    expires 1d;
}
```

`~*` means a case-sensitive regular expression that matches files such as `https://example.com/style.css`

In most cases, it's a good idea to turn off the `access_log` when requesting images, CSS, and js files. It spams the hell out of your access log file but doesn't really help you.

The other directives are:

- `add_header Cache-Control public;`: this adds a response header to enable caching of the static files by public caches such as browsers, proxies, and CDNs. Basically, this instructs the browser to store the files.
- `add_header Vary Accept-Encoding;`: this adds a response header to indicate that the content may vary based on the encoding of the request.
- `expires 1d;`: this sets the expiration time for the cached content to 1 day from the time of the request. There's no "perfect" time here. It depends on your deployment cycle, the usage, and so on. I usually use a shorter time since it doesn't cause too many errors. For example, if you cache JS files for 7 days because you deploy on a weekly basis it means you cannot release a bugfix confidently, because browsers might cache the old, buggy version. Of course, you can define a dedicated location directive

for JS, CSS files and another one for images. Something like this:

```
location ~* \.(css|js)$ {
    access_log off;
    add_header Cache-Control public;
    add_header Vary Accept-Encoding;
    expires 1d;
}

location ~* \.(png|jpg|gif|ico)$ {
    access_log off;
    add_header Cache-Control public;
    add_header Vary Accept-Encoding;
    expires 7d;
}
```

Assuming that images don't change that often.

As you can see, it was pretty easy. Caching static content with nginx is an effective way to improve the performance of your app, reduce server load, and enhance the user experience.

Caching fastcgi responses

Caching fastcgi responses with nginx is a technique used to improve the performance of dynamic web applications that use fastcgi to communicate with backend servers.

The `fastcgi_cache` directive can be used to store the responses from the fastcgi server on disk and serve them directly to clients without having to go through the backend server every time. So this is what happens:

- The browser sends a request to the API
- nginx forwards it to fastcgi which runs our application
- **nginx saves the response to the disk**
- It returns the response to the client
- Next time when a request comes into the same URL it won't forward the request to fastcgi. Instead, it loads the content from the disk and returns it immediately to the client.

Caching fastcgi responses can drastically reduce the load on backend servers, improve the response time of web applications, and enhance the user experience. It is particularly useful for websites that have high traffic and serve dynamic content that changes infrequently.

In the company I'm working for, we had a recurring performance problem. The application we're building is a platform for companies to handle their internal communication and other PR or HR-related workflows. One of the most important features of the app is posts and events. Admins can create a post and publish them. Employees get a notification and they can read the post.

Let's say a company has 10000 employees. They publish an important post that interests people. All 1000 employees get the notification in 60 seconds or so. And they all hit the page within a few minutes. That's a big spike compared to the usual traffic. The post details page (where employees go from the mail or push notification) is, let's say, not that optimal. It's legacy code and has many performance problems such as N+1 queries. The page triggers ~80 SQL queries. 10 000 x 80 = 800 000 SQL queries. Eight hundred thousand SQL queries in 5-10 minutes or so. That's bad.

There are at least two things we can do in such situations:

- Optimize the code and remove N+1 queries and other performance issues. This is outside of the scope but fortunately, there's Laracheck which can detect N+1 and other performance problems in your code! Now, that was a seamless plug, wasn't it? 😈
- Cache the contents of the API request in nginx.

In this case, caching is a very good idea, in my opinion:

- The API response doesn't change frequently. Only when admins update or delete the post.
- The response is independent of the current user. Every user sees the same title, content, etc so there's no personalization on the page. This is required because nginx doesn't know anything about users and their settings/preferences.
- Since we're trying to solve a traffic spike problem, it's a very good thing if we could handle it on the nginx-level. This means users won't even hit the API and Laravel. Even if you cache the result of a database query with Laravel `Cache` 10 000 requests will still come into your app.

- We can cache the posts for a very short time. For example, 1 minute. When the spike happens this 1 minute means thousands of users. But, using a short TTL means we cannot make big mistakes. Cache invalidation is hard. Harder than we think so it's always a safe bet to use shorter TTLs. In this case, it perfectly fits the use case.

I'll solve the same situation in the sample app. There's an `/api/posts/{post}` endpoint that we're gonna cache.

```
http {
  fastcgi_cache_path /tmp/nginx_cache levels=1:2 keys_zone=content_cache:100m
inactive=10m;

  add_header X-Cache $upstream_cache_status;
}
```

First, we need to tell nginx where to store the cache on the disk. This is done by using the `fastcgi_cache_path` directive. It has a few configurations:

- All cached content will be stored in the `/tmp/nginx` directory.
- `levels=1:2` tells nginx to create 2 levels of subdirectories inside this folder. The folder structure will be something like that:
  - e
    - 4**e**
      - b45cffe084dd3d20d928bee85e7b0f**4e**
      - 2c322014fccc0a5cfbaf94a4767db0**4e**
  - 2
    - 3**2**
      - e2446c34e2b8dba2b57a9bcba4854d**32**

So `levels=1:2` means that the first level of directories contains **1** character from the end of the hashed directory name. Such as `e` and `b45cffe084dd3d20d928bee85e7b0f4e`. And then on the second level, the directory's name contains **characters** from the end of the hash. Such as `4e` and `b45cffe084dd3d20d928bee85e7b0f4e`.

If you don't specify the `levels` option nginx will create only one level of directories. Which is fine for smaller sites. However, for bigger traffic, specifying the `levels` option is a good practice since it can boost the performance of nginx.

- `keys_zone=content_cache:100m` defines the key of this cache which we can reference later. The `100m` sets the size of the cache to 100MB.
- `inactive=10m` tells how long to keep a cache entry after it was last accessed. In this case, it's 10 minutes.

And now we can use it:

```
location ~\.php {
  fastcgi_cache_key $scheme$host$request_uri$request_method;
  fastcgi_cache content_cache;
  fastcgi_cache_valid 200 5m;
  fastcgi_cache_use_stale error timeout invalid_header http_500 http_503
http_404;
  fastcgi_ignore_headers Cache-Control Expires Set-Cookie;

  try_files $uri =404;
  include /etc/nginx/fastcgi.conf;
  fastcgi_pass unix:/run/php/php8.1-fpm.sock;
  fastcgi_index index.php;
  fastcgi_param PATH_INFO $fastcgi_path_info;
}
```

- `fastcgi_cache_key` defines the key for the given request. It looks like this: `HTTPSmysite.composts/1GET` This is the string that will be the filename after it's hashed.
- `fastcgi_cache` here we need to specify the key we used in the `keys_zone` option.
- The `fastcgi_cache_valid` directive sets the maximum time that a cached response can be considered valid. In this case, it's set to 5 minutes for only successful (200) responses.
- The `fastcgi_ignore_headers` directive specifies which response headers should be ignored when caching responses. Basically, they won't be cached at all. Caching cache-related headers with expiration dates does not make much sense.
- The `fastcgi_cache_use_stale` directive specifies which types of stale cached responses can be used if the backend server is unavailable or returns an error. A stale cached response is a response that has been previously cached by the server, but has exceeded its maximum allowed time to remain in the cache and is considered "stale". This basically means that even if the BE is currently down we can serve clients by using older cached responses. In this project, where the content is not changing that often it's a perfectly good strategy to ensure better availability.

All right, so we added these directives to the `location ~\.php` location so it will apply to **every** request. Which is not the desired outcome. The way we can control which locations should use cache looks like this:

```
fastcgi_cache_bypass 1;
fastcgi_no_cache 1;
```

- If `fastcgi_cache_bypass` is 1 then nginx will not use cache and forwards the request to the backend.
- If `fastcgi_no_cache` is 1 then nginx will not cache the response at all.

Obviously, we need a way to set these values dynamically. Fortunately, nginx can handle variables and if statements:

```
set $no_cache 1;

if ($request_uri ~* "\/posts\/([0-9]+)") {
  set $no_cache 0;
}

if ($request_method != GET) {
  set $no_cache 1;
}
```

This code will set the `$non_cache` variable to `0` only if the request is something like this: `GET /posts/12` Otherwise, it'll be `1`. Finally, we can use this variable:

```
location ~\.php {
  fastcgi_cache_key $scheme$host$request_uri$request_method;
  fastcgi_cache content_cache;
  fastcgi_cache_valid 200 5m;
  fastcgi_cache_use_stale error timeout invalid_header http_500 http_503
http_404;
  fastcgi_ignore_headers Cache-Control Expires Set-Cookie;
  fastcgi_cache_bypass $no_cache;
  fastcgi_no_cache $no_cache;

  try_files $uri =404;
  include /etc/nginx/fastcgi.conf;
  fastcgi_pass unix:/run/php/php8.1-fpm.sock;
  fastcgi_index index.php;
  fastcgi_param PATH_INFO $fastcgi_path_info;
}
```

With this simple config, we can cache every `posts/{post}` for 10 minutes. The 5 minutes is an arbitrary number and it's different for every use case. As I said earlier, with this example, I wanted to solve a performance problem that happens in a really short time. So caching responses for 5 minutes is a good solution to this problem. And of course, the shorter you cache something the less risk you take (by serving outdated responses).

An important thing about caching on the nginx level: it can be tricky (or even impossible) to cache user-dependent content. For example, what is users can see the Post in their preferred language? To make this possible we need to add the language to the cache key so one post will have many cache keys. One for each language. If you have the language key in the URL it's not a hard task, but if you don't, you have to refactor your application. Or you need to use Laravel cache (where you have access to the user object and preferences of course).

There are other kinds of settings that can cause problems. For example, what if every post has an audience? So you can define who can see your post (for example, only your followers or everyone, etc). To handle this, you probably need to add the user ID to the URL and the cache key as well.

Be aware of these scenarios.

There's also a `proxy_pass` option that you can use in a reverse proxy.

Backups and restore

Having a backup of your application's data is crucial for several reasons. Firstly, it provides an extra layer of protection against accidental data loss or corruption. Even the most robust and secure applications can fall victim to unforeseen events like hardware failure, power outages, or even human error. In such cases, having a backup can be the difference between a quick recovery and a catastrophic loss.

Secondly, backups are essential for disaster recovery. In the unfortunate event of a security breach, a backup can help you restore your data to a previous state, reducing the risk of data loss and minimizing the impact on your business.

Here are some important things about having backups:

- Store them on an **external storage** such as S3. Storing the backups on the same server as the application is not a great idea.
- Have **regular** backups. Of course, it depends on the nature of your application and the money you can spend on storage. I'd say having at least one backup per day is the minimum you need to do.
- Keep **at least a week** worth of backups in your storage. It also depends on your app but if you have some bug that causes invalid data, for example, you still have a week of backups to recover from.
- Have a **restore** script that can run at any time. It has to be as seamless as possible. Ideally, the whole process is automated and you only need to run a script or push a button.
- Include the database dump, redis dump, storage folder, and .env file in the backup. Don't include the `vendor` and `node_modules` folders.

In this chapter, we're going to implement all of these.

AWS S3

One of the best ways to store backups is using AWS S3. Amazon S3 (Simple Storage Service) is a highly scalable, secure, and durable object storage service provided by AWS. It allows us to store and retrieve any amount of data from anywhere on the web, making it a popular choice for cloud-based storage solutions. S3 provides a simple HTTP API that can be used to store and retrieve data from anywhere on the web.

In S3, data is stored in **buckets**, which are essentially containers for objects. Buckets are used to store and organize data and can be used to host static websites as well. It's important to note that buckets are not folders. It's more similar to a git repository where you can have any number of folders and files (objects). Or the `/` folder in Linux. So it's the root.

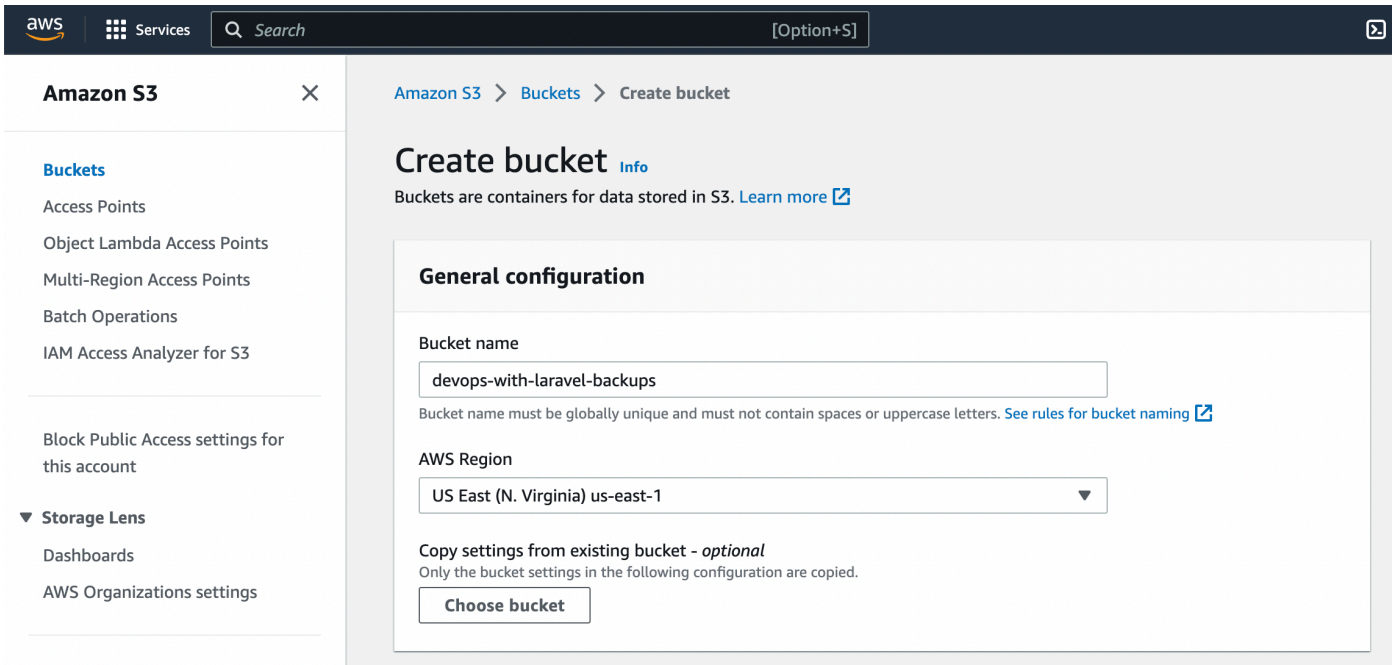
Objects are the individual files that are stored in buckets. Objects can be anything from a simple text file to a large video file or database backup (like in our case). Each object is identified by a unique key, which is used to retrieve the object from the bucket.

In my case, the bucket name will be `devops-with-laravel-backups` and we'll store ZIP files in that bucket. We will URLs such as this:

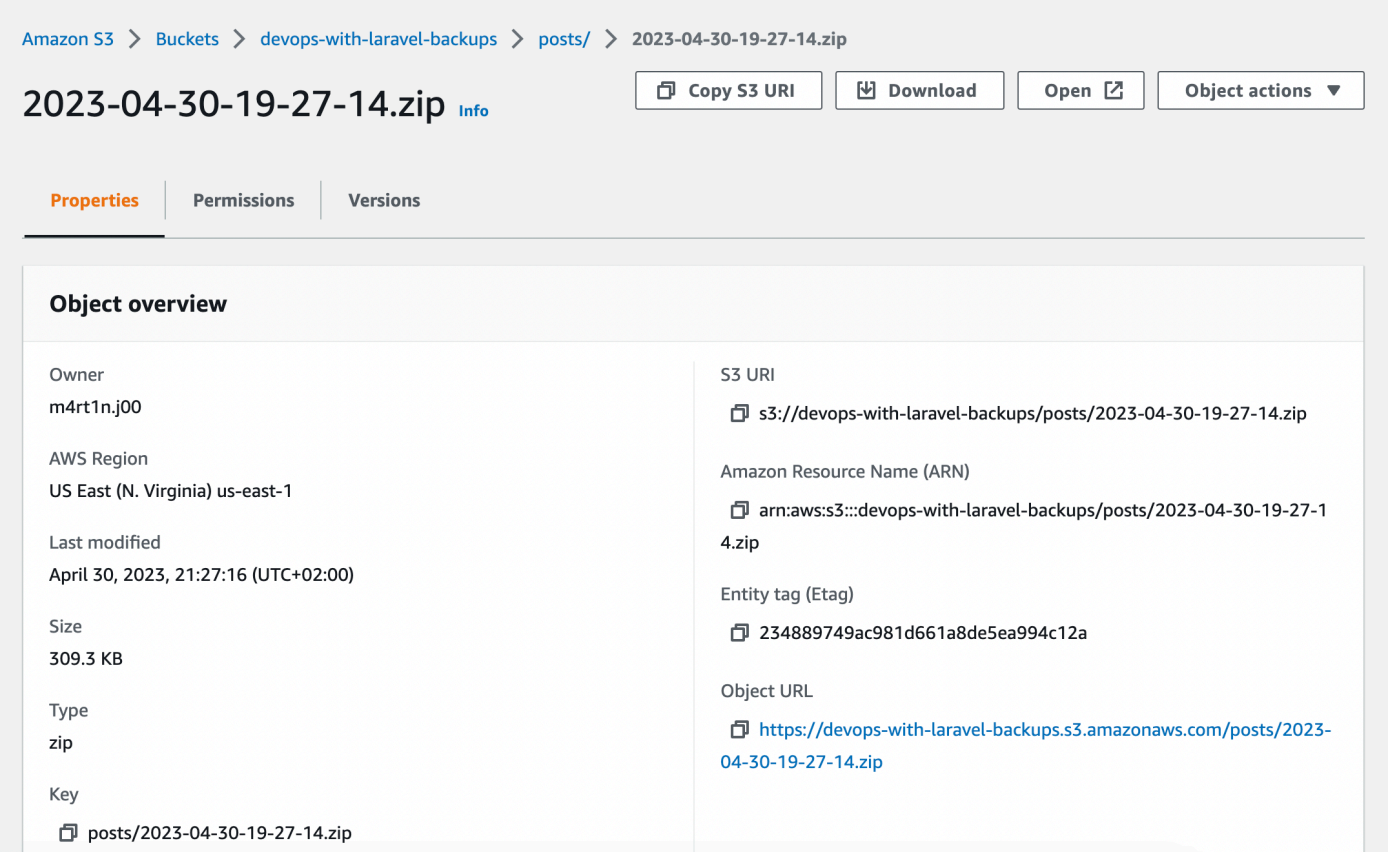
```
https://devops-with-laravel-backups.s3.us-west-2.amazonaws.com/2023-04-30-11-23-05.zip
```

As you can guess from this URL, bucket names must be globally unique.

To create a bucket you need an AWS account. Then login into the console and search for "S3." Then click on "Buckets" in the left navigation. We don't need to set any options, only a bucket name:



Now we have a bucket where we can upload files manually, or using the AWS CLI tool. If you upload a file you can see properties such as these:



There are two URLs:

- S3 URI: s3://devops-with-laravel-backups/posts/2023-04-30-19-27-14.zip
- Object URL: https://devops-with-laravel-backups.s3.amazonaws.com/posts/2023-04-30-19-27-14.zip

We want to access these backups from the restore bash script where we will use the AWS CLI tool. This tool works with S+ URIs. If you want to access files via HTTP you need to use the Object URL.

First, we need to install the AWS CLI tool. You can find the installation guide here, but on Linux it looks like this:

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"
unzip awscliv2.zip
./aws/install
```

I added these commands to the `provision_server.sh` script.

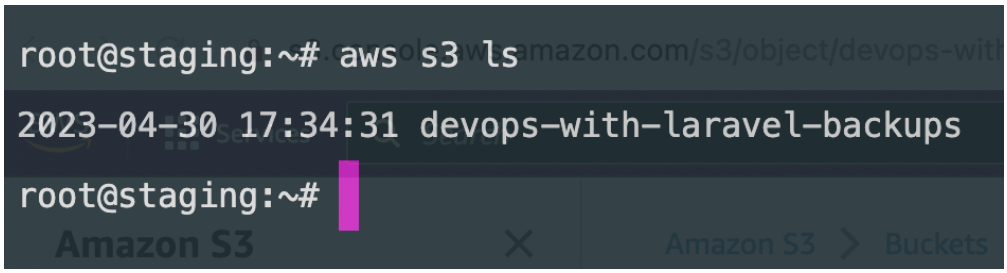
After that, you need an access token to use authorization. If you're working in a team you should enable the IAM identity center. I'm not going into too much detail because it's outside of the scope of this book, and it's well documented here. However, if you're just playing with AWS you can use a root user access key (which is not recommended in a company or production environment!). Just click your username in the upper right corner and click on the "Security credentials" link. There's an "Access keys" section where you can create a new one.

After that, you need to configure your CLI. Run this command:

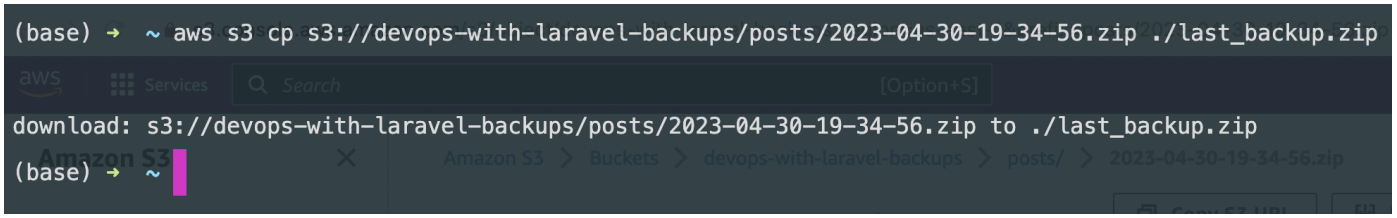
```
aws configure
```

It will ask for the access key.

After that, you should be able to run S3 commands. For example, list your buckets:



Or you can download specific files:



That's the command we need to use when restoring a backup.

To use S3 with Laravel, first, we need to set these environment variables:

```
AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
AWS_DEFAULT_REGION=us-east-1
AWS_BUCKET=devops-with-laravel-backups
AWS_USE_PATH_STYLE_ENDPOINT=false
```

After that, we need to install this package:

```
composer require league/flysystem-aws-s3-v3 "^3.0"
```

And that's it! Everything is ready.

By the way, you don't have to use S3 if you don't want to. Lots of cloud providers offer S3-compatible storage solutions. S3-compatible means it has the same API. DigitalOcean Spaces is a good example.

You can even run your own self-hosted S3 if you'd like to. There's an application called MinIO which is S3-compatible and you can run it on your own server. You can buy a VPS for ~$6 that has ~25GB of space and MinIO installed on it and you're probably good to go with smaller projects. You can even use SFTP (instead of MinIO) if you'd like to. spatie/laravel-backup (which we're gonna use in a minute) supports it.

If you'd like to switch to another S3-compatible storage all you need to do is configure Laravel's filesystem:

```
's3' => [
  'endpoint' => env('AWS_ENDPOINT', 'https://digital-ocean-spaces-url'),
],
```

spatie/laravel-backup

There's a pretty useful package by Spatie called laravel-backup. As the name suggests, it can create backups from a Laravel app. It's quite straightforward to setup and configure so I'm not gonna go into too much detail. They have great documentation here.

To configure the destination only this option needs to be updated in the `config/backup.php` file:

```
'disks' => [
  's3',
],
```

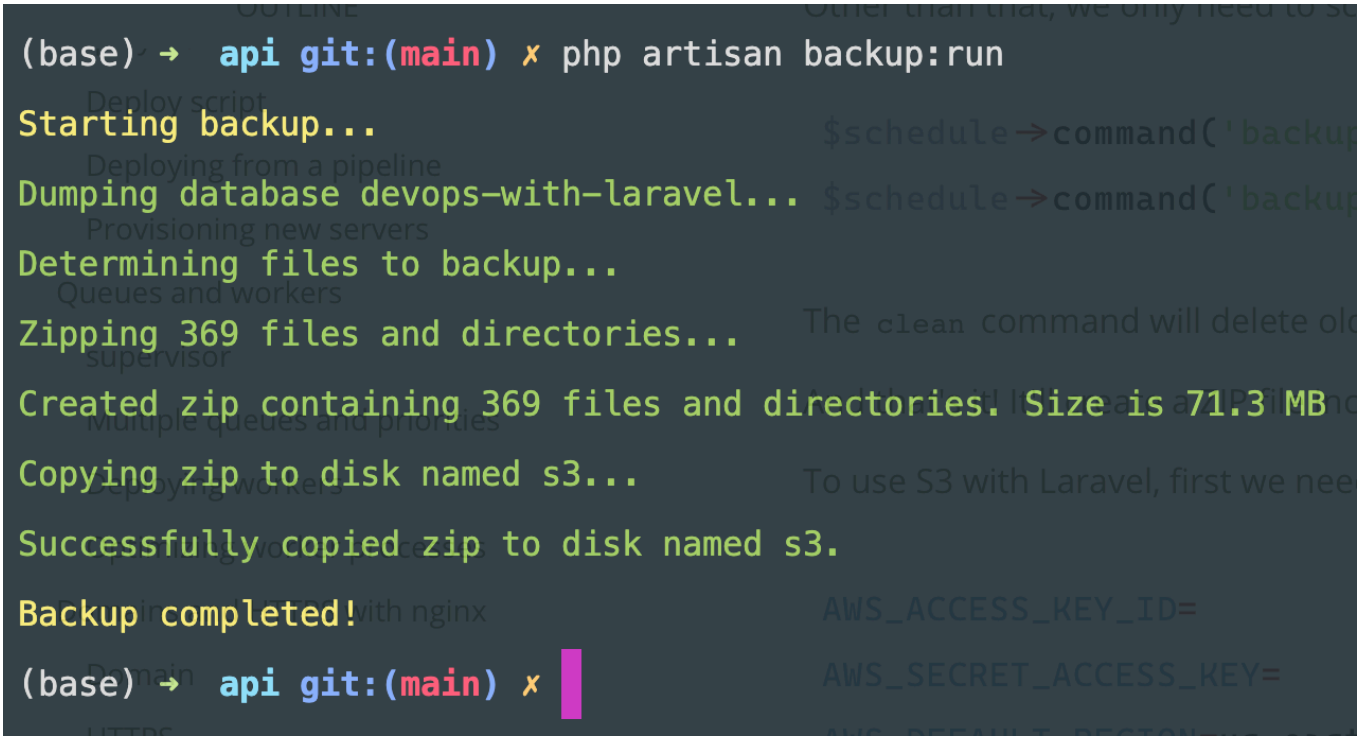
Other than that, we only need to schedule two commands:

```
$schedule->command('backup:clean')->daily()->at('01:00');

$schedule->command('backup:run')->daily()->at('01:30');
```

The `clean` command will delete old backups based on your config while the `run` command will create a new backup.

And that's it! It'll create a ZIP file including your databases and the whole directory of your application.

If you run `backup:run` manually you'll get the following message:



And you should see the file in S3. By the way, laravel-backup will create a folder with the `APP_NAME` inside S3, by default. I left this setting on but you can configure everything in the `config/backup.php` file.

Restore script

Restoring a backup means unzipping a ZIP archive first. To do that we need `unzip` on the server. I added this line to the provision server script:

```
apt install unzip
```

Now let's go through the script itself:

```
#!/bin/bash

set -e

BACKUP_FILENAME=$1
MYSQL_USER=$2
MYSQL_PASSWORD=$3

PROJECT_DIR="/var/www/html/posts"
BACKUP_DIR=$PROJECT_DIR"/api/storage/app/backup"
```

It takes three arguments:

- `BACKUP_FILENAME`: when running a restore we need to specify which backup file we want to restore from.
- `MYSQL_USER` and `MYSQL_PASSWORD`: the script will import a MySQL dump so it needs the user and the password.

The `BACKUP_DIR` is the folder where to script will download and extract the backup file.

The next step is to download the backup file:

```
aws s3 cp s3://devops-with-laravel-backups/$BACKUP_FILENAME
$PROJECT_DIR"/api/storage/app/backup.zip"

unzip -o $PROJECT_DIR"/api/storage/app/backup.zip" -d $BACKUP_DIR
```

As I discussed earlier, we need the `aws s3 cp` command which takes two arguments:

- source which is the S3 location of the file
- destination which is the local directory

After the file is downloaded from S3, `unzip` will decompress it into the `BACKUP_DIR` location.

The next step is very important and often forgotten:

```
php $PROJECT_DIR"/api/artisan" down
```

MySQL will drop tables and then create them from a dump file so it's important to shut down the application.

The actual import is quite easy:

```
mysql -u$MYSQL_USER -p$MYSQL_PASSWORD posts < $BACKUP_DIR"/db-dumps/mysql-posts.sql"
```

It will run the `mysql-posts.sql` scripts against the `posts` database. laravel-backup creates a dump file that uses

```
DROP TABLE EXISTS `posts`;
```

So first, it drops the table, then creates it, and finally inserts data into it.

Next, I want to make a copy from the current files:

```
mv $PROJECT_DIR"/api/.env" $PROJECT_DIR"/api/.env_before_restore"
mv $PROJECT_DIR"/api/storage/app/public"
$PROJECT_DIR"/api/storage/app/public_before_restore"
```

This is just for safety reasons. I don't want to lose any data if possible. In this application, every user uploaded content goes to `app/public`. But if you have other important folders such as `app/uploads` or `app/profile-pictures` make sure to create a copy of those as well.

And now we can restore the files from the backup:

```
mv $BACKUP_DIR"/"$PROJECT_DIR"/api/.env" $PROJECT_DIR"/api/.env"
mv $BACKUP_DIR"/"$PROJECT_DIR"/api/storage/app/public"
$PROJECT_DIR"/api/storage/app/public"
```

laravel-backup copies the whole directory structure in the ZIP file. So when you extract it, you don't see folders such as `app` or `storage` but you will see the absolute path of your project, which is in my case: `var/www/html/posts`

So the path `$BACKUP_DIR"/"$PROJECT_DIR` means:

- `BACKUP_DIR` = `/var/www/html/posts/api/storage/app/backup`
- `PROJECT_DIR` = `/var/www/html/posts`
- Together = `/var/www/html/posts/api/storage/app/backup/var/www/html/posts`

The last is also pretty important and often forgotten:

```
php $PROJECT_DIR"/api/artisan" optimize:clear
```

We just replaced the `.env` file. If there are any changes they won't apply until you clear the config cache.

And finally, we can start the app:

```
php $PROJECT_DIR"/api/artisan" up
```

As you can see, it's not that complicated.

Since restoring an application is pretty rare I don't think this script needs to be added to a pipeline or something like that. Because of that, I added a simple script that can be executed from your local machine.

You need to run the following:

```
./run_restore_from_local.sh root 163.92.129.186 posts/2023-04-30-19-27-14.zip
root XVENmqj4JwNz
```

It takes:

- An ssh user
- The server's IP address
- The file path in S3 (starting with "posts")
- The MySQL username
- And the password

Here's the whole script:

```
#!/bin/bash

set -e

BACKUP_FILENAME=$1
MYSQL_USER=$2
MYSQL_PASSWORD=$3

PROJECT_DIR="/var/www/html/posts"
BACKUP_DIR=$PROJECT_DIR"/api/storage/app/backup"

aws s3 cp s3://devops-with-laravel-backups/$BACKUP_FILENAME
$PROJECT_DIR"/api/storage/app/backup.zip"
unzip -o $PROJECT_DIR"/api/storage/app/backup.zip" -d $BACKUP_DIR

php $PROJECT_DIR"/api/artisan" down

# Restore database
mysql -u$MYSQL_USER -p$MYSQL_PASSWORD posts < $BACKUP_DIR"/db-dumps/mysql-
posts.sql"

# Copy the current files
mv $PROJECT_DIR"/api/.env" $PROJECT_DIR"/api/.env_before_restore"
mv $PROJECT_DIR"/api/storage/app/public"
$PROJECT_DIR"/api/storage/app/public_before_restore"

# Restore old files from backup
mv $BACKUP_DIR"/"$PROJECT_DIR"/api/.env" $PROJECT_DIR"/api/.env"
mv $BACKUP_DIR"/"$PROJECT_DIR"/api/storage/app/public"
$PROJECT_DIR"/api/storage/app/public"

php $PROJECT_DIR"/api/artisan" storage:link
php $PROJECT_DIR"/api/artisan" optimize:clear

php $PROJECT_DIR"/api/artisan" up
```

Before moving to Docker

In the previous chapters, we learned a lot about deploying Laravel projects without Docker and an orchestrator platform. This method has its obvious disadvantages (we'll talk about them in the next chapter when introducing Docker), however, it has a few advantages as well.

For example, if you'd like to ship a side project this can be easily the fastest way of deploying your app. No long-running pipelines, no need to build images, push them to DockerHub, etc. If you're working on a solo project, you probably don't even need a pipeline. Just a deploy script that you can run from your local machine. In 30 seconds the new feature is available for your users. I don't say it's a good model, however, I shipped applications like this, and Pieter Levels always does that. I think he has one $500 VPS to serve 50M+ requests/month and he used SFTP to upload code. Nowadays he's using `git`.

Docker, orchestrators, and cloud services have a learning curve. It's not a shame to use shell scripts and a simple VPS.

However, if you're working in a team, I think Docker is probably a better way to deploy your applications.

Docker

The project files are located in the `3-docker` folder.

The basics in theory

Before we start dockerizing the application and the deployment process let me explain what is it.

Docker is a platform that allows us to create, deploy, and run applications in containers. Containers are a lightweight and portable way to package software and its dependencies, allowing applications to run consistently across different environments.

The most important thing is to package software and its dependencies.

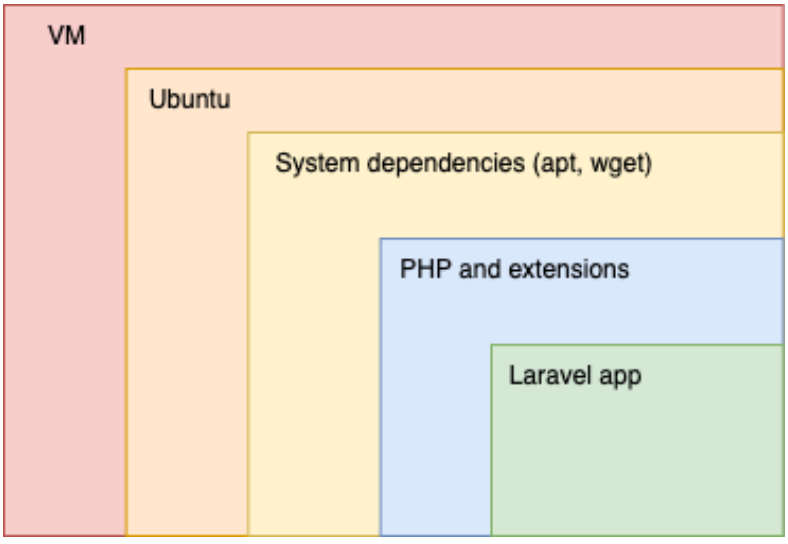
If we want to run the deploy script (from the previous chapters) on a random server we would face some errors. The application and the deploy script assume that the server already has php, nginx, mysql, gd, and so on. This is why we wrote the `provision_server.sh` script. It takes care of installing system dependencies on a new server. But this is still not really "portable" or platform-independent since the script has things like this:

```
wget https://nodejs.org/dist/v14.21.3/node-v14.21.3-linux-x64.tar.xz
apt install php8.1-common php8.1-cli -y
```

How do you know that the server has `wget` and `apt`? For example, `apt` is the package manager on Debian and Debian-based distros (such as Ubuntu or Mint). It works for us only because we created DigitalOcean droplets with Ubuntu. If your VM has CentOS it would fail because their package manager is `yum`.

Despite the fact that we can automatically deploy to a specific VM and cloud provider we still have a problem: the application is not **self-contained**.

Right now, we're looking at layers such as this:



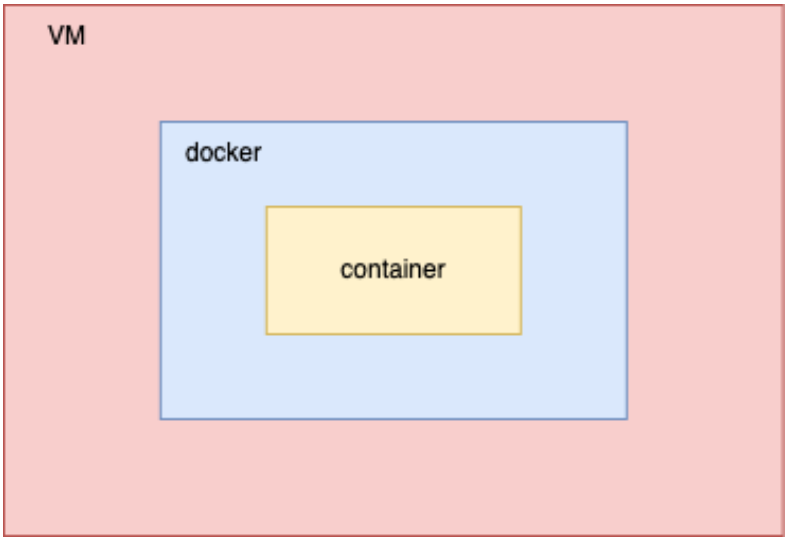
Technically apt or wget is not much different from PHP but they are different from our perspective because we need a specific version of PHP but we probably don't care if the current system has wget 1.14 or 1.13.4

Let's see how much control we have over those different layers:

- Laravel app: 100% control, we write the code.
- PHP and extensions: the deploy script assumes that there is PHP installed on the server. But who installs it? In this case, I choose a specific VM on DigitalOcean that already has PHP. Then again, it was PHP8.0 but the app needs 8.1 so I updated PHP in the `provision_server` script. Can you run the same script on a server with no PHP? I don't know. The deploy script also assumes that nginx is installed. Which is because of the specific VM type I choose. For this reason, I left out the nginx installation from the `provision_server` script. So our current deployment process only works on a server that has nginx pre-installed on it! Of course, you can add nginx to the provision script.
- System dependencies: we just assumed it's an Ubuntu system with apt, wget, git, sed, ufw, php, mysql, curl, and unzip installed on it. And I'm not even talking about the specific version numbers. Certainly, you can add all of these to the provision script if you have a free weekend.
- Ubuntu: we haven't even considered the specific version number of Ubuntu in the previous chapters. We just went with the one on DigitalOcean VMs and it worked.
- VM: we don't know much about the specific hardware but fortunately we don't need to.

As you can see, there are some areas where we are "out of control." Or in other words: the application is not self-contained. It has a lot of moving parts. And if you think about the development process as well it gets even more clunky. Developers have Mac, Linux, and Windows machines. Everyone needs to manually install the right PHP version, extensions, and so on.

Fortunately, with Docker, the whole situation looks like this:



It takes **everything** from the OS to your Laravel source code and packages it into one container. It only needs a machine that has Docker installed on it.

Here are the fundamental concepts of docker:

- **Dockerfile:** It's something like a mixture of a config file and shell scripts. In the previous chapters, we created an Ubuntu-based VM on DigitalOcean and then ran the provision script that installed some dependencies such as php8.1 or nodejs. In the Docker world, we do these steps inside a Dockerfile. I'm going to explain it in more detail but here's a pretty simple example:

```
FROM php:8.1.0-fpm

RUN apt-get install git
```

This config describes a Linux-based image that has PHP, fpm, and git. `FROM` and `RUN` comes from Docker syntax, while `apt-get install` is just a Linux command.

- **Image:** An image is like an ISO file lots of years ago. We have a Dockerfile and then we can build an image from it. For example, if you build the Dockerfile above Docker will create an image that contains every file, folder, and dependency required by Linux, PHP, and git. In this example, PHP8.1 is the base image (the `FROM` part) which is stored in Docker Hub so Docker will download it, and run the commands (the `RUN` part). These commands will also download files from the internet. Finally, everything is stored in a new image stored on your local machine.
- **Container:** A container is a running instance of an image. It's like when you actually mount and play the ISO file. It's an isolated environment that contains everything needed to run an application, including code, libraries, and system tools.

So a Dockerfile is a blueprint that describes your application and its environment (including the operating system). An image is a collection of files and folders based on that blueprint. And a container is a running instance of an image.

And as you might guess images can inherit from each other just like classes in PHP. In this example, the first line means that we want to use the official PHP8.1 image stored in Docker Hub. And the official PHP image uses an official Linux image.

Virtual machines vs containers

If a running container has its own Linux then it's a virtual machine, right? Almost, but no. The main difference is that virtual machines emulate **an entire operating system**, including the kernel, which can be resource-intensive. In contrast, Docker containers share the **host operating system kernel**, which makes them much faster and more efficient than virtual machines. Containers are much smaller and faster to start up than virtual machines.

And also this is the reason (the sharing of host OS kernel) that Docker did not really work on Windows for a long time. Years ago only the pro version of Windows was able to run Docker (not very stable though). And even nowadays you need WSL which is basically a virtual Linux.

The basics in practice

Let's start with running an `index.php` script from a container. The script looks like this:

```
<?php

echo 'Hi mom!';
```

As I mentioned earlier, every major vendor such as PHP, nginx, or MySQL has an official Docker image we can use. These images contain some Linux distro, every system dependency that the vendor needs, and the actual product. In our case, it's going to be PHP 8.1:

```
FROM php:8.1-cli
```

The `FROM` command defines the base image. Every image has a name and a version number in the format of `<vendor>:<version>` The version number is called a tag. These images and tags can be found on Docker Hub.

As you can see, in this case, the tag is not only the version number but `8.1-cli`. CLI means that you can use this PHP image to run a CLI script such as my "Hi mom!" script. You cannot use a CLI image to server requests through a web server. For this, we're gonna use another tag.

As I mentioned earlier, a Docker image is just a collection of files and folders. So let's put some files into it:

```
FROM php:8.1-cli

COPY ./index.php /usr/src/my-app/index.php
```

The `COPY` command copies files from your local machine to the Docker image. In this case, the `index.php` needs to be in the folder as the Dockerfile we're writing right now:

- my-app
  - index.php
  - Dockerfile

So this command copies the `index.php` from your local machine to the Docker image inside the `/usr/src/my-app` folder.

After that, we can define a "working directory:"

```
FROM php:8.1-cli

COPY ./index.php /usr/src/my-app/index.php

WORKDIR /usr/src/my-app
```

It's basically a `cd` command. So any command we issue after the `WORKDIR` will run in the `/usr/src/my-app` folder.

And the last part is:

```
FROM php:8.1-cli

COPY ./index.php /usr/src/my-app/index.php

WORKDIR /usr/src/my-app

CMD [ "php", "./index.php" ]
```

`CMD` runs any CLI command in a weird syntax, where every argument is a new entry in an array. So this is equivalent to this: `php ./index.php`. Remember, the current working directory is `/usr/src/my-app` where `index.php` is located so `./index.php` is a valid path.

So we have the blueprint for the image. Now let's build it:

```
docker build -t hi-mom:0.1 .
```

- `docker build`: This is the command to build a Docker image.
- `-t hi-mom:0.1`: This option tags the image with the name "hi-mom". The name can be anything you choose, but it's typically in the format of "repository:tag". In fact, it's the format of "registry/repository:tag" but we haven't talked about registries.
- `.`: This specifies the build context, which is the location of the Dockerfile and any files it references. In this case, we're using the current directory as the build context because it contains both the Dockerfile and `index.php` as well.

After running the command you should see something like that:

```
docker build -t hi-mom:0.1 .

[+] Building 1.1s (8/8) FINISHED
 => [internal] load .dockerignore                                                                                    0.0s
 => => transferring context: 2B                                                                                      0.0s
 => [internal] load build definition from Dockerfile                                                                 0.0s
 => => transferring dockerfile: 152B                                                                                 0.0s
 => [internal] load metadata for docker.io/library/php:8.1-cli                                                       1.0s
 => [internal] load build context                                                                                    0.0s
 => => transferring context: 30B                                                                                     0.0s
 => CACHED [1/3] FROM docker.io/library/php:8.1-cli@sha256:7d41d8fdb6fd8ef5ab9e53df39894d88297b34615da5bf7cd1242a27c7d9e87a  0.0s
 => [2/3] COPY ./index.php /usr/src/my-app/index.php                                                                 0.0s
 => [3/3] WORKDIR /usr/src/my-app                                                                                    0.0s
 => exporting to image                                                                                               0.0s
 => => exporting layers                                                                                              0.0s
 => => writing image sha256:83d7a7e5ee2bd68ff782295b5325744f5caede61857351493b638a19a0cd610c                         0.0s
 => => naming to docker.io/library/hi-mom:0.1                                                                        0.0s
```

If you run the `docker images` command you should see an image called `hi-mom`:

```
docker images
REPOSITORY                                 TAG           IMAGE ID       CREATED              SIZE
hi-mom                                     0.1           83d7a7e5ee2b   About a minute ago   524MB
```

As you can see, the size is 524MB because Linux (Debian), PHP, and all of the dependencies take up that much space.

To run the image and create a running container out of it all you need to do is:

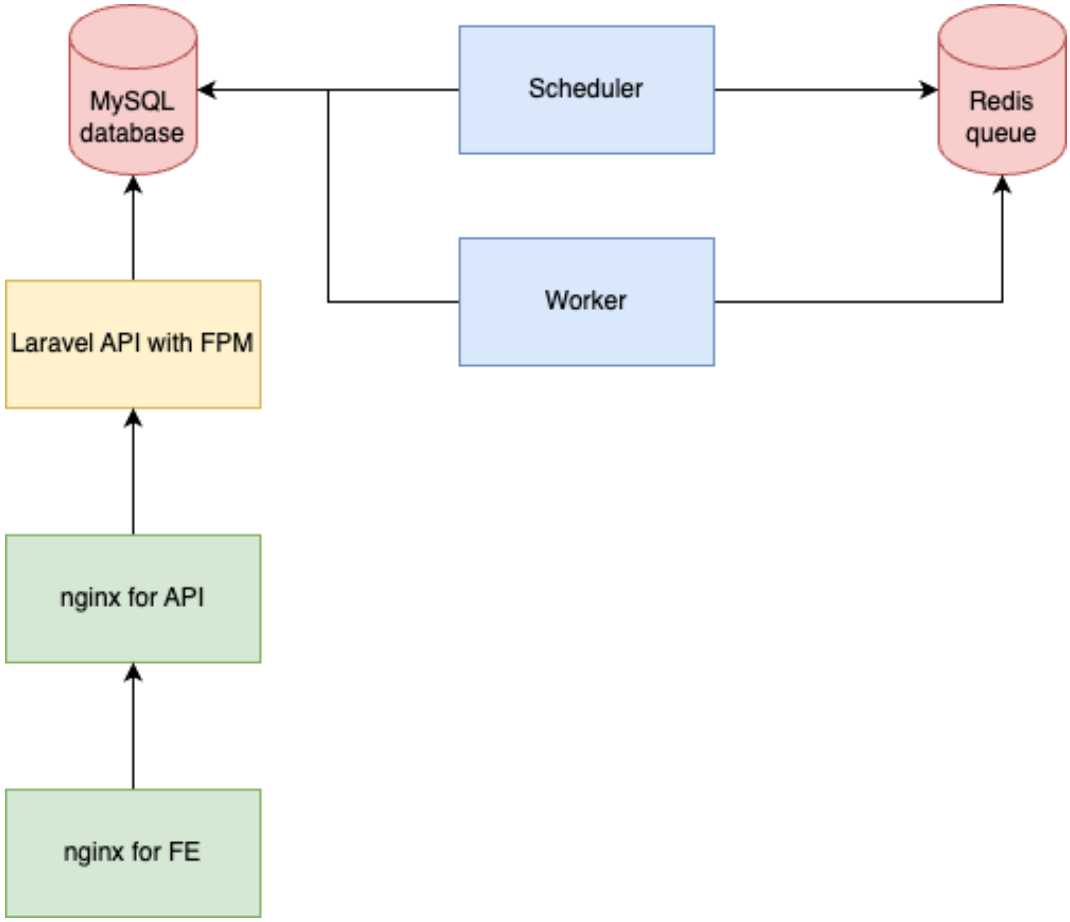
```
docker run hi-mom:0.1
```

And you can see the output:

So these are the basics of running a Docker container. This image contains a single command that runs a script that executes and exits immediately. So it has no long-running processes. Of course, when we want to serve an API via nginx or run 8 worker processes managed by supervisor we need some more advanced Dockerfiles.

Overview of the application

To dockerize the sample application we need the following architecture:



Each box is a container and they have the following responsibilities:

- nginx for FE serves the static files of the Vue frontend
- nginx for API: this nginx instance accepts requests from the Vue app
- Laravel API with FPM: this is where the application lives. nginx forwards HTTP requests to this container, FPM accepts them, and finally, it forwards them to our Laravel app.
- MySQL has a separate container using the official MySQL image
- The scheduler container contains the same Laravel code as the "Laravel API with FPM" container. But instead of running PHP FPM, this container will run `php artisan schedule:run` once every 60 seconds. Just like we used crontab in the previous chapters. The scheduler might dispatch queue jobs, and it might also need the database for various reasons.
- The worker container is similar to the scheduler but it runs the `php artisan queue:work` command so it will pick up jobs from the queue. It also contains the source code, since it runs jobs.
- And finally, we have a container for Redis. Just like MySQL, it uses the official Redis image.

This is the basic architecture of the application. Later, I'm gonna scale containers, use nginx to load balance, and use supervisor to scale workers but for now, the goal is to dockerize everything and use docker-compose to orchestrate the containers.

Dockerizing a Laravel API

Let's start by writing a Dockerfile to the Laravel API:

```
FROM php:8.1-fpm

WORKDIR /usr/src
```

The base image is `php:8.1-fpm`. You can use any version you want, but it's important to use the FPM variant of the image. It has PHP and PHP-FPM preinstalled. You can check out the official Dockerfile here.

In Dockerfiles, I used to use `/usr/src` as the root of the project. Basically, you can use almost any folder you'd like to. Some other examples I encountered with:

- `/usr/local/src`
- `/application`
- `/laravel`
- `/var/www`
- `/var/www/html`

The next step is to install system packages:

```
RUN apt-get update && apt-get install -y \
    libpng-dev \
    libonig-dev \
    libxml2-dev \
    libzip-dev \
    git \
    curl \
    zip \
    unzip \
    supervisor \
    default-mysql-client
```

First, we run an `apt-get update` and then install the following libs:

- `libpng-dev` is needed to deal with PNG files.
- `libonig-dev` is the Oniguruma regular expression library.
- `libxml2-dev` is a widely-used XML parsing and manipulation library written in C.

- `libzip-dev` deals with ZIP files.

These libs are C programs and are needed by PHP or a particular composer package or Laravel itself. Other than these low-level libs we install standard user-facing programs such as git, curl, zip, unzip, supervisor. `default-mysql-client` contains `mysqldump` which is required by `laravel-backup`.

After installing packages, it's recommended to run these commands:

```
RUN apt-get clean && rm -rf /var/lib/apt/lists/*
```

- `apt-get clean` cleans the cache memory of downloaded package files.
- `rm -rf /var/lib/apt/lists/*` removes the lists of available packages and their dependencies. They're automatically regenerated the next time `apt-get update` is run.

The great thing about the official PHP image (compared to starting from Debian and installing PHP manually) is that it has a helper called `docker-php-ext-install`. It can be used to install PHP extensions very easily:

```
RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd zip
```

By the way, `docker-php-ext-install` is a simple shell script included in the official image.

This installs the following extensions:

- `pdo_mysql` is used by Laravel to interact with databases. PDO stands for PHP Data Objects which is another PHP extension that ships with PHP by default.
- `mbstring` stands for "multibyte string" and provides functions for working with multibyte encodings in PHP. A long time ago one character was one exactly byte. But nowadays, when we have UTF8 characters that require more than 1 byte. Hence the name multibyte string.
- `exif` provides functions for reading and manipulating metadata embedded in image files.
- `pcntl` is used for managing processes and signals.
- `bcmath` is a PHP extension that provides arbitrary precision arithmetic functions for working with numbers that are too large or too precise to be represented using the standard floating-point data type.
- `gd` handles in various formats, such as JPEG, PNG, GIF, and BMP. It is required to create PDFs as well.
- And finally `zip` is pretty self-explanatory.

`docker-php-ext-install` can only install PHP core libraries. Usually, the low-level ones. If you need something else you can use PECL:

```
RUN pecl install redis
```

In this example, I'm going to use Redis as a queue so we need to install it.

In a Dockerfile we also have another, kind of unusual but pretty fast way of installing stuff: copying the binary from another Docker image:

```
COPY --from=composer:2.5.8 /usr/bin/composer /usr/bin/composer
```

There's a `--from` option to the `COPY` command in which we can specify from which Docker image we want to copy files. Composer also has an official image. If you run the image

```
docker run --rm -it composer:2.5.8
```

you can find the `composer` executable file in the `/usr/bin` directory. The `COPY --from=composer:2.5.8 /usr/bin/composer /usr/bin/composer` downloads that file from the composer image and copies it into our own image.

The next line copies the source code from the host machine to the container:

```
COPY . .
```

`.` means the current working directory which is `/usr/src` in the image.

And finally, we can copy our PHP and FPM config files. If you remember the project has this structure:

- api
  - Dockerfile
- frontend
- deployment
  - config

The config files are located in `deployment/config` and right now I'm editing the `api/Dockerfile` file. So the `COPY` command looks like this:

```
COPY ../deployment/config/php-fpm/php.ini /usr/local/etc/php/conf.d/php.ini
COPY ../deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf
```

But this would fail miserably if you run a `docker build` from the `api` folder, such as this: `docker build -t api:0.1 .`

The result is this:

```
 => ERROR [stage-0 11/12] COPY ../deployment/config/php-fpm/php.ini /usr/local/etc/php/conf.d/php.ini      0.0s
 => ERROR [stage-0 12/12] COPY ../deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf     0.0s
------
 > [stage-0 10/12] COPY ./api .:
------
------
 > [stage-0 11/12] COPY ../deployment/config/php-fpm/php.ini /usr/local/etc/php/conf.d/php.ini:
------
------
 > [stage-0 12/12] COPY ../deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf:
------
Dockerfile:36
--------------------
  34 |
  35 |     COPY ../deployment/config/php-fpm/php.ini /usr/local/etc/php/conf.d/php.ini
  36 | >>> COPY ../deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf
  37 |
--------------------
ERROR: failed to solve: failed to compute cache key: failed to calculate checksum of ref 5b072298-d242-4c3e-8780-23314c52f675::kfnsvx16xtfgv5ji7e4bnzi2d: "/deployment/config/php-fpm/www.conf": not found
```

The reason is that in a Dockerfile you cannot reference folders outside the current working directory. So `../deployment` does not work.

One solution is to move the Dockerfile up:

- api
- frontend
- deployment
- Dockerfile

But this solution is confusing since frontend will also have its own Dockerfile. And of course, we might have other services with their own Dockerfiles. So I really want to store this file in the `api` folder.

Another solution is to leave the Dockerfile in the `api` folder but when you build it (or use it in docker-compose) you set the context to the root directory. So you don't build it the `api` but the parent directory:

```
(base) ~/code/devops-with-laravel/docker git:(main) (0.041s)
ls
api          deployment frontend

(base) ~/code/devops-with-laravel/docker git:(main)±26
docker build -t api:0.1 . -f ./api/Dockerfile api/
```

There are three arguments for this command:

- `-t api:0.1` sets the image tag.
- `-f ./api/Dockerfile` sets the Dockerfile we want to build the image from.
- `.` is the context. So it's not the `api` but the current (root) folder.

With a setup like this, we can rewrite the `COPY` commands:

```
COPY ./api .

COPY ./deployment/config/php-fpm/php.ini /usr/local/etc/php/conf.d/php.ini
COPY ./deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf
```

`COPY . .` becomes `COPY ./api .` because we are one folder above so the source code is located in `./api`. And the same goes for the deployment folder as well which became `./deployment` instead `../deployment`.

Now the image can be built successfully. However, we don't need to manually build images right now.

The last thing is installing composer packages after the project files have been copied:

```
COPY ./api .

RUN composer install
```

It's an easy but important step. Now the image is entirely self-contained, meaning it has:

- Linux
- PHP
- System dependencies
- PHP extensions
- Project files
- Project dependencies

And this is the final result:

```
FROM php:8.1-fpm

WORKDIR /usr/src

RUN apt-get update && apt-get install -y \
    git \
    curl \
    libpng-dev \
    libonig-dev \
    libxml2-dev \
    libzip-dev \
```

```
    zip \
    unzip \
    supervisor

RUN apt-get clean && rm -rf /var/lib/apt/lists/*

RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd zip

RUN pecl install redis

COPY --from=composer:2.5.8 /usr/bin/composer /usr/bin/composer

COPY ./api .

RUN composer install

COPY ./deployment/config/php-fpm/php.ini /usr/local/etc/php/conf.d/php.ini
COPY ./deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf
```

To summarize:

- It extends the official PHP 8.1 FPM image.
- We install some basic dependencies that are needed by Laravel, the app itself, and the composer packages it uses.
- We install the necessary PHP extensions, Redis, and composer.
- Then we copy files from the host machine into the image.

Before we move on let's build it and run it. The build command is the same as before. Remember, you need to run it from the project root folder (where the `api` folder is located):

```
docker build -t api:0.1 . -f ./api/Dockerfile
```

If you now run the `docker images` command you should see the newly build image:

```
(base) ~/code/devops-with-laravel/docker git:(main) (0.135s)
docker images
REPOSITORY                                TAG             IMAGE ID       CREATED         SIZE
api                                       0.1             fd3df74ded11   7 seconds ago   1.08GB
```

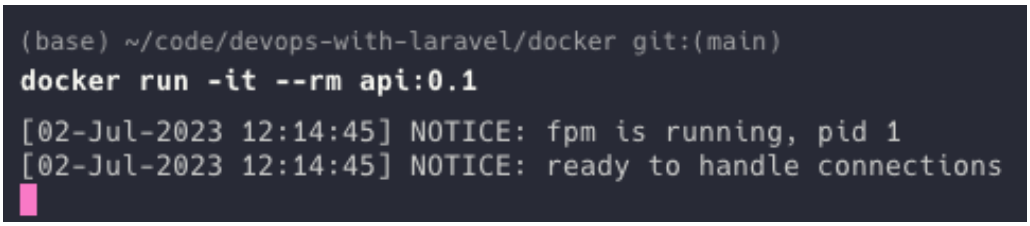
To run the image execute the following command:

```
docker run -it --rm api:0.1
```

There are two important flags here:

- `-it` will start an interactive shell session inside the container. The `i` flag stands for interactive, which keeps STDIN open, and the `t` flag stands for terminal, which allocates a pseudo-TTY. This means that you can interact with the container's shell as if you were using a local terminal.
- `--rm` will automatically remove the container when it exits. Otherwise, Docker would keep the container in a `stopped` status. Removing means more free space on your disk.

After running the command you should see something like this:



As you can see it started PHP-FPM. But why? Our Dockerfile doesn't do anything except copying files from the host.

If you check out the php8.1-fpm image you can see it ends with these two commands:

```
EXPOSE 9000
CMD ["php-fpm"]
```

`CMD` is pretty similar to `RUN`. It runs a command, which is `php-fpm` in this case. However, there's a big difference between `RUN` and `CMD`:

- `RUN` is used to execute commands during the **build process** of an image. This can include installing packages, updating the system, or running any other command that needs to be executed to set up the environment for the container. Just as we did.
- `CMD`, on the other hand, is used to define the **default command** that should be executed when a container is started from the image. This can be a shell script, an executable file, or any other command that is required to run the application inside the container.p

Since the official PHP image contains a `CMD ["php-fpm"]` command our image will inherit this and run php-fpm on startup. Of course, we can override it with another `CMD` in our own Dockerfile but we don't need to right now.

There's another command in the PHP image:

```
EXPOSE 9000
```

It means that the container exposes port 9000 to the outside world. We can verify this by running `docker ps`:

```
(base) ~/code/devops-with-laravel/docker git:(main) (0.357s)
docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS      NAMES
31c75e572b07   api:0.1   "docker-php-entrypoi…"   10 minutes ago   Up 10 minutes   9000/tcp   exciting_driscoll
```

You can see `9000/tcp` is exposed from the container. Right now, we don't need to use it, but later it's going to be very important. This is the port where FPM can be reached by nginx.

`docker ps` also gives us the container ID which we can use to actually go inside the container:

```
docker exec -it 31c75e572b07 bash
```

With `docker exec` you can run commands inside your containers. In this example, I run the `bash` command in interactive mode (`-it`) which essentially gives me a terminal inside the container where I can run commands such as `ls -la`:

```
(base) ~/code/devops-with-laravel/docker git:(main)
docker exec -it 31c75e572b07 bash

root@31c75e572b07:/usr/src# ls -la
total 12068
drwxr-xr-x  1 root root     4096 Jul  2 12:11 .
drwxr-xr-x  1 root root     4096 Jun 12 00:00 ..
-rw-r--r--  1 root root     8196 Apr 30 18:37 .DS_Store
-rw-r--r--  1 root root      258 Mar  8 16:57 .editorconfig
-rw-r--r--  1 root root     1217 Jun 30 08:57 .env
-rw-r--r--  1 root root     1087 May  2 17:58 .env.ci
-rw-r--r--  1 root root     1069 Mar  8 16:57 .env.example
-rw-r--r--  1 root root      186 Mar  8 16:57 .gitattributes
-rw-r--r--  1 root root      273 Jun 30 17:56 .gitignore
-rw-r--r--  1 root root     1038 Apr 10 20:20 .phpunit.result.cache
-rw-r--r--  1 root root      796 Jul  2 12:10 Dockerfile
-rw-r--r--  1 root root     4158 Mar  8 16:57 README.md
drwxr-xr-x 12 root root     4096 Apr  7 11:39 app
-rwxr-xr-x  1 root root     1686 Mar  8 16:57 artisan
drwxr-xr-x  3 root root     4096 Mar  8 16:57 bootstrap
-rw-r--r--  1 root root     2161 Apr 30 18:47 composer.json
-rw-r--r--  1 root root   401409 Apr 30 18:47 composer.lock
drwxr-xr-x  2 root root     4096 Jun 30 09:22 config
drwxr-xr-x  5 root root     4096 Apr  6 17:36 database
-rw-r--r--  1 root root     2288 Jul  1 16:22 docker-compose.yml
drwxr-xr-x  2 root root     4096 Jun 30 17:41 frontend
drwxr-xr-x  3 root root     4096 Apr 30 18:44 lang
-rw-r--r--  1 root root      226 Mar  8 16:57 package.json
-rw-r--r--  1 root root 11793228 Jun 13 23:37 php.tar.xz
-rw-r--r--  1 root root      833 Jun 13 23:37 php.tar.xz.asc
-rw-r--r--  1 root root      111 Apr  7 11:45 phpstan.neon
-rw-r--r--  1 root root     1079 Apr 10 17:43 phpunit.xml
drwxr-xr-x  3 root root     4096 Jul  2 12:11 public
drwxr-xr-x  5 root root     4096 Apr  6 17:36 resources
drwxr-xr-x  2 root root     4096 Apr  6 20:09 routes
drwxrwxrwx  8 root root     4096 Jun 30 17:55 storage
drwxr-xr-x  4 root root     4096 Apr  6 17:36 tests
drwxr-xr-x 55 root root     4096 Jun 30 05:19 vendor
-rw-r--r--  1 root root      263 Mar  8 16:57 vite.config.js
-rwxr-xr-x  1 root root     5227 Jun 30 04:58 wait-for-it.sh
root@31c75e572b07:/usr/src#
```

As you can see, we are in the `/usr/src` directory and the project's files are copied successfully. Here you can run commands such as `composer install` or `php artisan tinker`.

There's only one problem, though:

```
root@31c75e572b07:/usr/src# whoami
root
root@31c75e572b07:/usr/src#
```

We are running the container as root. Which is not a really good idea. It can pose a security risk. When a container runs as root, it has root-level privileges on the host system, which means that it can potentially access and modify any file or process on the host system. This can lead to accidental or intentional damage to the host system.

It can also cause annoying bugs. For example, if the storage directory doesn't have 777 permissions and is owned by root, Laravel is unable to write into the log files, etc.

By default, Docker containers run as the root user, but it is recommended to create a new user inside the container. So let's do that!

```
RUN useradd -G www-data,root -u 1000 -d /home/martin martin
```

This is how you create a new user in Linux:

- `-G www-data,root` adds the user to two groups, `www-data` and `root`. If you search for `www-data` in the fpm Dockerfile you can see that FPM is running as `www-data` and this user is in the `www-data` group. So it's important that our own user is also part of that group.
- `-u 1000` specifies the user ID for the new user. UID is really just an integer number, 1000 is not special at all but it is commonly used as the default UID for the first non-root user created on a system. The UID 0 is reserved for root.
- `-d /home/martin` sets the home directory.
- `martin` is the name of my user.

After that, we need to run this:

```
RUN mkdir -p /home/martin/.composer && \
    chown -R martin:martin /home/martin && \
    chown -R martin:martin /usr/src
```

It creates a folder for composer and then sets the ownerships of the home and `/usr/src` directories to the new user.

And at the end of the Dockerfile, we need to specify that the container should run as `martin`:

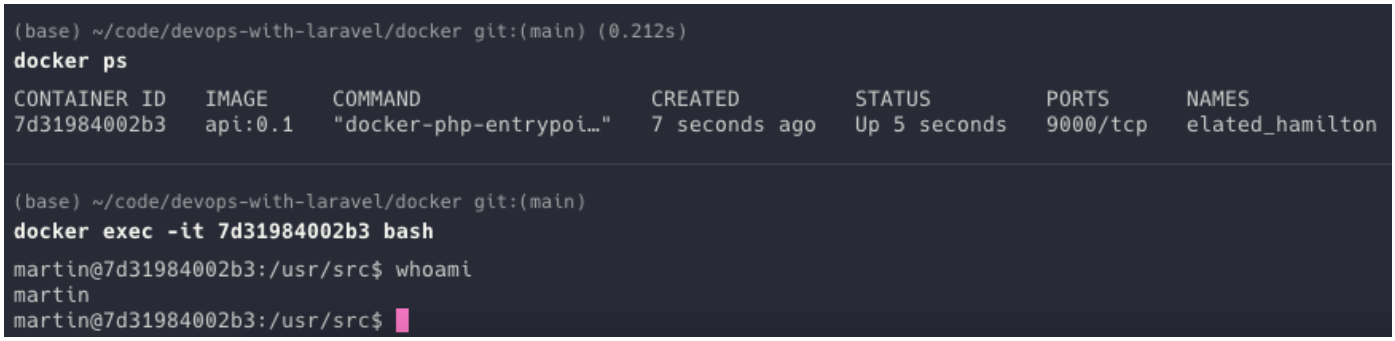
```
USER martin
```

These commands need to run before we copy files into the container, so the new Dockerfile looks like this:

```
FROM php:8.1-fpm

WORKDIR /usr/src

RUN apt-get update && apt-get install -y \
    git \
    curl \
    libpng-dev \
    libonig-dev \
    libxml2-dev \
    libzip-dev \
    zip \
    unzip \
    supervisor

RUN apt-get clean && rm -rf /var/lib/apt/lists/*

RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd zip

RUN pecl install redis

COPY --from=composer:2.5.8 /usr/bin/composer /usr/bin/composer

RUN useradd -G www-data,root -u 1000 -d /home/martin martin

RUN mkdir -p /home/martin/.composer && \
    chown -R martin:martin /home/martin && \
    chown -R martin:martin /usr/src

COPY ./api .

RUN composer install

COPY ./deployment/config/php-fpm/php.ini /usr/local/etc/php/conf.d/php.ini
COPY ./deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf
```

```
USER martin
```

Let's build a new image and then check the output of `whoami` again:



Now the container's running as `martin`.

This should be perfectly fine, however, we can make the whole thing dynamic. So we don't need to hardcore the user `martin` in the Dockerfile but rather we can use build arguments. To do that, we need to add two lines at the beginning of the file:

```
FROM php:8.1-fpm

WORKDIR /usr/src

ARG user
ARG uid
```

These are called build arguments but they really just variables. We can pass anything when building the image (or later, running the image from docker-compose) and we can reference it in the Dockerfile:

```
RUN useradd -G www-data,root -u $uid -d /home/$user $user

RUN mkdir -p /home/$user/.composer && \
    chown -R $user:$user /home/$user && \
    chown -R $user:$user /usr/src

COPY ./api .

RUN composer install

COPY ./deployment/config/php-fpm/php.ini /usr/local/etc/php/conf.d/php.ini
COPY ./deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf
```

```
USER $user
```

If you now run the same `docker build -t api:0.1 . -f ./api/Dockerfile` command it fails with this error message:

The command `useradd` failed with an invalid user ID since we didn't pass it the `docker build`. So this is the correct command:

```
docker build -t api:0.1 -f ./api/Dockerfile --build-arg user=joe --build-arg uid=1000 .
```

After building and running the image, you can see it's running as `joe`:

So this is the final Dockerfile for the Laravel API:

```
FROM php:8.1-fpm
```

```
WORKDIR /usr/src

ARG user
ARG uid

RUN apt-get update && apt-get install -y \
    git \
    curl \
    libpng-dev \
    libonig-dev \
    libxml2-dev \
    libzip-dev \
    zip \
    unzip \
    supervisor

RUN apt-get clean && rm -rf /var/lib/apt/lists/*

RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd zip

RUN pecl install redis

COPY --from=composer:2.5.8 /usr/bin/composer /usr/bin/composer

RUN useradd -G www-data,root -u $uid -d /home/$user $user

RUN mkdir -p /home/$user/.composer && \
    chown -R $user:$user /home/$user && \
    chown -R $user:$user /usr/src

COPY ./api .

RUN composer install

COPY ./deployment/config/php-fpm/php.ini /usr/local/etc/php/conf.d/php.ini
COPY ./deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf
```

```
USER $user
```

After all, it wasn't that complicated.

Dockerizing a Vue app

Now, we can move on to the Vue app. Just as with bare-bone nginx and shell scripts, the frontend is significantly easier than the backend. But there is a big difference between the two. When running the API it doesn't matter if you're in production or in development. The Dockerfile is the same, the PHP version is the same, and the project is the same. In the case of the Vue app, there's a difference between production and development. In prod, you want to build your assets and serve static files. On the other hand, in development you want to run `npm run serve` that watches your file changes.

Let's start with production and we can figure out what to do in development mode.

```
FROM node:14.21.3-alpine
WORKDIR /usr/src
```

Building the project requires nodejs so this is the base image. Once again, I'm using the `/usr/src` as the root of the project.

Next, we need to install npm packages:

```
COPY ./frontend/package.* ./
RUN npm install
```

First, it copies `package.json` and `package-lock.json` into the container then it runs `npm install`. You can see I reference `./frontend/package.*` instead of `./package.*`. In a minute, you'll see why.

After we installed the npm packages we can copy the source code and build it:

```
COPY ./frontend .
RUN npm run build
```

This is what it looks like so far:

```
FROM node:14.21.3-alpine
WORKDIR /usr/src
COPY ./frontend/package.* ./
RUN npm install
COPY ./frontend .
RUN npm run build
```

Now we have the `/usr/src/dist` folder with all the static files in it. The last step is to serve it via nginx. So we need to install and configure nginx manually.

Actually not. We don't need to install nginx. There's a much better solution in Docker called multi-stage builds.

Image something like this:

```
FROM node:14.21.3-alpine AS build
#  ...
RUN npm run build

FROM nginx:1.25.1-alpine AS prod
COPY --from=base /usr/src/dist /usr/share/nginx/html
```

A few important things going on here. We have multiple `FROM` expressions each with an "alias" using the `AS` expression. They are called stages. In a multi-stage Docker build, you define multiple build stages, just like `build` and `prod`, each with its own set of instructions. Each stage can use a **different base image** and can **copy files** from previous stages.

The main advantage is that we can use different base images. This is why we don't need to install and configure nginx. We just use the official nginx image with a fresh installation in it and use it!

The first stage (`build`) ends with an `npm run build` command that generates a lot of files inside the `usr/src/dist` folder. The second stage (`prod`) can just grab these files and paste them into a folder that is configured in an nginx config (which we'll see in a minute).

So after we have the build all we need to do is to start nginx:

```
FROM nginx:1.25.1-alpine AS prod
COPY --from=base /usr/src/dist /usr/share/nginx/html
COPY ./deployment/config/nginx-frontend.conf /etc/nginx/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

On the last line, we override the default command of the image with `nginx -g daemon off`. By default, when you start nginx using the `nginx` command, it runs as a background process, detaching from the shell.

However, when running a docker container it needs a long-running process. Otherwise, it will exit. `daemon off` means that nginx runs in the foreground. It's a long-running process that "keeps the container alive." The `-g` option allows you to pass a configuration directive to nginx.

So, when you run `nginx -g daemon off`, it starts up and runs in the foreground, outputting log messages and other information to your terminal session. And more importantly, keeping the container in a running state.

As you can see, we copy the `nginx-frontend.conf` file from the deployment folder. Once again, this is the reason we're referencing the source as `./frontend` instead of `.` The same as before with the backend. Of course, you can move these config files directly into the `api` and `frontend` folders but I like them in a separate directory with all the shell script and other deployment-related stuff.

Here's the nginx config:

```
worker_processes auto;

events {
  worker_connections 1024;
}

http {
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;
  include /etc/nginx/mime.types;

  gzip on;
  gzip_comp_level 4;
  gzip_types text/css application/javascript image/jpeg image/png;

  server {
    server_name posts.today www.posts.today;
    listen 80;
    root /usr/share/nginx/html;
    index index.html;

    location / {
      try_files $uri $uri/ /index.html;
      gzip_static on;
    }

    location ~* \.(css|js|png|jpg|gif|ico)$ {
      access_log off;
      add_header Cache-Control public;
      add_header Vary Accept-Encoding;
      expires 1d;
    }
```

```
  }
}
```

You've already seen configurations such as this so there's nothing new here. There's no SSL yet. Since the official Vue documentation uses the `/usr/share/nginx/html` folder as the root, I just stick to the same folder in my Dockerfiles.

The last thing we need to do is the development image. If you think about it, all we need is this:

```
FROM node:14.21.3-alpine AS base
WORKDIR /usr/src
COPY ./frontend/package.* ./
RUN npm install
COPY ./frontend .
CMD ["npm", "run", "serve"]
```

So it's just the `base` stage but instead of `npm run build` we need `npm run serve`. To do so we can extract a brand new stage:

```
FROM node:14.21.3-alpine AS base
WORKDIR /usr/src
COPY ./frontend/package.* ./
RUN npm install
COPY ./frontend .

FROM base AS dev
EXPOSE 8080
CMD ["npm", "run", "serve"]

FROM base AS build
RUN npm run build
```

- `base` installs dependencies and copies files
- `dev` starts a development server.
- `build` runs `npm run build`

The `dev` stage also exposes the port `8080`. This is the default port used by `vue-cli-service serve` (which is the underlying command being run by `npm run serve`). Later, in the docker-compose chapter, we're going to use this port.

And after all that we can include the `prod` stage that copies the files from the stage `build`:

```
FROM nginx:1.25.1-alpine AS prod
COPY --from=build /usr/src/dist /usr/share/nginx/html
COPY ./deployment/config/nginx-frontend.conf /etc/nginx/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

The whole Dockerfile looks like this:

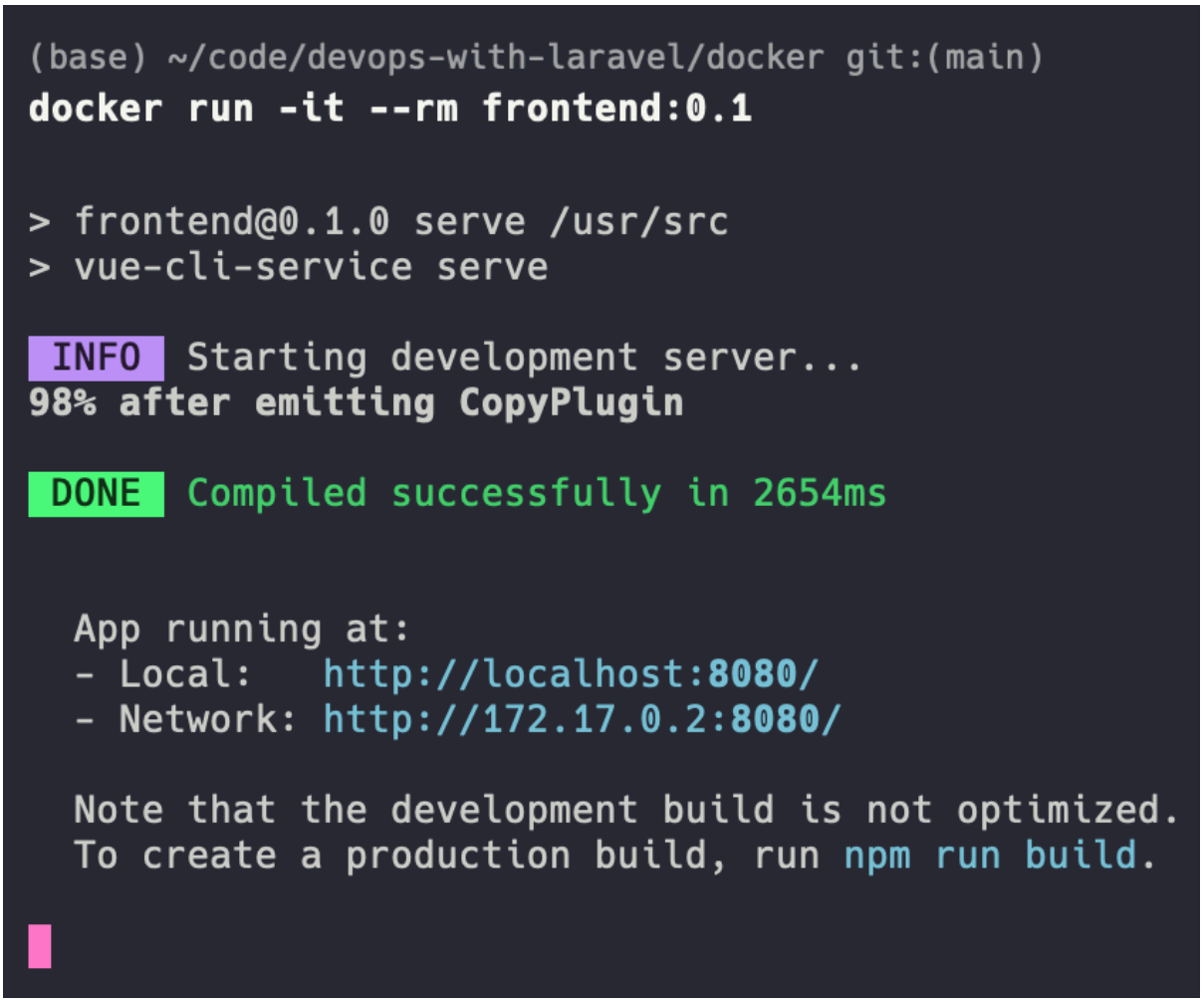
```
FROM node:14.21.3-alpine AS base
WORKDIR /usr/src
COPY ./frontend/package.* ./
RUN npm install
COPY ./frontend .

FROM base AS dev
EXPOSE 8080
CMD ["npm", "run", "serve"]

FROM base AS build
RUN npm run build

FROM nginx:1.25.1-alpine AS prod
COPY --from=build /usr/src/dist /usr/share/nginx/html
COPY ./deployment/config/nginx-frontend.conf /etc/nginx/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

If you now want to build a dev image you do so by running this:

```
docker build -t frontend:0.1 -f ./frontend/Dockerfile --target=dev .
```

There's a `target` option to the `build` command that targets a specific stage from the Dockerfile. In this case, the build ends at this line: `CMD ["npm", "run", "serve"]`

If you target the `dev` stage and run the image you'll get this result:



It starts a development server. However, if you target the `prod` stage:

```
docker build -t frontend:0.1 -f ./frontend/Dockerfile --target=prod .
```

You'll get this output:

```
(base) ~/code/devops-with-laravel/docker git:(main)
docker run -it --rm frontend:0.1-prod
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
```

It's a running nginx instance.

Dockerizing a scheduler and a worker

I have good news. The scheduler needs the same Dockerfile as the API. It's the same Laravel application but instead of HTTP requests it cares about console commands. The worker also needs the same Dockerfile. So, we don't need to do much.

There's only one difference.

- The worker runs this command as its entry point: `nice -n 10 php /usr/src/artisan queue:work --tries=3 --verbose --timeout=30 --sleep=3 --rest=1 --max-jobs=1000 --max-time=3600`
- And the scheduler runs this: `nice -n 10 sleep 60 && php /usr/src/artisan schedule:run --verbose --no-interaction`. It executes `schedule:run` every 60 seconds.

We can handle this in (at least) two ways:

- Adding two new stages to the existing Dockerfile where we override the default php-fpm command with those commands.
- Not touching the Dockerfile and adding these commands to the `docker-compose.yml` file (which we don't have yet).

Both of them are good options, but I prefer the first one. In my opinion, it's always a good idea to be independent of the orchestration platform and be as self-contained as possible. So let's do the first one.

We need two more stages at the end of the API Dockerfile:

```
FROM api AS worker
CMD ["/bin/sh", "-c", "nice -n 10 php /usr/src/artisan queue:work --tries=3 --verbose --timeout=30 --sleep=3 --rest=1 --max-jobs=1000 --max-time=3600"]

FROM api AS scheduler
CMD ["/bin/sh", "-c", "nice -n 10 sleep 60 && php /usr/src/artisan schedule:run --verbose --no-interaction"]
```

The `-c` option tells the shell to read the command that follows as a string and execute it as a command.

And of course, I added the alias at the beginning of the Dockerfile as well:

```
FROM php:8.1-fpm as api
```

When you have no target (wanna build the API itself) Docker builds the base stage and inherits the default command (`CMD`) from the base `php:8.1-fpm` image.

But of course, if you target the `worker` stage you'll get an image that runs `queue:work`:

```
docker build -t api:0.1 . -f ./api/Dockerfile --build-arg user=joe --build-arg uid=1000 --target=worker
```

And that's it! We just dockerized everything in the application. I mean, we still have a few components, such as MySQL, Redis, and an nginx for the API. But we don't need to "dockerize" these. These are only external services. We'll use official images in a docker-compose config.

docker-compose

docker-compose is an amazing container orchestration tool that you can use in production and on your local machine as well. Container orchestration is really just a fancy word to say: you can run and manage multiple containers with it.

Right now, if you'd like to run the project you'd need to run all the containers one by one which is not quite convenient. docker-compose solves that problem by having a simple `docker-compose.yml` configuration file. The file is located in the root folder at the same level as the `api` or the `frontend` folders:

- api
- frontend
- deployment
- docker-compose.yml

Right now, the goal is to create a compose file that works for local development. Production comes later (however, it's not that different).

Frontend

Let's start with the frontend:

```
version: "3.8"
services:
  frontend:
    build:
      context: .
      dockerfile: ./frontend/Dockerfile
      target: dev
    ports:
      - "3000:8080"
```

That's the most basic configuration:

- `version` defined the compose file version you want to use. 3.8 is the newest right now.
- `services` defines a set of services your project consists of. In our case, we're gonna have services such as `frontend`, `api`, `scheduler`, `worker`, etc.
- `frontend` is one of the services.
- `ports` is an important part. If you remember the Dockerfile has an `EXPOSE 8080` instruction. This means that the container exposes port 8080. In the docker-compose file can bind the port from the container to the host machine. So `3000:8080` means that `localhost:3000` should be mapped to `container:8080` where the container is a made-up name referring to the frontend container.
- `build`: just like with the `docker build` command docker-compose will build an image when we run it.

Remember when we used the `docker build` command? It looked something like this:

```
docker build -t frontend:0.1 -f ./frontend/Dockerfile .
```

`-f ./frontend/Dockerfile` is the same as `dockerfile: ./frontend/Dockerfile` and the `.` is the equivalent of `context: .`

However, the full `build` command looked like this:

```
docker build -t frontend:0.1 -f ./frontend/Dockerfile --target=dev .
```

Right now, docker-compose doesn't provide the target so let's do these now:

```
version: "3.8"
services:
  frontend:
    build:
      context: .
      dockerfile: ./frontend/Dockerfile
      target: dev
    ports:
      - "3000:8080"
```

With `target` we can target specific build stages. In this case, I'd like to run a dev container.

The purpose of the frontend dev container is to run a development server with hot reload. So whenever you change a file Vue should restart the server and reload the page. To do this we need some real-time connection between the files on your local machine and the files inside the container. We copy files in the Dockerfile but that's not real-time, of course. That happens at build time and that's it.

To solve this problem we can use docker volumes:

```
version: "3.8"
services:
  frontend:
    build:
      context: .
      dockerfile: ./frontend/Dockerfile
      target: dev
    ports:
      - "3000:8080"
    volumes:
      - ./frontend:/usr/src
```

Basically, we can bind a folder from the host machine to a folder in the container. This means whenever you change a file in the `frontend` folder (or its subfolders) it gets copied into the container.

And of course, we can provide environment variables to the container as well:

```
version: "3.8"
services:
  frontend:
    build:
      context: .
      dockerfile: ./frontend/Dockerfile
      target: dev
    ports:
      - "3000:8080"
    volumes:
      - ./frontend:/usr/src
    environment:
      - NODE_ENV=local
```

That's all the config we need for the frontend. Now let's take care about the api.

API

This is what the `api` services looks like in `docker-compose.yml`:

```
api:
  build:
    args:
      user: martin
      uid: 1000
    context: .
    dockerfile: ./api/Dockerfile
    target: api
  restart: unless-stopped
  volumes:
    - ./api/app:/usr/src/app
    - ./api/config:/usr/src/config
    - ./api/database:/usr/src/database
    - ./api/routes:/usr/src/routes
    - ./api/storage:/usr/src/storage
    - ./api/tests:/usr/src/tests
    - ./api/.env:/usr/src/.env
```

Only one new keyword which is `restart`. As I said earlier, docker-composer compose is a container orchestrator so it can start, restart and manage containers. `restart` makes it possible to automatically restart containers if they stop. It has the following values:

- `no`: Containers should never be automatically restarted, no matter what happens.
- `always`: Containers should always be automatically restarted, no matter what happens.
- `on-failure`: Containers should be automatically restarted if they fail, but not if they are stopped manually.
- `unless-stopped`: Containers should always be automatically restarted, unless they are stopped manually.

`always` is a bit tricky because it'll restart containers that you manually stopped on purpose.

The difference between `on-failure` and `unless-stopped` is this: `on-failure` only restarts containers if they fail so if they exit with a non-0 status code. `unless-stopped` will restart the container even if it stopped with 0.

In the case of the API, it's not a big difference since php-fpm is a long-running process and it won't stop with 0. However, when we run a scheduler container it'll stop immediately with a status of 0 (because it runs `schedule:work` and then it stops). So in this case, `unless-stopped` is the best option. In general, most of the time I use `unless-stopped`.

The next weird thing is this:

```
volumes:
  - ./api/app:/usr/src/app
  - ./api/config:/usr/src/config
  - ./api/database:/usr/src/database
  - ./api/routes:/usr/src/routes
  - ./api/storage:/usr/src/storage
  - ./api/tests:/usr/src/tests
  - ./api/.env:/usr/src/.env
```

Why `./api:/usr/src` isn't enough? If you check out these folders there's an important one that is missing. It's `vendor`.

So in the Dockerfile, we installed composer packages, right? If we mount `./api` to `/usr/src` Docker creates a shared folder on your machine. It builds the image, so it has a `vendor` folder. Then it copies the files from `./api` to the shared folder. At this point, there's no `vendor` folder on your local machine. You just pulled the repo and now you want to start the project.

It's important to note that existing files or directories in the container at the specified mount point will not be copied to the host machine. Only files that are created or modified after the volume is mounted will be shared between the host and container. So when the project is running Docker will delete the `vendor` folder from the container! To be honest, I'm not sure why is that but you can try it. Just delete the `vendor` folder on your host, change the volume to `./api:/usr/src`, and run `docker-compose up`.

So there are two solutions to that problem:

- The one I just showed you: mounting every folder but `vendor`
- Running `composer-install` in `docker-composer.yml`

The second one is also a valid option, and it looks like this:

```
api:
  build: ...
  command: sh -c "composer install && php-fpm"
  volumes:
    - ./api:/usr/src
```

We run `composer install` every time the container is started. I, personally don't like this option because it's not part of how the containers should be orchestrated, therefore it shouldn't be in `docker-compose.yml`. And it's not just about separation of concern but this solution causes real problems when it comes to scaling (later on that).

That's the reason I use multiple volumes for the different folders.

If you wonder why only those folders and files are included: you only need volumes for the files you're editing while writing code. You certainly wilt not edit `artisan` or the contents of the `bootstrap` folder so you don't need to mount those.

However, this solution has a drawback: you won't have a `vendor` folder on your host machine. Meaning, there's no autocompletion in your idea. We can solve this problem by copying the folder from the container:

```
#!/bin/sh

ID=$(docker ps --filter "name=api" --format "{{.ID}}")

docker cp $ID:/usr/src/vendor ./api
```

You can find this script as `copy-vendor.sh`. You only need to run it when you initialize a project or when you install new packages. It's a good trade-off in my opinion.

This config doesn't work just yet, because we need MySQL and Redis.

Databases

We need Redis for our queue and jobs so let's add another service:

```
redis:
  image: redis:7.0.11-alpine
  restart: unless-stopped
  volumes:
    - ./redisdata:/data
  ports:
    - "63790:6379"
```

That's it. For Redis, we don't have a custom Dockerfile, but we want to use the official image instead.

Remember, this compose file is for local development, so we can expose a port to use some GUI if you need it. In a containerized environment we need to use volumes to preserve the state of databases. In this example, I use a folder named `redisdata` to store all the data inside Redis. This way, if I stop the stack the `redisdata` folder will store the whole database and the next time I run it the database has its previous state. Without this volume, Redis would start in an empty state each time you run the project.

Next up, we need MySQL:

```
mysql:
  image: mysql:8.0.33
  restart: unless-stopped
  volumes:
    - ./deployment/config/mysql/create_database.sql:/docker-entrypoint-
initdb.d/create_database.sql
    - ./deployment/config/mysql/mysql.cnf:/etc/mysql/conf.d/mysql.cnf
    - ./mysqldata:/var/lib/mysql
  ports:
    - "33060:3306"
  environment:
    MYSQL_ROOT_PASSWORD: root
```

It's pretty similar to Redis, except for a few things. We mount the `create_database.sql` file into the `docker-entrypoint-initdb.d` folder. It's a great feature of the MySQL image. Any SQL file you copy into this folder gets executed at startup. It's a great way to create the database on the first run. You can see the source here.

The `MYSQL_ROOT_PASSWORD` environment variable sets the password for the root user. We use the same password in the `.env` file to connect to the database. Once again, this compose config is for development so it's not a security issue to expose the password.

Migrations

Now that we have MySQL there's a tricky question: who and when is going to run the migrations?

It needs to be the API container before it starts. Fortunately, it's easy to solve this with docker-compose:

```
api:
  build:
    args:
      user: martin
      uid: 1000
    context: .
    dockerfile: ./api/Dockerfile
  command: sh -c "php /usr/src/artisan migrate --force && php-fpm"
```

We override the image's default command (which is `php-fpm`) with a shell script that runs the migrations and then starts `php-fpm`. It's a pretty simple solution, however, it introduces some problems:

- **Separation of concerns**. It's pretty similar to running `composer install` in the docker-compose file. As I said earlier, I don't think it's the responsibility of the orchestrator platform.
- **Portability issues**. It works with docker-compose for sure. But let's say we want to migrate the whole application to DigitalOcean App Platform. Or GCP App Engine. Or Kubernetes. Would this solution still be working? I don't know. Probably not.
- **Scalability issues**. What happens if we want to run the API container in 8 instances (we can do this with docker-compose by the way)? Do all eight processes want to migrate the database concurrently? Well, yes. And that's a real problem with real consequences. By the way, we'd run into this same problem with `composer install`.

I know, we're writing this compose file for local development, but we would still run into this problem if it was for production. And it's always a good idea to have our development and production stack as close to each other as possible.

Of course, when we have only one instance of the container it's not a big deal, but later I want to scale these services so we have to take care of this problem anyway.

To solve this problem in a "cloud native", production-ready way, we actually need to introduce a separate container for migrating the database. I know it sounds weird, but:

- It "obeys" at least two factors from 12factors: dev/prod parity and admin processes. Later, we're gonna talk about these factors in detail.
- This is the standard way of running migrations in the cloud world as well. For example, in DigitalOcean App Platform.

- This is an example of the sidecar container pattern. In a nutshell, you have the main container, such as the `api` in this case, and then you have a supporting container that runs some other, smaller tasks. These tasks can mean lots of different things such as collecting/formatting logs, collecting some metrics, monitoring, running migrations, etc. Here's a good article that shows some other examples.

So, let's create a migration container:

```
migrate:
  build:
    args:
      user: martin
      uid: 1000
    context: .
    dockerfile: ./api/Dockerfile
  command: sh -c "php /usr/src/artisan migrate --force"
  restart: no
```

That's it. We are using the same Dockerfile to build an image but the command is simply: `php /usr/src/artisan migrate --force` This is a one-off process that exits immediately. If we use the `restart: no` it means that the container runs the migrations and then exits. Which is exactly what we want.

There are only two things I didn't show you yet. In a docker-compose config, we can define dependencies between services. For example, we can configure things such as the API needs MySQL and Redis to function properly. In this case, docker-compose will start MySQL and Redis first, and then the API.

In this case, the migration container clearly needs the MySQL container before it can run:

```
migrate:
  build: ...
  command: sh -c "php /usr/src/artisan migrate --force"
  restart: no
  depends_on:
    - mysql
```

The `depends_on` expression can be used to list services that the current service depends on.

But unfortunately, waiting for MySQL using `depends_on` doesn't guarantee that MySQL is healthy and is responding. docker-compose doesn't know anything about the internals of MySQL. It doesn't know when it is considered to be ready to use. `depends_on` only means that the MySQL container started and now it's running. But in reality, it might not be responsive for X seconds after it starts. If it's still in the "startup" phase and the migration container starts running it will fail.

Fortunately, there's a pretty simple solution to that problem. There's an excellent script called `wait-for-it.sh`. You can find the source here. It's a single sh script that can be downloaded and used from `docker-compose.yml`. If you check out the sample project you can find it inside the `api` folder. Here's how to use it:

```
./wait-for-it.sh mysql:3306 -t 30
```

It tries to send a TCP request to the mysql container on port 3306. It sends a request every second for 30 seconds controlled by the `-t` option. Using it with docker-compose is pretty easy:

```
migrate:
  build: ...
  command: sh -c "./wait-for-it.sh mysql:3306 -t 30 && php /usr/src/artisan migrate --force"
  restart: no
  depends_on:
    - mysql
```

Now that we know about `depends_on` and `wait_for_it` we can use them in the `api` service as well:

```
api:
  build: ...
  command: sh -c "./wait-for-it.sh mysql:3306 -t 30 && ./wait-for-it.sh redis:6379 -t 30 && php-fpm"
  restart: unless-stopped
  volumes: ...
  depends_on:
    - migrate
    - mysql
    - redis
```

It's important that the `api` depends on the `migrate` container!

nginx

Now that the database and the API is ready, it's time to add a web server:

```
nginx:
  image: nginx:1.25.1-alpine
  restart: unless-stopped
  volumes:
    - ./api:/usr/src
    - ./deployment/config/nginx.conf:/etc/nginx/nginx.conf
  ports:
    - "8000:80"
  depends_on:
    - api
```

We need to mount the project's source to the container since nginx will forward requests to `public/index.php`. Another solution would be to write a Dockerfile where you copy the files into the container and then the service would look something like this:

```
nginx:
  build:
    context: .
    dockerfile: ./api/Dockerfile.nginx
  restart: unless-stopped
  ports:
    - "8000:80"
  depends_on:
    - api
```

But usually, people just use the official image and then mount the source code.

One important change in the configuration. If you remember from the first part, we wrote a config such as this:

```
location ~\.php {
    try_files $uri =404;
    include /etc/nginx/fastcgi.conf;
    fastcgi_pass unix:/run/php/php8.1-fpm.sock; # This is the important line
    fastcgi_index index.php;
    fastcgi_param PATH_INFO $fastcgi_path_info;
}
```

We communicated with FPM via a Unix socket. But with docker and docker-compose we want to send the request to a completely different container. The `api` container. If you remember the php-fpm image exposes port 9000 by default. That's the TCP where FPM accepts requests. So let's change the config:

```
location ~\.php {
    try_files $uri =404;
    include /etc/nginx/fastcgi.conf;
    fastcgi_pass api:9000; # This is the important line
    fastcgi_index index.php;
    fastcgi_param PATH_INFO $fastcgi_path_info;
}
```

When running a docker-compose stack it creates an internal network. Each service is basically a domain name. When they expose a port we can reach it by `host:port`. This is exactly what happens here. The nginx container sends PHP requests to http://api:9000 which is the `api` container. In the PHP-FPM chapter, I discussed that FPM can accept requests via Unix sockets or TCP connections. Now we're using TCP connections.

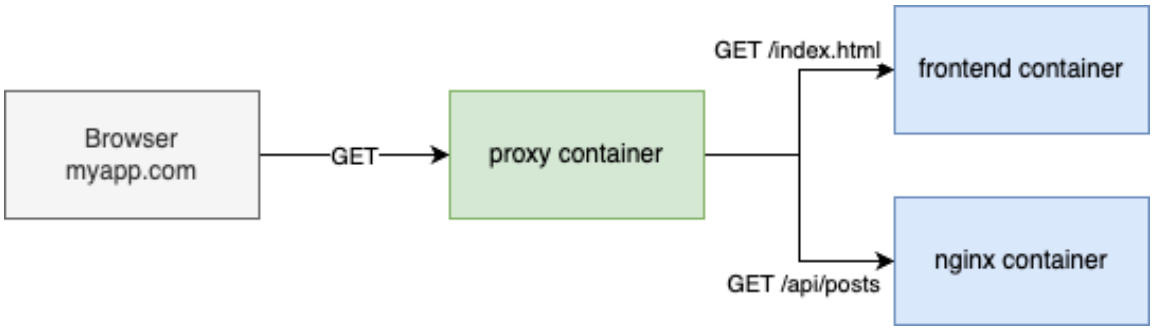
Reverse proxy

Right now, we have a problem. Frontend exposes port 80 (or 443 if you run Certbot and add a certificate) and nginx exposes 8000 for the API. This means that the browser needs to send requests to

```
http://myapp.com:8000/api/posts
```

which is not secure at all. This is the same problem we faced in the first chapter without Docker. But using Docker and docker-compose we can easily add a reverse proxy.

This is what it looks like:



The reverse proxy is the only container that exposes any port to the outside world. It accepts all incoming traffic and routes them based on some criteria:

- Requests prefixed with `/api` go to the `nginx` container
- Requests to static files or to the `/` root URI go to the `frontend` container

`nginx` and `frontend` do not bind any port to the host machine and `proxy` listens on 80 (443).

This is called a reverse proxy. And the configuration is quite easy:

```
worker_processes auto;

events {
  worker_connections 1024;
}

http {
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;
  include /etc/nginx/mime.types;

  gzip on;
  gzip_comp_level 4;
```

```
  gzip_types text/css application/javascript image/jpeg image/png;

  server {
    listen 80;
    server_name posts.today;

    location / {
        proxy_pass http://frontend;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /api {
        proxy_pass http://nginx;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
  }
}
```

The `/api` location catches every request such as `/api/posts` or `/api/posts/1/comments` and the other location catches everything else. The `nginx` container exposes port 80 by default and in a minute we'll change the frontend to expose it as well even in development (which is 8080 by default). As I earlier said, in the same docker-compose stack, containers can access each other by using the container as a hostname such as `http://nginx`.

The next step is to expose 80 from the frontend container:

```
FROM base AS dev
EXPOSE 80
CMD ["npm", "run", "serve"]
```

In `package.json` I added a new flag to `vue-cli-service`:

```
"scripts": {
  "serve": "vue-cli-service serve --port 80"
}
```

This way both the `frontend` container exposes port 80 both in development and production. No need for `if` statements and clunky solutions.

Now that we have a proxy and every incoming request goes through it, we don't need access and error logs in the other nginx configs.

`nginx-frontend.conf`:

```
http {
  access_log off;
  error_log off;
}
```

`nginx.conf`:

```
http {
  access_log off;
  error_log off;
}
```

`proxy.conf`:

```
http {
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;
}
```

It will log every incoming request just as before.

The compose service is quite simple

```
proxy:
  build:
    context: .
    dockerfile: ./Dockerfile.proxy
  restart: unless-stopped
  ports:
    - "3000:80"
  volumes:
    - ./deployment/config/proxy.conf:/etc/nginx/nginx.conf
  depends_on:
    - frontend
    - nginx
```

In the development stack, it listens on port 3000. Of course, in the `docker-compose.prod.yml` stack it listens on 80 (443). And now the `frontend` and `nginx` containers do not expose any ports at all.

With this proxy, we don't need to use build arguments anymore.

Scheduler and worker

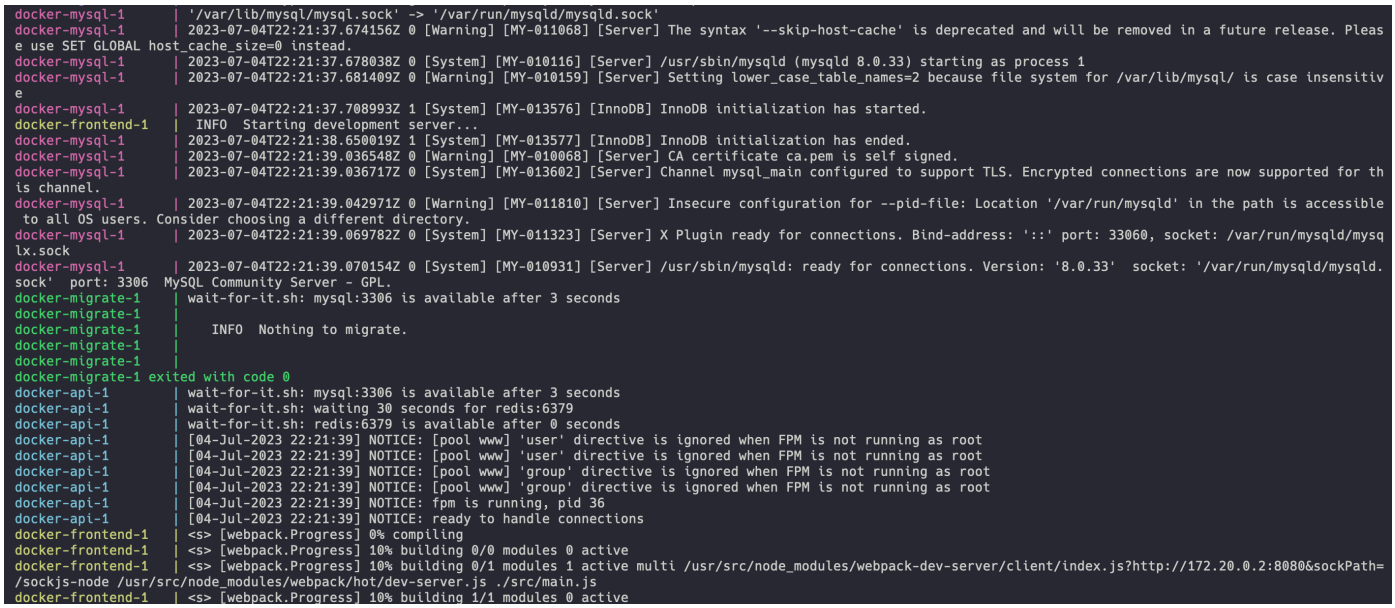
And finally the last two services:

```
scheduler:
  build:
    args:
        user: martin
        uid: 1000
    context: .
    dockerfile: ./api/Dockerfile
    target: scheduler
  restart: unless-stopped
  volumes:
    - ./api/app:/usr/src/app
    - ./api/config:/usr/src/config
    - ./api/database:/usr/src/database
    - ./api/routes:/usr/src/routes
    - ./api/storage:/usr/src/storage
    - ./api/tests:/usr/src/tests
    - ./api/.env:/usr/src/.env
  depends_on:
    - migrate
    - mysql
    - redis

  worker:
    build:
      args:
        user: martin
        uid: 1000
      context: .
      dockerfile: ./api/Dockerfile
      target: worker
    restart: unless-stopped
    volumes:
      - ./api/app:/usr/src/app
```

```
      - ./api/config:/usr/src/config
      - ./api/database:/usr/src/database
      - ./api/routes:/usr/src/routes
      - ./api/storage:/usr/src/storage
      - ./api/tests:/usr/src/tests
      - ./api/.env:/usr/src/.env
  depends_on:
    - migrate
    - mysql
    - redis
```

They are basically the same as the `api` service but they target specific build stages.

And docker-compose is ready to run. Just execute `docker-compose up` and you should see something like that:



In this image you can see the order:

- MySQL starts
- Then migrations are executed
- Then the API is up

And the scheduler runs every 60 seconds and then exits with status code 0 so it gets restarted and runes again after 60 seconds:

```
docker-scheduler-1  |
docker-scheduler-1  |
docker-scheduler-1  |   2023-07-04 22:41:50 Running [App\Jobs\PublishPostsJob]
docker-scheduler-1  | ..... 10ms DONE
docker-scheduler-1  |
docker-scheduler-1  |
docker-scheduler-1 exited with code 0
docker-worker-1     |   2023-07-04 22:41:53 App\Jobs\PublishPostsJob gSxCjYHiJkQyJBGQ05VMaTtwhVUN4U6W  RUNNING
docker-worker-1     |   2023-07-04 22:41:53 App\Jobs\PublishPostsJob gSxCjYHiJkQyJBGQ05VMaTtwhVUN4U6W  4.46ms DONE
docker-scheduler-1  |
docker-scheduler-1  |
docker-scheduler-1  |   2023-07-04 22:42:51 Running [App\Jobs\PublishPostsJob]
docker-scheduler-1  | ..... 16ms DONE
docker-scheduler-1  |
docker-scheduler-1  |
docker-scheduler-1 exited with code 0
docker-worker-1     |   2023-07-04 22:42:52 App\Jobs\PublishPostsJob Y4Hd54Suze1kC1dmIQaHJOjJf1QUYlvf  RUNNING
docker-worker-1     |   2023-07-04 22:42:52 App\Jobs\PublishPostsJob Y4Hd54Suze1kC1dmIQaHJOjJf1QUYlvf  8.95ms
docker-worker-1     |  DONE
```

Before we move on here are a few important `docker-compose` commands.

`docker-compose exec api bash` is the most commonly used by me. You can execute commands in a given container. In this example, I'm starting a new terminal in the `api` container so I can run any command. The other one I often use is `docker-compose exec api php artisan tinker` which gives me a tinker session in the container:

```
(base) ~/code/devops-with-laravel/docker git:(main)
docker-compose exec api php artisan tinker

Psy Shell v0.11.15 (PHP 8.1.20 — cli) by Justin Hileman
> User::count()
[!] Aliasing 'User' to 'App\Models\User' for this Tinker session.
= 1

>
```

`docker-compose ps` gives you a list of the containers in the current docker-compose configuration:

```
(base) ~/code/devops-with-laravel/docker git:(main) (0.283s)
docker-compose ps
NAME                IMAGE               COMMAND                  SERVICE             CREATED             STATUS              PORTS
docker-api-1        docker-api          "docker-php-entrypoi…"   api                 4 minutes ago       Up 4 minutes        9000/tcp
docker-frontend-1   docker-frontend     "docker-entrypoint.s…"   frontend            4 minutes ago       Up 4 minutes        0.0.0.0:3000->8080/tcp
docker-mysql-1      mysql:8.0.33        "docker-entrypoint.s…"   mysql               4 minutes ago       Up 4 minutes        33060/tcp, 0.0.0.0:33060->3306/tcp
docker-nginx-1      nginx:1.25.1-alpine "/docker-entrypoint.…"   nginx               4 minutes ago       Up 4 minutes        0.0.0.0:8000->80/tcp
docker-redis-1      redis:7.0.11-alpine "docker-entrypoint.s…"   redis               4 minutes ago       Up 4 minutes        0.0.0.0:63790->6379/tcp
docker-scheduler-1  docker-scheduler    "docker-php-entrypoi…"   scheduler           4 minutes ago       Up 30 seconds       9000/tcp
docker-worker-1     docker-worker       "docker-php-entrypoi…"   worker              4 minutes ago       Up 4 minutes        9000/tcp
```

`docker-compose restart <service>` restarts the given service. I often use this command to restart the worker.

`docker-compose up -d` starts the application in detached mode so it runs in the background. We're going to use it in production.

`docker-compose up --build` rebuilds the images and then starts the whole project. It's useful when you're modifying the Dockerfiles and test if it works.

`docker-compose logs -f --tail=500 <service>` shows the last 500 lines of logs of a given service. If you don't specify the service it shows every container.

`docker-compose down` stops and removes containers.

supervisor

The last we need to take care of is supervisor. Right now, the worker simply runs the `queue:work` command which is fine for development but it's not great in production. And as I said earlier, we want the development and production environments as close to each other as possible.

First, we need a few changes in the `supervisord.conf`. When you're running containers it's important to redirect every log to `stdout` and `stderr`:

```
[supervisord]
logfile=/dev/null
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0
```

We can do the same thing for every program in the config:

```
[program:default-worker]
command=nice -n 10 php /usr/src/artisan queue:work --
queue=default,notification --tries=3 --verbose --timeout=30 --sleep=3 --max-
jobs=1000 --max-time=3600
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0

[program:notifications-worker]
command=nice -n 10 php /usr/src//artisan queue:work --
queue=notifications,default --tries=3 --verbose --timeout=30 --sleep=3 --max-
jobs=1000 --max-time=3600
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0
```

Now supervisor logs everything to `stdout` and `stderr` which can be seen in your terminal if you're running `docker-compose` in attached mode (without the `-d` option). Or you can see them with `docker-compose logs -f worker` as well:

```
docker-worker-1  | 2023-07-04 23:20:11,785 INFO supervisord started with pid 7
docker-worker-1  | 2023-07-04 23:20:12,791 INFO spawned: 'default-worker_00' with pid 8
docker-worker-1  | 2023-07-04 23:20:12,793 INFO spawned: 'default-worker_01' with pid 9
docker-worker-1  | 2023-07-04 23:20:12,794 INFO spawned: 'notifications-worker_00' with pid 10
docker-worker-1  | 2023-07-04 23:20:12,795 INFO spawned: 'notifications-worker_01' with pid 11
docker-worker-1  | 2023-07-04 23:20:13,797 INFO success: default-worker_00 entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
docker-worker-1  | 2023-07-04 23:20:13,797 INFO success: default-worker_01 entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
docker-worker-1  | 2023-07-04 23:20:13,797 INFO success: notifications-worker_00 entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
docker-worker-1  | 2023-07-04 23:20:13,797 INFO success: notifications-worker_01 entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
docker-worker-1  |   2023-07-04 23:21:13 App\Jobs\PublishPostsJob JicD1SVWG9XeDAaJk0jXZFC5zEVt74ox  RUNNING
docker-worker-1  |   2023-07-04 23:21:13 App\Jobs\PublishPostsJob JicD1SVWG9XeDAaJk0jXZFC5zEVt74ox  68.77ms DONE
```

But the most important change in supervisor config is this line:

```
[supervisord]
nodaemon=true
```

Remember the frontend Dockerfile? The last line is `nginx -g daemon off`. The same thing happens here. We want a long-running process in the container so we want supervisor to run in the foreground as a "main process". Otherwise, the container would exit immediately. Try it out for yourself. Comment out this line, then run `docker-compose up --build`. The container will exit immediately, then docker-compose restarts it, then it exists, and so on.

Now that we changed the config it's time to copy it into the container and change the `CMD` of the `worker` stage in the `./api/Dockerfile`:

```
FROM base AS worker
COPY ./deployment/config/supervisor/supervisord.conf
/etc/supervisor/conf.d/supervisor.conf
CMD ["/bin/sh", "-c", "supervisord -c
/etc/supervisor/conf.d/supervisor.conf"]
```

We don't need to change `docker-compose.yml`. With these changes, everything should work.

Here's the whole config:

```
version: "3.8"
services:
  proxy:
    build:
      context: .
      dockerfile: ./Dockerfile.proxy
    restart: unless-stopped
    ports:
```

```
      - "3000:80"
    volumes:
      - ./deployment/config/proxy.conf:/etc/nginx/nginx.conf
    depends_on:
      - frontend
      - nginx

  frontend:
    build:
      context: .
      dockerfile: ./frontend/Dockerfile
      target: dev
    restart: unless-stopped
    volumes:
      - ./frontend:/usr/src
    environment:
      - NODE_ENV=local

  api:
    build:
      args:
        user: martin
        uid: 1000
      context: .
      dockerfile: ./api/Dockerfile
      target: api
    command: sh -c "./wait-for-it.sh mysql:3306 -t 30 && ./wait-for-it.sh
redis:6379 -t 30 && php-fpm"
    restart: unless-stopped
    volumes:
      - ./api/app:/usr/src/app
      - ./api/config:/usr/src/config
      - ./api/database:/usr/src/database
      - ./api/routes:/usr/src/routes
      - ./api/storage:/usr/src/storage
      - ./api/tests:/usr/src/tests
      - ./api/composer.json:/usr/src/composer.json
```

```
      - ./api/composer.lock:/usr/src/composer.lock
      - ./.env:/usr/src/.env
      - ./deployment/config/php-fpm/php-
dev.ini:/usr/local/etc/php/conf.d/php.ini
    depends_on:
      - update
      - mysql
      - redis

  scheduler:
    build:
      args:
          user: martin
          uid: 1000
      context: .
      dockerfile: ./api/Dockerfile
      target: scheduler
    restart: unless-stopped
    volumes:
      - ./api/app:/usr/src/app
      - ./api/config:/usr/src/config
      - ./api/database:/usr/src/database
      - ./api/routes:/usr/src/routes
      - ./api/storage:/usr/src/storage
      - ./api/tests:/usr/src/tests
      - ./api/composer.json:/usr/src/composer.json
      - ./api/composer.lock:/usr/src/composer.lock
      - ./.env:/usr/src/.env
    depends_on:
      - update
      - mysql
      - redis

  worker:
    build:
      args:
        user: martin
```

```
        uid: 1000
      context: .
      dockerfile: ./api/Dockerfile
      target: worker
    restart: unless-stopped
    volumes:
      - ./api/app:/usr/src/app
      - ./api/config:/usr/src/config
      - ./api/database:/usr/src/database
      - ./api/routes:/usr/src/routes
      - ./api/storage:/usr/src/storage
      - ./api/tests:/usr/src/tests
      - ./api/composer.json:/usr/src/composer.json
      - ./api/composer.lock:/usr/src/composer.lock
      - ./.env:/usr/src/.env
    depends_on:
      - update
      - mysql
      - redis

  nginx:
    build:
      context: .
      dockerfile: ./Dockerfile.nginx
    restart: unless-stopped
    volumes:
      - ./api:/usr/src
      - ./deployment/config/nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - api

  mysql:
    build:
      args:
        password: ${DB_PASSWORD}
      context: .
      dockerfile: ./Dockerfile.mysql
```

```
    restart: unless-stopped
    volumes:
      - ./mysqldata:/var/lib/mysql
    ports:
      - "33060:3306"
    environment:
      - MYSQL_ROOT_PASSWORD=${DB_PASSWORD}

  redis:
    image: redis:7.0.11-alpine
    restart: unless-stopped
    volumes:
      - ./redisdata:/data
    ports:
      - "63790:6379"

  update:
    build:
      args:
        user: martin
        uid: 1000
      context: .
      dockerfile: ./api/Dockerfile
    command: sh -c "./wait-for-it.sh mysql:3306 -t 30 && ./update.sh"
    restart: no
    volumes:
      - ./api/composer.json:/usr/src/composer.json
      - ./api/composer.lock:/usr/src/composer.lock
      - ./.env:/usr/src/.env
      - ./deployment/bin/update.sh:/usr/src/update.sh
    depends_on:
      - mysql
```

Deploying Docker containers to production

In this chapter, we're going to do three main things:

- Make a 100% production-ready docker-compose config
- Change the pipeline to build and use docker images
- Change the deploy script to ship docker containers

Custom-built images for MySQL and nginx

Before we start writing a pipeline, first let's discuss something. Right, now in the `docker-compose` config we have a MySQL service such as this:

```
mysql:
  image: mysql:8.0.33
  restart: unless-stopped
  volumes:
    - ./deployment/config/mysql/create_database.sql:/docker-entrypoint-initdb.d/create_database.sql
    - ./deployment/config/mysql/set_native_password.sql:/docker-entrypoint-initdb.d/set_native_password.sql
    - ./deployment/config/mysql/mysql.cnf:/etc/mysql/conf.d/mysql.cnf
    - ./mysqldata:/var/lib/mysql
  ports:
    - "33060:3306"
  environment:
    MYSQL_ROOT_PASSWORD: root
```

We basically use the first three volumes to put files into the container so it has a database a user and a config file. As I said earlier, we can build our own image that already has these files.

Let's just do this with MySQL before we move on:

```
FROM mysql:8.0.33

COPY ./deployment/config/mysql/create_database.sql /docker-entrypoint-initdb.d/create_database.sql
COPY ./deployment/config/mysql/create_database.sql /docker-entrypoint-initdb.d/set_native_password.sql
COPY ./deployment/config/mysql/mysql.cnf /etc/mysql/conf.d/mysql.cnf
```

That was easy. Now the `docker-compose.yml` looks like this:

```
mysql:
  build:
    context: .
    dockerfile: ./Dockerfile.mysql
  restart: unless-stopped
  volumes:
    - ./mysqldata:/var/lib/mysql
  ports:
    - "33060:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=root
```

If you check out the `set_native_password.sql` file (from the fundamental chapter's source code) it has a pretty big security issue:

```
ALTER USER 'root'@'%' IDENTIFIED WITH mysql_native_password BY 'root';
```

It contains the database's root password (and the docker-compose file above also contains it). So far it wasn't a problem because we were talking about development databases.

This is how to get rid of the password in the SQL file:

```
FROM mysql:8.0.33

ARG password

RUN echo "ALTER USER 'root'@'%' IDENTIFIED WITH mysql_native_password BY
'${password}';" >> /docker-entrypoint-initdb.d/set_native_password.sql
```

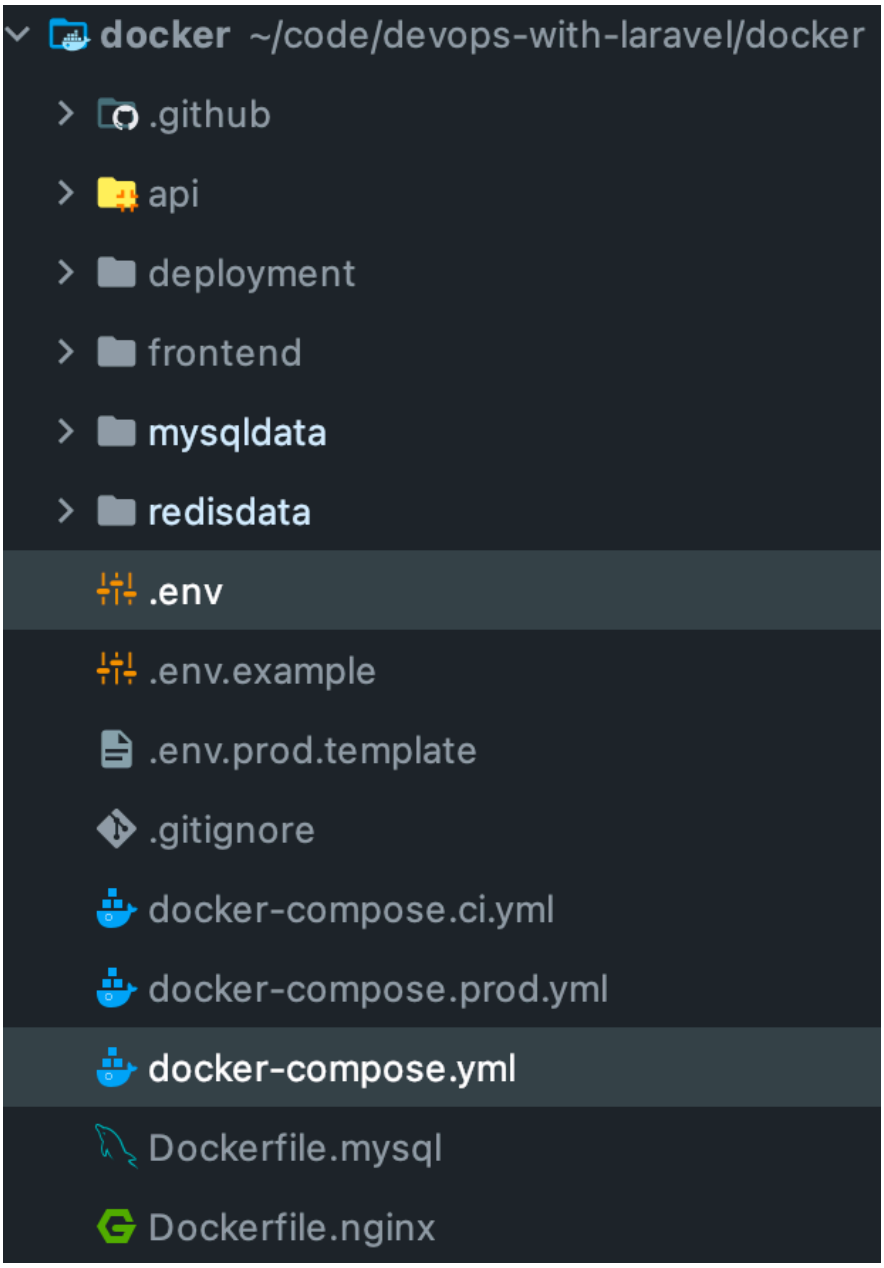
Instead of storing the script as a file in the repository, the following happens here:

- The dockerfile has a new build argument called `password`
- It contains the script as a string and puts it into a file. Inside the container.

So now we can rewrite the docker-compose config such as this:

```
mysql:
  build:
    args:
      password: ${DB_PASSWORD}
    context: .
    dockerfile: ./Dockerfile.mysql
  restart: unless-stopped
  volumes:
    - ./mysqldata:/var/lib/mysql
  ports:
    - "33060:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=${DB_PASSWORD}
```

The `DB_PASSWORD` comes from the environment. Fortunately, docker-compose always load the `.env` file from the current directory. If you check out the source code you can see that the `.env` file is located in the root directory:



So we store the `DB_PASSWORD` in the `.env` file, docker-compose loads it then uses it as a build argument and an environment variable.

We can do the same thing with nginx:

```
FROM nginx:1.25.1-alpine

COPY ./api /usr/src
COPY ./deployment/config/nginx.conf /etc/nginx/nginx.conf

EXPOSE 80
```

And then in `docker-compose.yml`:

```
nginx:
  build:
    context: .
    dockerfile: ./Dockerfile.nginx
  restart: unless-stopped
  depends_on:
    - api
```

And the last one is the Dockerfile of proxy:

```
FROM nginx:1.25.1-alpine

COPY ./deployment/config/proxy.conf /etc/nginx/nginx.conf

EXPOSE 80
```

These simple changes make the deployment process much easier. And as a result, the production server will look like this:

```
$ cd /usr/src
$ ls -la
total 16
drwxr-xr-x  2 martin martin 4096 Jul  7 17:21 .
drwxr-xr-x 14 root   root   4096 Mar 17 02:08 ..
-rw-rw-r--  1 martin martin  638 Jul  7 17:21 .env
-rw-r--r--  1 martin martin 1855 Jul  7 17:21 docker-compose.prod.yml
$
```

Do you see how the server doesn't have source code on it?

Building images in a pipeline

In order to ship docker containers, first we need to build them. The docker-compose config we created in the previous chapter is great for development. It takes the code and builds the images. However, in production, we don't want to do that. We only want to care about images.

So when you have a containerized PHP application you don't actually need source code on your server. When I first learned about this, my mind was blown away. But that's true, and this is what a production server looks like:

```
$ cd /usr/src
$ ls -la
total 16
drwxr-xr-x  2 martin martin 4096 Jul  7 17:21 .
drwxr-xr-x 14 root   root   4096 Mar 17 02:08 ..
-rw-rw-r--  1 martin martin  638 Jul  7 17:21 .env
-rw-r--r--  1 martin martin 1855 Jul  7 17:21 docker-compose.prod.yml
$
```

It only needs a `.env` file and the `docker-compose.yml`. We don't need to run `git` commands or `php artisan key:generate` or anything like that. Instead of these commands or source code, we only need to run containers from images:

```
$ cd /usr/src
$ ls -la
total 16
drwxr-xr-x  2 martin martin 4096 Jul  7 17:21 .
drwxr-xr-x 14 root   root   4096 Mar 17 02:08 ..
-rw-rw-r--  1 martin martin  638 Jul  7 17:21 .env
-rw-r--r--  1 martin martin 1855 Jul  7 17:21 docker-compose.prod.yml
$ sudo docker ps
CONTAINER ID   IMAGE                                                                COMMAND                  CREATED             STATUS             PORTS                                       NAMES
4e2fe2ae5fd3   martinjoo/posts-nginx:9d933619f743ce267d74fce3a08f210643f4ceec        "/docker-entrypoint.…"   About an hour ago   Up About an hour   0.0.0.0:8000->80/tcp, :::8000->80/tcp   posts-nginx-1
21516b98c738   martinjoo/posts-worker:9d933619f743ce267d74fce3a08f210643f4ceec       "docker-php-entrypoi…"   About an hour ago   Up About an hour   9000/tcp                                    posts-worker-1
5e4bbffeb86a   martinjoo/posts-api:9d933619f743ce267d74fce3a08f210643f4ceec          "docker-php-entrypoi…"   About an hour ago   Up About an hour   9000/tcp                                    posts-api-1
8ec6eaf5b2c4   martinjoo/posts-scheduler:9d933619f743ce267d74fce3a08f210643f4ceec    "docker-php-entrypoi…"   About an hour ago   Up 4 seconds       9000/tcp                                    posts-scheduler-1
7fdf3b9f4bd7   martinjoo/posts-frontend:9d933619f743ce267d74fce3a08f210643f4ceec     "/docker-entrypoint.…"   About an hour ago   Up About an hour   0.0.0.0:80->80/tcp, :::80->80/tcp       posts-frontend-1
7b4353783502   redis:7.0.11-alpine                                                  "docker-entrypoint.s…"   About an hour ago   Up About an hour   6379/tcp                                    posts-redis-1
9ca5dc9ef313   martinjoo/posts-mysql:9d933619f743ce267d74fce3a08f210643f4ceec       "docker-entrypoint.s…"   About an hour ago   Up About an hour   3306/tcp, 33060/tcp                         posts-mysql-1
$
```

The first thing that needs to be done is to build and version images in an automated way. The GitHub pipeline is a perfect place to do that.

If you want to build and store docker images you need a registry. It's a place where you can upload (push) and download (pull) docker images. DockerHub is an excellent provider. This is the official registry of Docker, so anytime you use an official image (such as PHP or MySQL) you pull images from DockerHub.

The project we're building requires multiple images:

- API
- Scheduler
- Worker
- Frontend
- MySQL

- Proxy

I created one repository for each of them on DockerHub:

- martinjoo/posts-api
- martinjoo/posts-scheduler
- martinjoo/posts-worker
- martinjoo/posts-frontend
- martinjoo/posts-mysql
- martinjoo/posts-proxy

This is where the pipeline is going to push images. The production server will pull images from these repositories because in the docker-compose file, we're going to have values such as:

```
api:
  image: martinjoo/posts-api:<version>
```

Let's review the pipeline from the first chapter (when we didn't use docker) and let's see how we can dockerize it. These are the first steps:

```
- uses: shivammathur/setup-php@15c43e89cdef867065b0213be354c2841860869e
  with:
    php-version: '8.1'
- uses: actions/checkout@v3
- name: Copy .env
  run: |
    cd api
    cp .env.ci .env
- name: Install Dependencies
  run: |
    cd api
    composer install -q --no-ansi --no-interaction --no-scripts --no-progress
--prefer-dist
- name: Generate key
  run: |
    cd api
    php artisan key:generate
- name: Directory Permissions
  run: |
```

```
    cd api
    chmod -R 777 storage bootstrap/cache
```

The new pipeline doesn't need any of them. These are all steps that are already being done in the image:

- The image has composer dependencies so no need to run `compose install`
- The image has the source with the right permissions and users so no need to run `chmod`
- However, the image doesn't have a `.env` file with an `APP_KEY` in it. Fortunately, we can solve these pretty easily without running these steps in the pipeline.

So these initialization steps can be replaced by using images. After these, the original pipeline runs code analysis tools, and tests, and then deploys the code. These steps will be similar in the new pipeline as well.

In order to run code analysis and the tests we need to run a docker-compose stack. For this purpose, I usually create a `docker-compose.ci.yml` file. It looks like this:

```
version: "2.4"
services:
  api:
    image: ${API_IMAGE}
    environment:
      - APP_NAME=posts
      - APP_ENV=local
      - APP_KEY=base64:aL6o/U2e1ziUTXsyTkfzNziH9l4crCISoWMwC8LX4B0=
      - APP_DEBUG=true
      - APP_URL=http://localhost
      - LOG_CHANNEL=stack
      - LOG_LEVEL=debug
      - DB_CONNECTION=mysql
      - DB_HOST=mysql-test
      - DB_PORT=3306
      - DB_DATABASE=posts
      - DB_USERNAME=root
      - DB_PASSWORD=${DB_PASSWORD}
      - QUEUE_CONNECTION=sync
      - MAIL_MAILER=log
    depends_on:
      migrate:
        condition: service_started
      mysql-test:
```

```
        condition: service_healthy
```

As you can see the container lists every environment variable. This is because I don't want to copy the `.env.example` file and then change some variables inside it. We can just do that instead. I found this solution better in the pipeline. It lists everything it needs and it doesn't depend on anything.

The other new thing is this line:

```
image: ${API_IMAGE}
```

As I said we want to version our images. There are lots of different strategies but the most straightforward one is to use commit SHAs. The idea is that whenever you push to the main branch, you build a new image with a specific commit SHA. So you'll have images such as this: `martinjoo/posts-api:9d933619f743ce267d74fce3a08f210643f4ceec` It doesn't look good, but generally speaking, you won't be thinking or talking too much about version numbers when it comes to docker images.

So the image tag changes based on the commit SHA. In the pipeline, we can create an environment variable called `API_IMAGE` which can be used in the `docker-compose.ci.yml` file as shown above.

The other new thing is this:

```
depends_on:
  migrate:
    condition: service_started
  mysql-test:
    condition: service_healthy
```

We used `wait-for-it.sh` earlier. This is essentially the same but with built-in docker-compose directives. It starts the `api` container only when `mysql-test` is healthy. It's a cool feature, however, "healthy" means different things for every container. You can use `wait-for-it.sh` for anything the same way, but you need to write different health checks to make `service_healthy` work. So in the `mysql-test` service, we need to define what it means to be "healthy":

```
mysql-test:
  image: ${MYSQL_IMAGE}
  healthcheck:
    test: [ "CMD", "mysqladmin", "ping" ]
    interval: 10s
    timeout: 5s
    retries: 5
  environment:
    - MYSQL_ROOT_PASSWORD=${DB_PASSWORD}
```

It's a `ping` command that runs every 10 seconds and it's repeated 5 times if MySQL is not responding. If the 5th check fails the container stops.

And we also have the `migrate` service:

```
migrate:
  image: ${API_IMAGE}
  environment:
    - APP_ENV=local
    - APP_KEY=base64:aL6o/U2e1ziUTXsyTkfzNziH9l4crCISoWMwC8LX4B0=
    - APP_DEBUG=true
    - LOG_CHANNEL=stack
    - LOG_LEVEL=debug
    - DB_CONNECTION=mysql
    - DB_HOST=mysql-test
    - DB_PORT=3306
    - DB_DATABASE=posts
    - DB_USERNAME=root
    - DB_PASSWORD=${DB_PASSWORD}
  depends_on:
    mysql-test:
      condition: service_healthy
```

Each of these services uses a dynamic image name coming from an environment variable.

The important part is this: we need to build an `api` or any `mysql` image before we can run the tests and the code analysis.

So let's start writing the pipeline:

```yaml
name: Tests

on:
  push:
    branches: [ "main" ]

env:
  API_IMAGE: martinjoo/posts-api:${{ github.sha }}
  SCHEDULER_IMAGE: martinjoo/posts-scheduler:${{ github.sha }}
  WORKER_IMAGE: martinjoo/posts-worker:${{ github.sha }}
  NGINX_IMAGE: martinjoo/posts-nginx:${{ github.sha }}
  PROXY_IMAGE: martinjoo/posts-proxy:${{ github.sha }}
  MYSQL_IMAGE: martinjoo/posts-mysql:${{ github.sha }}
  FRONTEND_IMAGE: martinjoo/posts-frontend:${{ github.sha }}
```

It runs when you push commits or open a PR to the main branch (GitFlow coming later).

The `env` key defines environment variables that can be used in the entire pipeline. Here we can construct the image names. GitHub provides some special variables for us. `github.sha` is one of them and it contains the current commit SHA so we can create image names such as `martinjoo/posts-api:9d933619f743ce267d74fce3a08f210643f4ceec`

Let's move on to the jobs:

```yaml
jobs:
  tests:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
```

We have already discussed the basic syntax and `actions/checkout@v3` in the "Building a pipeline" chapter. This pipeline will not just build but also push images to DockerHub. Because of that, we need to log in to DockerHub. I created two secrets in GitHub (also discussed earlier) so I can log in with my credentials without compromising them.

We can now build the API and MySQL images:

```
- name: Build API image
  run: docker build -t $API_IMAGE --target=api --build-arg user=martin --
build-arg uid=1000 -f ./api/Dockerfile .
- name: Build MySQL image
  run: docker build -t $MYSQL_IMAGE --build-arg password=${{
secrets.DB_PASSWORD }} -f ./Dockerfile.mysql .
```

We've already seen this `docker build` command but now we're using environment variables to pass the image name and version.

To run static code analysis we don't actually need a database, so we can simply run the `api` image alone:

```
- name: Run phpstan
  run: docker run --rm -t $API_IMAGE ./vendor/bin/phpstan analyze --memory-
limit=1G
- name: Run phpinsights
  run: docker run --rm -t $API_IMAGE php artisan insights --no-interaction --
min-quality=90 --min-complexity=90 --min-architecture=90 --min-style=90 --
ansi --format=github-action
```

`docker run --rm -t $API_IMAGE ./vendor/bin/phpstan analyze --memory-limit=1G` means:

- Run the `$API_IMAGE` image (`-t`)
- Remove the container after it is finished (`--rm`)
- Run the following command inside the running container: `./vendor/bin/phpstan analyze --memory-limit=1G`

The two steps are basically the same as in the first chapter but now there's a `docker run` before `phpstan`.

If everything goes well we can run the tests:

```
- name: Run tests
  run: |
    docker-compose -f docker-compose.ci.yml up -d
    docker-compose -f docker-compose.ci.yml exec -T api php artisan test
    docker-compose -f docker-compose.ci.yml down
```

If the compose file's name is `docker-compose.yml` you need to set it explicitly using the `-f` flag. `-d` means detach mode so it starts the container and then puts it into the background so it doesn't run in the foreground.

After that command, we have two running containers: `api` and `mysql-test`. In order to run the test we need to go inside the `api` container and execute `php artisan test`:

```
docker-compose -f docker-compose.ci.yml exec -T api php artisan test
```

And after that, if everything goes well we can stop the stack with `docker-compose down`. If you're using the default GitHub runners you don't even need to do that. It's because every job is picked up by a random runner with a completely empty state.

If the tests pass we can build the other images:

```
- name: Build scheduler image
  run: docker build -t $SCHEDULER_IMAGE --target=scheduler --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .
- name: Build worker image
  run: docker build -t $WORKER_IMAGE --target=worker --build-arg user=martin
--build-arg uid=1000 -f ./api/Dockerfile .
- name: Build nginx image
  run: docker build -t $NGINX_IMAGE -f ./Dockerfile.nginx .
- name: Build frontend image
  run: docker build -t $FRONTEND_IMAGE --target=prod -f ./frontend/Dockerfile
.
- name: Build proxy image
  run: docker build -t $PROXY_IMAGE -f ./Dockerfile.proxy .
```

As you can see, it's the same command over and over again but with different image names. Don't forget to target the right stage when building the scheduler and worker images.

After we have all the images we need to push them to DockerHub:

```
- name: Push API image
  run: docker push $API_IMAGE
- name: Push scheduler image
  run: docker push $SCHEDULER_IMAGE
- name: Push worker image
  run: docker push $WORKER_IMAGE
```

```
- name: Push nginx image
  run: docker push $NGINX_IMAGE
- name: Push MySQL image
  run: docker push $MYSQL_IMAGE
- name: Push frontend image
  run: docker push $FRONTEND_IMAGE
- name: Push proxy image
  run: docker push $PROXY_IMAGE
```

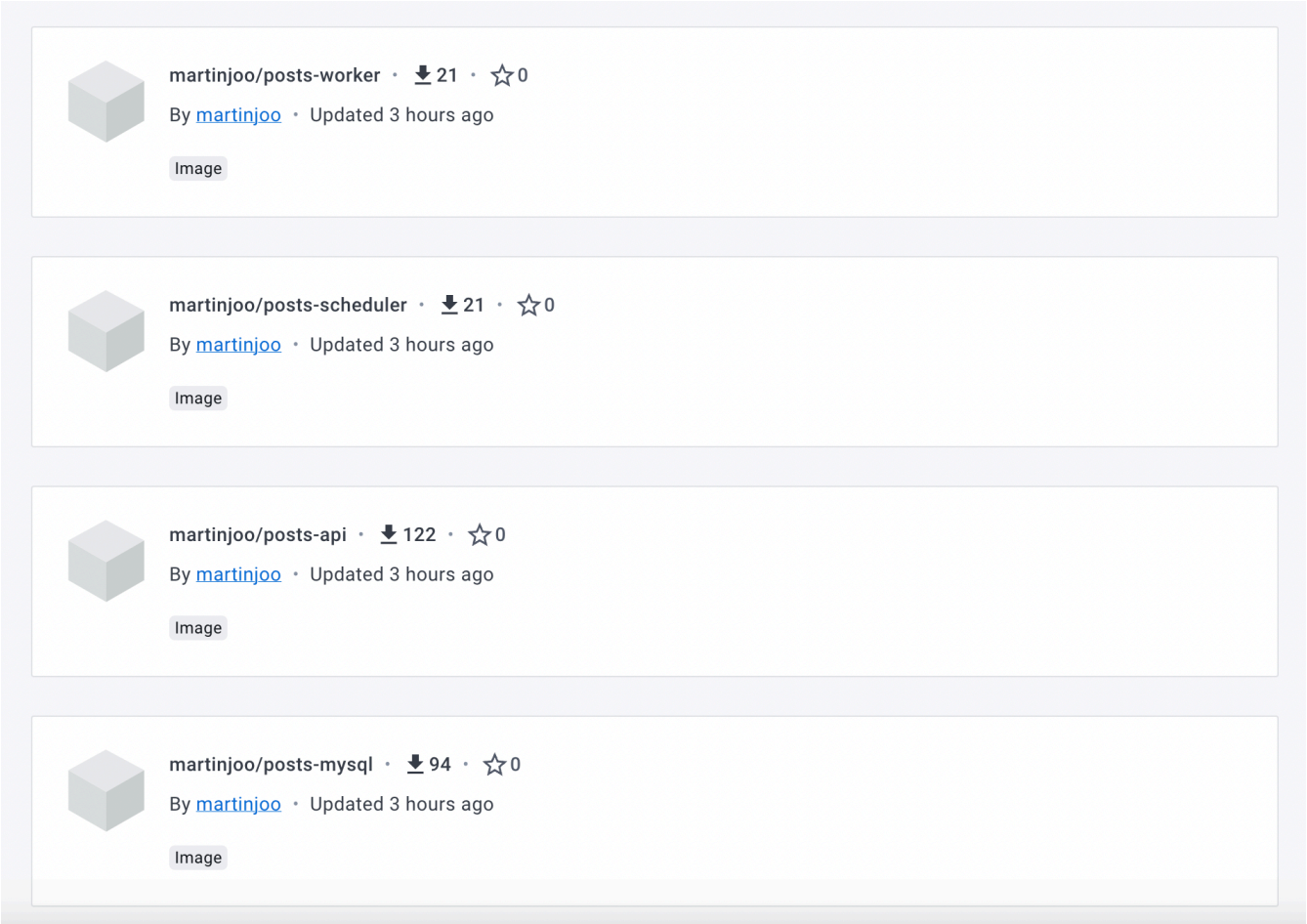
This is the whole pipeline:

```
name: Tests

on:
  push:
    branches: [ "main" ]

env:
  API_IMAGE: martinjoo/posts-api:${{ github.sha }}
  SCHEDULER_IMAGE: martinjoo/posts-scheduler:${{ github.sha }}
  WORKER_IMAGE: martinjoo/posts-worker:${{ github.sha }}
  NGINX_IMAGE: martinjoo/posts-nginx:${{ github.sha }}
  PROXY_IMAGE: martinjoo/posts-proxy:${{ github.sha }}
  MYSQL_IMAGE: martinjoo/posts-mysql:${{ github.sha }}
  FRONTEND_IMAGE: martinjoo/posts-frontend:${{ github.sha }}

jobs:
  tests:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build API image
```

```
        run: docker build -t $API_IMAGE --target=api --build-arg user=martin
--build-arg uid=1000 -f ./api/Dockerfile .
      - name: Build MySQL image
        run: docker build -t $MYSQL_IMAGE -f ./Dockerfile.mysql .
      - name: Run phpstan
        run: docker run --rm -t $API_IMAGE ./vendor/bin/phpstan analyze --
memory-limit=1G
      - name: Run phpinsights
        run: docker run --rm -t $API_IMAGE php artisan insights --no-
interaction --min-quality=90 --min-complexity=90 --min-architecture=90 --min-
style=90 --ansi
      - name: Run tests
        run: |
          docker-compose -f docker-compose.ci.yml up -d
          docker-compose -f docker-compose.ci.yml exec -T api php artisan
test
      - name: Build scheduler image
        run: docker build -t $SCHEDULER_IMAGE --target=scheduler --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .
      - name: Build worker image
        run: docker build -t $WORKER_IMAGE --target=worker --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .
      - name: Build nginx image
        run: docker build -t $NGINX_IMAGE -f ./Dockerfile.nginx .
      - name: Build frontend image
        run: docker build -t $FRONTEND_IMAGE -f ./frontend/Dockerfile .
      - name: Build proxy image
        run: docker build -t $PROXY_IMAGE -f ./Dockerfile.proxy .
      - name: Push API image
        run: docker push $API_IMAGE
      - name: Push scheduler image
        run: docker push $SCHEDULER_IMAGE
      - name: Push worker image
        run: docker push $WORKER_IMAGE
      - name: Push nginx image
        run: docker push $NGINX_IMAGE
      - name: Push MySQL image
```

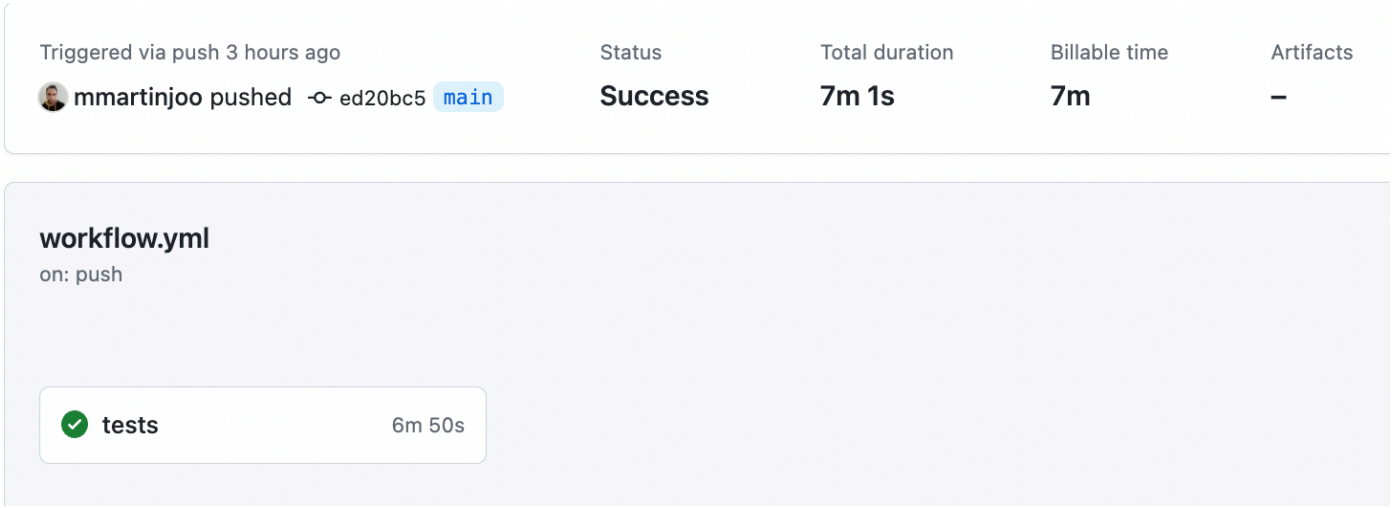
```
      run: docker push $MYSQL_IMAGE
    - name: Push frontend image
      run: docker push $FRONTEND_IMAGE
    - name: Push proxy image
      run: docker push $PROXY_IMAGE
```

If the pipeline was successful the images can be found on DockerHub:

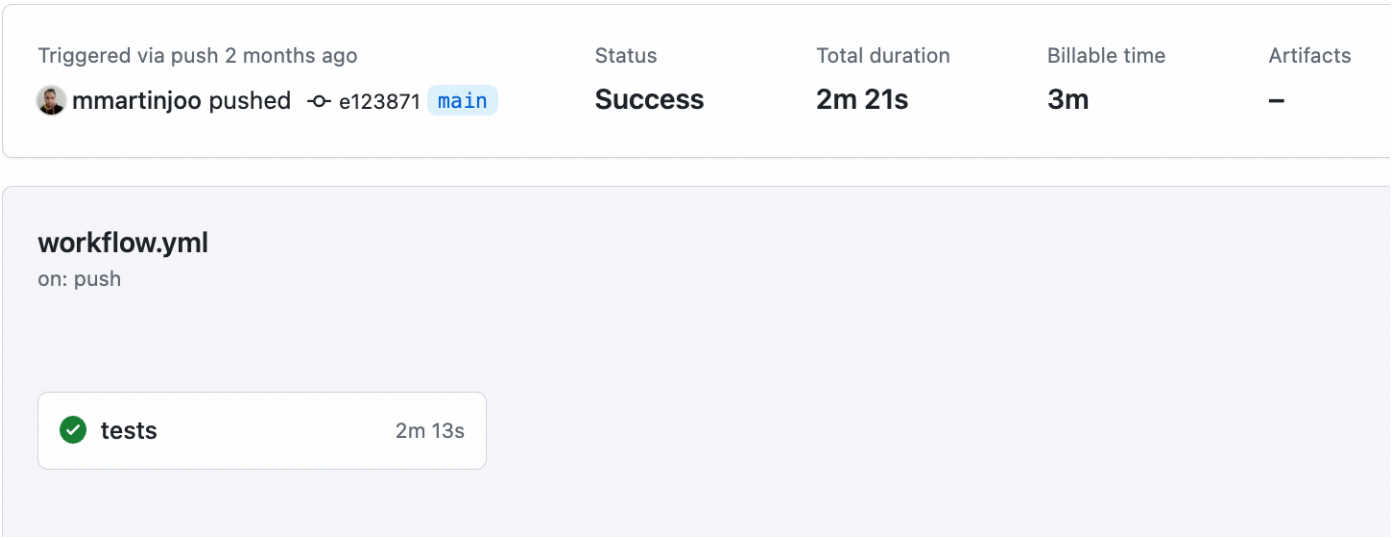


Optimizing the pipeline

Building and pushing docker images in a pipeline has a big disadvantage: it takes forever to run. The one I just presented to you takes ~7 minutes:



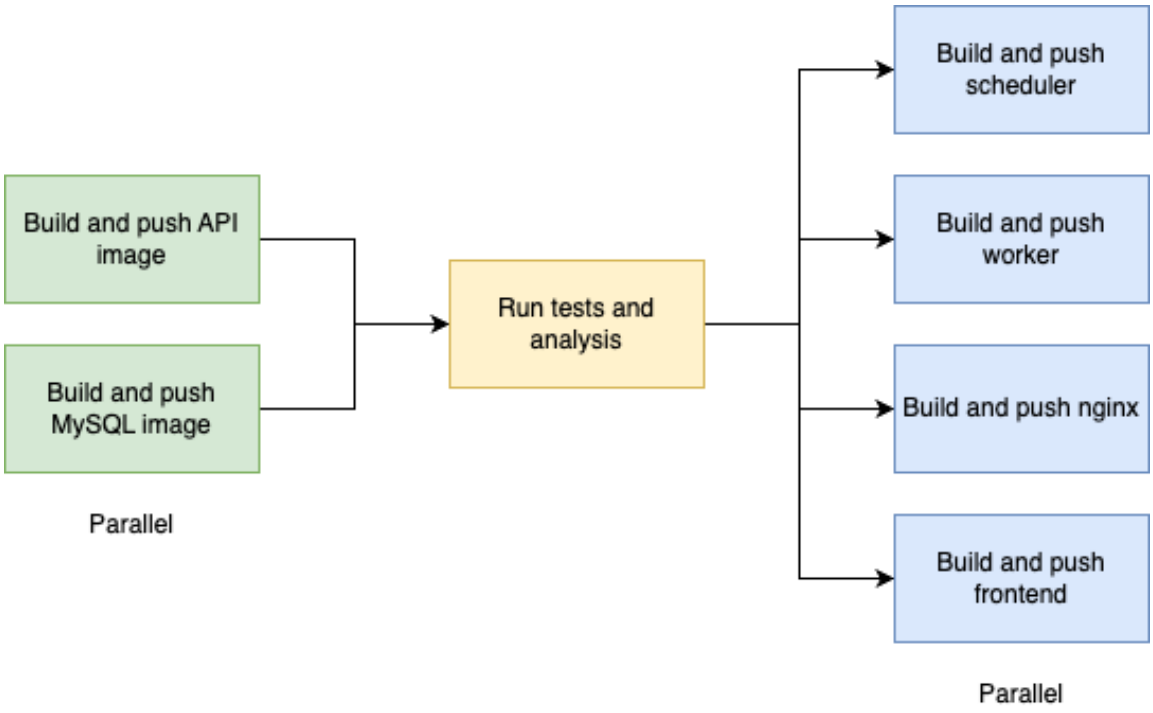
Without docker images, it took between 2 and 3 minutes:



However, we can optimize it a little bit. This is how GitHub actions work:

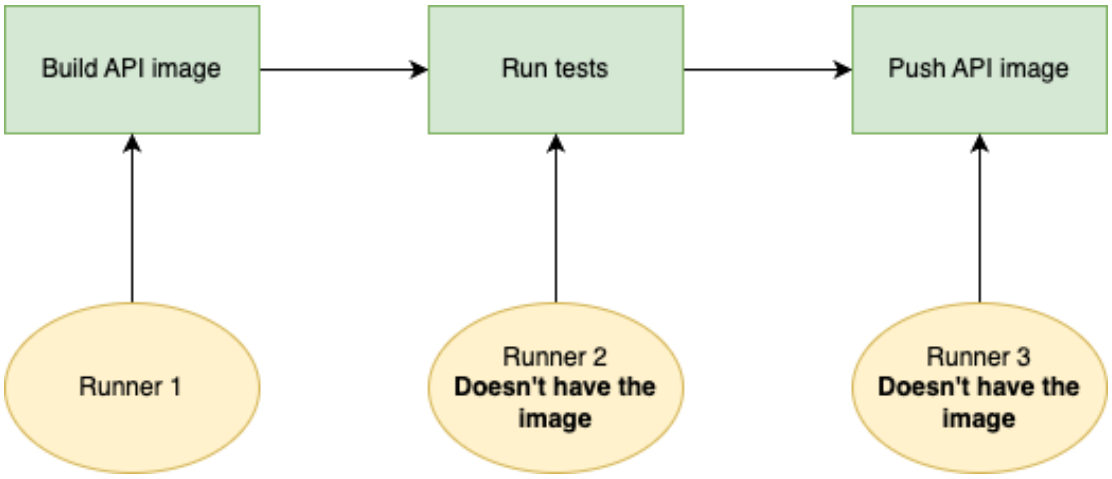
- Each job is picked by a random runner with an **empty state**
- Jobs run parallel by default
- But we can define job dependencies

We can optimize for concurrent jobs, something like this:

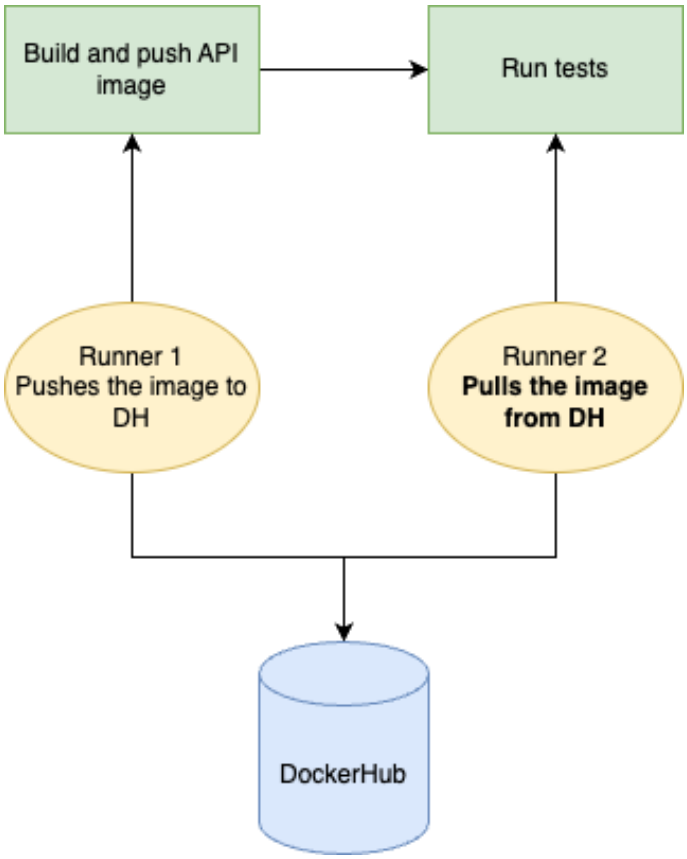


Each box is a job. Each job with a matching color can run in parallel such as "Build and push API image" and "Build and push MySQL image".

Building and pushing images happens in one step now. This is because each job starts with an **empty state.** This wouldn't work, for example:



Runner 1 builds the image locally but of course, Runner 2 doesn't have this image so it cannot start `docker-compose`. Runner 3 doesn't have the image either so it doesn't know what to push. We can solve this problem by building and pushing in one job (of course it brings another problem we need to solve later):



This is how we can refactor the pipeline:

```
jobs:
  build-api:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build API image
        run: docker build -t $API_IMAGE --build-arg user=martin --build-arg
uid=1000 -f ./api/Dockerfile .
      - name: Push API image
        run: docker push $API_IMAGE

  build-mysql:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

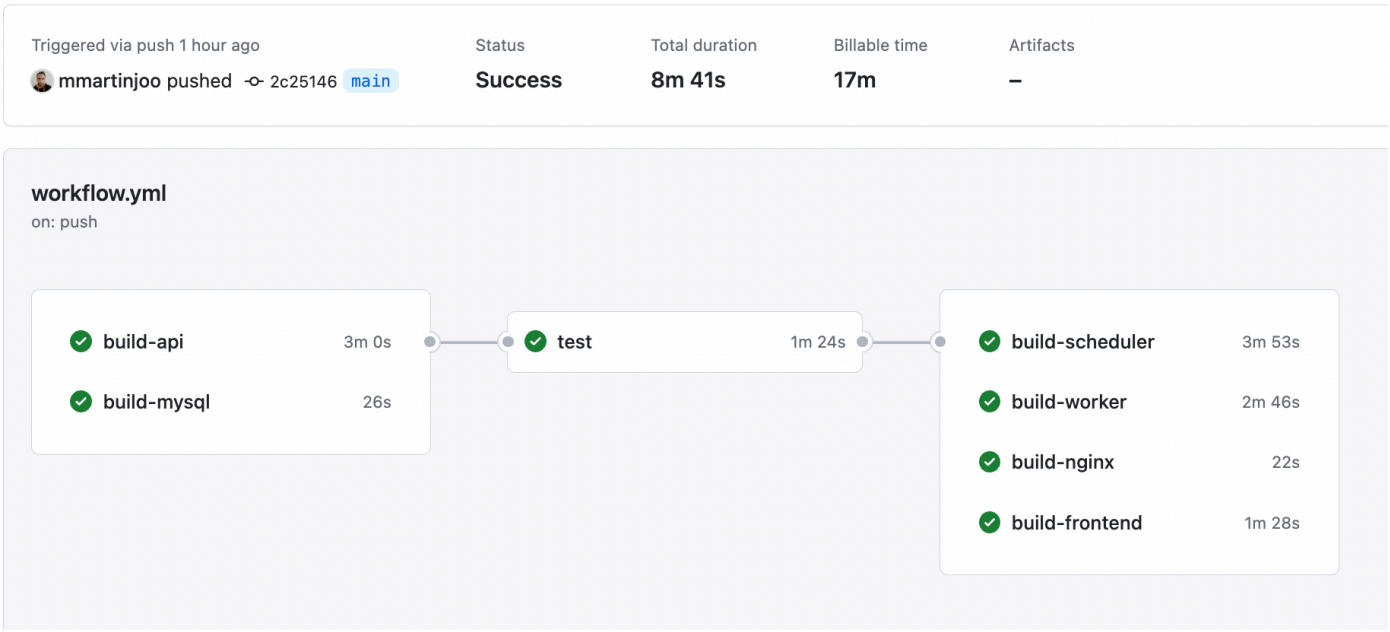
```
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build MySQL image
        run: docker build -t $MYSQL_IMAGE -f ./Dockerfile.mysql .
      - name: Push Mysql image
        run: docker push $MYSQL_IMAGE

  test:
    needs: [build-api, build-mysql]
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Pull API image
        run: docker pull $API_IMAGE
      - name: Pull MySQL image
        run: docker pull $MYSQL_IMAGE
      - name: Run phpstan
        run: docker run --rm -t $API_IMAGE ./vendor/bin/phpstan analyze --
memory-limit=1G
      - name: Run phpinsights
        run: docker run --rm -t $API_IMAGE php artisan insights --no-
interaction --min-quality=90 --min-complexity=90 --min-architecture=90 --min-
style=90 --ansi --format=github-action
      - name: Run tests
        run: |
          docker-compose -f docker-compose.ci.yml up -d --force-recreate
          docker-compose -f docker-compose.ci.yml exec -T api php artisan
test
          docker-compose -f docker-compose.ci.yml down

  build-scheduler:
```

```
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build scheduler image
        run: |
          docker build -t $SCHEDULER_IMAGE --target=scheduler --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .
          docker push $SCHEDULER_IMAGE

  build-worker:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build worker image
        run: |
          docker build -t $WORKER_IMAGE --target=worker --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .
          docker push $WORKER_IMAGE

  build-nginx:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
```

```
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build nginx image
        run: |
          docker build -t $NGINX_IMAGE -f ./Dockerfile.nginx .
          docker push $NGINX_IMAGE

  build-frontend:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build frontend image
        run: |
          docker build -t $FRONTEND_IMAGE --target=prod -f
./frontend/Dockerfile .
          docker push $FRONTEND_IMAGE

  build-proxy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build frontend image
        run: |
          docker build -t $PROXY_IMAGE --target=prod -f ./Dockerfile.proxy .
          docker push $PROXY_IMAGE
```

Technically only the `needs` key is new. It defines a dependency between two or more jobs. For example, the `test` job depends on the first two build jobs: `needs: [build-api, build-mysql]`

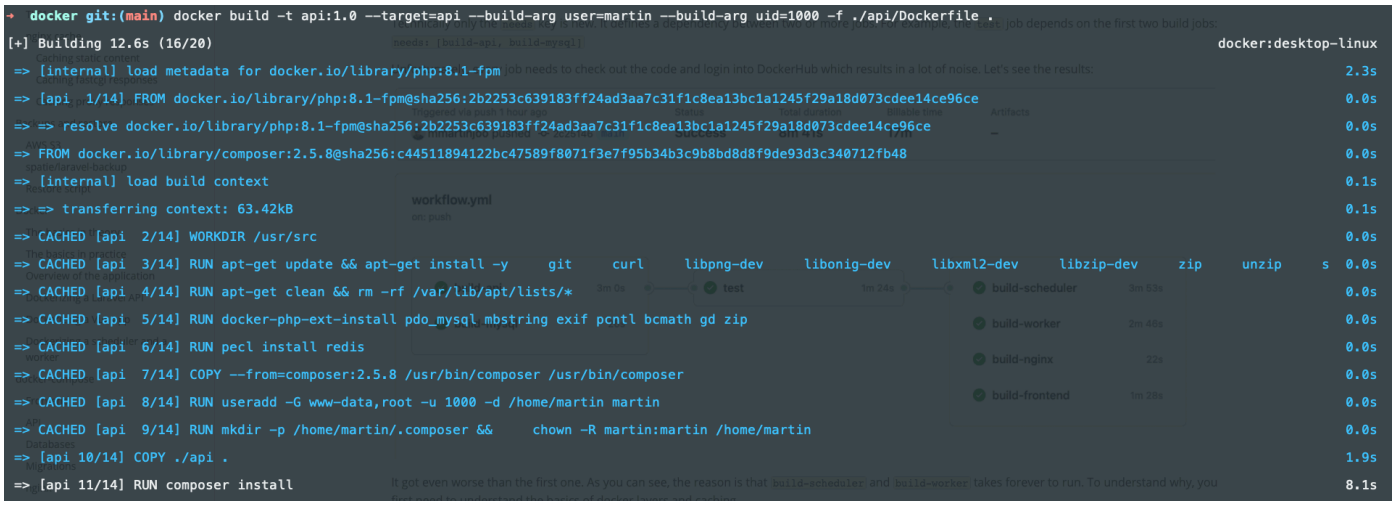
Unfortunately, every job needs to check out the code and login into DockerHub which results in a lot of noise. Let's see the results:



It got even worse than the first one. As you can see, the reason is that `build-scheduler` and `build-worker` takes forever to run. To understand why, you first need to understand the basics of docker layers and caching.

Docker layers

When you write a command (such as `RUN ls -la`) in a Dockerfile it creates a layer. You can see it in the output of `docker build`:



The image I'm building has 14 layers. You can also see from layer 2 to layer 9 they are all cached and take 0.0s to run. So these layers are cacheable. They are essentially building blocks and each layer adds more content to the previous ones. For example, layer 9 adds the `/home/martin/.composer` folder to the existing files. When a layer changes it needs to be re-built by docker. And not just that specific layer, but all the ones that follow.

You can see that effect on the image. I changed some source files in the `./api` directory so layer 10 is the first one that needs to be rebuilt. But layer 11 also needs a rebuild because layer 10 changed!

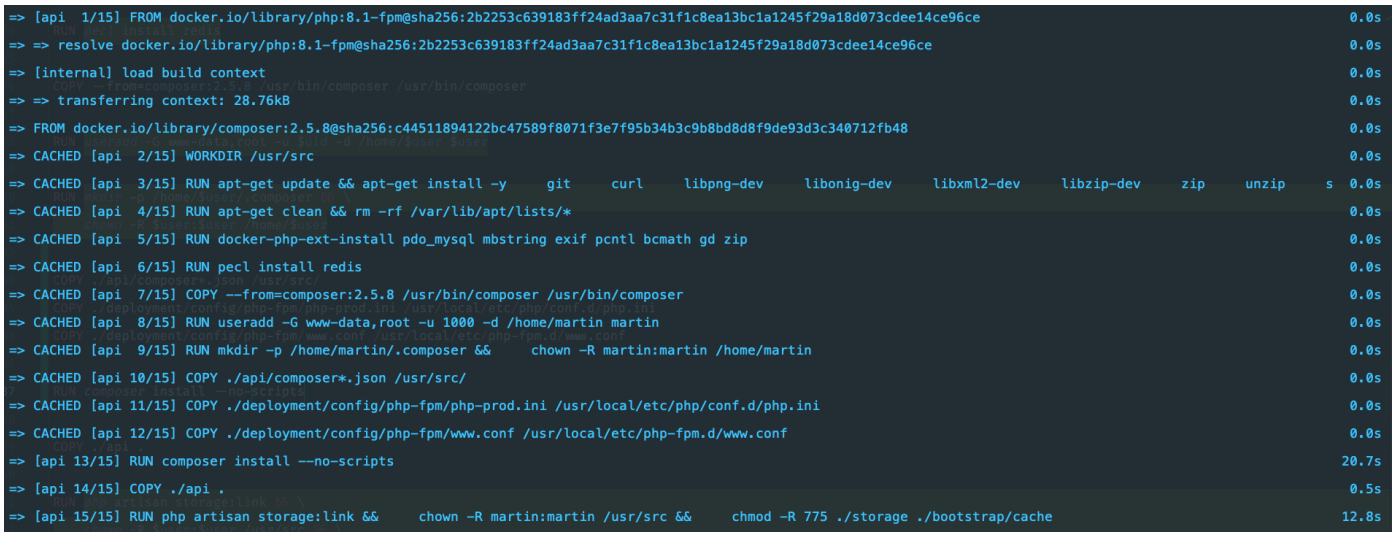
That teaches us at least two things:

- When a job is picked up by a random runner every layer needs to be built from scratch! The runner doesn't have the previous state, and it doesn't cache layers. Which is a good thing in general, but it's terrible in our current situation. If you're using your own runner, it would be not a problem for you because your runner would have a build cache (probably).
- It does matter how you structure your Dockerfiles. In the example above, if I change anything inside the `./api` directory (so basically if I'm writing code) docker will install composer packages from scratch. However, we only need to run `composer install` if `composer.json` or `composer-lock.json` changes, right? Yes.

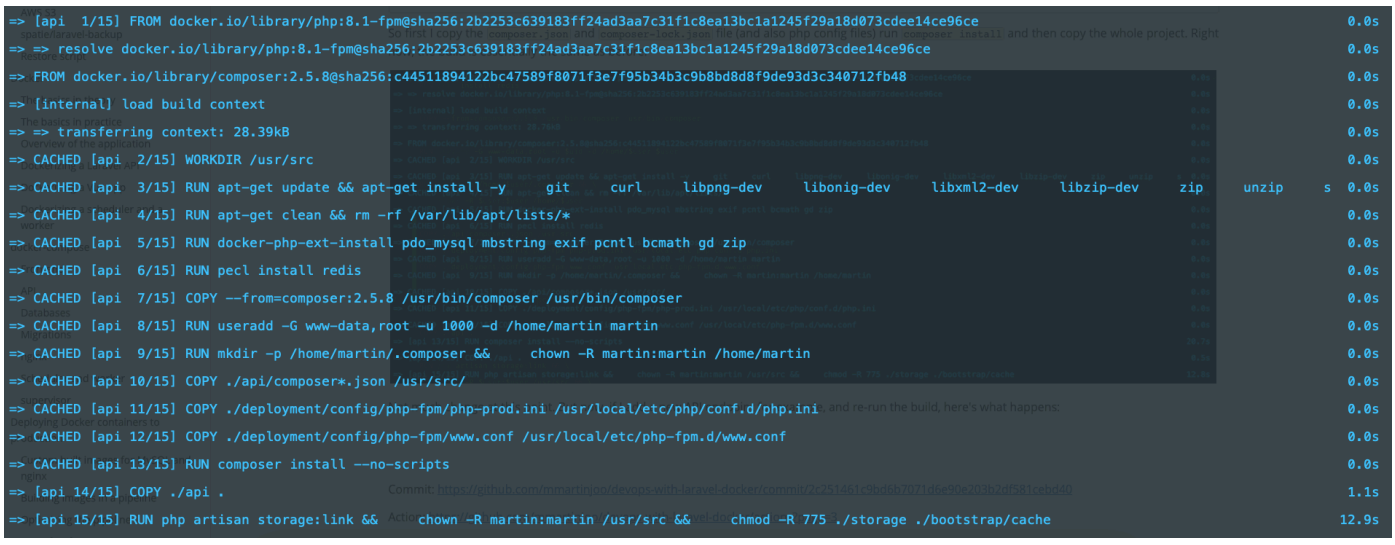
So we can optimize the `api` Dockerfile by doing this:

```
COPY ./api/composer*.json /usr/src/
COPY ./deployment/config/php-fpm/php-prod.ini
/usr/local/etc/php/conf.d/php.ini
COPY ./deployment/config/php-fpm/www.conf /usr/local/etc/php-fpm.d/www.conf

RUN composer install --no-script

COPY ./api .
```

So first I copy the `composer.json` and `composer-lock.json` files (and also php config files) run `composer install` and then copy the whole project. Right now, the build is essentially the same as before:



Not much change at this point. But now, if I add a new API endpoint for example, and re-run the build, here's what happens:



Can you see the difference? Layer 13 (composer install) is cached. Which saves us a good 20 seconds!

By the way, these layers are the reason why I'm writing multiple commands as one such as this:

```
RUN apt-get update && apt-get install -y \
    git \
    curl \
    libpng-dev \
```

It only creates one layer. If I wrote like this:

```
RUN apt-get update
RUN apt-get install -y git
RUN apt-get install -y curl
RUN apt-get install -y libpng-dev
```

It would create 4 layers.

Back to the pipeline

So the problem is that the scheduler stage has only one extra layer to the api stage. The worker stage 2 extra layers. These are minimal differences but we need to build 15 layers from scratch.

To fix this we need to build these three images in one job:

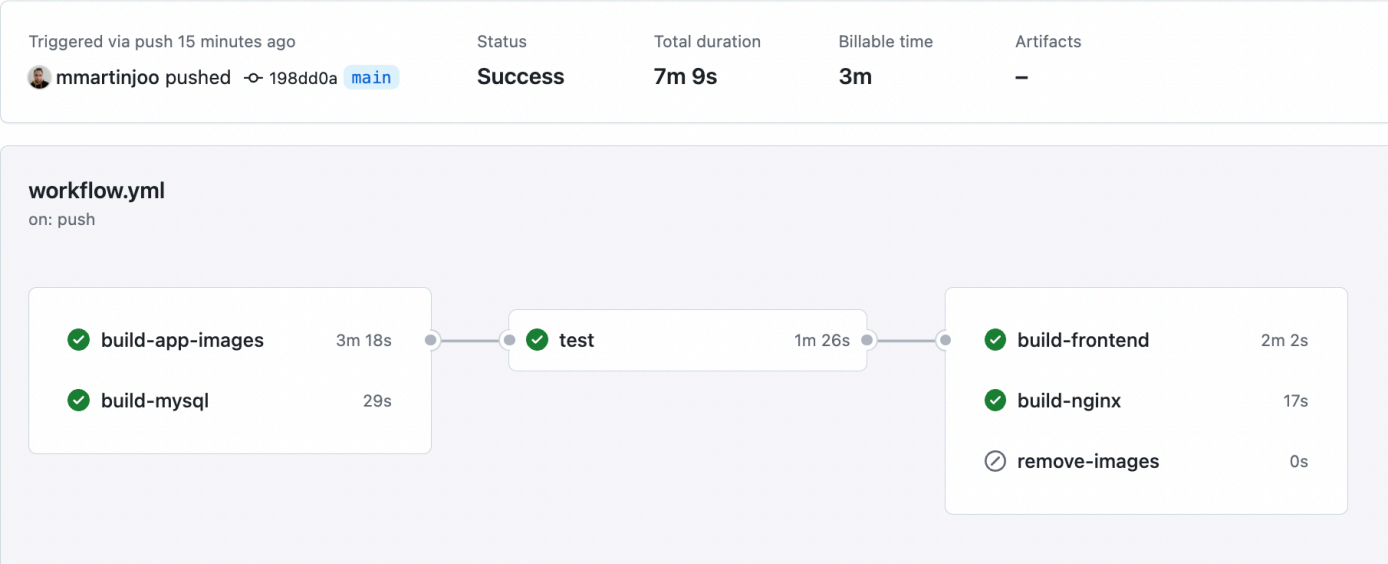
- `api`
- `scheduler`
- `worker`

When the runner builds the `api` image it creates and caches 13 or so layers. When it's time to build the `scheduler` image it can re-use these 13 layers from the build cache and then build the only layer specific to the `scheduler`.

So here's the optimized pipeline:

```
jobs:
  build-app-images:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build images
        run: |
          docker build -t $API_IMAGE --build-arg user=martin --build-arg
uid=1000 -f ./api/Dockerfile .
          docker build -t $SCHEDULER_IMAGE --target=scheduler --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .
          docker build -t $WORKER_IMAGE --target=worker --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .
      - name: Push images
        run: |
          docker push $API_IMAGE
          docker push $SCHEDULER_IMAGE
          docker push $WORKER_IMAGE
```

I call the new job `build-app-images` and it builds three images and then pushes them. This job took 3 minutes to run which is a significant boost compared to the previous one:



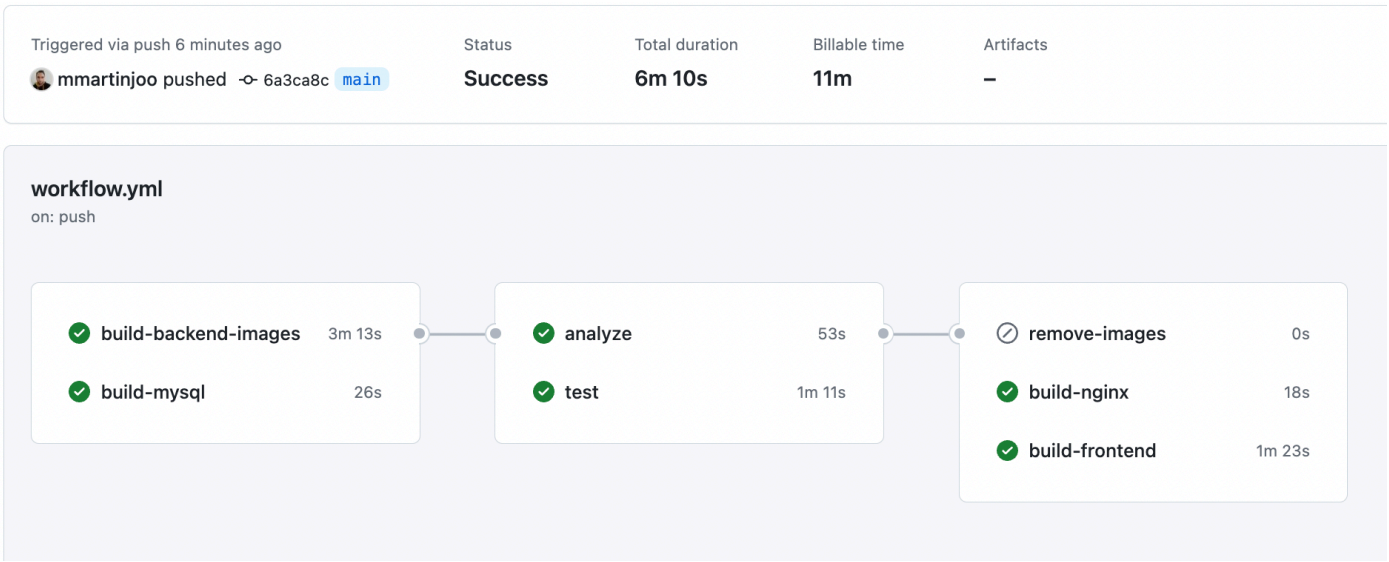
The whole pipeline takes 7 minutes to run. Don't worry about the `remove-images` job right now.

The test job first runs code analysis tools and then unit tests. If you think about it, we can make these parallels. It looks something like this:

```
analyze:
  needs: [build-backend-images, build-mysql]
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
    - uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKERHUB_USERNAME }}
        password: ${{ secrets.DOCKERHUB_TOKEN }}
    - name: Pull API image
      run: docker pull $API_IMAGE
    - name: Pull MySQL image
      run: docker pull $MYSQL_IMAGE
    - name: Run phpstan
      run: docker run --rm -t $API_IMAGE ./vendor/bin/phpstan analyze --
memory-limit=1G
    - name: Run phpinsights
      run: docker run --rm -t $API_IMAGE php artisan insights --no-
interaction --min-quality=90 --min-complexity=90 --min-architecture=90 --min-
style=90 --ansi --format=github-action
```

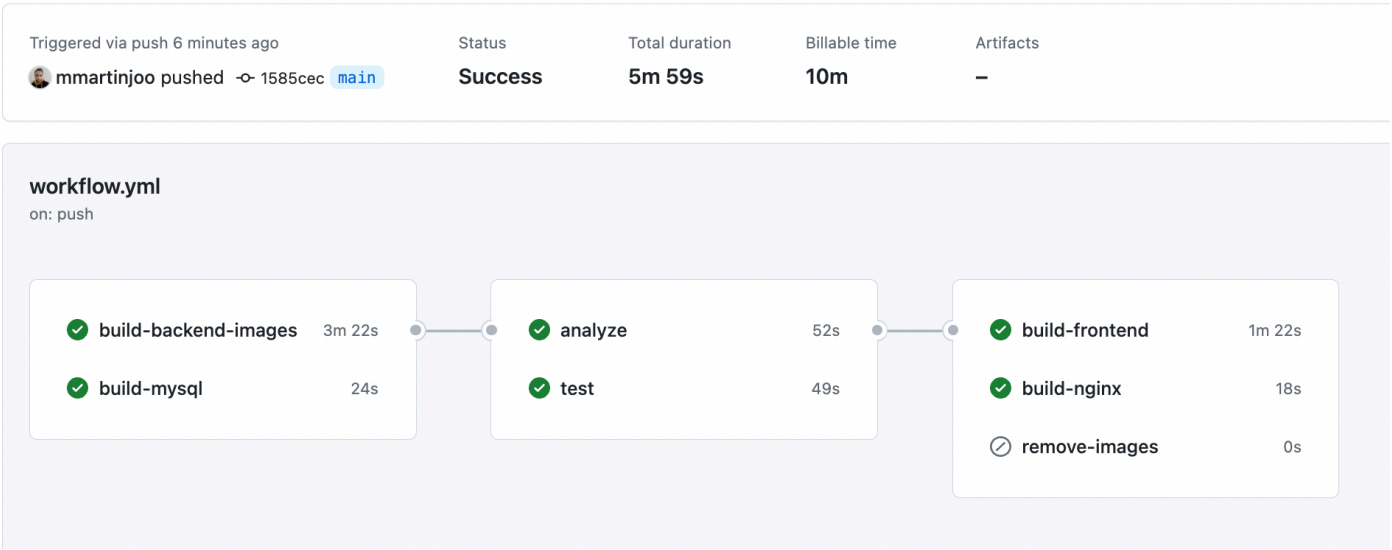
```yaml
test:
  needs: [ build-backend-images, build-mysql ]
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
    - uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKERHUB_USERNAME }}
        password: ${{ secrets.DOCKERHUB_TOKEN }}
    - name: Pull API image
      run: docker pull $API_IMAGE
    - name: Pull MySQL image
      run: docker pull $MYSQL_IMAGE
    - name: Run tests
      run: |
        docker-compose -f docker-compose.ci.yml up -d --force-recreate
        docker-compose -f docker-compose.ci.yml exec -T api php artisan test
        docker-compose -f docker-compose.ci.yml down
```

All that happened is that the steps are now separated into two different jobs. The whole pipeline takes 6 minutes to run:



Naming things is hard so now I'm using the `build-backend-images` name.

As you can see, this pipeline runs a `docker-compose down` after tests passed. As I said earlier, this is unnecessary when using GitHub runners. By removing it we can save ~10 extra seconds which is not bad for one small command:



So with a few changes, we went from the original 7 minutes to 6 minutes. Doesn't sound that much but it's 14%.

What about the `remove-images` job?

Right now, the following images are pushed to DockerHub regardless of the status of the tests:

- `api`
- `scheduler`
- `worker`
- `mysql`

This means even if the code is buggy, these images will be pushed. To prevent that, we can write a job that only if the `test` or the `analyze` job fails. If that happens we can easily remove the newly created tags from DockerHub using their API.

This is the job:

```
remove-images:
  needs: [ analyze, test ]
  runs-on: ubuntu-latest
  if: ${{ always() && contains(needs.*.result, 'failure') }}
  steps:
    - uses: actions/checkout@v3
    - name: Remove images
      run: |
        ./deployment/bin/remove-image.sh martinjoo/posts-api ${{ github.sha
}} ${{ secrets.DOCKERHUB_USERNAME }} ${{ secrets.DOCKERHUB_PASSWORD }}
        ./deployment/bin/remove-image.sh martinjoo/posts-mysql ${{ github.sha
}} ${{ secrets.DOCKERHUB_USERNAME }} ${{ secrets.DOCKERHUB_PASSWORD }}
        ./deployment/bin/remove-image.sh martinjoo/posts-worker ${{
github.sha }} ${{ secrets.DOCKERHUB_USERNAME }} ${{
secrets.DOCKERHUB_PASSWORD }}
        ./deployment/bin/remove-image.sh martinjoo/posts-scheduler ${{
github.sha }} ${{ secrets.DOCKERHUB_USERNAME }} ${{
secrets.DOCKERHUB_PASSWORD }}
```

In a GitHub workflow, we can use conditionals to trigger jobs only if a specific condition is true. This condition `if: ${{ always() && contains(needs.*.result, 'failure') }}` will result in true only if the `test` or the `analyze` job failed.

If that happens we can run a simple shell script that calls the DockerHub API and removes the new tags (the current SHA):

```
#!/bin/bash

set -e

IMAGE=$1
TAG=$2
DOCKERHUB_USERNAME=$3
DOCKERHUB_PASSWORD=$4

curl -s -X POST \
  -H "Accept: application/json" \
  -H "Content-Type: application/json" \
```

```
  -d "{\"username\":\"$DOCKERHUB_USERNAME\",
\"password\":\"$DOCKERHUB_PASSWORD\"}" \
  https://hub.docker.com/v2/users/login \
  -o response.json

token=$(jq -r '.token' response.json)

curl -i -X DELETE \
  -H "Accept: application/json" \
  -H "Authorization: JWT $token" \
  https://hub.docker.com/v2/repositories/martinjoo/$IMAGE/tags/$TAG
```

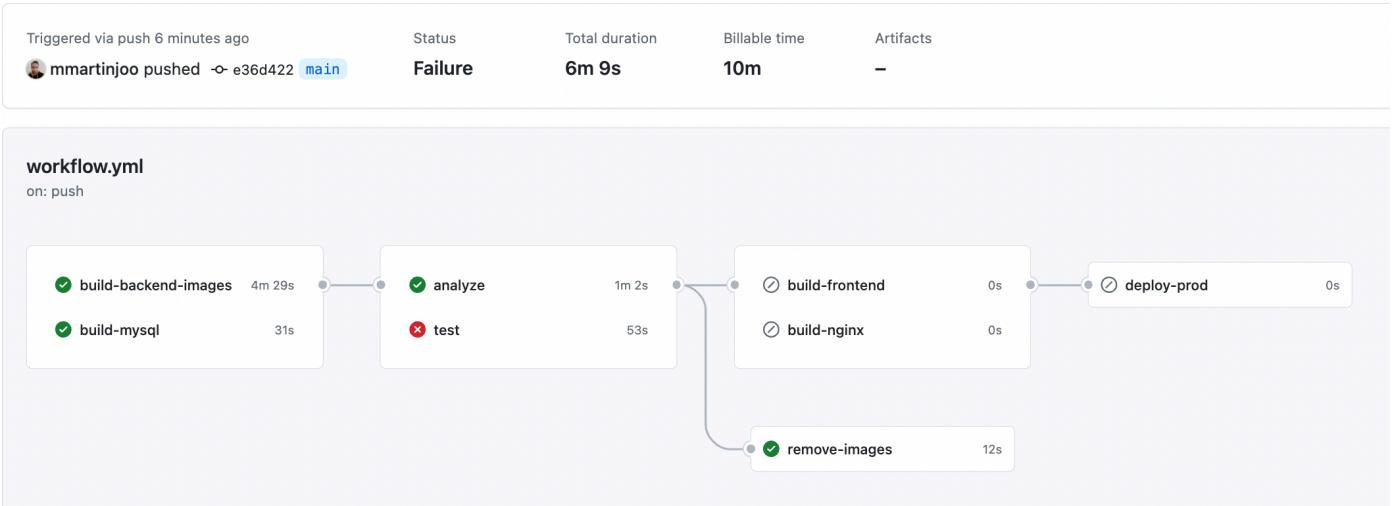
It's basically just two simple `curl` commands.

The only interesting part is `jq`. It's a CLI tool to parse JSON and run query selectors against it. In this example, the `response.json` contains a simple JSON such as this:

```
{
  "token": "abc-123"
}
```

To get the value of the token key we can use this command: `jq -r '.token' response.json`

And this is what it looks like in action:



Since `build-frontend` and `build-nginx` both depend on `analyze` and `test` they will not run when the tests fail. Instead, `remove-images` will run and remove the image tags from DockerHub.

Of course, another alternative would be to not use the latest images in the `docker-compose.ci.yml` but build images on-the-fly instead. But I don't really like that idea for one reason: I simply want to test **the exact** images I'm about to ship into production.

Important note. I only use the `remove-images` job because how GitHub runners work. As I said earlier, every job runs on a new machine. This means when you build an image in the first job, the `test` job won't be able to use that image because it does not exist on that machine.

There are two other solutions to avoid this `remove-images` job:

- Use your own runner where every runs on the same machine. This way, `build-backend-images` and `build-mysql` can build images and tag them but no need to push them. The test job can use the exact same images from the local image repository. If it fails, we don't need to do anything. The images are never pushed to DockerHub.
- If you use GitHub owned runners you have another option. In the `docker-compose.ci.yml` you can just build images directly. So instead of using an image name and tag, you can do this:

```
api:
  build:
    args:
      user: martin
      uid: 1000
    context: .
    dockerfile: ./api/Dockerfile
    target: api
```

So the `test` job builds images locally and use them. This way, `test` and `analyze` would be the first jobs to run. If they pass we can build the images and push them. No need for the `remove-images` job.

Production-ready docker-compose

Finally, let's see the production-ready compose config! It's not that different from the development one.

Here's the proxy, for example:

```
proxy:
  image: martinjoo/posts-proxy:${IMAGE_TAG}
  restart: unless-stopped
  ports:
    - "80:80"
  depends_on:
    - frontend
    - nginx
```

We use the image from the image registry and publish port 80.

This is what the api looks like:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "./wait-for-it.sh mysql:3306 -t 30 && ./wait-for-it.sh
redis:6379 -t 30 && php-fpm"
  restart: unless-stopped
  volumes:
    - ./.env:/usr/src/.env
    - type: volume
      source: storage
      target: /usr/src/storage
  depends_on:
    - migrate
    - mysql
    - redis
```

One difference compared to the development config is that there's this new syntax in the `volumes` key.

Docker named volumes vs bind mounts

There are two kinds of volumes in docker:

- Named volumes
- Bind mounts

So far we've been using bind mounts:

```
volumes:
  - ./api:/usr/src
```

Bind mounts are directories or files on the host machine that are mounted into a container. With bind mounts, you can directly access files and directories from the host filesystem within the container. Any changes made to the files or directories from within the container will also be reflected on the host, and vice versa. We can basically sync files between the host and the container. It's a great tool for developing.

Named volumes are created and managed by Docker itself. You can specify a name for the volume when creating a container or let Docker generate one for you. Docker ensures that the **data in the named volume is preserved** even if no containers are currently using it. Named volumes are independent of container lifecycles and can be shared among multiple containers.

A named volume is basically just a folder in the container that is being stored on the host machine as well so it's preserved. Docker will copy these named volumes from the container into the `/var/lib/docker/volumes` folder.

All we need is to name the volume and define the path inside the container:

```
volumes:
  - type: volume
    source: storage
    target: /usr/src/storage
```

This means a volume named `storage` will be created and the contents of the volume are the `/usr/src/storage` folder inside the `api` container.

If you go to `/var/lib/docker` you should see folders like these:

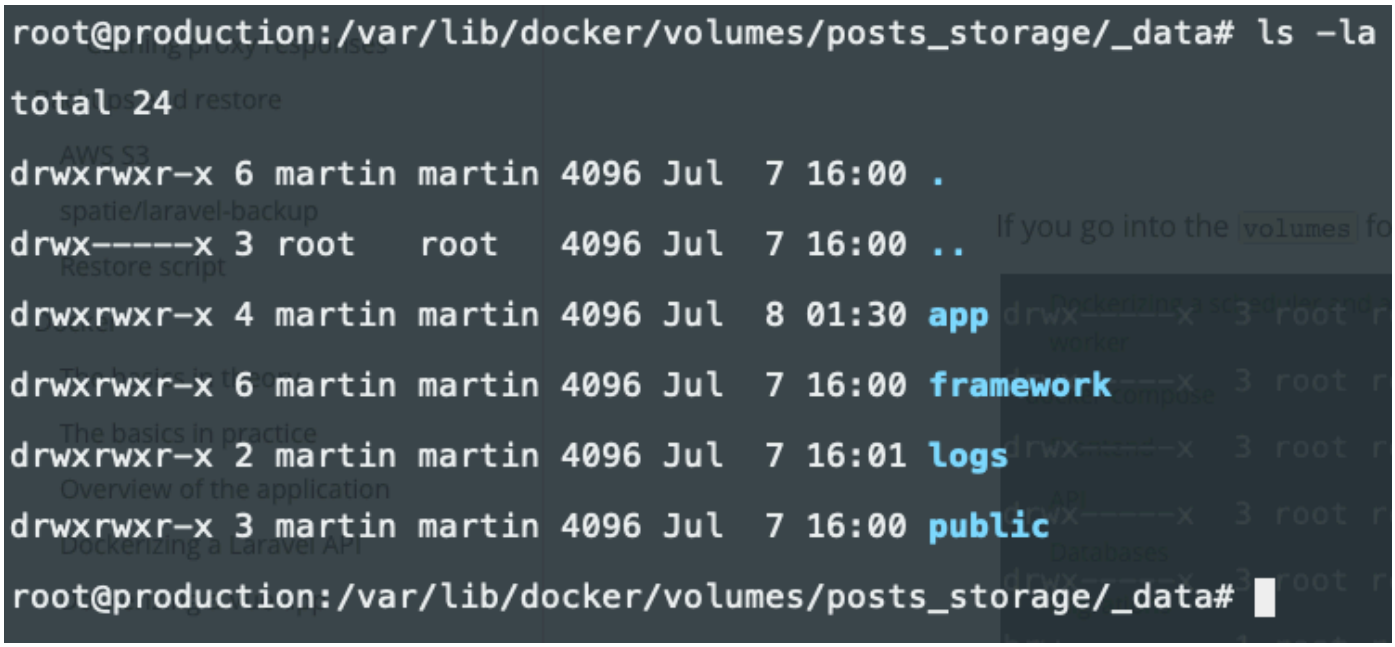
```
root@production:/var/lib/docker# ls -la
total 88
drwx--x---  12 root root  4096 Jul  6 20:07 .
drwxr-xr-x  41 root root  4096 May 11 08:40 ..
drwx--x--x   4 root root  4096 May 11 08:40 buildkit
drwx--x---  10 root root  4096 Jul  8 13:47 containers
-rw-------   1 root root    36 May 11 08:40 engine-id
drwx------   3 root root  4096 May 11 08:40 image
drwxr-x---   3 root root  4096 May 11 08:40 network
drwx--x--- 155 root root 36864 Jul  8 13:47 overlay2
drwx------   4 root root  4096 May 11 08:40 plugins
drwx------   2 root root  4096 Jul  6 20:07 runtimes
drwx------   2 root root  4096 May 11 08:40 swarm
drwx------   2 root root  4096 Jul  8 13:47 tmp
drwx-----x  22 root root  4096 Jul  7 16:00 volumes
root@production:/var/lib/docker#
```

If you go into the `volumes` folder you should see a bunch of folders:

```
drwx-----x  3 root root   4096 Jul  6 19:33 6cab01979ece635dc918794a9e5eeb959a02e2aa5a0b2e594d4ed23a2c06fcf5
drwx-----x  3 root root   4096 Jul  6 12:00 8989494bee9ffadcb2f15e917148b260a4170cc6a2b0bf8fad1851c7f979e95e
drwx-----x  3 root root   4096 Jul  6 19:25 8eb4dc33a8ec577f9c89a8c3c48c2e07efcbd640644b8377cbb1fe320c2d6ec3
drwx-----x  3 root root   4096 Jul  6 19:06 914c1bf86b717b90a610777498d5e2e569c77b77a454bfea52faf38b87dbdb84
drwx-----x  3 root root   4096 Jul  6 18:07 9a6be200cf32c27dcf1f584a79d1628891fc52ef3da2dfabde8f02e370d54b33
brw-------  1 root root 252, 1 Jul  6 20:07 backingFsBlockDev
drwx-----x  3 root root   4096 Jul  6 19:45 c90670f28ddc4c11afee96b98c8f67ddb9ab10103e6b0d7a419e59bf6b081126
drwx-----x  3 root root   4096 Jul  6 17:43 cd197cd0287ef2dc91c926ef6e042357c5c67f44596393df2b1b200d1be94f66
drwx-----x  3 root root   4096 Jul  6 18:28 db82ae06e51c813c9b7ca438faca014d3409a6d8b156d78bdb6e15ce4426599b
drwx-----x  3 root root   4096 Jul  6 17:30 fe79c935d7852060371a62e0df9244c4943e75e6cf089f205756440581181613
drwx-----x  3 root root   4096 Jul  6 18:07 ff62479685279e26fb00f6adeed2adf19bcdca4bf4cbb18bc9f95e16b58c76da
-rw-------  1 root root  65536 Jul  7 16:00 metadata.db
drwx-----x  3 root root   4096 Jul  7 16:00 posts_mysqldata
drwx-----x  3 root root   4096 Jul  7 16:00 posts_storage
```

The last folder's name is `posts_storage`. This is the volume I just showed you.

If you go into it you should see pretty familiar stuff:



This is the `storage` folder of our Laravel API. As I said, docker copies the folder from the container.

The reason I'm creating a named volume for the storage folder is because I want to preserve user-generated content, such as file uploads.

To make this work, we need another key at the end of the `docker-compose.yml` file:

```
volumes:
  storage:
  mysqldata:
  redisdata:
```

We need to list the volume names at the end of the file.

I also mount the `.env` file:

```
volumes:
  - ./.env:/usr/src/.env
```

There's no `.env` file in the container but obviously, it needs one. I'm using a bind mount so it's synced between the host and the container. It can be useful if you want to quickly change something. Changes in the `.env` file are immediately picked up by Laravel thanks to this volume.

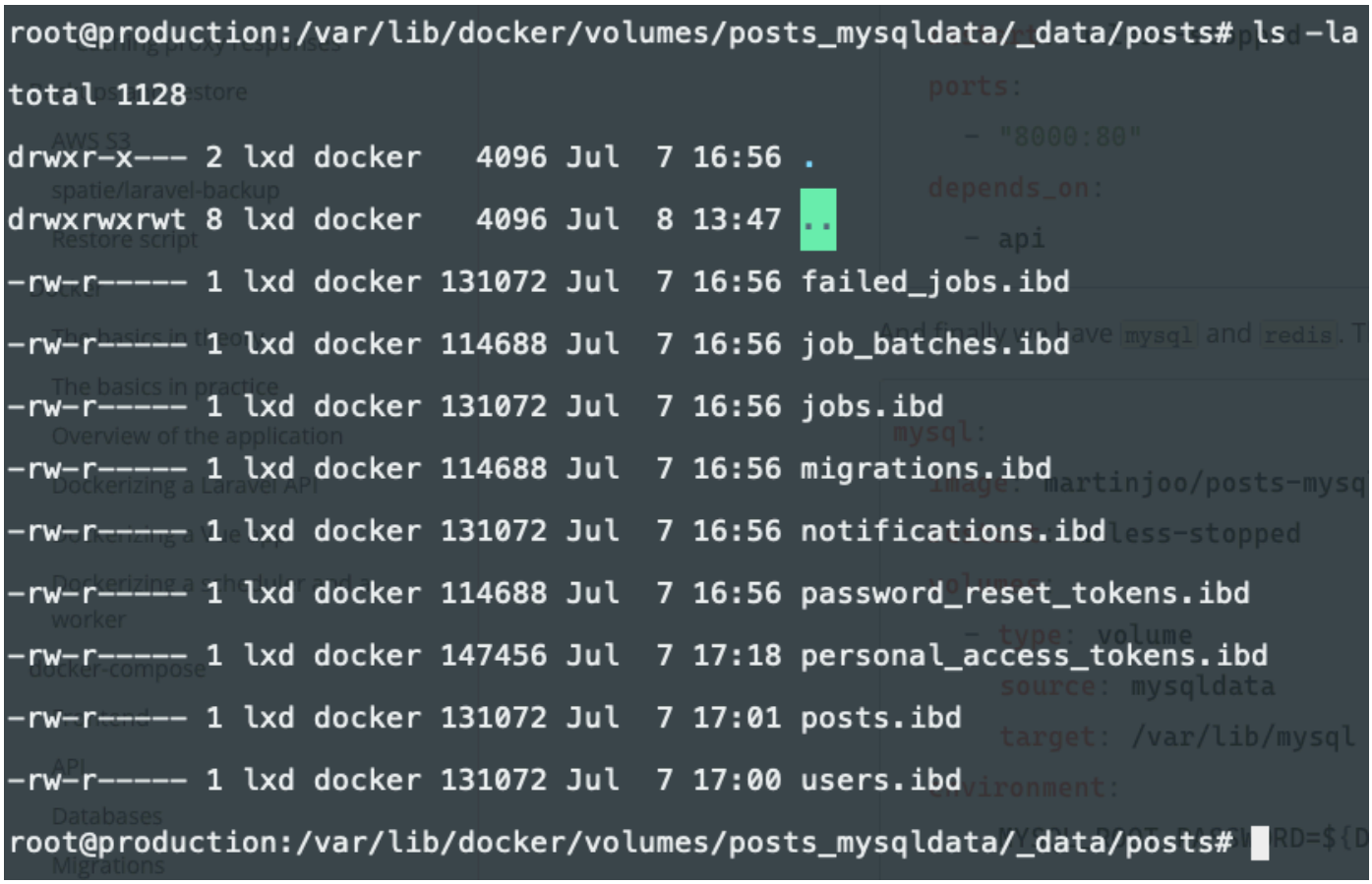
The config for `scheduler` and `worker` is basically the same as for the `api` but with different images. `nginx` is also pretty straightforward:

```
nginx:
  image: martinjoo/posts-nginx:${IMAGE_TAG}
  restart: unless-stopped
  ports:
    - "8000:80"
  depends_on:
    - api
```

And finally, we have `mysql` and `redis`. These containers contain state (data) so we need named volumes as well:

```
mysql:
  image: martinjoo/posts-mysql:${IMAGE_TAG}
  restart: unless-stopped
  volumes:
    - type: volume
      source: mysqldata
      target: /var/lib/mysql
  environment:
    - MYSQL_ROOT_PASSWORD=${DB_PASSWORD}

redis:
  image: redis:7.0.11-alpine
  restart: unless-stopped
  volumes:
    - type: volume
      source: redisdata
      target: /data
```

Thanks to the `mysqldata` we have permanent data on the host machine. All the tables and database-related files can be found in `/var/lib/docker`:



And that's it! We have a production-ready docker-compose config. You can check out the `docker-compose.prod.yml` file to see the whole configuration. And now, let's ship it!

Deployment

Deploy script

Do you remember the deploy script from the first chapter? It was 55 lines. Check out the new `deploy.sh`:

```
#!/bin/bash

set -e

cd /usr/src

sudo docker-compose -f docker-compose.prod.yml down
sudo docker-compose -f docker-compose.prod.yml up -d
```

That's it! It's just a `docker-compose down` and then `up`. We stop the previous stack with the old images and start the new ones. If you remember there has to be a `.env` file on the server with the `IMAGE_TAG` set to the current version. In a minute, I'll show you how this file gets there (another 5 lines of code).

But first, let's discuss why we can't use `docker-compose restart`. It restarts all services as the name suggests. However, it doesn't care about changes in your `docker-compose.yml` file which is a problem in our case since the compose file contains the new images. So we need to bring the whole stack down and then start it.

Deploying from a pipeline

The next step is to add a new job to the pipeline:

```
deploy-prod:
  needs: [ build-frontend, build-nginx ]
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
    - name: Copy SSH key
      run: |
        echo "${{ secrets.SSH_KEY }}" >> ./id_rsa
        chmod 600 id_rsa
    - name: Deploy app
      run: |
        scp -i ./id_rsa ./deployment/bin/deploy.sh ${{
secrets.SSH_CONNECTION_PROD }}:/home/martin/deploy.sh
        scp -i ./id_rsa ./docker-compose.prod.yml ${{
secrets.SSH_CONNECTION_PROD }}:/usr/src/docker-compose.prod.yml
        scp -i ./id_rsa ./.env.prod.template ${{ secrets.SSH_CONNECTION_PROD
}}:/usr/src/.env
        ssh -tt -i ./id_rsa ${{ secrets.SSH_CONNECTION_PROD }} "chmod +x
/home/martin/deploy.sh"
        ssh -tt -i ./id_rsa ${{ secrets.SSH_CONNECTION_PROD }} "
          sed -i "/IMAGE_TAG/c\IMAGE_TAG=${{ github.sha }}" /usr/src/.env
          sed -i "/DB_PASSWORD/c\DB_PASSWORD=${{ secrets.DB_PASSWORD }}"
/usr/src/.env
          sed -i "/AWS_ACCESS_KEY_ID/c\AWS_ACCESS_KEY_ID=${{
secrets.AWS_ACCESS_KEY_ID }}" /usr/src/.env
          sed -i "/AWS_SECRET_ACCESS_KEY/c\AWS_SECRET_ACCESS_KEY=${{
secrets.AWS_SECRET_ACCESS_KEY }}" /usr/src/.env
          sed -i "/APP_KEY/c\APP_KEY=${{ secrets.APP_KEY }}" /usr/src/.env
          sed -i "/COMPOSE_PROJECT_NAME/c\COMPOSE_PROJECT_NAME=posts"
/usr/src/.env"
        ssh -tt -o StrictHostKeyChecking=no -i ./id_rsa ${{
secrets.SSH_CONNECTION_PROD }} "/home/martin/deploy.sh"
```

It's pretty similar to the one we wrote in the first chapter. Here are the steps:

- It copies an SSH key from GitHub secrets to the runner.
- Using this key it copies the deploy script to the production server using another secret called `SSH_CONNECTIOn_PROD`. It puts the script to `/home/martin/deploy.sh`
- It copies the `docker-compose.prod.yml` to the production server. The exact location is `/usr/src/docker-compose.prod.yml`
- It copies the `.env.prod.template` file to the server as well. It puts the file into `/usr/src/.env`. This file contains every important environment variable that production needs. The extension `template` has no special meaning, I just use this because it feels a bit more important than `example`.
- It makes the `deploy.sh` executable on the production server.

Finally, it runs the following script:

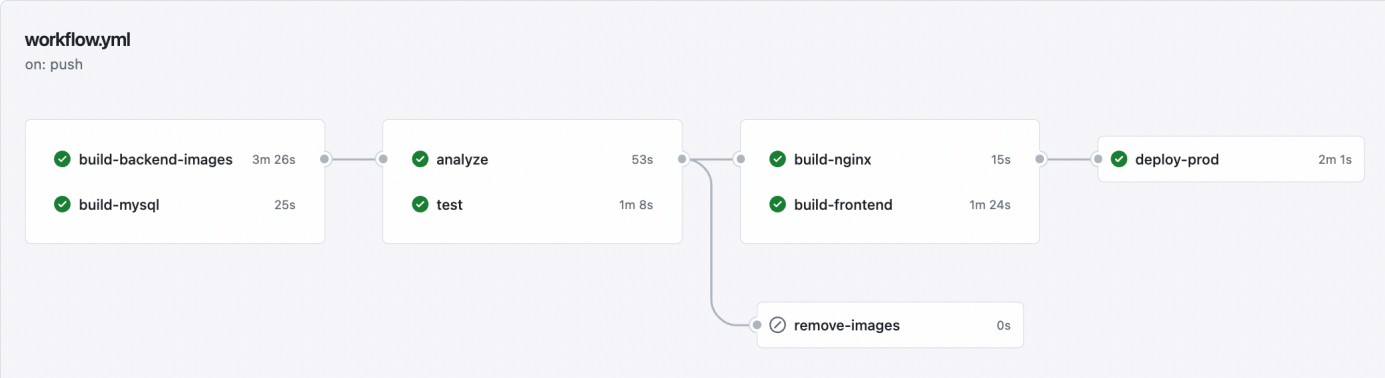
```
sed -i "/IMAGE_TAG/c\IMAGE_TAG=${{ github.sha }}" /usr/src/.env
sed -i "/COMPOSE_PROJECT_NAME/c\COMPOSE_PROJECT_NAME=posts" /usr/src/.env"

sed -i "/DB_PASSWORD/c\DB_PASSWORD=${{ secrets.DB_PASSWORD }}" /usr/src/.env
sed -i "/AWS_ACCESS_KEY_ID/c\AWS_ACCESS_KEY_ID=${{ secrets.AWS_ACCESS_KEY_ID
}}" /usr/src/.env
sed -i "/AWS_SECRET_ACCESS_KEY/c\AWS_SECRET_ACCESS_KEY=${{
secrets.AWS_SECRET_ACCESS_KEY }}" /usr/src/.env
sed -i "/APP_KEY/c\APP_KEY=${{ secrets.APP_KEY }}" /usr/src/.env
```

We have seen similar things before. `sed` will replace these environment variables with values coming from secrets. There are two special ones:

- `IMAGE_TAG` is not needed by the application itself but the `docker-compose.yml` file. This is how we get the newest version number to the server.
- `COMPOSE_PROJECT_NAME` is needed by docker-compose. I mean, it's not needed, but if you don't set it docker-compose will use the current folder name as the prefix in container names. I'm using the `/usr/src` folder so container names would look like `src_api_1` and so on. By setting this env to `posts` it will spin up containers such as `posts_api_1` etc.

You can absolutely extract these lines to a script, for example, `prepare_env.sh` or something like that. After preparing the `.env` file the pipeline calls the deploy script.

After these steps you should be able to deploy your containers:



And the containers should be running:

```
$ docker-compose ps
NAME                IMAGE                                                COMMAND                  SERVICE             CREATED             STATUS              PORT
S
posts-api-1         martinjoo/posts-api:596d11b6310cfce71a1394d9278cbcdfb1199324         "docker-php-entrypoi…"   api                 2 hours ago         Up 2 hours          9000
/tcp
posts-frontend-1    martinjoo/posts-frontend:596d11b6310cfce71a1394d9278cbcdfb1199324    "/docker-entrypoint.…"   frontend            2 hours ago         Up 2 hours          0.0.
0.0:80->80/tcp, :::80->80/tcp
posts-mysql-1       martinjoo/posts-mysql:596d11b6310cfce71a1394d9278cbcdfb1199324       "docker-entrypoint.s…"   mysql               2 hours ago         Up 2 hours          3306
/tcp, 33060/tcp
posts-nginx-1       martinjoo/posts-nginx:596d11b6310cfce71a1394d9278cbcdfb1199324       "/docker-entrypoint.…"   nginx               2 hours ago         Up 2 hours          0.0.
0.0:8000->80/tcp, :::8000->80/tcp
posts-redis-1       redis:7.0.11-alpine                                  "docker-entrypoint.s…"   redis               2 hours ago         Up 2 hours          6379
/tcp
posts-scheduler-1   martinjoo/posts-scheduler:596d11b6310cfce71a1394d9278cbcdfb1199324   "docker-php-entrypoi…"   scheduler           2 hours ago         Up 29 seconds       9000
/tcp
posts-worker-1      martinjoo/posts-worker:596d11b6310cfce71a1394d9278cbcdfb1199324      "docker-php-entrypoi…"   worker              2 hours ago         Up 2 hours          9000
/tcp
```

Provisioning new servers

Do you remember the provision script from the first chapter? It was 81 lines. Check out the new `provision_server.sh`:

```
#!/bin/bash

set -e

# Install docker-compose
sudo curl -L
"https://github.com/docker/compose/releases/download/v2.19.1/docker-compose-
linux-x86_64" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

Once again, I'm using the Docker marketplace image on DigitalOcean so the server already has Docker installed on it. We basically just need to install docker-compose.

And of course, we can create a user as well so we don't need to login as `root`:

```
#!/bin/bash

set -e

SSH_KEY=$1

useradd -G www-data,root,sudo,docker -u 1000 -d /home/martin martin
mkdir -p /home/martin/.ssh
touch /home/martin/.ssh/authorized_keys
chown -R martin:martin /home/martin
chown -R martin:martin /usr/src
chmod 700 /home/martin/.ssh
chmod 644 /home/martin/.ssh/authorized_keys
echo "$SSH_KEY" >> /home/martin/.ssh/authorized_keys

echo "martin ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers.d/martin
```

```
# Install docker-compose
sudo curl -L
"https://github.com/docker/compose/releases/download/v2.19.1/docker-compose-
linux-x86_64" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

We've already discussed these commands. The only thing worth noting is that the script accepts an SSH public key as its only argument and then pastes it in the `authorized_keys` file. Basically, I want to authorize my own key from my local machine for the new user so I can do this `ssh martin@1.2.3.4`

One of the great things about cloud providers such as DigitalOcean is that they provide CLI tools to create and manage resources. So we can write a pretty basic script that will spin up new docker machines for us. So we don't even need to open a browser to create a staging or demo server.

To be able to run this script you need `doctl` on your machine:

```
#!/bin/bash

set -ex

NAME=$1
SIZE=${2:-s-1vcpu-1gb}

SSH_FINGERPRINT_DEPLOY=$(doctl compute ssh-key list --no-header | grep
"devops-with-laravel-deploy" | awk '{ print $3 }')
SSH_FINGERPRINT_OWN=$(doctl compute ssh-key list --no-header | grep "MacBook
Air" | awk '{ print $4 }')

PUBLIC_KEY=$(cat $HOME/.ssh/id_rsa.pub)

OUTPUT=$(doctl compute droplet create --image docker-20-04 --size s-1vcpu-1gb
--region nyc1 --ssh-keys $SSH_FINGERPRINT_DEPLOY --ssh-keys
$SSH_FINGERPRINT_OWN --no-header $NAME)

DROPLET_ID=$(echo $OUTPUT | awk '{ print $1 }')

sleep 120
```

```
doctl projects resources assign cad78098-bfb8-4861-bfb2-e969cee18f16 --
resource=do:droplet:$DROPLET_ID

SERVER_IP=$(doctl compute droplet get $DROPLET_ID --format PublicIPv4 --no-
header)

scp -C -o StrictHostKeyChecking=no -i $HOME/.ssh/id_ed25519
$HOME/.ssh/id_ed25519 root@$SERVER_IP:~/.ssh/id_rsa
scp -C -o StrictHostKeyChecking=no -i $HOME/.ssh/id_ed25519
./provision_server.sh root@$SERVER_IP:./provision_server.sh
ssh -tt -o StrictHostKeyChecking=no -i $HOME/.ssh/id_ed25519 root@$SERVER_IP
"chmod +x ./provision_server.sh && ./provision_server.sh \"$PUBLIC_KEY\""
```

I'm not going to go through the details since this script is not that crucial but here's what's happening in a nutshell:

- I have added my SSH keys to the DigitalOcean UI. There's one for automatic deployments from the pipeline and there's another one which is for me to ssh into servers manually from my machine. The script gets the fingerprints of these keys from the DO API.
- It creates a new server (droplet) and passes these ssh fingerprints to the DO API. So DO enables the keys.
- It waits for 120 seconds to make sure the server is created. Usually, it takes only about 30-40 seconds.
- After the droplet is created it assigns it to a project. It's just a convenient thing to organize my servers.
- After that it gets the droplet's IP address.
- And then it copies and runs the `provision_server.sh` on the server with my public key. Passing the fingerprint to the `create` command only allows the key for the `root` user. But in the provision I create a new user, so it also needs my ssh key.

update service

There's one thing I didn't show you from the deploy script. It runs three additional commands so in fact it looks like this:

```
sudo docker-compose -f docker-compose.prod.yml down
sudo docker-compose -f docker-compose.prod.yml up -d

sudo docker-compose -f docker-compose.prod.yml exec -T api php artisan
config:cache
sudo docker-compose -f docker-compose.prod.yml exec -T api php artisan
route:cache
sudo docker-compose -f docker-compose.prod.yml exec -T api php artisan
view:cache
```

It caches configs, routes, and views. Just as we did in the first chapter. That's completely fine but it raises a question: migrations are run by a docker-compose service but the cache is done by the deploy script?

It's a good question. We can add a little bit more responsibility to the migrate container and call it the update container:

```
update:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "./wait-for-it.sh mysql:3306 -t 30 && php /usr/src/artisan
migrate --force && php artisan config:cache && php artisan route:cache && php
artisan view:cache"
  restart: no
  volumes:
    - ./.env:/usr/src/.env
  depends_on:
    - mysql
```

But of course, it looks a bit busy so let's create an `update.sh` in the `deployment` folder:

```
php artisan down

php artisan migrate --froce

php artisan config:cache
php artisan route:cache
php artisan view:cache

php artisan up
```

The file gets copied into the image when the pipeline builds it and `docker-compose.yml` looks like this:

```
update:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "./wait-for-it.sh mysql:3306 -t 30 && ./update.sh"
  restart: no
  volumes:
    - ./.env:/usr/src/.env
  depends_on:
    - mysql
```

After starting the stack it gives us the following output:

```
→ docker git:(main) ✗ dc logs -f --tail=200 update
docker-update-1  | wait-for-it.sh: waiting 30 seconds for mysql:3306
docker-update-1  | wait-for-it.sh: mysql:3306 is available after 8 seconds
docker-update-1  |
docker-update-1  |
docker-update-1  |    INFO  Application is now in maintenance mode.
docker-update-1  |
docker-update-1  |
docker-update-1  |
docker-update-1  |
docker-update-1  |    INFO  Nothing to migrate.
docker-update-1  |
docker-update-1  |
docker-update-1  |
docker-update-1  |
docker-update-1  |    INFO  Configuration cached successfully.
docker-update-1  |
docker-update-1  |
docker-update-1  |
docker-update-1  |
docker-update-1  |    INFO  Routes cached successfully.
```

Since the file is located in the `deployment` folder we need to explicitly copy it into the `Dockerfile`:

```
COPY ./deployment/bin/update.sh /usr/src/update.sh
```

Congratulations! You just shipped a few docker containers in a docker-compose stack to production via a fully automated CI/CD pipeline.

Restore

Fortunately, the restore process is almost the same as in the first part of the book. There are only two slight changes.

The restore script the `aws` CLI tool to download backups so it needs to be installed in the image:

```
RUN curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip" && \
    unzip awscliv2.zip && \
    ./aws/install
```

The commands are the same as before.

The last change is to add `docker-compose` to the `restore.sh` itself. For example, downloading the backup from S3 looked like this before:

```
aws s3 cp s3://devops-with-laravel-backups/$BACKUP_FILENAME
$PROJECT_DIR"/storage/app/backup.zip"
```

A simple `aws` command that runs on the production server. Now the host machine doesn't have `aws` so we need to run these commands inside the container:

```
docker-compose exec api aws s3 cp s3://devops-with-laravel-
backups/$BACKUP_FILENAME $PROJECT_DIR"/storage/app/backup.zip"
```

That's the only difference. The whole process and every command is the same as before.

Rollback

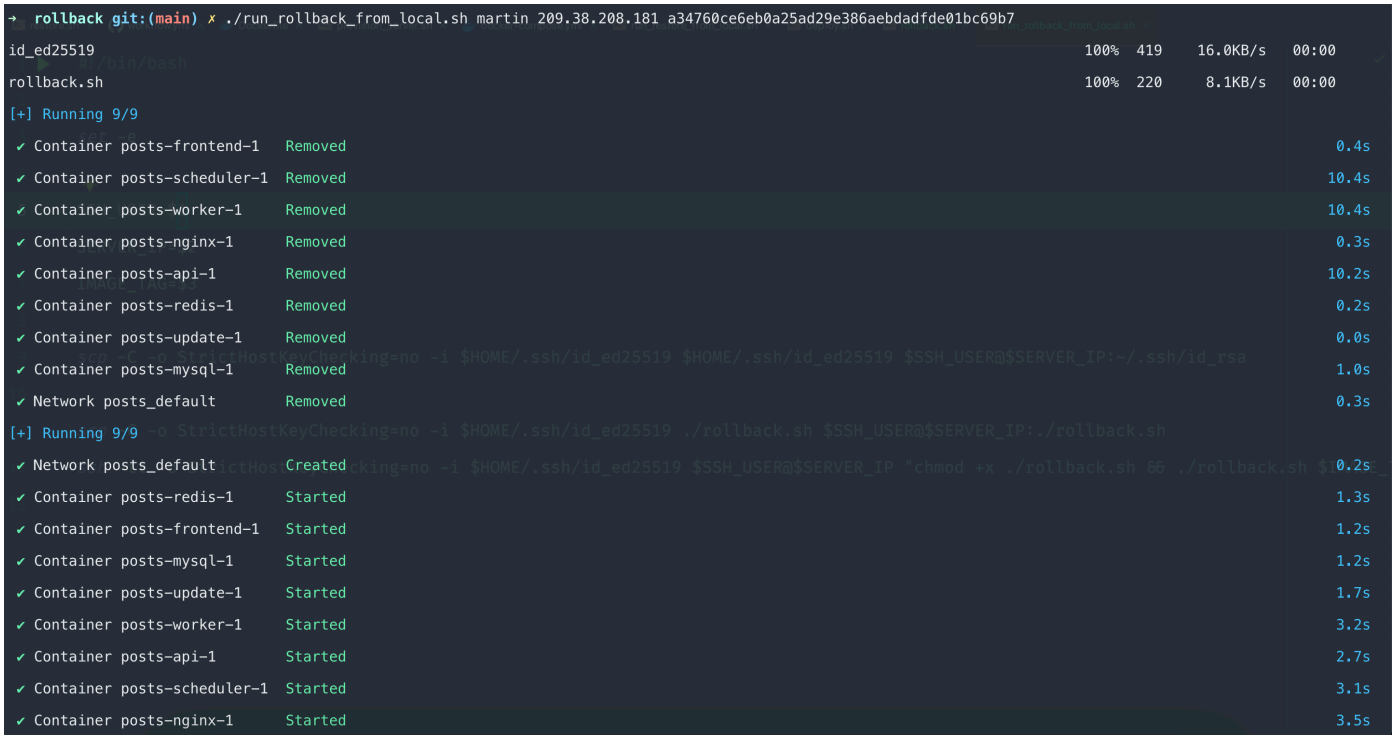
Sometimes when sh*t happens we need to rollback the application to a previous deployment. Fortunately, using commit SHAs as image versions it's a pretty simple process. All we need to do is:

- Set the `IMAGE_TAG` environment variable in the `.env` to the version we want to rollback to
- Bring the stack down
- Start the stack

The script looks like this:

```
#!/bin/bash

TAG=$1

sed -i "/IMAGE_TAG/c\IMAGE_TAG=$TAG" /usr/src/.env

sudo docker-compose -f docker-compose.prod.yml down
sudo docker-compose -f docker-compose.prod.yml up -d --remove-orphans
```

That's the whole script. You can find it in the `deployment/rollback` folder. I also created a `run_rollback_from_local.sh` script that you can use and here's the output:



First, it brings down the compose stack then it starts the container. And here you can see the different versions.

`f309` was the current version when I ran the script. After a few seconds, it started the `a347` images:

```
root@production:/usr/src# docker-compose ps
NAME                IMAGE                                                COMMAND                  SERVICE             CREATED             STAT
US                  PORTS
posts-api-1         martinjoo/posts-api:f3096ff5f1c9f40686d5596281f21dbb50ca19a9         "docker-php-entrypoi…"   api                 39 minutes ago      Up 3
9 minutes           9000/tcp
posts-frontend-1    martinjoo/posts-frontend:f3096ff5f1c9f40686d5596281f21dbb50ca19a9    "/docker-entrypoint.…"   frontend            39 minutes ago      Up 3
9 minutes           0.0.0.0:80->80/tcp, :::80->80/tcp
posts-mysql-1       martinjoo/posts-mysql:f3096ff5f1c9f40686d5596281f21dbb50ca19a9       "docker-entrypoint.s…"   mysql               39 minutes ago      Up 3
9 minutes           3306/tcp, 33060/tcp
posts-nginx-1       martinjoo/posts-nginx:f3096ff5f1c9f40686d5596281f21dbb50ca19a9       "/docker-entrypoint.…"   nginx               39 minutes ago      Up 3
9 minutes           0.0.0.0:8000->80/tcp, :::8000->80/tcp
posts-redis-1       redis:7.0.11-alpine                                  "docker-entrypoint.s…"   redis               39 minutes ago      Up 3
9 minutes           6379/tcp
posts-scheduler-1   martinjoo/posts-scheduler:f3096ff5f1c9f40686d5596281f21dbb50ca19a9   "docker-php-entrypoi…"   scheduler           39 minutes ago      Up 3
4 seconds           9000/tcp
posts-worker-1      martinjoo/posts-worker:f3096ff5f1c9f40686d5596281f21dbb50ca19a9      "docker-php-entrypoi…"   worker              39 minutes ago      Up 3
9 minutes           9000/tcp
root@production:/usr/src# docker-compose ps
NAME                IMAGE                                                COMMAND                  SERVICE             CREATED             STAT
US                  PORTS
posts-api-1         martinjoo/posts-api:a34760ce6eb0a25ad29e386aebdadfde01bc69b7         "docker-php-entrypoi…"   api                 40 seconds ago      Up 3
7 seconds           9000/tcp
posts-frontend-1    martinjoo/posts-frontend:a34760ce6eb0a25ad29e386aebdadfde01bc69b7    "/docker-entrypoint.…"   frontend            40 seconds ago      Up 3
```

Of course, the database schema is still the newest one. In my opinion, it's a bit risky to try to automate database rollbacks with scripts. If you want to go back to a previous schema, just run `php artisan migrate:rollback` manually after you ran the `rollback` script.

Automatic image updates

What if I told you that we can create a fully automatic deployment process with 7 lines of config? By fully automatic, I mean:

- You push a new commit to main
- The pipeline builds the new images
- Containers are updated on the production server
- **All of this without the deploy script and the deploy-prod job**

It's possible with Watchtower. It's a package that does the following:

- You add it to your docker-compose stack
- It detects what images you are running
- Every few minutes it sends requests to DockerHub (or your registry of choice) and checks if there is a new version of the images.
- If there is a new version it pulls the image
- Then it gracefully stops your existing containers and starts the new ones

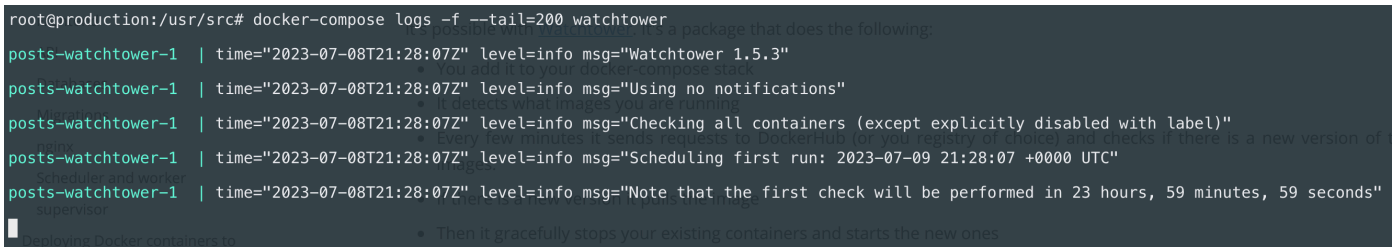
Important note! Watchtower only works with "rolling" image tags. So it has to be something like *latest* or *stable*. When you run *myimage:1.0* and you push *myimage:1.1* Wathtower won't udpate your image. But if you run *myimage:stable* and you push a new version of *stable* it detects the change (because the digest of the image changes) and updates your image.

It basically does the same as the deploy script and the deploy-prod job. This is the `docker-composer` service:

```
watchtower:
  image: containrrr/watchtower:1.5.3
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
  environment:
    - REPO_USER=${DOCKERHUB_USERNAME}
    - REPO_PASS=${DOCKERHUB_PASSWORD}
```

We need to connect the docker socket on the host machine to the container so Watchtower can query the images that are running on the host machine. It also takes the DockerHub username and password. I added these variables to the `.env.prod.template` file and added them to the `deploy-prod` job so they are updated on the server as well (you can do this manually too).

If everything goes well Watchtower creates these log entries:



The fastest way to test it is to change something in the code, build an image locally, and push it to DockerHub:

```
docker build -t martinjoo/posts-api:latest --target=api --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .

docker push martinjoo/posts-api:latest
```

Then the new `latest` should be running on your server after a few minutes.

One of the advantages of the `update` container we created is that it works with this deployment as well. If you push a new tag, the `update` container runs and updates the application. If we wrote these commands in shell as part of the pipeline it wouldn't work with this scenario.

There's another limitation to Watchtower. If you change `docker-compose.yml` itself the new changes won't be updated on your production server. Watchtower can only update images not config files or `.env` files.

Try it out, it can be a great addition to your toolset. I use Watchtower in solo projects where the workflow looks like this:

- I use two branches: develop and main. Locally, I'm working on develop.
- I push it to GitLab and the pipeline builds new images. These images are tagged as `latest`
- It runs the tests and then pushes the images into GitLab container registry.
- Watchtower updates the images on the staging server and the new version is out.
- When everything seems fine I rebase main to develop. Now another pipeline runs that tags the images as `stable`.
- Watchtower updates the images on the production server.

It's a simple and fast workflow.

GitFlow

The project files are located in the `4-docker-gitflow` folder.

The great thing about images and containers is that they are pretty flexible. It's relatively easy to implement GitFlow at the infrastructure level. What I mean is this:

- When you work on a feature branch and open a PR to develop the pipeline create a demo server running your code.
- When you merge a PR to develop or push a commit directly the pipeline updates the staging server.
- When you merge a PR to main or push commits directly, or rebase it the pipeline updates the production server.

So the infrastructure follows or supports the development flow. Let's think about what events the pipeline needs to handle.

Opening a feature branch PR to main is going to be a pull_request to main:

```
on:
  pull_request:
    branches: [ "develop" ]
```

Merging a feature branch into develop is going to be a push to develop:

```
on:
  push:
    branches: [ "develop" ]
```

And finally merging or rebasing develop to main is also going to be a push but to main:

```
on:
  push:
    branches: [ "main" ]
```

What do we want to happen when these events occur? Deploy either the staging or the production server.

So these two cases look pretty similar to me. We probably can handle them easily. So for now, let's just forget about feature branches and demo servers.

Pushing to develop or main

I'm going to create a new, dedicated workflow to handle only these events. Later, we're going to talk about reusing actions across workflows.

Let's summarize the jobs we already have, and think about if they need any changes at all:

```
name: Push

on:
  push:
    branches: [ "main", "develop" ]

build-backend-images: # No difference between main and develop
build-mysql: # No difference
analyze: # No difference
test: # No difference
remove-images: # No difference
build-nginx: # No difference
build-frontend: # No difference
build-proxy: # No difference
deploy: # Different servers and credentials
```

As you might be expected, building images or running tests do not have a difference at all. There is only one job we need to change. In `deploy` we need to execute the same script but with different variables.

This is what it looks like:

```
deploy-staging:
  needs: [ build-frontend, build-nginx ]
  if: github.ref == 'refs/heads/develop'
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
    - name: Copy SSH key
      run: |
        echo "${{ secrets.SSH_KEY }}" >> ./id_rsa
        chmod 600 id_rsa
    - name: Deploy app
```

```
      run: |
        scp -i ./id_rsa ./deployment/bin/deploy.sh ${{
secrets.SSH_CONNECTION_STAGING }}:/home/martin/deploy.sh
        ssh -tt -i ./id_rsa ${{ secrets.SSH_CONNECTION_STAGING }} "
          sed -i "/AWS_BUCKET/c\AWS_BUCKET=devops-with-laravel-staging-
backups" /usr/src/.env
        "
        ...
```

If `github.ref` is `refs/head/develop` it means we pushed to the develop branch. The `deploy-staging` job should run in only this case.

If the current branch is develop then we need to use staging-related variables in the deploy script such `SSH_CONNECTION_STAGING` and we need to set staging-related env variables such as the AWS bucket name going to be `devops-with-laravel-staging-backups`. If your project needs a mail server here you can set up Mailtrap for example.

Of course, the `deploy-prod` job checks if the branch is main:

```
deploy-prod:
  needs: [ build-frontend, build-nginx ]
  if: github.ref == 'refs/heads/main'
  runs-on: ubuntu-latest
```

And basically, that's it! With these two simple if statements we handled the deployment of staging and production.

Once again, this workflow runs when you push commits to develop or main.

Opening a PR to main

This is also going to be a separate workflow file.

Let's think about what should happen when want to merge develop into main. Do we need to deploy a server? In this case, no. Staging is already up-to-date and we don't want to deploy production. Do we need to push images? No. It's not a new version of the application. It's just an existing version (develop) requested to be merged into main.

So we actually only need to make sure that code is fine. Tests pass and analysis runs.

So this workflow is pretty simple. It contains only one job:

```
name: Pull request

on:
  pull_request:
    branches: [ "main" ]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build images
        run: |
          docker build -t $API_IMAGE --target=api --build-arg user=martin --
build-arg uid=1000 -f ./api/Dockerfile .
          docker build -t $SCHEDULER_IMAGE --target=scheduler --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .
          docker build -t $WORKER_IMAGE --target=worker --build-arg
user=martin --build-arg uid=1000 -f ./api/Dockerfile .
          docker build -t $MYSQL_IMAGE --build-arg password=${{
secrets.DB_PASSWORD }} -f ./Dockerfile.mysql .
      - name: Run phpstan
```

```
        run: docker run --rm -t $API_IMAGE ./vendor/bin/phpstan analyze --
memory-limit=1G
      - name: Run phpinsights
        run: docker run --rm -t $API_IMAGE php artisan insights --no-
interaction --min-quality=90 --min-complexity=90 --min-architecture=90 --min-
style=90 --ansi --format=github-action
      - name: Run tests
        run: |
          docker-compose -f docker-compose.ci.yml up -d
          docker-compose -f docker-compose.ci.yml exec -T api php artisan
test
```

In theory, we wouldn't even need to build images because the current SHA should already have images. The current SHA is the latest commit on develop. It was created either when you merged a feature branch, or when you pushed a commit to develop. Either way, the pipeline ran and built the according images. However, there are always exceptions and failures so I decided to build images in this job.

Opening a PR to develop (feature branches)

So far it was quite obvious. Feature branches are a bit more tricky. First, think about what's going to happen:

- Wo do the usual things: build images, run tests, run code analysis.
- Then we need to create a new server for this feature. I call them demo servers. Developers also call them review or preview servers.
- After that, we need to deploy the images to the new server.

We already have a `provision_server.sh` script that can be used to create a new server. We also have a `run_provision_server_from_local.sh`. You can call this script from your local machine and it creates a new server and libraries on it. However, running the script from the pipeline requires a few more steps so I created a new `run_provision_server_from_pipeline.sh` script.

It takes a few parameters:

```
NAME=$1
DOCTL_TOKEN=$2
SSH_KEY_PRIVATE_LOCATION=$3
SIZE=${4:-s-2vcpu-2gb}
```

`DOCTL_TOKEN` is an access token to the DigitalOcean API. It is stored in GitHub secret store. `SIZE` is the size of the server with a default value of `2vcpu-2gb`. And then `SSH_KEY_PRIVATE_LOCATION` is the location of the private SSH key used to SSH into the server.

We already know this step from the pipeline:

```
- name: Copy SSH key
  run: |
    echo "${{ secrets.SSH_KEY }}" >> ./id_rsa
    chmod 600 ./id_rsa
```

It copies the private key's content to a file inside the current directory. However, when we call the `run_provision_server_from_pipeline.sh` script it'll look something like this:

```
./deployment/bin/provision_server/run_provision_server_from_pipeline.sh
$SERVER_NAME ${{ secrets.DOCTL_TOKEN }} "$(pwd)"
```

So in the script itself, we need the full path of the SSH key. The "Copy SSH key" step copies it into the worker's working directory, but the run provision script is located elsewhere. So I just pass $(pwd) to the script as an argument that contains the current working dit.

Next, `doctl` (DigitalOcean CLI) needs to be installed on the runner:

```
wget https://github.com/digitalocean/doctl/releases/download/v1.94.0/doctl-1.94.0-linux-amd64.tar.gz
tar xf ./doctl-1.94.0-linux-amd64.tar.gz
mv ./doctl /usr/local/bin
```

We've already discussed how we can create a new droplet with SSH keys using `doctl`:

```
SSH_FINGERPRINT_DEPLOY=$(doctl compute ssh-key list --no-header --access-token $DOCTL_TOKEN | grep "devops-with-laravel-deploy" | awk '{ print $3 }')

SSH_FINGERPRINT_OWN=$(doctl compute ssh-key list --no-header --access-token $DOCTL_TOKEN | grep "MacBook Air" | awk '{ print $4 }')

OUTPUT=$(doctl compute droplet create --image docker-20-04 --size $SIZE --region nyc1 --ssh-keys $SSH_FINGERPRINT_DEPLOY --ssh-keys $SSH_FINGERPRINT_OWN --no-header --access-token $DOCTL_TOKEN $NAME)
```

The only difference is that the runner isn't logged in to `doctl` but uses an access token instead. Every command contains an extra flag: `--access-token $DOCTL_TOKEN $NAME` So this command creates a new server and authorizes my two SSH keys for the root user.

The command returns something like this:

```
→ ~ doctl compute droplet create --image docker-20-04 --size s-1vcpu-1gb --region nyc1  --no-header test
364892974    test              1024    1    25    nyc1    Ubuntu Docker 23.0.6 on Ubuntu 22.04        new        droplet_agent
→ ~
```

The first number is the droplet ID. We can grab this using `awk`:

```
DROPLET_ID=$(echo $OUTPUT | awk '{ print $1 }')
```

`awk` is a powerful pattern scanning and processing tool. `print $1` is a statement that tells it to print the first field (or column) of the input. Which is the droplet ID in this case.

The `droplet create` command is async meaning that it immediately returns the ID and some other information but the droplet is not yet created. In my experience, it takes something like 30-50 seconds. We have to wait for it because we want to SSH into it and then run some commands.

So I just do a

```
sleep 120
```

120 seconds is way more than needed so you can lower it but I wouldn't go below 60 just to be safe.

The next step is completely optional:

```
doctl projects resources assign cad78098-bfb8-4861-bfb2-e969cee18f16 --
resource=do:droplet:$DROPLET_ID --access-token $DOCTL_TOKEN
```

It moves the droplet to the project with the ID of `cad78098-bfb8-4861-bfb2-e969cee18f16` which is my playground project for this book.

In order to SSH into the server, we need its IP address:

```
SERVER_IP=$(doctl compute droplet get $DROPLET_ID --format PublicIPv4 --no-
header --access-token $DOCTL_TOKEN)
```

Thanks to the `--format PublicIPv4` it returns only the IP address. No need to use `awk`.

And then the usual steps:

```
scp -i $SSH_KEY_PRIVATE_LOCATION/id_rsa ./provision_server.sh
root@$SERVER_IP:./provision_server.sh
ssh -tt -i $SSH_KEY_PRIVATE_LOCATION/id_rsa root@$SERVER_IP "chmod +x
./provision_server.sh && ./provision_server.sh"
```

Two easy steps we've already seen before:

- Copy the `provision_server.sh` script to the server
- Make it executable and then run it

But in this case, the script needs to somehow return the droplet's IP address because later in the pipeline we need SSH into it and start the compose stack. But unfortunately, a shell script doesn't have a return value. Only functions can use the `return` keyword. And even functions must return an integer between 0 and 255.

To "return" a string from a script you need to use `echo`. Something like this:

```
# in my_script.sh

#!/bin/bash

echo "Return value"

# The caller
VALUE=$(./my_script.sh)
```

So the last of the script is a simple:

```
echo $SERVER_IP
```

And here's the job that uses this script:

```
provision-demo-server:
  needs: [ analyze, test ]
  runs-on: ubuntu-latest
  outputs:
    demo_server_ip: ${{ steps.create-demo-server.outputs.SERVER_IP }}
  steps:
    - uses: actions/checkout@v3
    - uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKERHUB_USERNAME }}
        password: ${{ secrets.DOCKERHUB_TOKEN }}
    - name: Copy SSH key
      run: |
        echo "${{ secrets.SSH_KEY }}" >> ./id_rsa
        chmod 600 ./id_rsa
    - name: Create demo server
      id: create-demo-server
      run: |
        cp ./deployment/bin/provision_server/provision_server.sh .
        SERVER_NAME=$(echo $GITHUB_HEAD_REF | sed 's/.*\///')
```

```
SERVER_IP=$(./deployment/bin/provision_server/run_provision_server_from_pipeli
ne.sh $SERVER_NAME ${{ secrets.DOCTL_TOKEN }} "$(pwd)")
        echo "$SERVER_IP" >> "$GITHUB_OUTPUT"
```

Let's start with the `Create demo server` step:

- It creates the server name from the `$GITHUB_HEAD_REF` variable which is a string such as `feature/my-feature`. The `sed` command converts this into just `my-feature`. That's going to be the server's name.
- It call the script with `SERVER_NAME`, `secrets.DOCTL_TOKEN` and `$(pwd)`. We need `pwd` to access the `id_rsa` file created in the previous step.
- It stores the output into `SERVER_IP`
- Then it writes these values into `GITHUB_OUTPUT` It's a special GitHub variable that enables jobs to have outputs. This is why we used the env-like format in the files created by the script. In `GITHU_OUTPUT` we need values such as `FOO=foo`

Because the job writes to `GITHUB_OUTPUT` it can have outputs which you can see in the job config:

```
outputs:
  demo_server_ip: ${{ steps.create-demo-server.outputs.SERVER_IP }}
```

The job has one output, coming from the `create-demo-server` step. `create-demo-server` is the ID specified in the step itself:

```
- name: Create demo server
  id: create-demo-server
  run:  ...
```

The variable names must match the ones in `GITHUB_OUTPUT`. So once again:

- The provision script runs this command: `echo $SERVER_IP`
- The pipeline uses this return value: `SERVER_IP=$(...)`
- It writes the value into `GITHUB_OUTPUT`: `echo "$SERVER_IP" >> "$GITHUB_OUTPUT"`
- And finally, we can mark this as an output of the job.

Job outputs can be used in other jobs. For example, in the `deploy-demo` job:

```
deploy-demo:
    needs: [ build-frontend, build-nginx, provision-demo-server ]
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Copy SSH key
        run: |
          echo "${{ secrets.SSH_KEY }}" >> ./id_rsa
          chmod 600 id_rsa
      - name: Deploy app
        run: |
          SSH_CONNECTION_DEMO=root@${{ needs.provision-demo-
server.outputs.demo_server_ip }}
          scp ./id_rsa ./deployment/bin/deploy.sh
$SSH_CONNECTION_DEMO:/home/martin/deploy.sh
          scp ./id_rsa ./docker-compose.prod.yml
$SSH_CONNECTION_DEMO:/usr/src/docker-compose.prod.yml
          scp -i ./id_rsa ./.env.prod.template
$SSH_CONNECTION_DEMO:/usr/src/.env
           ...
```

Outputs can be accessed like this: `needs.provision-demo-server.outputs.demo_server_ip`

It uses the `provision-demo-server` job's output to construct the `SSH_CONNECTION_DEMO` variable. Other than that, it's the same as `deploy-staging` or `deploy-prod`. It copies `deploy.sh`, `docker-compose.yml`, and the `.env` file to the demo server.

After the `.env` file is copied to the server, it's time to set some variables:

```
ssh -tt -o StrictHostKeyChecking=no -i ./id_rsa $SSH_CONNECTION_DEMO "
sed -i '/IMAGE_TAG/c\IMAGE_TAG=${{ github.sha }}' /usr/src/.env

sed -i '/AWS_ACCESS_KEY_ID/c\AWS_ACCESS_KEY_ID=${{ secrets.AWS_ACCESS_KEY_ID
}}' /usr/src/.env
sed -i '/AWS_SECRET_ACCESS_KEY/c\AWS_SECRET_ACCESS_KEY=${{
secrets.AWS_SECRET_ACCESS_KEY }}' /usr/src/.env
sed -i "/AWS_BUCKET/c\AWS_BUCKET=devops-with-laravel-demo-backups"
/usr/src/.env

sed -i '/DB_PASSWORD/c\DB_PASSWORD=${{ secrets.DB_PASSWORD }}' /usr/src/.env

sed -i '/APP_KEY/c\APP_KEY=${{ secrets.APP_KEY }}' /usr/src/.env"
```

That's it. Basically, this job is the same as before but the SSH connection comes from the output of a previous job.

Reusing jobs (composite actions)

Right now, we have lots of duplications in the pipeline. Just to name a few:

- Building images (api, worker, scheduler, mysql, frontend)
- QA steps (analyze, test)
- Deploying staging and production are identical

Fortunately, in GitHub, we can use composite actions to remove these duplications. You can copy a job, put it into a dedicated yaml file, and re-use it in your pipeline. These small, reusable jobs are called compose actions in GitHub.

Let's refactor `build-mysql`. I created a new folder in `.github/workflows/jobs/build-mysql` and created a new file called `action.yml`. This filename is a requirement by GitHub. The yaml file looks like this:

```
name: 'Build MySQL'
description: 'Build MySQL image'
inputs:
  dockerhub-username:
    description: 'DockerHub username'
    required: true
  dockerhub-token:
    description: 'DockerHub access token'
    required: true
  db-password:
    description: 'Database password'
    required: true

runs:
  using: "composite"
  steps:
    - uses: docker/login-action@v2
      with:
        username: ${{ inputs.dockerhub-username }}
        password: ${{ inputs.dockerhub-token }}
    - run: docker build -t $MYSQL_IMAGE --build-arg password=${{ inputs.db-
password }} -f ./Dockerfile.mysql .
      shell: bash
    - run: docker push $MYSQL_IMAGE
      shell: bash
```

In order to build and push MySQL we need three inputs:

- DockerHub username
- DockerHub password
- DB password

These are all required inputs as you can see.

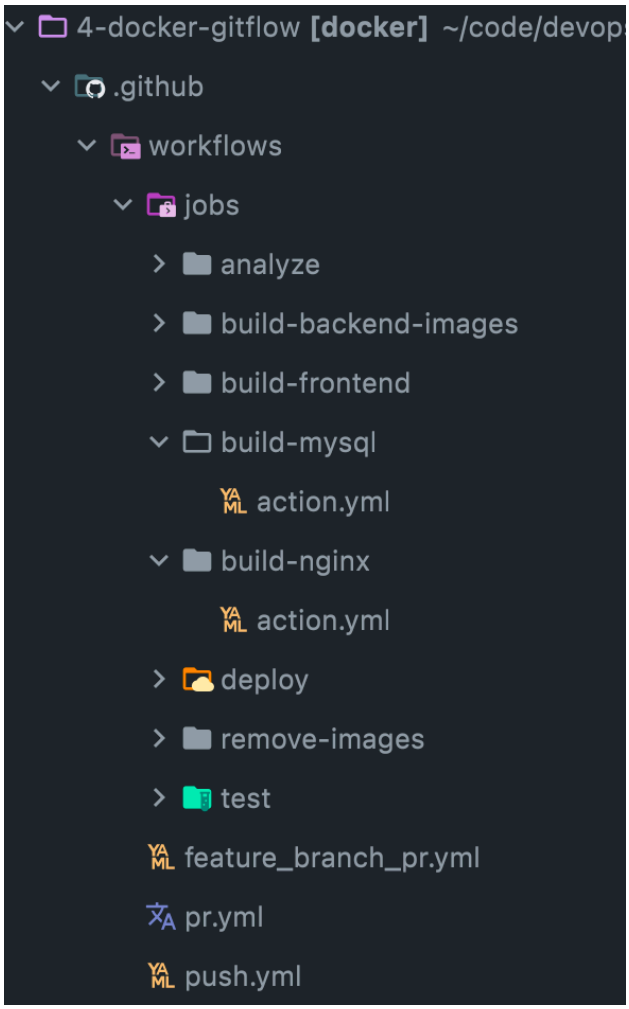
`using: "composite"` tells GitHub it's a composite action that can be reused in other workflows. Every step looks the same as before but we have to add the `shell: bash` to each of them.

It can be used like this:

```
- uses: ./.github/workflows/jobs/build-mysql
  with:
    dockerhub-username: ${{ secrets.DOCKERHUB_USERNAME }}
    dockerhub-token: ${{ secrets.DOCKERHUB_TOKEN }}
    db-password: ${{ secrets.DB_PASSWORD }}
```

Just like other actions such as `actions/checkout` or `docker/login-action`.

I did the same with every duplicate and created the following folder structure:



And the usage looks like this:

```yaml
jobs:
  build-backend-images:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: ./.github/workflows/jobs/build-backend-images
        with:
          dockerhub-username: ${{ secrets.DOCKERHUB_USERNAME }}
          dockerhub-token: ${{ secrets.DOCKERHUB_TOKEN }}

  build-mysql:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: ./.github/workflows/jobs/build-mysql
        with:
```

```
          dockerhub-username: ${{ secrets.DOCKERHUB_USERNAME }}
          dockerhub-token: ${{ secrets.DOCKERHUB_TOKEN }}
          db-password: ${{ secrets.DB_PASSWORD }}

  analyze:
    needs: [build-backend-images, build-mysql]
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: ./.github/workflows/jobs/analyze
        with:
          dockerhub-username: ${{ secrets.DOCKERHUB_USERNAME }}
          dockerhub-token: ${{ secrets.DOCKERHUB_TOKEN }}

  test:
    needs: [ build-backend-images, build-mysql ]
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: ./.github/workflows/jobs/test
        with:
          dockerhub-username: ${{ secrets.DOCKERHUB_USERNAME }}
          dockerhub-token: ${{ secrets.DOCKERHUB_TOKEN }}
```

It's much better. We use the same jobs with different inputs.

Final touches

Frontend, nginx, and proxy

The `docker-compose.yml` I showed you in this chapter still misses two things: there's no `wait-for-it.sh` in the `frontend`, `nginx`, and `proxy` services. It's not the end of the world because it does not cause errors, but of course, we can fix it pretty easily. Unfortunately, `wait-for-it.sh` won't run on `alpine` Linux, so first we have to change the Dockerfile.

This is the frontend Dockerfile:

```
FROM base AS dev
COPY ./api/wait-for-it.sh /usr/wait-for-it.sh
...

FROM nginx:1.25.1 AS prod
COPY ./api/wait-for-it.sh /usr/wait-for-it.sh
...
```

Now it's a simple Debian-based image and it also contains the script.

And now the service looks like this:

```
frontend:
  image: martinjoo/posts-frontend:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh nginx:80 -t 60 && nginx -g \"daemon
off;\""
  restart: unless-stopped
  ports:
    - "80:80"
```

The main command we need to run is `nginx -g "daemon off;"` This is the same command as the base image runs. You can find it here.

The `command` in the development compose file looks like this:

```
command: sh -c "/usr/src/wait-for-it.sh nginx:80 -t 60 && npm run serve"
```

It's `npm run serve` instead of `nginx`.

The other service we need to change is `nginx`:

```
nginx:
  image: martinjoo/posts-nginx:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh api:9000 -t 60 && nginx -g \"daemon
off;\""
```

It's essentially the same command but it waits for the `api`. The development config is the same as this one.

And the last is the proxy which needs both frontend and nginx:

```
proxy:
  image: martinjoo/posts-proxy:${IMAGE_TAG}
  command: sh -c "/usr/wait-for-it.sh nginx:80 -t 120 && /usr/wait-for-it.sh
frontend:80 -t 120 && nginx -g \"daemon off;\""
```

Worker & Scheduler

The `worker` and `scheduler` containers also need to wait for `mysql` and `redis`. These containers already contain the `wait-for-it.sh` script, so all we need to do is this:

```
scheduler:
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 30 && /usr/src/wait-for-it.sh redis:6379 -t 30 && /usr/src/scheduler.sh"

worker:
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 30 && /usr/src/wait-for-it.sh redis:6379 -t 30 && /usr/src/worker.sh"
```

These commands would be so long that I moved the original commands into `sh` files. They are very simple one-liners:

```
# worker.sh
supervisord -c /etc/supervisor/conf.d/supervisor.conf

# scheduler.sh
nice -n 10 sleep 60 && php /usr/src/artisan schedule:run --verbose --no-interaction
```

They both live inside the `api` folder so they are copied into the images.

And of course, I also changed the `CMD` in the Dockerfile:

```
FROM api AS worker
COPY ./deployment/config/supervisor/supervisord.conf /etc/supervisor/conf.d/supervisor.conf
CMD ["/bin/sh", "/usr/src/worker.sh"]

FROM api AS scheduler
CMD ["/bin/sh", "/usr/src/scheduler.sh"]
```

Now, everything is ready.

Limitations of docker-compose

Let's quickly talk about the advantages and disadvantages of docker-compose.

The advantages:

- It's pretty easy to learn and use
- Also easy to deploy. The deploy script was effectively 2 lines of code.

So it's a fast and easy way to ship containers into production. However, it has some disadvantages as well:

- It's not designed to be used in production. Compose was created to be used in only development. However, lots of teams use it in production.
- No rolling updates. No zero-downtime deployments. It may or may not be a problem for you but it's a fact. When you deploy with docker-compose downtime is guaranteed.
- No rollbacks.
- It's a single-machine solution. docker-compose only works on a single machine. You cannot have a cluster of servers with it. Which might not be a problem for your project, of course. It might be a problem, however, when you decide to be highly available and you need some scale.

Conclusions

The first question you need to decide: do I need Docker or don't? There's not a single answer to this question but here are my thoughts:

- If you're working in a team it's probably a good idea to use Docker. It just removes so many "moving parts" from your environment. Your application has a Docker image, which is a self-contained, deployable unit that can be used on any server that has Docker installed on it.
- If you're working on a solo project, however, I think you're better off without Docker. It adds complexity to your workflow and slows down the deployment process. This is not a big deal in a company environment but when you're launching your indie SaaS project a 30s pipeline vs a 4-minute one makes all the difference, in my experience.

It also depends on your orchestrator platform. For example, if you'd like to use Kubernetes, you **have to** have images. On the other hand, if you want to deploy your application as Lambda function(s) you don't need images at all.

Thank you very much for reading this book! If you liked it, don't forget to Tweet about it. If you have questions you can reach out to me here.