Docker Swarm

The project files are located in the `5-swarm` folder.

I know Docker Swarm is not the sexiest thing in the world, but here's my offer:

If you are already experienced with docker-compose, you can go from a single machine setup to a highly available 100-server cluster in ~24 hours by learning just ~10 new commands.

Also:

- Deploys are going to be near zero downtime.
- **You can use the same `docker-compose.yml` as before.**
- Adding or removing servers are no-brainers.
- Scaling out services requires one line of code.
- Rolling back to previous versions is easy.
- The number of options related to updating or rolling back services is huge.
- Your local environment remains the same.
- It also works on one machine.
- Switching from Swarm to Kubernetes is not that hard. The principles are almost the same.
- It's a "native" tool developed by the Docker team.

Now that you are excited, let's talk about the downsides of scaled applications.

State

State is the number one enemy of distributed applications. By distributed, I mean, running on multiple servers. Imagine that you have an API running on two different servers. Users can upload and download files. You are using Laravel's default local storage.

- User A uploads `1.png` to `/storage/app/public/1.png` on Server 1.
- User B wants to download the image but his request gets served by Server 2. But there's no `/storage/app/public/1.png` on Server 2 because User A uploaded it onto Server 1.

So state means when you store something on the current server's:

- Filesystem
- Or memory

Here are some other examples of states:

- Databases such as MySQL. MySQL does not just use state, it is the state itself. So you cannot just run a MySQL container on a random node or in a replicated way. Being replicated means that, for example, 4 containers are running at the same time on multiple hosts. This is what we want to do with **stateless** services but not with a database.
- Redis also means state. The only difference is that it uses memory (but it also persists data on the SSD).
- Local storage just as we discussed earlier.
- File-based sessions (`SESSION_DRIVER=file`).
- File-based cache (`CACHE_DRIVER=file`).
- `.env` file (kind of...)

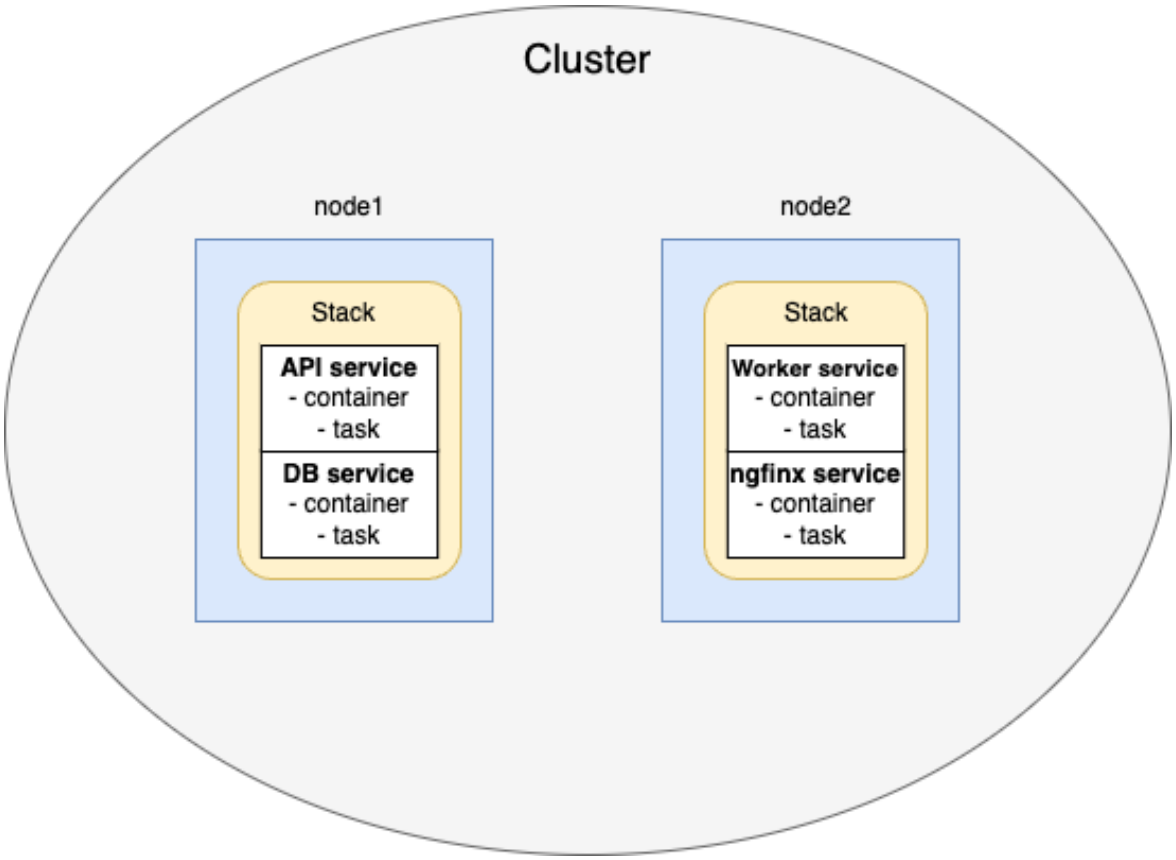
When I deployed my first distributed application it took me 4-6 hours of debugging to realize these facts.

All of these problems can be solved relatively easily, so don't worry, we're going to look into different solutions.

Basic concepts

Docker Swarm is a container orchestrator tool that works on multiple servers. These are the basic terms:

- **Node** is a server with Docker installed on it.
- **Cluster** is a set of nodes. Usually, one project runs on one cluster with multiple nodes.
- **Stack** is a `docker-compose.yml` file. A project can have multiple stacks such as the application itself and a monitoring-related stack. In the Docker chapter, I referred to the `docker-compose.yml` file as "stack" because of this.
- **Service** is the same thing as before. It's a service inside a stack. But in Swarm, a service can run in a replicated way. For example, we can scale the API to run in 4 or 6 replicas.
- **Container** is also the same as before. It's a running image.
- **Task** is a tricky one. Each running container will have a task. And a task is the smallest unit of work that can be scheduled and deployed in a Swarm cluster. A task represents a single instance of a service running on a node in the cluster. Each task represents a specific container with its associated configuration and resources. It's basically a scheduling entity. Don't worry, I didn't understand it either. Just start playing with Swarm and you'll have a better idea after a few days.



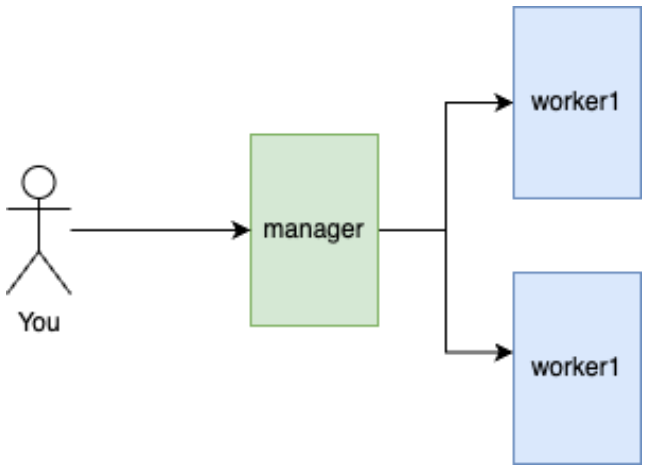
Workers, managers, and leaders

There are two kinds of nodes in a cluster:

- **Worker** is a node that doesn't care about the world but gets instructions and runs containers. It has no other responsibilities.
- **Manager** is a node that instructs the workers to do something. When you configure a service that needs to run in 6 replicas you need to issue the command on a manager node and it will distribute your 6 replicas across the available workers. The manager also checks the cluster and restarts services if needed.

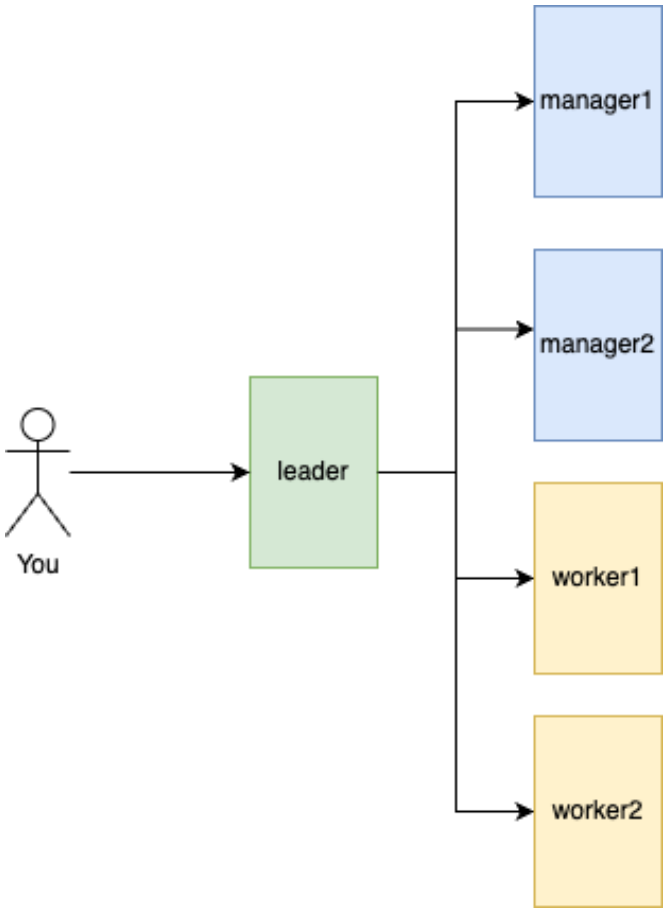
So it's a master-worker architecture. The only difference is that a manager is also a worker. So it not only gives the worker commands but also runs containers itself.

It looks like this:



You can see that this architecture has a big disadvantage: the manager is a **single point of failure**. If it goes down the cluster becomes unreliable. There's no node that runs checks on the other nodes or monitors the number of replicates and so on.

To solve that problem, we can have many manager nodes. But one of them is a **leader**. At any given time, there's only one leader and that node manages the cluster. If the leader goes down, the other managers elect a new leader.



And now, you are the only single point of failure. But that's okay!

Here's the fault tolerance and the recommended number of managers by Docker:

Number of managers	Failed managers tolerated
1	0
2	0
3	1
4	1
5	2
6	2
7	3

If you have 4 manager nodes and one of them goes down the cluster stops making "progress." The general formula is `N / 2 - 1` where N is the number of managers. Because of that formula, it doesn't make sense to have an even number of managers. As you can see it doesn't make a difference if you have 3 or 4 managers. The fault tolerance is 1 in both cases.

I said that the cluster stops making "progress." But what is progress in this context? The manager nodes need to agree on a specific state or command and perform the required actions based on the agreement. These decisions and operations involve maintaining a consistent and replicated state across all nodes in the cluster. For example, having 6 replicas of the `api` service but only two can run on the same node at the same time. We're talking about these kinds of decisions, operations, and progress. Other than that managers also have some special tasks such as:

- Leader election: The managers in the cluster elect a leader responsible for coordinating operations and making decisions on behalf of the cluster. It's an important process.
- Log replication: The leader receives commands from clients and appends them to its log. It then replicates the log entries to other nodes in the cluster to maintain consistency. On a manager node, you can see every log entry from all the nodes produced by a specific service. For example, here are the logs of the `worker` service running in 4 replicas on 4 nodes:

```
posts_worker.4.vejjt8udtv4c@node3    |   2023-07-19 14:47:06 App\Jobs\PublishPostsJob mHvk3Bj8bqPvV56XOVbODIJDU0L1KoAG  1.91ms DONE
posts_worker.4.vejjt8udtv4c@node3    |   2023-07-19 14:50:12 App\Jobs\PublishPostsJob lqbu2WNZg4l1XqX8oE5lZ1UEqgKGOrOB  RUNNING
posts_worker.4.vejjt8udtv4c@node3    |   2023-07-19 14:50:12 App\Jobs\PublishPostsJob lqbu2WNZg4l1XqX8oE5lZ1UEqgKGOrOB  2.01ms DONE
posts_worker.4.vejjt8udtv4c@node3    |   2023-07-19 14:51:12 App\Jobs\PublishPostsJob 42H2AEZWMXuoKGzSAA2JuJmzisowHhhK  RUNNING
posts_worker.4.vejjt8udtv4c@node3    |   2023-07-19 14:51:12 App\Jobs\PublishPostsJob 42H2AEZWMXuoKGzSAA2JuJmzisowHhhK  1.91ms DONE
posts_worker.4.vejjt8udtv4c@node3    |   2023-07-19 14:52:15 App\Jobs\PublishPostsJob w1Icm8NIgAFuilvyhlk5KlU8QRRsUGVr  RUNNING
posts_worker.4.vejjt8udtv4c@node3    |   2023-07-19 14:52:15 App\Jobs\PublishPostsJob w1Icm8NIgAFuilvyhlk5KlU8QRRsUGVr  1.54ms DONE
posts_worker.3.yb32copy10w6@node4    | wait-for-it.sh: waiting 60 seconds for mysql:3306
posts_worker.3.yb32copy10w6@node4    | wait-for-it.sh: mysql:3306 is available after 13 seconds
posts_worker.3.yb32copy10w6@node4    | wait-for-it.sh: waiting 60 seconds for redis:6379
posts_worker.3.yb32copy10w6@node4    | wait-for-it.sh: redis:6379 is available after 0 seconds
posts_worker.3.yb32copy10w6@node4    |   2023-07-19 15:06:44 App\Jobs\PublishPostsJob or346Y3KbAvVyKsegksytH7qUz83qOoO  RUNNING
posts_worker.3.yb32copy10w6@node4    |   2023-07-19 15:06:45 App\Jobs\PublishPostsJob or346Y3KbAvVyKsegksytH7qUz83qOoO  1s DONE
posts_worker.3.yb32copy10w6@node4    |   2023-07-19 15:07:45 App\Jobs\PublishPostsJob SLfxBIJ3DIrT4IVLy4xwrxOoZznyIMwt  RUNNING
posts_worker.3.yb32copy10w6@node4    |   2023-07-19 15:07:46 App\Jobs\PublishPostsJob SLfxBIJ3DIrT4IVLy4xwrxOoZznyIMwt  266.36ms DONE
```

- And other cluster management-related tasks.

Docker Swarm implements the Raft Consensus Algorithm.

Creating a cluster

I recommend you rent a few $6 droplets to create a playground cluster with me. Choose the Docker image and name them node1, node2, etc.

To run a cluster you need to open the following ports **on all nodes**:

- 2377
- 7949
- 4789

They are used by Docker for communication. On DigitalOcean droplets you can run this command:

```
ufw allow 2377 && ufw allow 7946 && ufw allow 4789
```

You also need to login into your Docker registry **on every node**. For me, it's Docker Hub so the command is simply:

```
docker login -u <username> -p <token>
```

And here's the command to create a cluster:

```
docker swarm init
```

You're basically done! Now you have a 1-node cluster. This node is the **leader**. If you run this command it gives a token. This token can be used to join the cluster as a **worker** node.

Create another droplet and run this command:

```
docker swarm join --token <your-token> 1.2.3.4:2377
```

`1.2.3.4` is the leader's IP address and `2377` is one of the ports you opened before.

If you run

```
docker node ls
```

on the manager node, you should see something like this:

```
root@node1:~# docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
rsomn2y3d200ja1upl57vhu2q *   node1      Ready     Active         Leader           23.0.6
zruwq8f9qoib3wjcbtysp7zry     node2      Ready     Active                          23.0.6
ela1mwsl2vvnv06zoutjqupfe     node3      Ready     Active                          23.0.6
po9sldtuxjur7zn3av36xpgh0     node4      Ready     Active                          23.0.6
root@node1:~#
```

I ran the command on `node1` which is the leader. It lists all of the nodes that are part of the cluster. All of the other nodes are workers since the `Manager Status` column is empty. It can be `Leader`, `Reachable`, or empty. `Reachable` means manager.

If I run the same command on a worker node such as `node2` it returns an error:

```
root@node2:~# docker node ls
Error response from daemon: This node is not a swarm manager. Worker nodes can't be used to view or modify cluster state. Please run this command on a manager
 node or promote the current node to a manager.
root@node2:~#
```

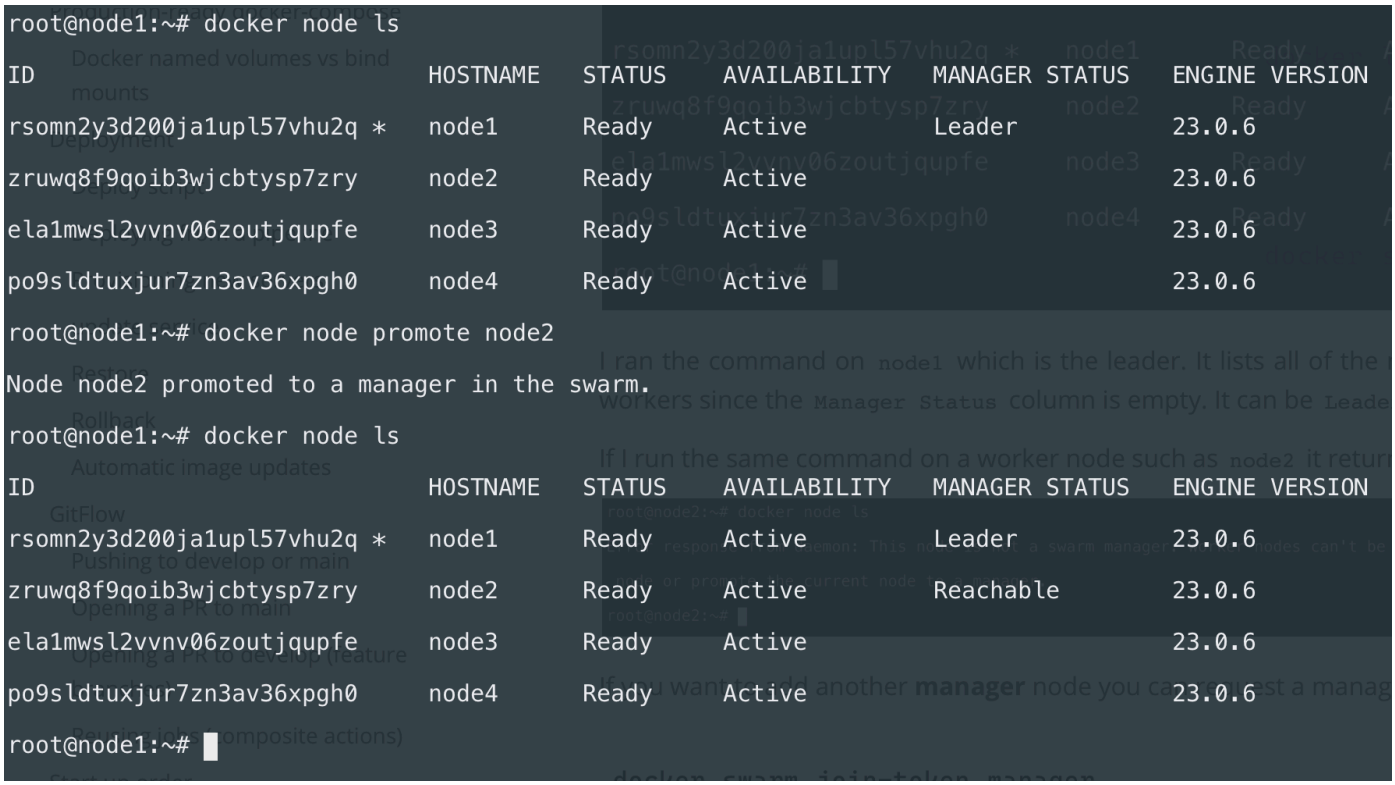
If you want to add another **manager** node you can request a manager token on the current manager node:

```
docker swarm join-token manager
```

If you run `docker swarm join` with this token, the new node is going to be a manager.

Or if you have an existing worker node but want to promote it to the manager you can run this command on a **manager** node:

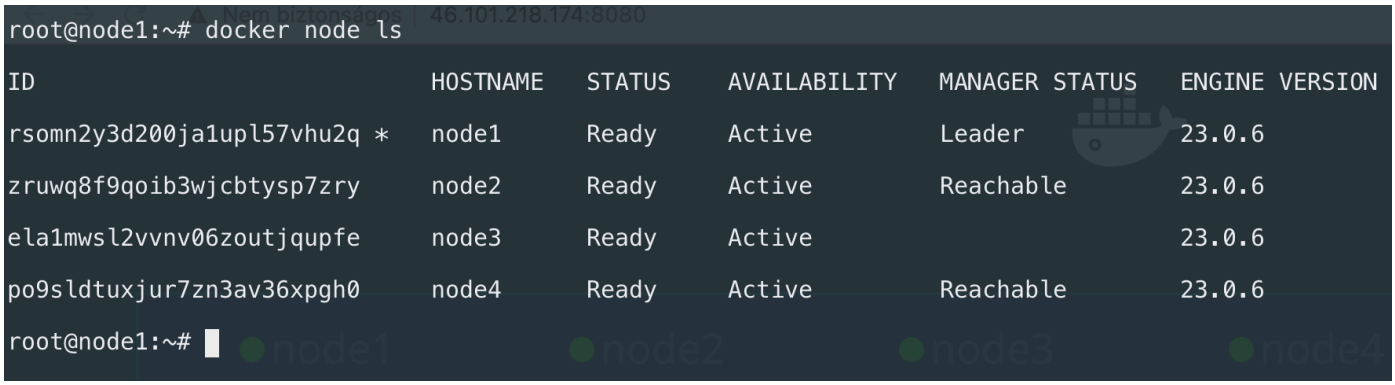
```
docker node promote node2
```



You can demote a node by running:

```
docker node demote node3
```

I created a quick playground cluster such as this:



Application-level changes

To take an app from one machine to a cluster you most likely need to make some changes. As I said in the introduction, the state is our biggest enemy right now. You need to make sure that you don't use these:

- Local storage
- File-based session
- File-based caches

Later, we're gonna talk about MySQL and Redis too. The sample application contains an endpoint with file upload. To make it work in a cluster our best shot is to use S3, DigitalOcean Spaces, CloudFlare R2, or some other object storage.

I'm going to use S3 in this book but if it was my own project I would not use it because the whole AWS seems brutally overpriced. And quite ugly, to be honest.

First, you need to set some env variables:

```
FILESYSTEM_DISK=s3

AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
AWS_DEFAULT_REGION=us-east-1
AWS_BUCKET=devops-with-laravel-storage
AWS_USE_PATH_STYLE_ENDPOINT=false
AWS_URL=https://devops-with-laravel-storage.s3.us-east-1.amazonaws.com/
```

You can get all this information from the AWS console.

Then, in the `PostController` we can upload the file:

```
private function storeCoverPhoto(UpsertPostRequest $request, Post $post): void
{
  $file = $request->file('cover_photo');

  if ($file) {
    $filename = Str::slug($post->title, '_')
      . '-' . $post->id . '.' . $file->extension();

    $file->storeAs('post_cover_photos', $filename);
```

```
    $post->cover_photo_path =
      'post_cover_photos' . DIRECTORY_SEPARATOR . $filename;

    $post->save();
  }
}
```

Since S3 is the default driver there's no need to explicitly specify it. But of course, you can do this as well:

```
$file->storeAs('post_cover_photos', $filename, 's3');
```

Then in the `PostResource` class, I create a temporary URL to show the file:

```
class PostResource extends JsonResource
{
  public function toArray(Request $request): array
  {
    return [
      'id' => $this->id,
      'title' => $this->title,
      'headline' => $this->headline,
      'content' => $this->content,
      'is_published' => $this->is_published,
      'publish_at' => $this->publish_at,
      'author' => UserResource::make($this->whenLoaded('author')),
      'cover_photo_url' => Storage::disk('s3')
        ->temporaryUrl($this->cover_photo_path, now()->addMinutes(10)),
      ];
  }
}
```

If you use file-based session or cache you need to change these variables as well:

```
CACHE_DRIVER=redis
SESSION_DRIVER=database
```

Of course, they don't need to be these exact values.

Deploying a stack

The next step is to make the existing compose files "Swarm-compatible." Everything is going to be the same, there's only one, but big difference: Docker Swarm doesn't read the `.env` file in the current working directory. `docker-compose` does. This is why we deployed it this way:

- Copy the .env template in the pipeline
- Overwrite some values from secrets and pipeline variables
- Run `docker-compose up` that reads the `.env` and everything's good to go

Docker Swarm doesn't do this for us, so we need to make a few changes.

the first step is to add environment variables to the services:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && php-fpm"
  environment:
    - APP_NAME=posts
    - APP_ENV=production
    - APP_KEY=${APP_KEY}
    - APP_DEBUG=false
    - APP_URL=http://localhost
    - LOG_CHANNEL=stack
    - LOG_LEVEL=error
    - DB_CONNECTION=mysql
    - DB_HOST=mysql
    - DB_PORT=3306
    - DB_DATABASE=posts
    - DB_USERNAME=${DB_USERNAME}
    - DB_PASSWORD=${DB_PASSWORD}
    - QUEUE_CONNECTION=redis
    - REDIS_HOST=redis
    - REDIS_PORT=6379
    - MAIL_MAILER=log
    - AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID}
    - AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY}
    - AWS_BUCKET=devops-with-laravel-storage
```

```
    - AWS_DEFAULT_REGION=us-east-1
    - FILESYSTEM_DISK=s3
  depends_on:
    - update
    - mysql
    - redis
```

Every Laravel service needs these variables so I put these under the `scheduler`, `worker`, and `update` as well.

As you can see, I'm using the template format `${DB_USERNAME}`. This will be substituted with the current environment variables in the running process. I'm talking about something like `export DB_USERNAME=username && some-command`. If you run this command the `DB_USERNAME` will be available for the `some-command` process.

Before deploying a stack (the compose file), we need to make sure that we export the right environment variables. So we still can have an `.env` file but it needs to be exported:

```
export $(cat .env)
```

This command will export every variable from the `.env` file. And after that the stack can be deployed and Docker Swarm will substitute the placeholders with the environment variables.

So technically we will still have a `.env` file but it's only going to be used on the manager node before running the deploy command. After that, it can be deleted because the variables are in the current session and Swarm handles the rest.

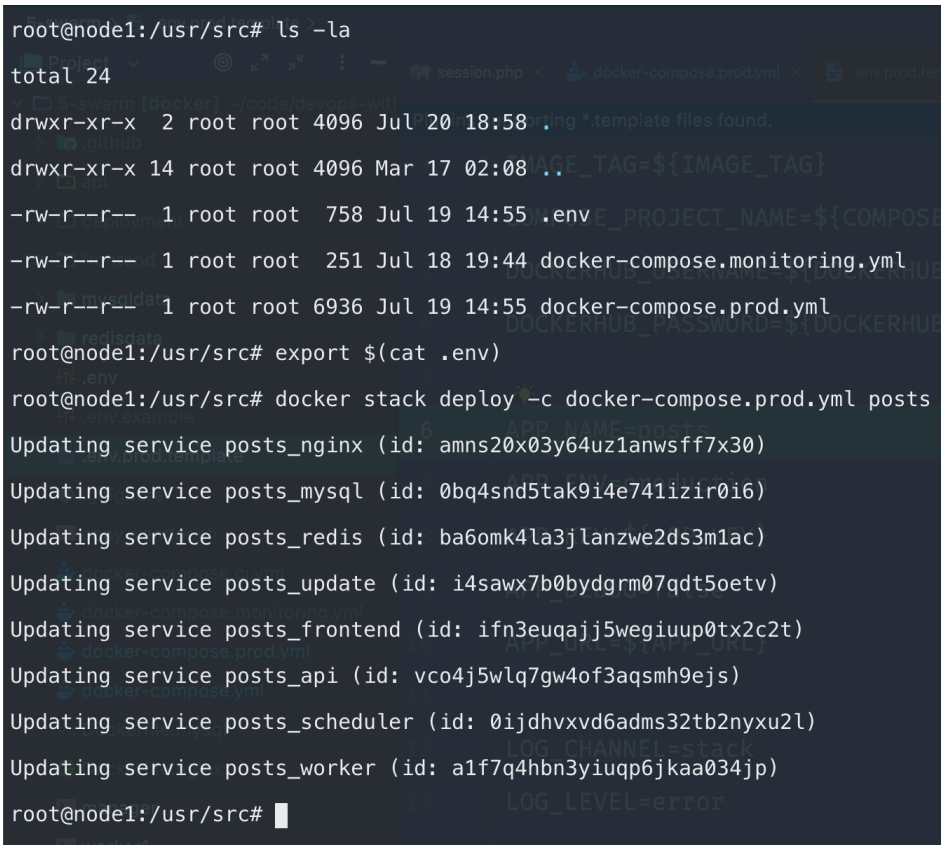
So if you check out the project files you can prepare a quick, manual deployment by running these commands:

```
scp ./.env.prod.template user@1.2.3.4:/usr/src/.env
scp ./docker-compose.prod.yml user@1.2.3.4:/usr/src
```

Where `1.2.3.4` is the IP of a manager node. Then change the values inside the `.env` file and run this on the manager:

```
export $(cat .env)
docker stack deploy -c docker-compose.prod.yml posts
```

You should see something like that:



`docker stack deploy` is the most important command. This is how you can deploy a stack. It takes one argument which is `posts` in my case. It's the stack's name. Basically, it's the name of your project. With the `-c` you can define which `docker-comose` file you want to deploy. You always need to specify this, even if your file is called `docker-compose.yml`.

If you now run

```
docker stack ls
```

You should see the stacks that have been deployed. In this case, `posts` is the only stack.

To check the services inside the stack, just run:

```
docker service ls
```

You should see the services listed in the `docker-compose.yml` file:

```
root@node1:/usr/src# docker service ls
ID             NAME                   MODE             REPLICAS           IMAGE                                                                  PORTS
js7mpg4f5gia   monitoring_visualizer  replicated       1/1                dockersamples/visualizer:stable                                        *:8080->808
0/tcp
87vcsqvbsvpo   posts_api              replicated       6/6                martinjoo/posts-api:a5b15afc99f881b6475fbd30de800ea9a29a1881
oum3qb62u65w   posts_frontend         replicated       1/1                martinjoo/posts-frontend:a5b15afc99f881b6475fbd30de800ea9a29a1881       *:80->80/tc
p
gzt2nv38mo3p   posts_mysql            replicated       1/1                martinjoo/posts-mysql:a5b15afc99f881b6475fbd30de800ea9a29a1881          *:3306->330
6/tcp
etyxvkx5hyhr   posts_nginx            replicated       1/1                martinjoo/posts-nginx:a5b15afc99f881b6475fbd30de800ea9a29a1881          *:8000->80/
tcp
sq9aa0j0usk6   posts_redis            replicated       1/1                redis:7.0.11-alpine                                                    *:6379->637
9/tcp
wt1qopa2u0vy   posts_scheduler        replicated       0/1                martinjoo/posts-scheduler:a5b15afc99f881b6475fbd30de800ea9a29a1881
upm76c7q2s4p   posts_update           replicated job   0/1 (1/1 completed)   martinjoo/posts-api:a5b15afc99f881b6475fbd30de800ea9a29a1881
ql8itymvhbwm   posts_worker           replicated       4/4                martinjoo/posts-worker:a5b15afc99f881b6475fbd30de800ea9a29a1881
root@node1:/usr/src#
```

Ignore `REPLICAS` and `replicated job` for now. If everything went well you should see 1/1 everywhere.

If you now open the IP address of your manager node, you can see the frontend.

So let's recap what happened so far:

- We Created 3 servers.
- Made the app stateless.
- Created a cluster with this command: `docker swarm init`
- Added some worker nodes with this one: `docker swarm join --token <your-token> 1.2.3.4:2377`
- Added some environment variables to the existing `docker-compose.prod.yml` file
- And then deployed the stack using the `docker stack deploy` command

And now you have a 3-node cluster! If you still think Swarm is lame and k8s is the king...

Of course, we still need to fix two major problems:

- The database is being placed on a random node every time you deploy the stack. So basically each node will have about 33% of your records.
- Services are not replicated. I mean, it's not really a "major problem", but that's the main goal right now.

Service placements

One of the possible ways to solve the database problem is to label a node as "the database node." And then Swarm will always place the database services (MySQL and Redis) to that node. Of course, the other solution is to use a managed database cluster in a cloud provider. We're going to try it out later.

The first thing to do is adding a label to one of the nodes. First, list the nodes on the manager:

```
docker node ls
```

Then run the `docker node update` command:

```
docker node update --label-add db=true node3
```

- `label-add` adds a label to a node. These labels are visible to Swarm and they can be used in the `docker-compose.yml` config to place services to a particular node.
- `db=true` is the label itself. `db` is the label itself and `true` is the value. `true` has no special meaning. It can be `db=yes` if you like that. And these are not binary values, so you can use a label such as `type=database`
- `node3` is the name of the node returned by `docker node ls`

You can check out the labels of a node by running:

```
docker node inspect node3
```

It returns the full spec of the node:

```
root@node1:/usr/src# docker node inspect node3
[
    {
        "ID": "ela1mwsl2vvnv06zoutjqupfe",
        "Version": {
            "Index": 20252
        },
        "CreatedAt": "2023-07-18T18:54:43.310028328Z",
        "UpdatedAt": "2023-07-19T21:48:58.318689826Z",
        "Spec": {
            "Labels": {
                "db": "true"
            },
            "Role": "worker",
            "Availability": "active"
        },
        "Description": {
            "Hostname": "node3",
            "Platform": {
                "Architecture": "x86_64",
                "OS": "linux"
            },
```

So now, node3 is the database server. It will run the MySQL and Redis services so they can persist in their state.

The next is to use the `db=true` label in the `docker-compose` stack:

```
mysql:
  image: martinjoo/posts-mysql:${IMAGE_TAG}
  deploy:
    placement:
      constraints:
        - "node.labels.db==true"
  ports:
    - "3306:3306"
  volumes:
    - type: volume
      source: mysqldata
      target: /var/lib/mysql
  environment:
```

```
    - MYSQL_ROOT_PASSWORD=${DB_PASSWORD}
```

`deploy` is a special part of the compose config. We're going to use it a lot because it contains configuration that can be used only by Swarm (it's not entirely true, because `docker-compose` can also use a few of them).

In the `placement` key we can control how Swarm should place the containers of this service. Swarm offers a ton of options, just to name a few:

- Place certain containers only in manager nodes.
- Only to nodes with a specific operating system.
- Spread replicas evenly across specific labels. For example, you can use labels to label data center regions and then you can distribute your replicas across these data centers.

Under the `constraint` key we can use our specific constraint using the `==` or the `!=` operands. Here's a list of all available constraints.

`node.labels.db` refers to the specific label we want, and the value must be `true`. With this small change Swarm guarantees that MySQL will always run on `node3`.

Of course, we need the same placement for Redis as well:

```
redis:
  image: redis:7.0.11-alpine
  deploy:
    placement:
      constraints:
        - "node.labels.db==true"
  ports:
    - "6379:6379"
  volumes:
    - type: volume
      source: redisdata
      target: /data
```

Notice that I use the same `volumes` as earlier. It doesn't need to change at all.

After these changes, all you need to do is:

```
export $(cat .env)
docker stack deploy -c docker-compose.prod.yml posts
```

You can check the running tasks (replicas) of a service by running:

```
docker service ps posts_mysql
```

It gives you something like this:

The important thing is the task created by the `posts_mysql` service runs on `node3` now.

If you're still confused about tasks, maybe this helps:

- Each service can spin up multiple containers because we can scale them (we will later).
- Each container has exactly one task.

So a task is a container with some additional meta information (such as on which node it's running).

Let's summarize what happened:

- We labeled node3 as the database node by running `docker node update --label-add db=true node3`
- Added the `deploy`, `placement`, and `constraints` keys to the docker-compose config to tell Swarm that MySQL and Redis should run on nodes with the label `db=true`
- Then we re-deployed the stack with `docker stack deploy`

I suggest you try this on your own and after you deploy the stack run `docker service ls`. This way you can see how it's updating services. Meanwhile, just open the IP address of your manager node and you can see there's no downtime in the process.

Scaling services

API and nginx

And here comes the fun part. Let's scale out the api to 6 replicas:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && php-fpm"
  deploy:
    replicas: 6
   ...
```

Re-deploy the stack:

```
export $(cat .env)
docker stack deploy -c docker-compose.prod.yml posts
```

It's all done.

If you now list the service by running `docker services ls` you should see 6/6 replicas:

```
root@node1:/usr/src# docker service ls
ID             NAME                  MODE         REPLICAS   IMAGE                                                                 PORTS
js7mpg4f5gia   monitoring_visualizer   replicated   1/1        dockersamples/visualizer:stable                                       *:8080->808
0/tcp
87vcsqvbsvpo   posts_api             replicated   6/6        martinjoo/posts-api:a5b15afc99f881b6475fbd30de800ea9a29a1881
oum3qb62u65w   posts_frontend        replicated   1/1        martinjoo/posts-frontend:a5b15afc99f881b6475fbd30de800ea9a29a1881      *:80->80/tc
```

You can list each of these tasks by running `docker service ps posts_api`:

```
root@node1:/usr/src# docker service ps posts_api
ID             NAME              IMAGE                                                                 NODE      DESIRED STATE   CURRENT STATE            ERROR     POR
TS
i87ty04qz7o5   posts_api.1       martinjoo/posts-api:a5b15afc99f881b6475fbd30de800ea9a29a1881   node2     Running         Running 2 hours ago
xf805ecy35nx    \_ posts_api.1   martinjoo/posts-api:513997698d3eac459a14acbad576c864a1b3babf   node2     Shutdown        Shutdown 2 hours ago
ihgjt7aj6uki   posts_api.2       martinjoo/posts-api:a5b15afc99f881b6475fbd30de800ea9a29a1881   node4     Running         Running 2 hours ago
zfdgmzc86vrk    \_ posts_api.2   martinjoo/posts-api:513997698d3eac459a14acbad576c864a1b3babf   node4     Shutdown        Shutdown 2 hours ago
pjnojhdf5fes   posts_api.3       martinjoo/posts-api:a5b15afc99f881b6475fbd30de800ea9a29a1881   node3     Running         Running 2 hours ago
ums7x14rt8sy    \_ posts_api.3   martinjoo/posts-api:513997698d3eac459a14acbad576c864a1b3babf   node3     Shutdown        Shutdown 2 hours ago
hda8tf3cexpu   posts_api.4       martinjoo/posts-api:a5b15afc99f881b6475fbd30de800ea9a29a1881   node1     Running         Running 2 hours ago
zkdzkdzlx7k6    \_ posts_api.4   martinjoo/posts-api:513997698d3eac459a14acbad576c864a1b3babf   node1     Shutdown        Shutdown 2 hours ago
zek0sn4d32a3   posts_api.5       martinjoo/posts-api:a5b15afc99f881b6475fbd30de800ea9a29a1881   node2     Running         Running 2 hours ago
el565h6gmx6q    \_ posts_api.5   martinjoo/posts-api:513997698d3eac459a14acbad576c864a1b3babf   node2     Shutdown        Shutdown 2 hours ago
5kx0sa59a310   posts_api.6       martinjoo/posts-api:a5b15afc99f881b6475fbd30de800ea9a29a1881   node4     Running         Running 2 hours ago
cenmxc6kcbng    \_ posts_api.6   martinjoo/posts-api:513997698d3eac459a14acbad576c864a1b3babf   node4     Shutdown        Shutdown 2 hours ago
root@node1:/usr/src#
```

You can see that each node has some replicas of the `api` service. in this picture, you can see two kinds of desired states:

- Running
- Shutdown

It's because I was already running a replicated stack, and then I redeployed it. When you run `docker stack deploy` Swarm will shut down the currently running tasks and start new ones.

Scaling nginx vs the API

There's a frequently asked question when it comes to scaling an API: should I scale the nginx service that acts like a reverse proxy or the API itself or both?

To answer that, let's think about what nginx does in our application:

```
location ~\.php {
  try_files $uri =404;
  include /etc/nginx/fastcgi_params;
  fastcgi_pass api:9000;
  fastcgi_index index.php;
  fastcgi_param PATH_INFO $fastcgi_path_info;
  fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
}
```

It acts like a reverse proxy. It literally just receives requests and then forwards them right to the API. It's not a performance-heavy task.

If you just run in 1 replica on a server with 2 CPUs it means that it will spin up 2 worker processes with 1024 connections each. So it can handle a maximum of 2048 **concurrent** requests. Of course, it's not a guarantee, it's a maximum number. But it's quite a bug number. Now, I guarantee you that the bottleneck will be PHP and MySQL, not nginx.

In my opinion, you can scale your nginx but it won't bring you that many benefits. Meanwhile scaling the API is much more useful.

How many replicas?

Usually, the easiest way to decide on the number of replicas is to (kind of) match the number of CPUs in your server/cluster. My cluster has 4 nodes with 2 CPUs each. That's 8 CPUs. So in theory I can scale out the API to 8 replicas and Swarm will probably place 2 on each node. Which is great. But what about other services? Let's say node1 runs two API containers. But it also runs a worker. And nginx. And also the scheduler.

It's easy to see the problem. We have 8 CPUs so we just scale this way:

Container	Number of replicas
API	8
nginx	8
worker	8

(highlighting only the most important containers)

It's **important** to note that in Swarm I don't use supervisor. So a worker container just runs a `queue:work` process (details later).

Now there are 24 tasks running on 4 nodes and 8 CPUs. Meaning that each node runs 6 tasks with 2 CPUs. And I'm not even counting, MySQL, Redis, or the frontend.

If we reduce the number of replicas to something like that:

Container	Number of replicas
API	4
nginx	4
worker	4

Then there are 12 tasks running on 4 nodes with 8 CPUs. It means 3 tasks on each node with 2 CPUs. That sounds much better to me. You don't have to match the number of CPUs *exactly* because sometimes worker processes are idle for a long time. Other times, worker processes are busy but the API has a low traffic (for example, a few users running performance-heavy jobs).

But as I said, nginx can handle quite a lot of traffic so we can change the number of replicas such as this:

Container	Number of replicas
API	6
nginx	2
worker	4

We have the same number of containers but a different distribution:

- nginx does its job on the servers in 2 replicas. It can probably handle ~4000 concurrent connections.
- There's an API container running on all the nodes plus we have 2 extra ones, just in case. This is the container that will die the most and have the most trouble probably.
- There's a worker on every node. Of course, it depends highly on the nature of your application. For example, if you mainly use the queue to send notifications via an SMTP server you probably don't need 4 replicas.

Of course, it's not an exact science and it depends on your application and the nature of your traffic. So, unfortunately, I cannot give you exact numbers and formulas.

Worker

As I said earlier, I don't use supervisor in a cluster. The reason is that now worker processes can run on multiple servers so we don't really need 4 processes on one machine. It can be 4 containers across 4 nodes. Docker Swarm can play the part of a supervisor because it can:

- Run health checks and decide if a container is healthy or not
- Restart them if something went wrong
- Limit the resources (such as CPU or RAM) a container can use
- Limit the number of replicas a node can run from the same container

The configuration options offered by Swarm are huge.

As you might remember, the `worker` service defines a command that executes the `worker.sh` script:

```
worker:
  image: martinjoo/posts-worker:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && /usr/src/worker.sh"
```

I replaced `supervisord` in the `worker.sh` script to simply run this instead:

```
nice -10 php /usr/src/artisan queue:work --queue=default,notification --tries=3 --verbose --timeout=30 --sleep=3 --max-jobs=1000 --max-time=3600
```

With this config the `worker` service can be easily replicated:

```
worker:
  image: martinjoo/posts-worker:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && /usr/src/worker.sh"
  deploy:
    replicas: 4
```

Visualizing the cluster

There's a fantastic tool called `dockersamples/visualizer`. It's a very simple tool that can visualize a cluster, its node, the container, and its statuses.

It's a good idea to run these kinds of monitoring tools in a different stack. If you check out the project files, you can see a `docker-compose.monitoring.yml`:

```
version: "3.8"
services:
  visualizer:
    image: dockersamples/visualizer:stable
    deploy:
      placement:
        constraints:
          - "node.role==manager"
    ports:
      - "8080:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

Two important things:

- It needs to run on a manager node since it runs Swarm-related Docker commands to get information about the nodes and so on.
- It needs to have access to the `docker.sock` file on your machine. The way we can solve this is a volume. We mount the Docker socket from the host machine into the container. So when `visualizer` runs a command in its container such as `docker node ls` it actually receives information from the host machine.

I copied this file to the manager node and ran the following command:

```
docker stack deploy -c docker-compose.monitoring.yml monitoring
```

If I run the `docker stack ls` command it returns two stacks:

```
root@node1:/usr/src# docker stack ls
NAME          SERVICES
monitoring    1
posts         8
root@node1:/usr/src#
```

One for the project itself and one for the new monitoring stack I just deployed.

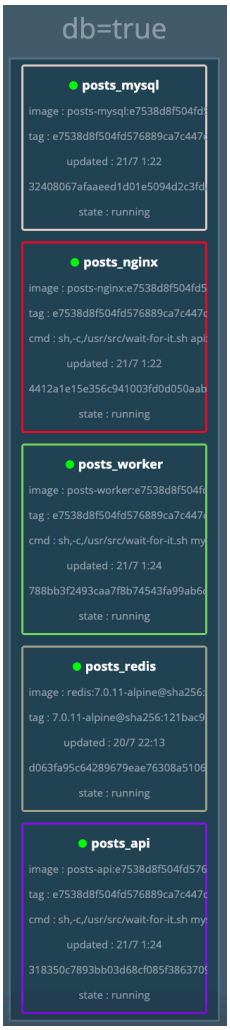
And if you now open one of your managers' IP addresses on port 8080 you'll see something like that:

You can see every node and the services running on them.

Visualizer works in real-time. So if you run a `docker stack deploy` you can see how your containers are being deployed in real-time.

Protecting the databases

If we zoom in on the visualizer image we can see that the database node is currently running 5 containers:



It runs the following services:

- MySQL
- Redis
- nginx
- API
- worker

It might be a little bit too much for a server with only 2 CPUs and 2G of RAM. If your application gets a big spike in traffic this node will suffer because:

- If you run lots of queries (and/or heavy ones) MySQL will consume a lot of your CPUs.
- The API container can also consume CPU based on your application and of course some RAM. In the optimization chapter, I showed you that an average PHP8 process in my case consumed 25Mb to 43MB of RAM. But I also showed you a legacy PHP7 project that sometimes consumes 130MB of RAM. And it's just one process, Now imagine if only 50 users are being served by this particular node.
- What if the worker process starts to generate big PDFs or resizes images uploaded by users?

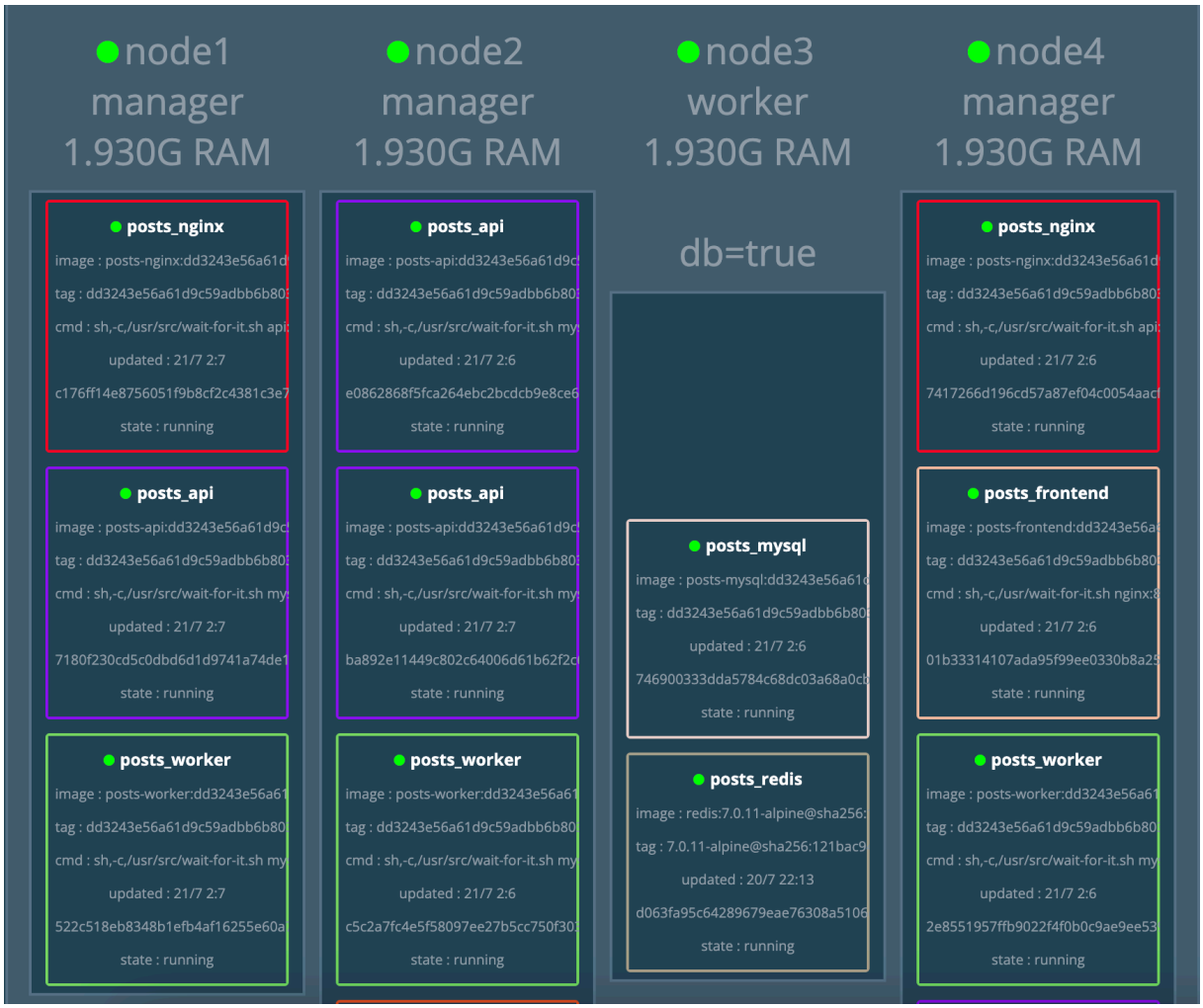
If you add all of these things together it's almost a guaranteed failure for your database node. Which makes the application and all of your other nodes useless. No database means no application. It's clearly a single-point failure right now. On top of that, we overload it with containers. And it doesn't need to be down necessarily. If this node is slowing down, probably your whole application is slowing down because the vast majority of your requests use a database.

Fortunately, it's easy to solve this problem. All we need to do is to place non-database containers on the other nodes:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && php-fpm"
  deploy:
    replicas: 6
    placement:
      constraints:
        - "node.labels.db≠true"
```

I did the same thing to the other non-database services as well.

And now node3 only runs MySQL and Redis:



Since now we effectively have a 3-node cluster, I decreased the number of replicas:

Container	Number of replicas
API	4
nginx	2
worker	3

I still want an extra API container. It means one of the nodes runs two APIs which is perfectly fine (node2 turned out to be the lucky one).

The result of this is a more reliable cluster and less risk.

Protecting user-facing service

Our cluster has another interesting attribute. It runs worker and nginx/frontend/API containers on the same nodes. nginx, frontend and API are containers directly responsible for the user experience and overall responsiveness of the application. If the worker starts a long-running heavy job, it can cause problems in the API container's performance. It consumes server resources from the other containers. Which might or might not be a problem in your application.

If your project has heavy queue jobs such as:

- Transcoding videos
- Processing audio
- Resizing and manipulating images
- Running antivirus scans for large files
- Creating large ZIP files or just working with many files in general

then it can be a problem since workers will overload the nodes and the API is going to be slow.

But there are other kinds of jobs that can be resource intensive as well. For example, importing or exporting large datasets. For example, I worked on a project where simply CSV imports caused real trouble.

This was the workflow:

- Tenants used our application to manage HR/PR-related tasks in their company.
- These companies already used some kind of ERP system where they had all the user information about their workforce.
- So they obviously didn't want to create and manage thousands (or tens of thousands) of users in another application as well.
- Because of that we had to sync users with CSV imports via API integrations.
- Some of the tenants were crazy (I don't want to name them, but you probably like their burgers) and they uploaded their CSV every hour.

So every hour a job started on the server that was scanning tens of thousands of rows and it:

- Queried the user in the database based on the CSV
- It updated the user if there were changes in the CSV
- It deleted the user based on a flag
- It created the user if they were new

Unfortunately, processing a user meant something like 7 or 8 queries. So just a 10 000 row CSV import can mean 80 000 database queries. Every hour (in this case). We had the worker processes on the same server as nginx or the API and you can imagine things were slow.

To avoid situations like that we can add dedicated worker nodes with a new label of `worker=true`. And we can place the worker container on these nodes:

```
worker:
  image: martinjoo/posts-worker:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && /usr/src/worker.sh"
  deploy:
    replicas: 2
    placement:
      constraints:
        - "node.labels.worker==true"
```

We also need to add another constraint to every other service:

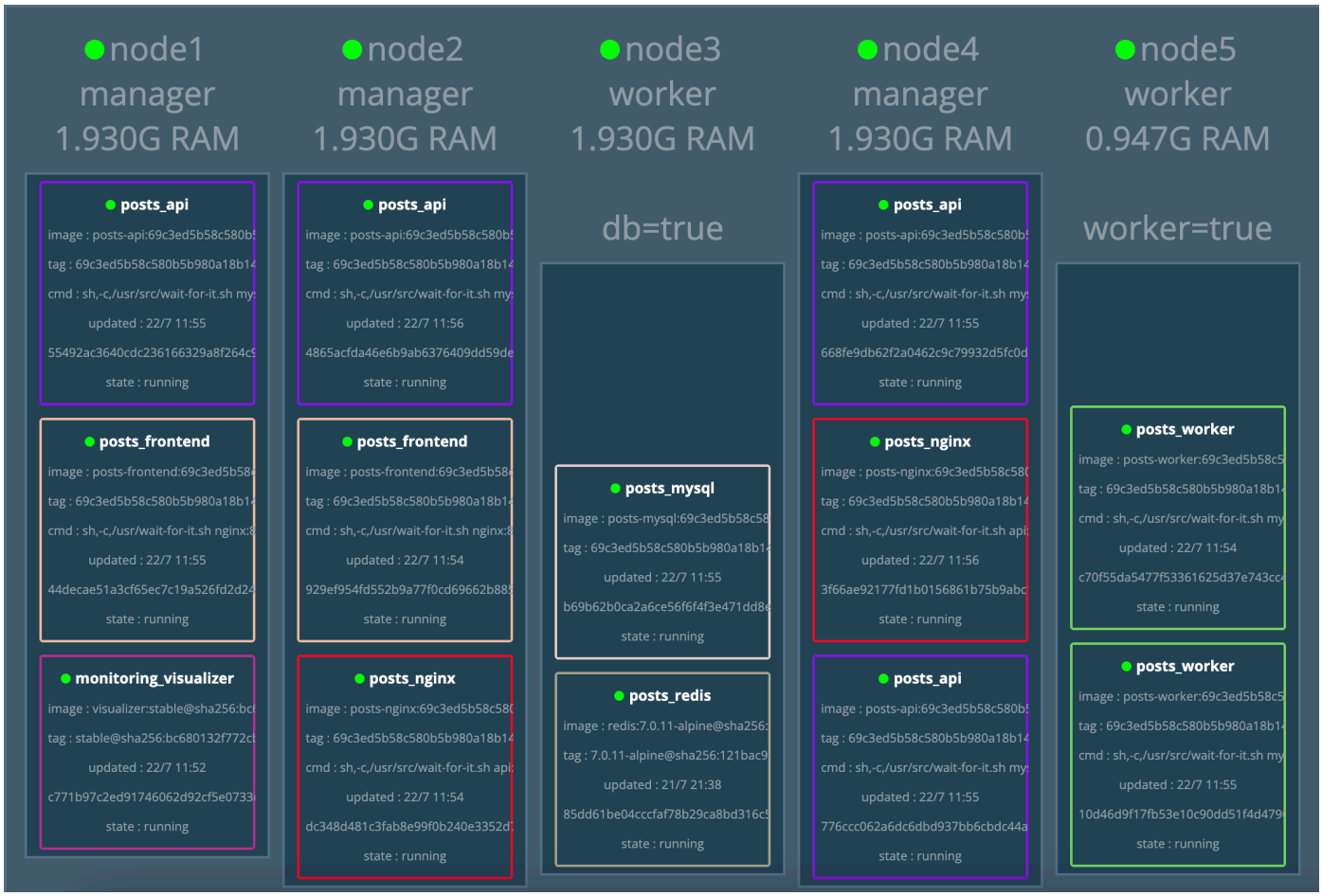
```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && php-fpm"
  deploy:
    replicas: 4
    placement:
      constraints:
        - "node.labels.db!=true"
        - "node.labels.worker!=true"
```

With this constraint, other containers won't be placed on the worker server.

To label a node we need to run this:

```
docker node update --label-add worker=true node5
```

And now we have a cluster such as this:



And of course, you can add even more worker servers, you can play around with the server size, the number of replicas, etc.

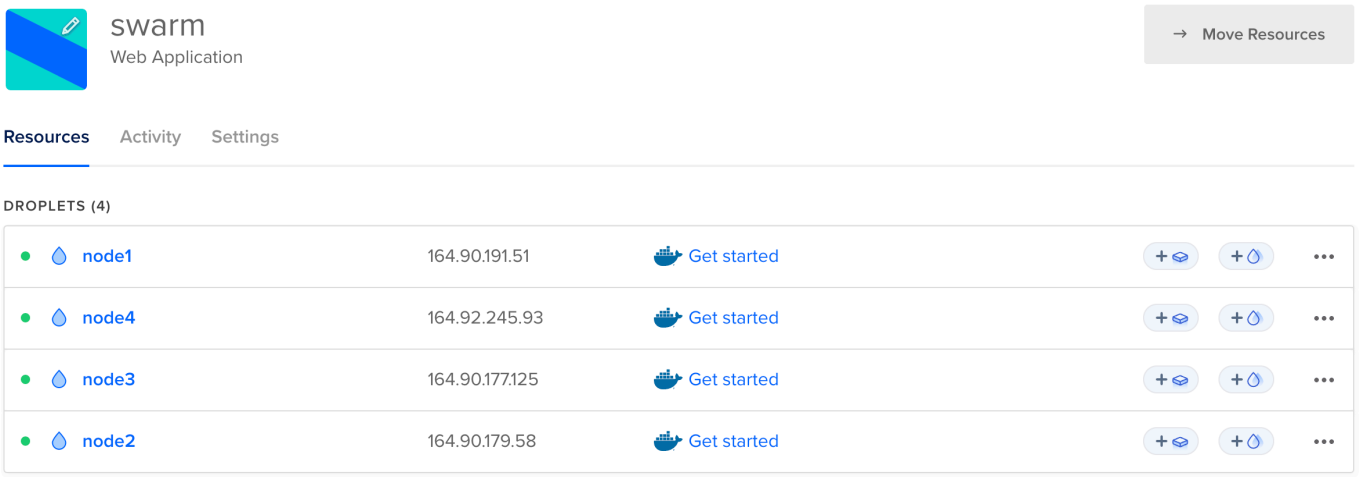
As I said earlier, lots of projects don't need this kind of separation.

You can find this configuration in the `docker-compose.dedicated-worker.example.yml` file.

Ingress routing mesh

In the cluster, I have 4 nodes. 3 of them are managers and of them is a worker. And the thing is that I can access the frontend or the API on any of these nodes.

Here are the nodes:



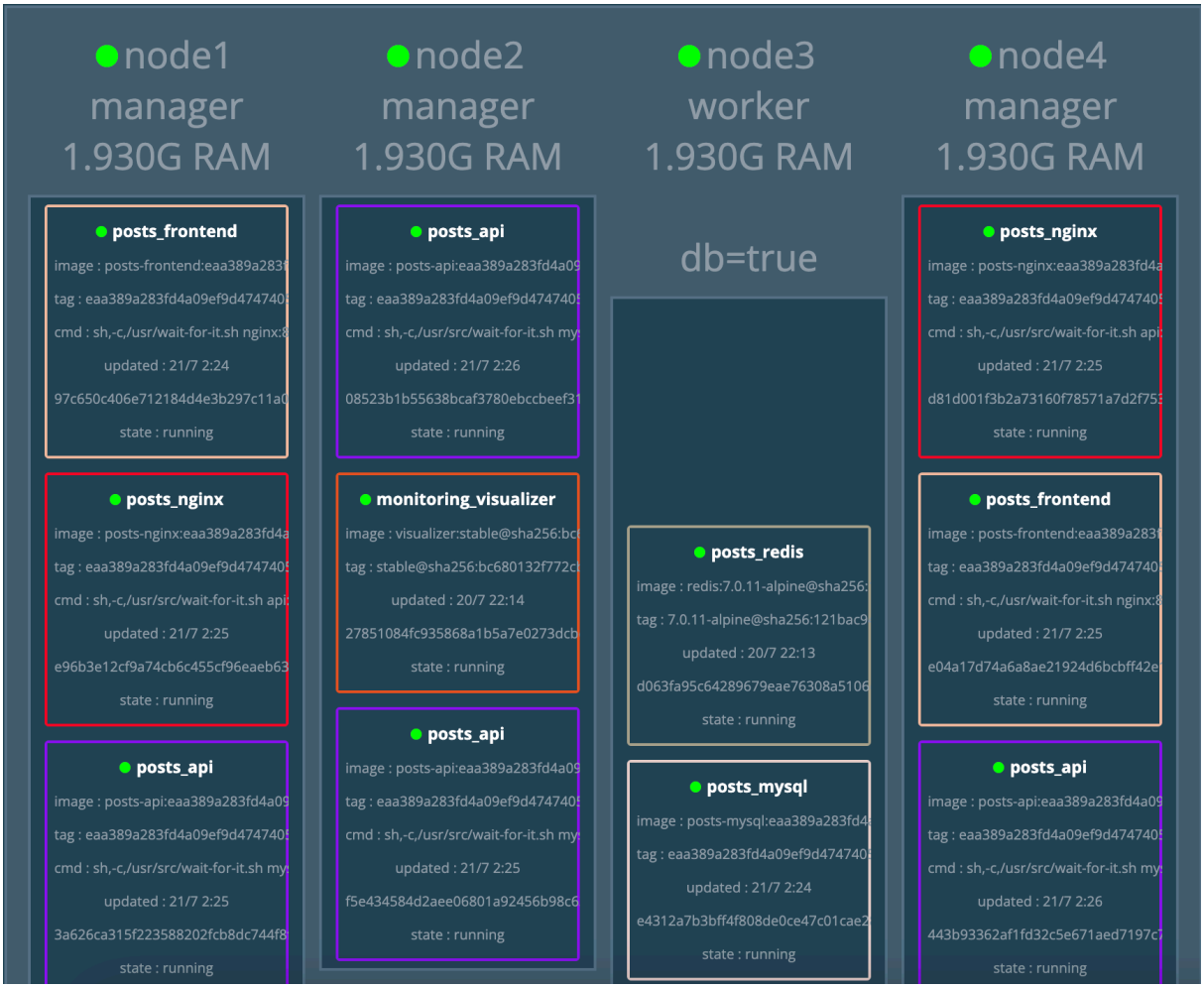
If I hit these IP addresses on port 80 I always got a 200 response:

```
→  ~ curl -s -o /dev/null -w "%{http_code}" 164.90.191.51
200%
→  ~ curl -s -o /dev/null -w "%{http_code}" 164.92.245.93
200%
→  ~ curl -s -o /dev/null -w "%{http_code}" 164.90.177.125
200%
→  ~ curl -s -o /dev/null -w "%{http_code}" 164.90.179.58
200%
```

The same happens if I send requests to the API:

```
→  ~ curl -H "Authorization: Bearer 1|WC73QSzfSZ6gkypbSvp2LMDtGBY1rF8mmtZqQQRE" http://164.90.179.58:8000/api/posts
{"data":[{"id":2,"title":"Second","headline":"second","content":"asdf","is_published":0,"publish_at":null,"author":{"id":1,"name":"Martin Joo","email":"m4rt1n
.j00@gmail.com"},"cover_photo_url":"https:\/\/devops-with-laravel-storage.s3.amazonaws.com\/post_cover_photos\/second-2.png?X-Amz-Content-Sha256=UNSIGNED-PAYL
OAD&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAVS45JDI3HZI5OSEW%2F20230721%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20230721T181940Z&X-Amz-Signed
Headers=host&X-Amz-Expires=600&X-Amz-Signature=5926e9c90c744d01238f3ddb46283b4ba804fab4ac7a4abd755031d2bcb55b99"},{"id":1,"title":"First","headline":"first","
content":"first","is_published":1,"publish_at":"2023-07-20T00:00:00.000000Z","author":{"id":1,"name":"Martin Joo","email":"m4rt1n.j00@gmail.com"},"cover_photo
_url":"https:\/\/devops-with-laravel-storage.s3.amazonaws.com\/post_cover_photos\/first-1.png?X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Algorithm=AWS4-HMAC-
SHA256&X-Amz-Credential=AKIAVS45JDI3HZI5OSEW%2F20230721%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20230721T181940Z&X-Amz-SignedHeaders=host&X-Amz-Expires=600
&X-Amz-Signature=c5d9d9f6239e780f68c72b09fca0bbd144c9a8e10e6ffe9d374f27023f07ad0b"}]}%
→  ~ curl -H "Authorization: Bearer 1|WC73QSzfSZ6gkypbSvp2LMDtGBY1rF8mmtZqQQRE" -s -o /dev/null -w "%{http_code}" http://164.92.245.93:8000/api/posts
200%
→  ~ curl -H "Authorization: Bearer 1|WC73QSzfSZ6gkypbSvp2LMDtGBY1rF8mmtZqQQRE" -s -o /dev/null -w "%{http_code}" http://164.90.177.125:8000/api/posts
200%
```

Yes, right now API runs on every node, but the FE does not. It runs in two replicas on node1 and node4:



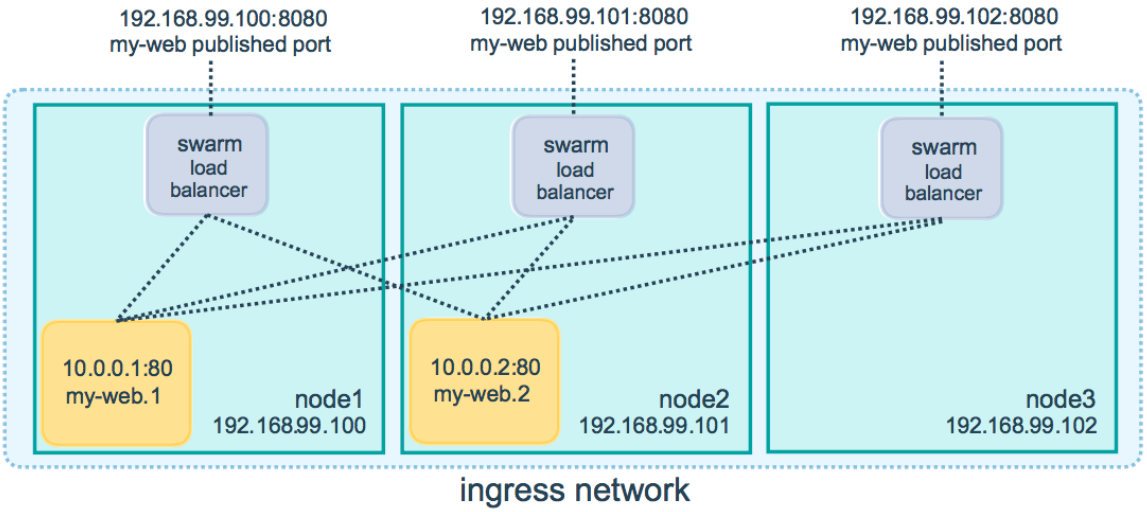
But I was still able to access every node. Something is clearly going on in the background.

This is called the **ingress routing mesh**. This is how Docker defines it:

> Docker Engine swarm mode makes it easy to publish ports for services to make them available to resources outside the swarm. All nodes participate in an ingress **routing mesh**. The routing mesh enables each node in the swarm to accept connections on published ports for any service running in the swarm, even if there's no task running on the node. The routing mesh routes all incoming requests to published ports on available nodes to an active container.

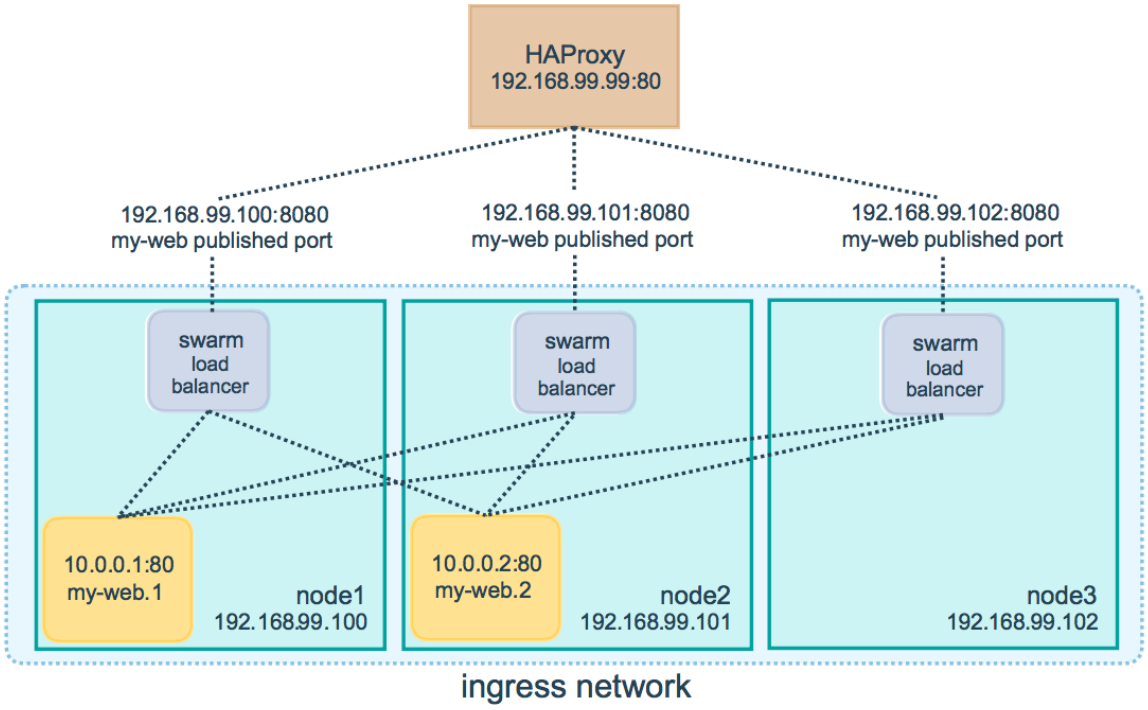
This is why we need to open ports `7946` and `4789` on the nodes. Routing mesh uses these ports for container discovery and communication.

This is what it looks like:



When you hit a port that is publicly accessible the request is going through an internal load balancer provided by Swarm and it routes the request to a container that can handle it. This is because you can access port 80 on every node. Swarm routes the request to a container on a node that listens to that port.

So basically Docker Swarm comes with a fully functional load balancer. Without any configuration. But you can still configure your own load balancer. If you want to use the routing mesh with your own load balancer that's all you need to do:



If you want to learn more check out the official Docker page about routing mesh where you can see the exact configuration of the load balancer.

Health checks

One of the great things about Docker Swarm is its health checks and self-healing features. We can define some pretty great health checks for the containers and Swarm will start and restart them according to their states.

This is a health check for the MySQL container:

```
mysql:
  image: martinjoo/posts-mysql:${IMAGE_TAG}
  healthcheck:
    test: [ "CMD", "mysqladmin", "ping" ]
    interval: 30s
    timeout: 5s
    retries: 3
    start_period: 30s
```

- It runs the `mysqladmin ping` command to check if the database is available
- It runs the command every 30s defined in `internal`. When the container is starting Swarm will wait 30s to run the first check. You can imagine like this: `sleep 30 && mysqladmin ping` instead of `mysqladmin ping && sleep 30`
- If the `ping` command takes more than 5s defined in `timeout` the check fails.
- If 3 check fails in a row to container is considered to be unhealthy. This is defined in `retries`
- We give 30s defined in `start_period` to the container to start. Failed health checks in the first 30s won't increase the retries count.

Redis also has a built-in `ping` command we can use:

```
redis:
  image: redis:7.0.11-alpine
  healthcheck:
    test: [ "CMD", "redis-cli", "ping" ]
    interval: 30s
    timeout: 5s
    retries: 3
    start_period: 30s
```

The API runs a php-fpm service on port 9000 so the best health check is to connect to that port. For this, I'm using `nc`:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  healthcheck:
    test: [ "CMD", "nc", "-zv", "localhost", "9000" ]
    interval: 30s
    timeout: 5s
    retries: 3
    start_period: 40s
```

Since the API depends on MySQL, Redis, and updates I gave it a bit more start period. We also need to install `nc` in the Dockerfile:

```
RUN apt-get update && apt-get install -y netcat-openbsd
```

After that, in the nginx container, we can actually send an HTTP request to the API as a health check:

```
nginx:
  image: martinjoo/posts-nginx:${IMAGE_TAG}
  healthcheck:
    test: [ "CMD", "curl", "-f", "http://localhost/api/health-check" ]
    interval: 30s
    timeout: 5s
    retries: 3
    start_period: 1m
```

nginx depends on even more containers so I raised the start_period to 1 minute. Of course, it won't take that much. The health check sends an HTTP request to localhost using curl. I added this pretty simple `health-check` endpoint:

```
Route::get('/health-check', function () {
  return response('', 200);
});
```

It doesn't need to return anything. It's either a `200` or a `5xx` response.

The worker container checks if it can access the queue:

```
worker:
  image: martinjoo/posts-worker:${IMAGE_TAG}
  healthcheck:
    test: [ "CMD", "php", "/usr/src/artisan", "queue:monitor", "default" ]
    interval: 30s
    timeout: 5s
    retries: 3
    start_period: 40s
```

It's a simple command that returns the available queues:

Scheduler doesn't need a health check since it's a one-off container.

The frontend container can also just curl itself on port 80:

```
frontend:
  image: martinjoo/posts-frontend:${IMAGE_TAG}
  healthcheck:
    test: [ "CMD", "curl", "-f", "http://localhost" ]
    interval: 30s
    timeout: 5s
    retries: 3
    start_period: 1m30s
```

It's the last container in the dependency chain so it has the largest `start_period`.

Of course, you can and should customize these numbers to your own need. You can also play with the `timeout` numbers (I'm using a default 5s for everything). But it can vary based on some criteria, for example:

- If you have a typical application usually it's okay if your worker is a bit slower for a period of time. So 5s can be perfect.
- But probably it's a problem if the API is slow. So maybe you can change it to 2 or 3 seconds.

Now that every service has health checks and they also use `wait-for-it` you can be sure that the startup state of your application is stable and flawless. But of course, health checks also play a big role in restarting containers when they run into some problem.

Restarting services

Swarm also comes with great restart abilities. We can define how we want to restart containers that are stopped or in an unhealthy state:

```
mysql:
  image: martinjoo/posts-mysql:${IMAGE_TAG}
  deploy:
    restart_policy:
      condition: any
      delay: 5s
      max_attempts: 3
      window: 60s
```

- `condition` can be `none`, `on-failure`, or `any`. In most cases, I'd like to use `any` since I can't imagine a scenario where a container stops working and there's no need to restart it. The `update` container it's an exception. It's a one-off job.
- `delay` means that Swarm will wait for 5s before restarting the container.
- `max_attempts` specifies how many times Swarm attempts to restart the container.
- `windows` specifies how long to wait before deciding if a restart has succeeded or not. This value should match the `start_period` in the `healthcheck` section and/or the time given to `wait-for-it`

Every container in the sample stack has a similar restart policy to this one except `update`. It doesn't need to be restarted at all.

And of course, `scheduler` requires exactly 60s of delay:

```
scheduler:
  image: martinjoo/posts-scheduler:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && /usr/src/scheduler.sh"
  deploy:
    restart_policy:
      condition: any
      delay: 60s
      window: 30s
```

It runs the `scheduler.sh` script, stops, waits for 60s seconds then restarts it. If you remember, with docker-compose we had this script:

```
nice -n 10 sleep 60 && php /usr/src/artisan schedule:run --verbose --no-interaction
```

We used `sleep 60` because docker-compose can't handle delays.

But now `delay 60s` replaces `sleep 60` so `scheduler.sh` looks like this:

```
nice -n 10 php /usr/src/artisan schedule:run --verbose --no-interaction
```

A tiny difference but I think it's better that the orchestrator does the scheduling as well.

Updating services

Swarm can also intelligently update the services when you re-deploy your stack. Meaning, it doesn't cause downtime.

The update process be configured such as this:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && php-fpm"
  deploy:
    replicas: 4
    update_config:
      parallelism: 2
      delay: 10s
      failure_action: pause
      monitor: 20s
      max_failure_ratio: 0.25
      order: stop-first
```

- Swarm will update two containers at a time defined by `parallelism`. It means that it stops 2 containers from the currently running ones and then starts two new ones with the new image.
- After stopping and starting two containers it waits for 10s because of the `delay`
- If the new containers fail it will `pause` the update. This means that all the containers that were running the previous version of the service will continue to run and serve traffic, while no new containers will be created or updated until the issue that caused the failure is resolved manually. So you had 4 working replicas, and you rolled out a deployment that fails. There are still 2 old containers serving traffic while you're working on the hotfix. `pause` is the default value so you can omit it. The other two options are `continue` and `rollback`. In most cases, `pause` is your best option. `continue` is a bit too risky, however, `rollback` can be useful. We're going to talk about it in a minute.
- Swarm will monitor the status of the new containers for the 20s defined by `monitor`. If they fail in this interval it's considered a failure and `failure_action` will be triggered. Otherwise, the update is considered successful. Updates happen in order. So when `api` is updated `mysql` and `redis` are already up and running according to their health checks.
- If 25% of the new containers fail the whole update process is considered a failure and is halted. This is defined by `max_failure_ratio` which is `0.25` which translates to 25% which means exactly 1 container since `api` runs in 4 replicas. So if one of the API containers fails, we stop the update of the `api` containers. However, in a real project, I don't want to tolerate failures at all, so I set `max_failure_ratio` to 0. Which is the default value so you can omit it.

- Swarm will first stop 2 runnings containers and then starts two new ones. This is controlled by `order: stop-first`. The other value is `start-first` `stop-first` is almost always the safest choice since `start-first` can cause problems. For example, you can write a constraint that prevents the `api` service to run more than 2 replicas on the same node. `start-first` might violate that constraint. It can also overload your node with too many tasks.

Let's see how we can use the `rollback` `failure_action`:

```
mysql:
  image: martinjoo/posts-mysql:${IMAGE_TAG}
  deploy:
    update_config:
      parallelism: 1
      failure_action: rollback
      monitor: 30s
      max_failure_ratio: 0
      order: stop-first
```

This is the MySQL service. It runs only one replica. It's a single point of failure. If the update fails you're in deep shit. The whole application is down. This is why I'm using this config.

```
failure_action: rollback
```

If something goes wrong with the update Swarm will roll back to the previous version. This is crucial. With `failure_action: pause` your whole application would be down until you figure out the problem. And in this case, a rollback is pretty much safe in my opinion. The MySQL image is basically just the official image with a config file in it. So an update will go wrong if you updated the config, didn't test it, and deployed it. Which doesn't happen that often. And if it does happen Swarm will roll back to the previous image. Which means either an older MySQL version or a different config. Hopefully, there's no such version of config that would cause serious problems in your application. So your application most likely will work with the older image.

Rolling back services

Sometimes you might want to roll back a specific service to its previous version. Maybe you deployed a new version of the API but it contains some critical bugs and it's easier/faster to roll back to the previous version until the fix is done. You can do so by running:

```
docker service rollback posts_api
```

It's a great feature that can save the day. Rollbacks can be configured as well, and the good news is that it has exactly the same properties as `update_config`:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  deploy:
    replicas: 4
    update_config:
      parallelism: 2
      delay: 10s
      failure_action: pause
      monitor: 15s
      max_failure_ratio: 0
      order: stop-first
    rollback_config:
      parallelism: 2
      delay: 10s
      failure_action: pause
      monitor: 15s
      max_failure_ratio: 0
      order: stop-first
```

As you can see the `rollback_config` is exactly the same as the `update_config`. For me, it makes sense to use the same values when rolling back.

Deployment

Deploying from a pipeline

Deploying from the pipeline it's almost entirely the same as it was with docker-compose. In the GitHub workflow, we need to do the exact same steps:

```
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"
      ssh -tt ./id_rsa ${{ secrets.SSH_CONNECTION_PROD }}
"/home/martin/deploy.sh"
```

You need set one of the manager nodes' IP address in a GitHub secret to SSH into it. Just like we did earlier.

After these steps the `.env` file is ready and the deploy script is on the server. It's super simple:

```
#!/bin/bash

set -e

cd /usr/src

export $(cat .env)
docker stack deploy -c docker-compose.prod.yml posts
```

That's it. The pipeline now can deploy the stack.

Update service

The `update` service (that runs migrations and caches configs) doesn't need to be changed at all. The only difference is that it's not a long-running service but a one-off short-lived task.

For this, Swarm offers a special service type called `replicated-job`:

```
update:
  image: martinjoo/posts-api:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && /usr/src/update.sh"
  deploy:
    mode: replicated-job
```

This will run the container in one replica and when the `update.sh` script exists Swarm will not restart the service. A `replicated-job` cannot have an update or rollback-related configs since it doesn't make sense.

Provisioning nodes

Provisioning a new node is exactly the same as before. In the Docker chapter, I explained the whole script in detail.

There are only three new steps we need to add:

```
docker login -u "$DOCKERHUB_USERNAME" -p "$DOCKERHUB_TOKEN"

ufw allow 2377 && ufw allow 7946 && ufw allow 4789

docker swarm join --token "$SWARM_TOKEN" "$SWARM_MANAGER_IP_PORT"
```

We log in to Docker Hub, then open the necessary ports. As the final step new server needs to join the cluster. For this, the script accepts a token as its parameter. If it's a manager token the new node is going to be a manager, if it's a worker token then a worker, of course. `$SWARM_MANAGER_IP_PORT` is also an argument. It's the manager node's IP address and port number such as `1.2.3.4:2377`

Monitoring and error tracking

The project files are located in the `6-log-collecting` folder.

Monitoring means a lot of things. In this chapter, I'm going to show some techniques you can use. We start with simpler solutions and then move forward to more complicated ones.

Uptime

This is the most basic and crucial monitoring tool you should have. Monitoring uptime means that there's a service that calls your API every minute and notifies you if it's not responding. We did the same thing on the container level with Swarm health checks. But in this case, I'm talking about an external tool on the "network level."

There are a lot of these tools, such as:

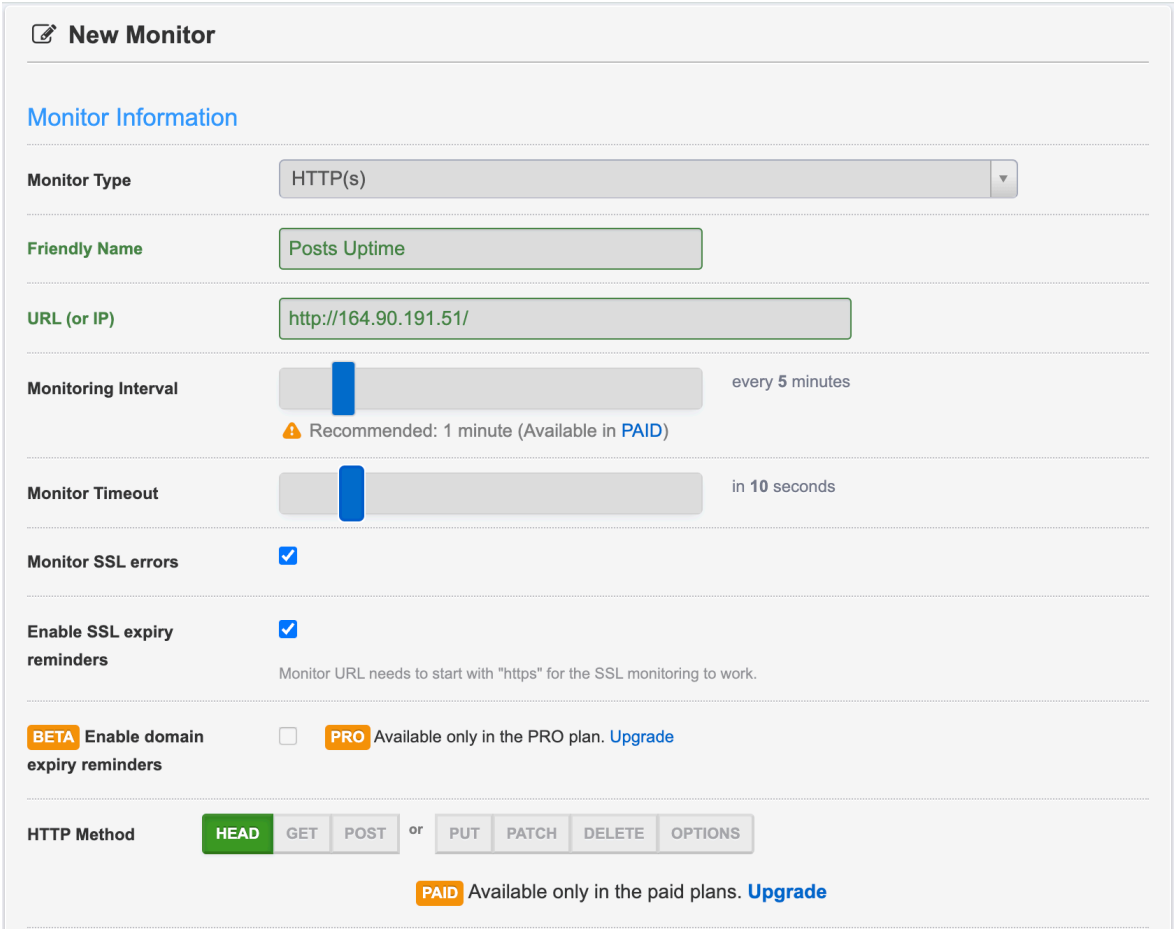
- Uptime robot
- Better uptime
- More advanced services such as Oh dear
- Uptime checks provided by your server provider

We're going to take a quick look at Uptime robot and DigitalOcean's uptime service.

Uptime robot

You need to configure a few basic things:

- Your domain
- The interval they send requests to your site
- Timeout after they send you an alert

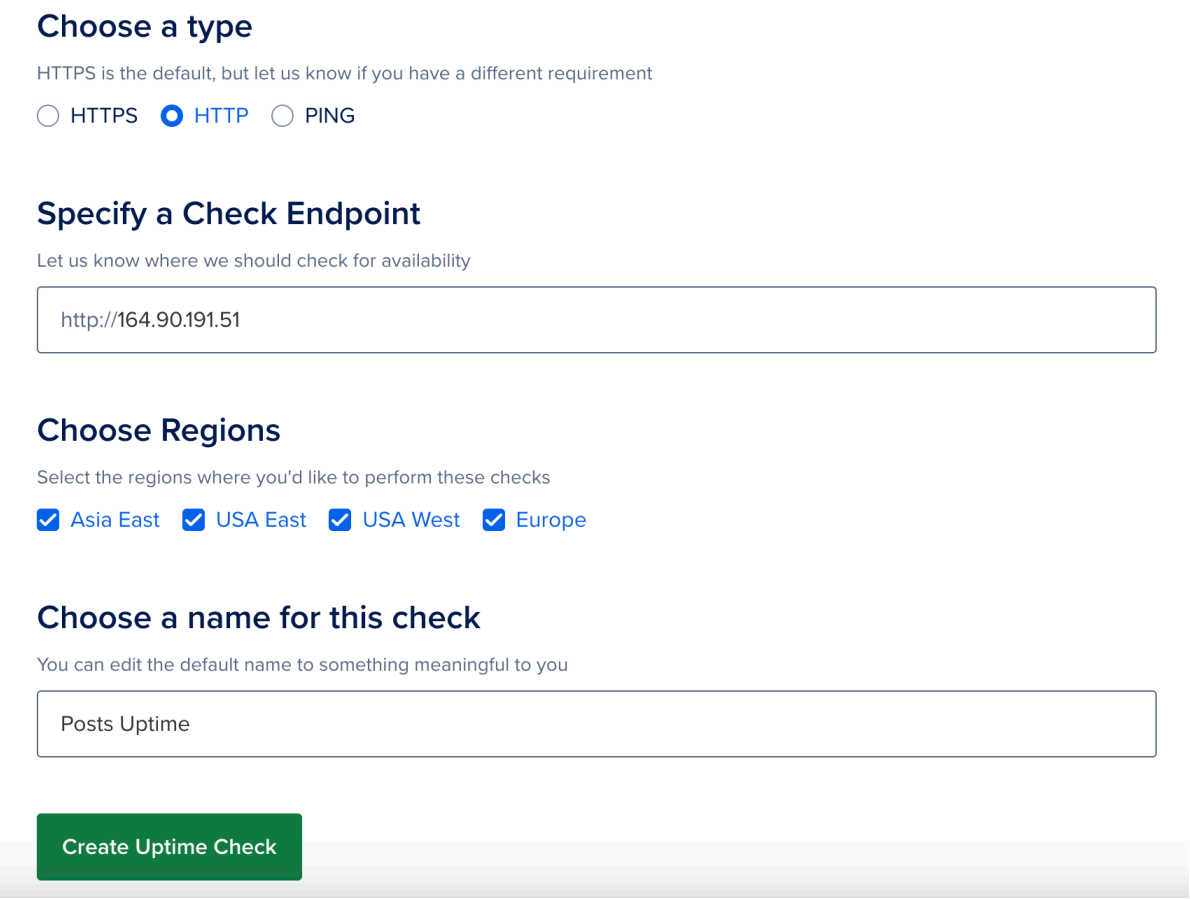


The free account only allows you to use 5-minute intervals. The ideal would 1 minute. This config will run every 5 minutes and sends an alert if the site isn't responding after 10 seconds or it returns a not-healthy status code (4xx, 5xx).

DigitalOcean

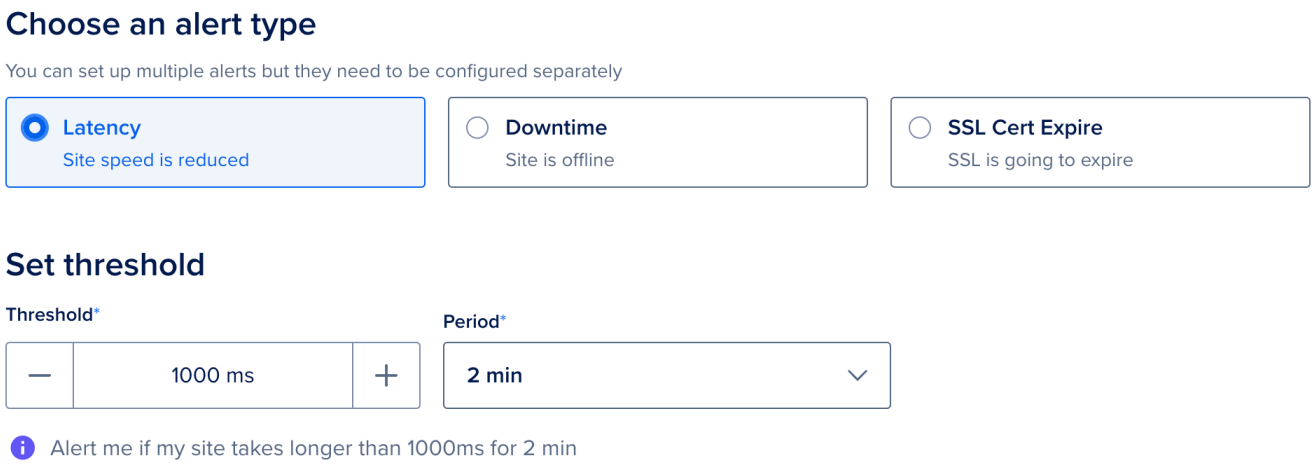
Every server provider provides you with monitoring tools. On DigitalOcean they are completely free. However, there are other providers (such as Azure) where it costs money.

They also have uptime monitoring for free. In the left sidebar choose `Manage` -> `Monitoring` -> `Uptime`:



They even have region settings. This means they try to hit your site from 4 different regions.

After you created your monitor, you can create different types of alerts:



It's pretty straightforward:

- Latency notifies you if your site isn't responding in 1s in a 2-minute period.
- Downtime notifies you if your site is down.
- An SSL Cert Expire notifies if your SSL certificate is going to expire in N days.

Depending on your server provider use these monitors and alerts, or choose something like Uptime robot or Oh dear.

Health check monitors

There are two excellent packages:

- spatie/laravel-health
- pragmarx/health

Both of them do the same thing. They can monitor the health of the application. Meaning:

- How much free disk space does your server have?
- Can the application connect to the database?
- What's the current CPU load?
- How many MySQL connections are there?
- etc

You can define the desired threshold and the package will notify you if necessary. I'm going to use the Spatie package but the other one is also pretty good. Spatie is more code-driven meanwhile the Pragmarx package is more configuration-driven.

Please refer to the installation guide here.

This is what a spatie/laravel-health configuration looks like:

```
<?php

namespace App\Providers;

class HealthCheckServiceProvider extends ServiceProvider
{
  public function boot(): void
  {
    Health::checks([
        UsedDiskSpaceCheck::new(),
        DatabaseCheck::new(),
        CpuLoadCheck::new()
          ->failWhenLoadIsHigherInTheLast5Minutes(2)
          ->failWhenLoadIsHigherInTheLast15Minutes(1.75),
        DatabaseConnectionCountCheck::new()
          ->warnWhenMoreConnectionsThan(50)
          ->failWhenMoreConnectionsThan(100),
        DebugModeCheck::new(),
        EnvironmentCheck::new(),
```

```
            RedisCheck::new(),
            RedisMemoryUsageCheck::new(),
            QuerySpeedCheck::new(),
        ]);
    }
}
```

We're using the following checks:

- `UsedDiskSpaceCheck` warns you if more than 70% of the disk is used and it sends an error message if more than 90% is used.
- `DatabaseCheck` checks if the database is available or not.
- `CpuLoadCheck` measures the CPU load (the numbers you can see when you open `htop`). It sends you a failure message if the 5-minute average load is more than 2 or if the 15-minute average is more than 1.75. I'm using 2-core machines in this project a load of 2 means both the cores run at 100%. If you have a 4-core CPU 4 means 100% load.
- `DatabaseConnectionCountCheck` sends you a warning if there are more than 50 connections and a failure message if there are more than 100 MySQL connections.
- `DebugModeCheck` simply checks if the `APP_DEBUG` is equal `true` .
- `EnvironmentCheck` makes sure that you're application is running in `production` mode.
- `RedisCheck` tries to connect to Redis and notifies you if the connection cannot be established.
- `RedisMemoryUsageCheck` sends you a message if Redis is using more than 500MB of memory.

These are the basic checks you can use in almost every project.

To be able to use the `CpuLoadCheck` and the `DatabaseConnectionCountCheck` you have to install these packages as well:

- `spatie/cpu-load-health-check`
- `doctrine/dbal`

The package can send you e-mail and Slack notifications as well. Just set up them in the `health.php` config file:

```
'notifications' => [
  Spatie\Health\Notifications\CheckFailedNotification::class => ['mail'],
],

'mail' => [
  'to' => env('HEALTH_CHECK_EMAIL', 'healthcheck@posts.today'),
  'from' => [
    'address' => env('MAIL_FROM_ADDRESS', 'healthcheck@posts.today'),
    'name' => env('MAIL_FROM_NAME', 'Health Check'),
  ],
],

'slack' => [
    'webhook_url' => env('HEALTH_SLACK_WEBHOOK_URL', ''),
    'channel' => null,
    'username' => null,
    'icon' => null,
],
```

Finally, you need to run the command provided by the package every minute:

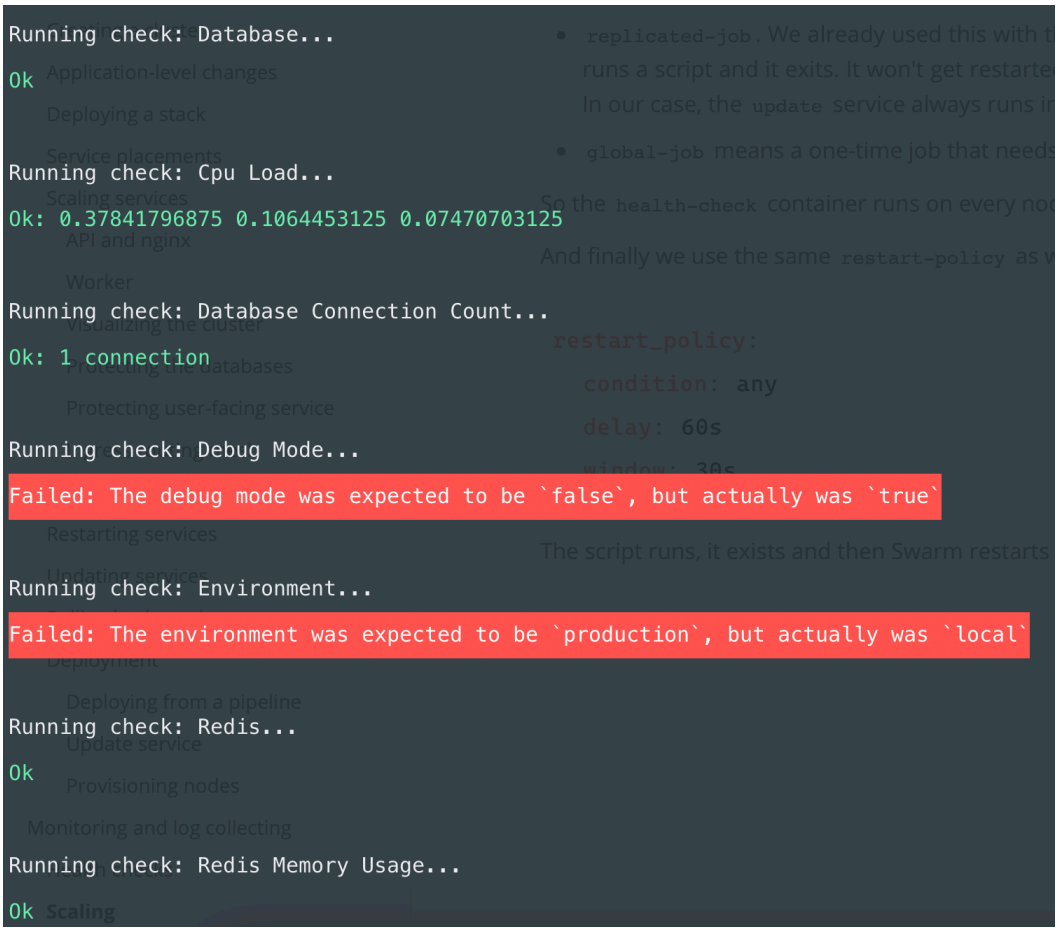
```
$schedule->command('health:check')->everyMinute();
```

You can also write your own checks. For example, I created a `QuerySpeedCheck` that simply measures the speed of an important query:

```php
namespace App\Checks;

class QuerySpeedCheck extends Check
{
  public function run(): Result
  {
    $result = Result::make();

    $executionTimeMs = Benchmark::measure(function () {
      Post::with('author')->orderBy('publish_at')->get();
    });

    if ($executionTimeMs ≥ 50 && $executionTimeMs ≤ 150) {
      return $result->warning('Database is starting to slow down');
    }

    if ($executionTimeMs > 150) {
      return $result->failed('Database is slow');
    }

    return $result->ok();
  }
}
```

In the sample application I don't have too many database queries, but selecting the posts with authors is pretty important and happens a lot. So if this query cannot be executed in a certain amount of time it means the application might be slow. The numbers used in this example are completely arbitrary. Please measure your own queries carefully before setting a threshold.

If you run the command it gives you an output such as this:


Running check: Database...
Ok
Running check: Cpu Load...
Ok: 0.37841796875 0.1064453125 0.07470703125
Running check: Database Connection Count...
Ok: 1 connection
Running check: Debug Mode...
Failed: The debug mode was expected to be `false`, but actually was `true`
Running check: Environment...
Failed: The environment was expected to be `production`, but actually was `local`
Running check: Redis...
Ok
Running check: Redis Memory Usage...
Ok

Health checks in a cluster

If your project runs on a single machine, you're done with health checks. Just schedule that command, and it should work.

However, in a cluster, we have to do some extra work. Just imagine the following:

- In the previous chapter I created a 4-node cluster.
- One of them was dedicated to databases
- The scheduler container runs in 1 replica on a random node (except the database node of course).
- This means that at any given time we're only monitoring one of the nodes. And the check will never run on the database server.

This is not optimal. We want these checks to run:

- Every minute
- On all the nodes

To accomplish this we need a new service that runs on every node.

The first step is to add a new stage to the API image. It's pretty similar to the worker or scheduler stages:

```
FROM api AS worker
CMD ["/bin/sh", "/usr/src/worker.sh"]

FROM api AS scheduler
CMD ["/bin/sh", "/usr/src/scheduler.sh"]

FROM api AS health-check
CMD ["/bin/sh", "/usr/src/health-check.sh"]
```

The `health-check.sh` contains one line:

```
#!/bin/bash

nice -n 10 php /usr/src/artisan health:check
```

It just runs the command provided by the Spatie package.

And now we can add a new service to the Swarm stack:

```
health-check:
  image: martinjoo/posts-health-check:${IMAGE_TAG}
  command: sh -c "/usr/src/wait-for-it.sh mysql:3306 -t 60 && /usr/src/wait-for-it.sh redis:6379 -t 60 && /usr/src/health-check.sh"
  deploy:
    mode: global
    restart_policy:
      condition: any
      delay: 60s
      window: 30s
  depends_on:
    - mysql
    - redis
```

I'm using the `martinjoo/posts-health-check` image which is built in the pipeline (just like worker or scheduler):

```
- 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 $HEALTH_CHECK_IMAGE --target=health-check --build-arg user=martin --build-arg uid=1000 -f ./api/Dockerfile .
```

But the most important thing is this:

```
deploy:
  mode: global
```

This guarantees that the container runs on every node in exactly 1 replica. This is called a global service. There are four different kinds of modes:

- `replicated` is the default. This means we want to run X number of replicas of service on random nodes (unless you use a placement constraint, of course).
- `global` means that each node should run exactly one replica of this service.
- `replicated-job`. We already used this with the `update` service. So a `job` means that it's a one-off task. The container starts, runs a script and it exits. It won't get restarted. It's a one-time job. `replicated-job` means it can run in X number of replicas. In our case, the `update` service always runs in 1 replica.
- `global-job` means a one-time job that needs to run on every node.

So the `health-check` container runs on every node. Which is great because it can measure the current server's health.

And finally, we use the same `restart-policy` as we used with the `scheduler` service:

```
restart_policy:
  condition: any
  delay: 60s
  window: 30s
```

The script runs, it exists and then Swarm restarts the container after a 60s delay so it runs every minute.

If you check out the logs with:

```
docker service logs -f -t posts_health-check
```

You can see it runs on different nodes:

```
2023-07-26T17:16:48.244138531Z posts_health-check.0.lhk45qfhcvc3@node4    | Running check: Query Speed...
2023-07-26T17:16:48.267243926Z posts_health-check.0.lhk45qfhcvc3@node4    | Ok
2023-07-26T17:16:48.327635955Z posts_health-check.0.vv6oc64jxna0@node1    |
2023-07-26T17:16:48.329718106Z posts_health-check.0.vv6oc64jxna0@node1    | All done!
2023-07-26T17:16:48.344471371Z posts_health-check.0.lhk45qfhcvc3@node4    |
2023-07-26T17:16:48.345218647Z posts_health-check.0.lhk45qfhcvc3@node4    | All done!
2023-07-26T17:17:23.303063961Z posts_health-check.0.pdwg70uf0bcu@node2    | wait-for-it.sh: waiting 60 seconds for mysql:3306
2023-07-26T17:17:23.319721228Z posts_health-check.0.pdwg70uf0bcu@node2    | wait-for-it.sh: mysql:3306 is available after 0 seconds
2023-07-26T17:17:23.345461062Z posts_health-check.0.pdwg70uf0bcu@node2    | wait-for-it.sh: waiting 60 seconds for redis:6379
2023-07-26T17:17:23.362317769Z posts_health-check.0.pdwg70uf0bcu@node2    | wait-for-it.sh: redis:6379 is available after 0 seconds
2023-07-26T17:17:23.897278500Z posts_health-check.0.pdwg70uf0bcu@node2    | Running checks...
2023-07-26T17:17:23.907132914Z posts_health-check.0.pdwg70uf0bcu@node2    |
2023-07-26T17:17:23.908436104Z posts_health-check.0.pdwg70uf0bcu@node2    | Running check: Used Disk Space...
2023-07-26T17:17:23.920543058Z posts_health-check.0.pdwg70uf0bcu@node2    | Ok: 70%
```

These log entries come from `node1`, `node2`, and `node4`.

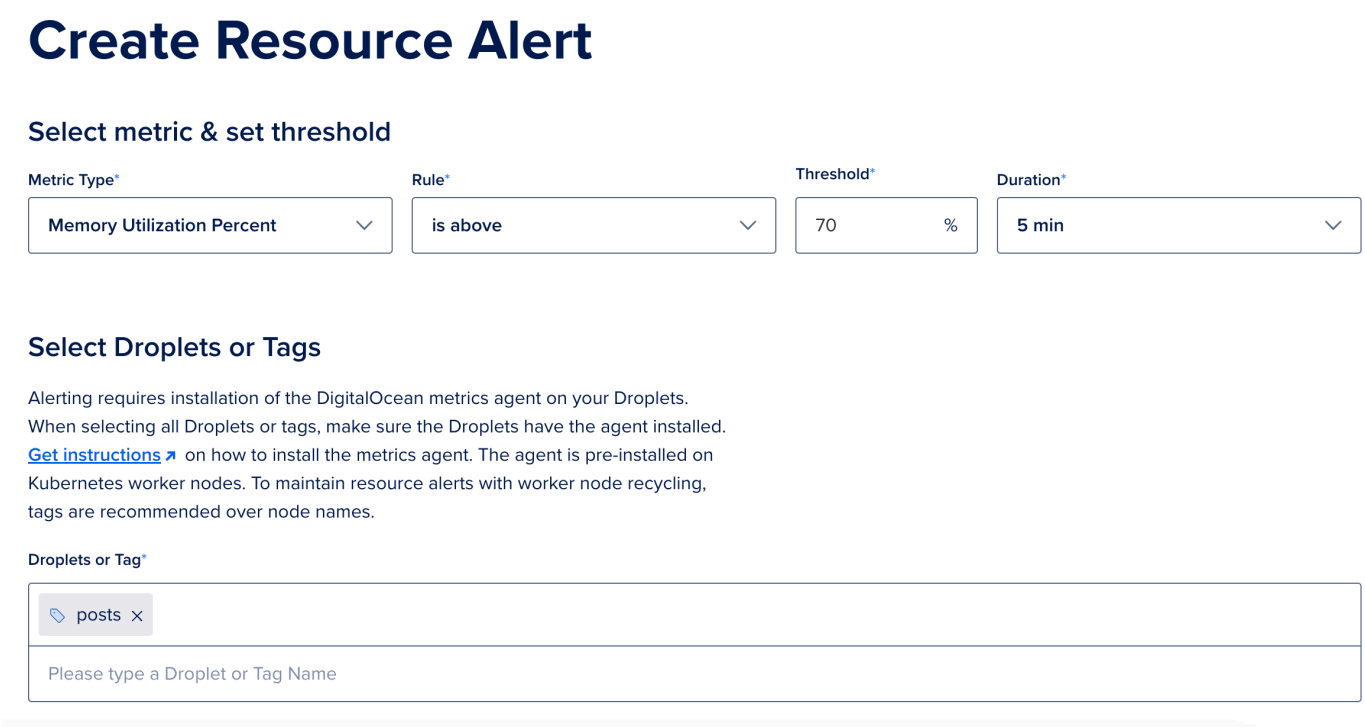
These health checks are very important in my opinion. And as you can see, it's pretty easy to set up them.

Server resource alerts

Server providers also provide resource monitoring tools. They monitor similar metrics as the health-check package:

- CPU load averages
- CPU utilization
- Disk space
- Memory
- Disk read and write I/O

There's no memory check in the Spatie package, so let's set up it on DigitalOcean:



This will alert me if the memory usage is above 70% on any of the Droplets tagged as `posts`.

To be able to run these alerts you need to install the DigitalOcean agent on your servers. Unfortunately, it needs to be running on the actual host, not inside the container so I added the installation script to the `provision_node` script:

```
curl -sSL https://repos.insights.digitalocean.com/install.sh | bash
```

This is how you can very that it's running:

```
ps aux | grep do-agent
```

Error tracking

The next topic I'd like to cover is error tracking. By default, you have only one tool to debug and understand the errors in your application: `laravel.log`. Laravel exceptions are quite nice, but let's be honest, we don't want to spend hours in 1000 lines long log files.

In a cluster environment, the situation gets even worse. In the previous chapter, I created a 4-node Swam cluster. The logs are spread across 3 or 4 nodes. The only command Swarm provides is the

```
docker service logs -f -t posts_api
```

Which can be quite chaotic.

To solve these issues you can introduce an error-tracking application such as:

- Rollbar
- Sentry
- Flare

All of them are pretty similar. Usually, they require the same installation and configuration steps. The dashboards they provide are also very similar. So check them out and choose your favorite one. I've been using Rollbar for years so I'm going to use it in the following examples as well.

Install Rollbar:

```
composer require rollbar/rollbar-laravel
```

Then create a project and a token on the UI and set it in your .env file:

```
ROLLBAR_TOKEN=your-token
```

We also need to add this to the `.env.prod.template` file:

```
ROLLBAR_TOKEN=${{ROLLBAR_TOKEN}}
```

And to the Swarm stack:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  ...
  environment:
    - ROLLBAR_TOKEN=${ROLLBAR_TOKEN}
```

Add a new GitHub secret called `ROLLBAR_TOKEN`, and finally change the pipeline so it copies the secret to the env file:

```
- name: Deploy app
  run: |
    ...
    sed -i "/ROLLBAR_TOKEN/c\ROLLBAR_TOKEN=${{ secrets.ROLLBAR_TOKEN }}"
/usr/src/.env
```

Of course, these steps depend on your deployment strategy. I'm assuming the same pipeline and Docker Swarm stack explained in the previous chapter.

Then set up a new log channel:

```
'rollbar' => [
  'driver' => 'monolog',
  'handler' => \Rollbar\Laravel\MonologHandler::class,
  'access_token' => env('ROLLBAR_TOKEN'),
  'level' => 'debug',
  'person_fn' => 'Auth::user',
  'capture_email' => true,
  'capture_username' => true,
],
```

Three important options:

- `person_fn` defines a function (`Auth::user()` in this case) that will be used by Rollbar to get the currently logged-in user.
- `capture_email` means that the `email` fields from the `User` model are going to be appended in the log entry.
- `capture_username` does the same but with the `username` field.

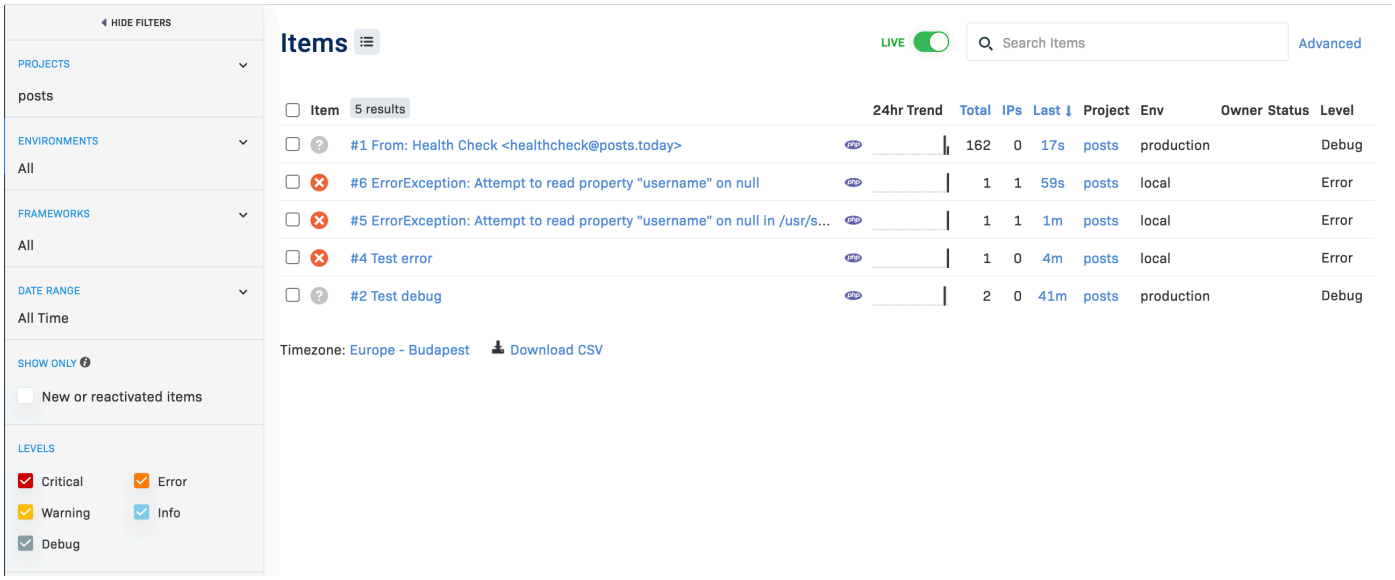
After we have a new channel, let's add it to the `stack` channel (which is the one I'm always using):

```
'stack' => [
  'driver' => 'stack',
  'channels' => ['single', 'daily', 'stdout', 'stderr', 'rollbar'],
  'ignore_exceptions' => false,
],
```

After that you can test it with:

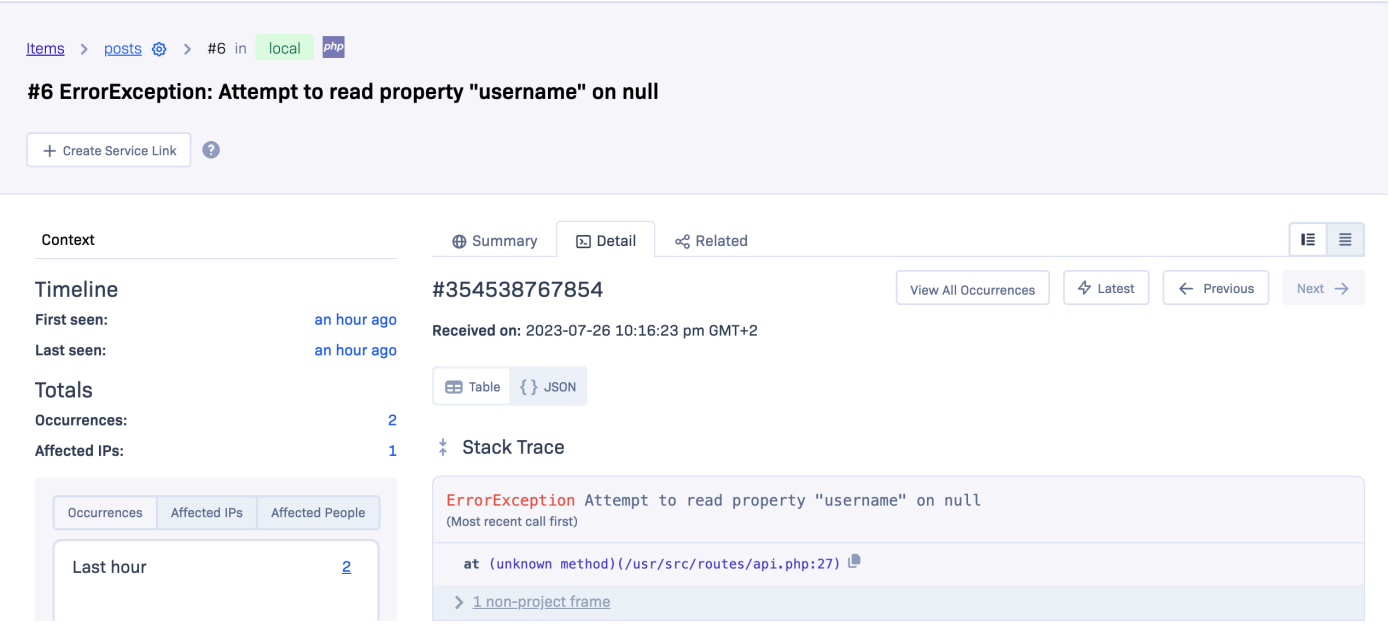
```
Log::debug('Test debug')
```

Or just simply throwing an exception. You should see every item on the dashboard:

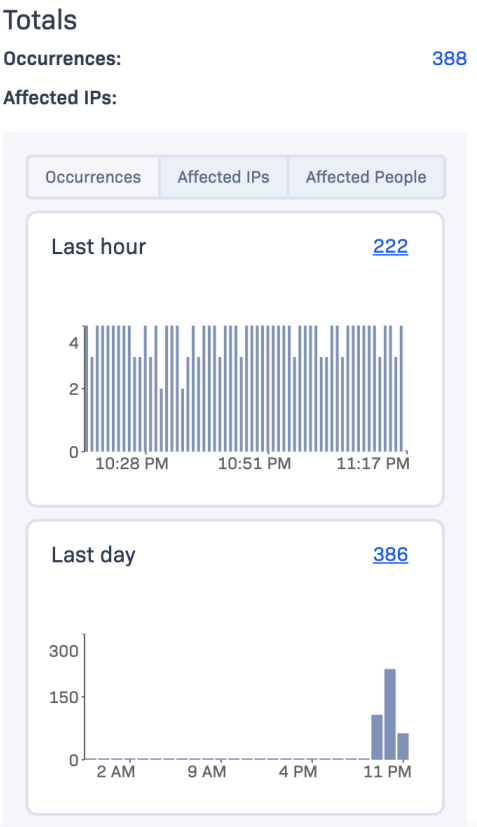


`Total` is an important column. It shows you how many times an error has occurred so far. In the sample application, I use a `log` mailer, and the first row on the screenshot is a log "e-mail" I'm getting from the health check package.

If you click on an item you can see more details:



You can also see the exact distribution of an error:



Or the exact details of every occurrence and lots of other things.

Having an error tracker is a must-have in my opinion.

Log management and dashboards

The project files are located in the `6-log-collecting` folder.

In this chapter, we're going to add log collecting to the Swarm cluster and also add some dashboards that help us debug the application.

For example, this dashboard shows the distribution of the different request methods, the top URIs, and the status code distribution:

This one lists only the failed requests and health check status codes:

To accomplish this we're going to use the following stack:

- fluentbit
- Loki
- Grafana

Usually, a monitoring stack consists of three layers:

- Collecting log messages from every node
- Storing them in a database optimized for storing, querying, and indexing unstructured log entries
- Visualizing the information on fancy dashboards

And this is the stack I'm going to show you:

- fluentbit is the **log collector**

- Loki is the **database**
- Grafana is the **visualizer**

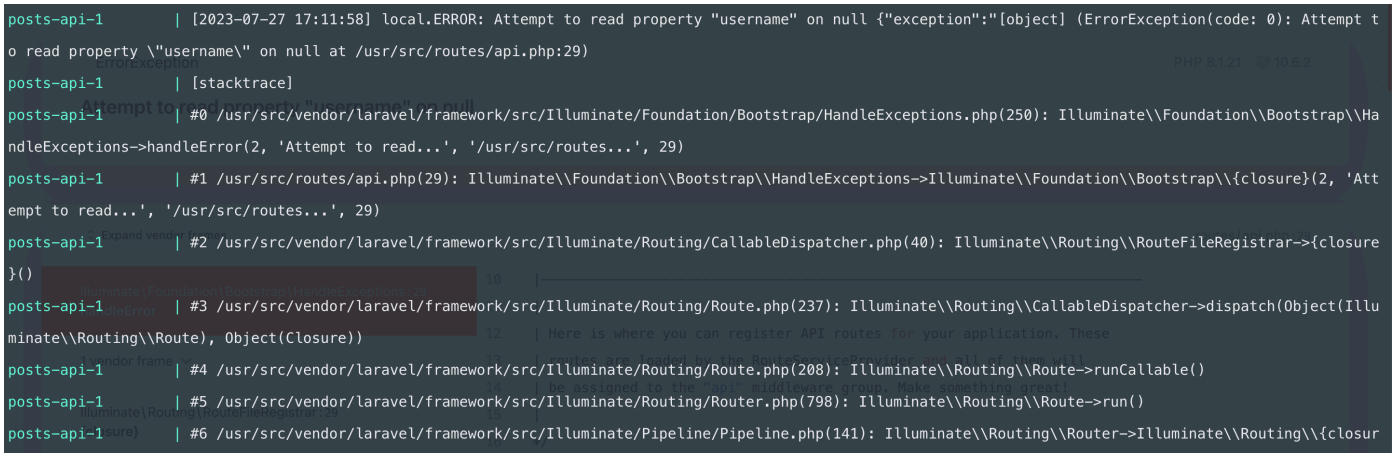
There are lots of other stacks but the main architecture/concept is usually the same. For example, the ELK stack contains these components:

- Logstash is the log collector
- Elasticsearch is the database
- Kibana is the visualizer

Another famous one is the EFK stack. It's almost the same as ELK but the letter "F" stands for fluentd. So it uses another log collector. And yes, there's a log collector called fluent**d** and another one called fluent**bit**. They are pretty similar but fluentbit is a bit simpler. For our purpose, it's perfect.

JSON logs

To be able to create these dashboards, first, we need a structured log format. The problem with the default format is that it produces multiline exceptions such as this:



A log collector (such as fluentbit) will collect this exception line-by-line and will become X different log entries in Grafana.

The other problematic thing is nginx access logs. If you want to make a dashboard such as the status code distribution above you need to somehow do calculations on the status codes. It's pretty hard if you have log entries such as this:

```
127.0.0.1 - - [27/Jul/2023:17:14:44 +0000] "GET /api/health-check HTTP/1.1"
200 5 "-" "curl/7.88.1"
```

With this format, your only hope is some magic with grep. Which is not good.

What I want is JSON log entries. I'd like to see logs such as these:

- The first one is a debug message from `artisan` from the API container
- The second one is an error message behind an API endpoint (directly in the `api.php` route file as you can see)
- The third one is the 500 access log entry caused by the API error
- And the last one is a successful health check made by docker-compose

Now imagine how much easier it is to filter for nginx log entries where the `status` property is either 4xx or 5xx. Or how easy it is to filter for `level_name = ERROR` in the API logs.

Here's the nginx configuration:

```
http {
    log_format json '{"time_local":"$time_local",'
                    '"remote_addr":"$remote_addr",'
                    '"request_method":"$request_method",'
                    '"request_uri":"$request_uri",'
                    '"status":"$status",'
                    '"body_bytes_sent":"$body_bytes_sent",'
                    '"http_referer":"$http_referer",'
                    '"http_user_agent":"$http_user_agent"}';

    access_log /var/log/nginx/access.log json;
     ...
}
```

We literally just need to create a JSON template using variables provided by nginx. The log format's name is `json`. Then we specify that access logs should use the new `json` log format. That's it! Now nginx will log in JSON.

After that go to your `logging.php` config and add a new `json` channel:

```
'json' => [
  'driver' => 'monolog',
  'level' => env('LOG_LEVEL', 'debug'),
  'handler' => StreamHandler::class,
  'formatter' => Monolog\Formatter\JsonFormatter::class,
  'with' => [
    'stream' => 'php://stdout',
  ],
  'processors' => [PsrLogMessageProcessor::class],
],
```

It logs to `stdout` using the `JsonFormatter` formatter. That's it! Now we have "cloud native" JSON logs.

I check the current environment at the beginning of `logging.php` so on localhost I can still enjoy dirty "not cloud native" format:

```
if (env('APP_ENV') === 'production') {
  $channels = ['json', 'rollbar'];
} else {
  $channels = ['single', 'daily', 'stdout'];
}
```

And then I use it in the `stack` channel:

```
'stack' => [
  'driver' => 'stack',
  'channels' => $channels,
  'ignore_exceptions' => false,
],
```

Grafana & fluentbit

Now that we have the correct log formats the next is to add the components such as fluentbit and the others. These are not strictly part of the application so I'm going to add them to the monitoring stack defined in `docker-compose.monitoring.yml`.

```
fluentbit:
  image: martinjoo/posts-fluentbit:${IMAGE_TAG}
  deploy:
    mode: global
  ports:
    - "24224:24224"
  environment:
    - LOKI_URL=http://loki:3100/loki/api/v1/push
```

Since fluentbit collects logs on every node it needs to run on every node so the deployment mode is set to `global`. It also needs a `LOKI_URL` environment variable so it can push the logs to Loki.

As you can see, I also built a custom image for fluentbit. It's pretty simple:

```
FROM fluent/fluent-bit:2.1.7

COPY ./deployment/config/fluent-bit.conf /fluent-bit/etc/fluent-bit.conf
```

Just as the nginx image it only contains a config file. You don't have to follow this practice, it's highly subjective. You can just use the official image and then mount the configuration file. However, I find the custom image solution more "self-contained," and more "stateless."

The config is surprisingly easy:

```
[SERVICE]
    Flush        5
    Log_Level    error
    Daemon       off

[INPUT]
    Name         forward
    Listen       0.0.0.0
    Port         24224
```

```
[OUTPUT]
    name      loki
    match     *
    host      loki
    labels    job=fluentbit, $sub['stream']
```

The `[SERVICE]` section configures fluentbit itself:

- `Flush 5` specifies the interval (in seconds) for flushing buffered records to the output plugin.
- `Log_Level error` means fluentbit itself only logs errors. It has nothing to do with your application logs. It means fluentbit should only report errors about itself.
- `Daemon off` means fluentbit runs in the foreground. In a Docker image, you must have at least one long-running process to keep alive and running. We did the same with nginx.

The `[INPUT]` section configures what fluenbit should do with its inputs. In a minute, we'll see that containers push logs to fluentbit. Containers will forward logs to fluentbit using a TCP connection.

We use the `forward` protocol with `0.0.0.0:24224`. This protocol will forward anything that comes in on port 24224. Containers send their logs to fluentbit:24224 and then fluentbit simply forwards them to the `OUTPUT`.

Finally, the `[OUTPUT]` section specifies what to do with the logs.

- `name` refers to the built-in Loki plugin. It will push logs to loki. This is why the image needs the `LOKI_URL` environment variable.
- `match *` means every log entry will be processed.
- `host loki` defines the target host where Loki is listening. It's going to be another container.
- `labels` helps us identify log entries later. `$sub['stream']` is a special variable provided by fluentbit. It contains the source of the log message which is `stdout` in most cases.

If you have trouble understanding these input and output protocols, here's an easier example:

```
[INPUT]
    Name            head
    Tag             head.cpu
    File            /proc/cpuinfo
    Lines           8
    Split_line      true

[OUTPUT]
    Name            stdout
    Match           *
```

The input protocol is `head` which is the `head` terminal command that reads the beginning of a file. So this config:

- Reads 8 lines from the `/proc/cpuinfo/` file
- And prints it to `stdout`

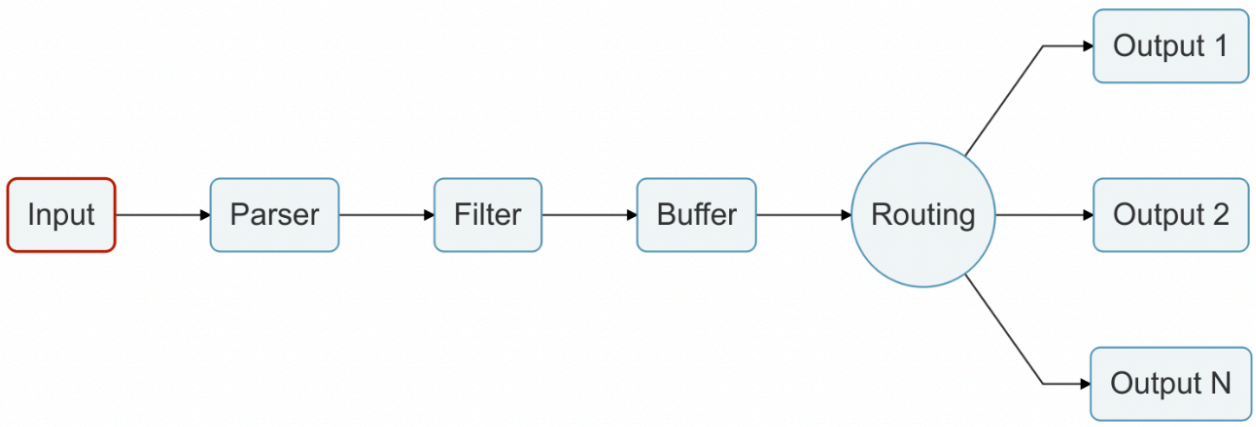
In our own configuration:

- "Reads" every request from `0.0.0.0:24224`
- And "prints" them out to Loki

fluentbit is very flexible and it has more of these configuration options, such as:

- Parser
- Filter
- Buffer

With these components, you can build complex data pipelines:



Now we can add another container:

```
loki:
  image: martinjoo/posts-loki:${IMAGE_TAG}
  deploy:
    placement:
      constraints:
        - "node.labels.db==true"
  volumes:
    - loki-data:/loki

volumes:
  loki-data:
```

Loki is a database so it needs a volume to store data. Because it has a state, it needs to be placed on the same node every time. The image follows the same principle, it's a Loki image with a config file. In most cases, you can just use a default config file from the examples.

The next step is configuring the containers to send logs to fluentbit:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  logging:
    driver: fluentd
```

That's it! fluentd and fluentbit are supported by Docker out of the box. They both work the same way from Docker's perspective so `driver: fluentd` is not a typo.

One more thing before moving on. It's a good idea to somehow label the log messages with a container name. Otherwise, you end up with tens of thousands of log messages and if you want to see only the API's log entries you won't be able to filter them. Fortunately, we can solve this issue with a simple label:

```
api:
  image: martinjoo/posts-api:${IMAGE_TAG}
  labels:
    service_name: "api"
  logging:
    driver: fluentd
    options:
      labels: "service_name"
```

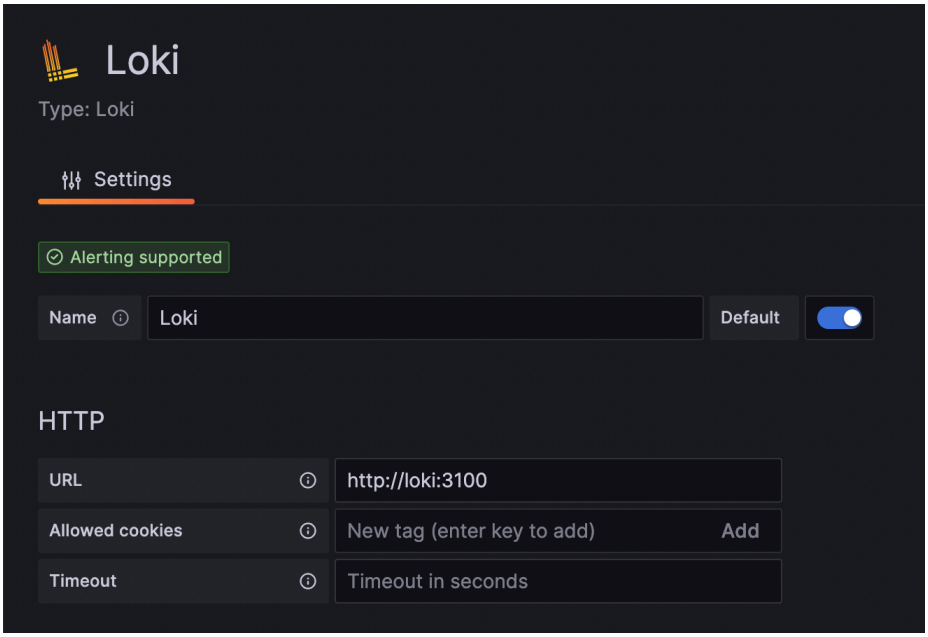
Swarm adds the label `api` to the running container and when it sends the logs to `fluentbit` it sends the label together with the message. Later, in Grafana we can use these labels in filters.

And the step is adding Grafana to the stack:

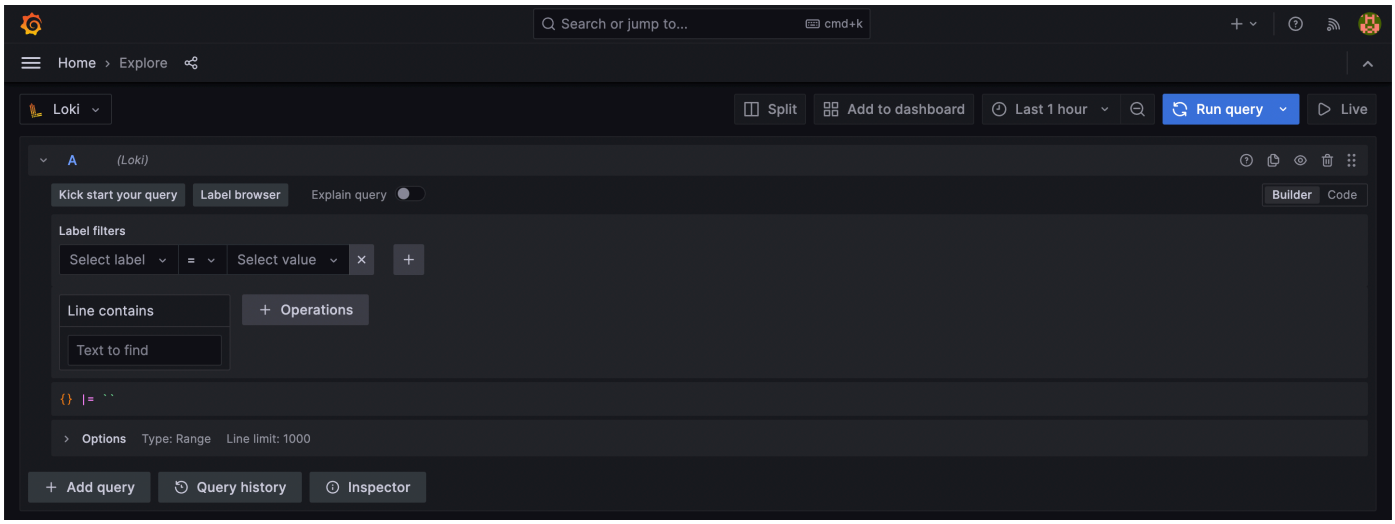
```
grafana:
  image: grafana/grafana:9.5.6
  deploy:
    placement:
      constraints:
        - "node.labels.db≠true"
  ports:
    - "3030:3000"
```

No configuration needed just a port.

After the stack is deployed Grafana can be accessed on port 3030. On the homepage, there's a link to choose a data source. Grafana needs to be configured to use Loki:



The URL should be `http://loki:3100` since they are in the same stack and Loki is listening on port 3100. After that, choose the menu 'Explore'. This is the view where you can build queries and you can save them as dashboards:



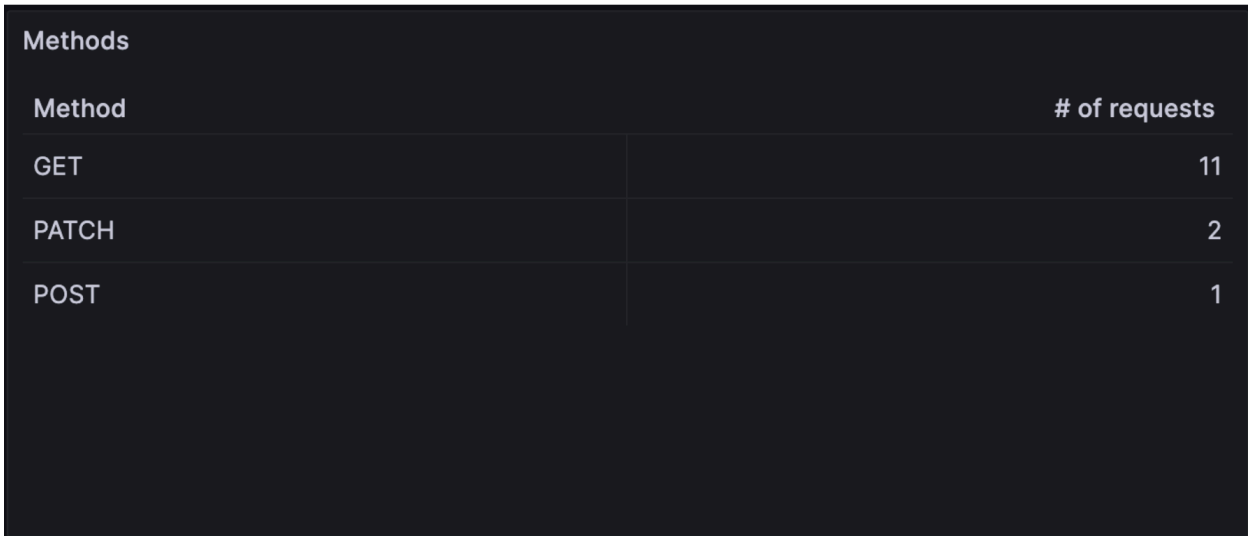
There are two modes of the explore view:

- Builder
- Code

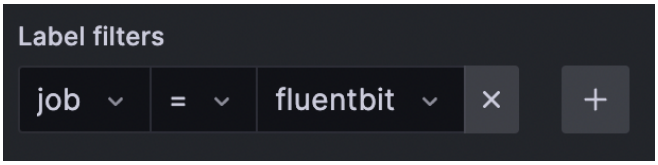
The builder view is what you can see in the image. In builder mode, you can visually build queries. in the code view, you can see and write the actual queries. The builder view is a great way to get used to this new query language. Which is `LogQL`. It has lots of features and introducing everything is outside the scope of this book. Check out their documentation and learn more about it if you're interested. Also, I'm not a monitoring expert at all, but I can show you some basic queries and dashboards.

Method distribution table

First, let's build a simple table that shows us how HTTP methods are distributed:



There's an input called `Label filter`. Always start your queries by choosing `job = fluentbit`



This is the label we set in the fluentbit config.

If you click on the `+ Operation` button you can see there are different types of operations, for example:

- Aggregations. The usual `MIN`, `SUM`, and `AVG` functions
- Range functions such as `Range over time`
- Formats such as `JSON` or `Logfmt`

`Formats` sounds like an easy start. A formatter reads a log entry and parses it in the given format so it can be used as an object later in the query. For example, now we have log entries such as this:

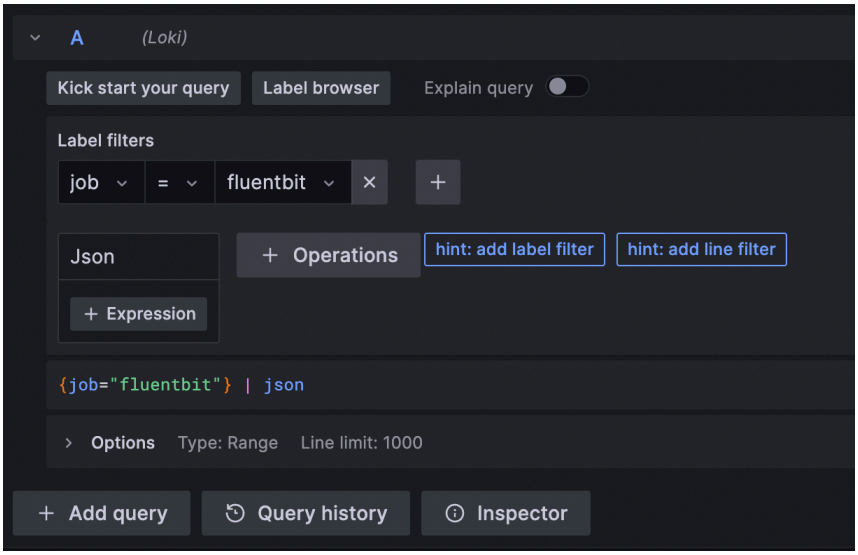
```
'{"level": 200, "level_name": "DEBUG"}'
```

But LogQL cannot use this in queries because it's just a string. First, we need to parse it:

```
{
  "level": 200,
  "level_name": "DEBUG"
}
```

And now we can access the properties such as `.level` or `.level_name` and they can be used in LogQL expressions.

So the "query" should look like that at this point:

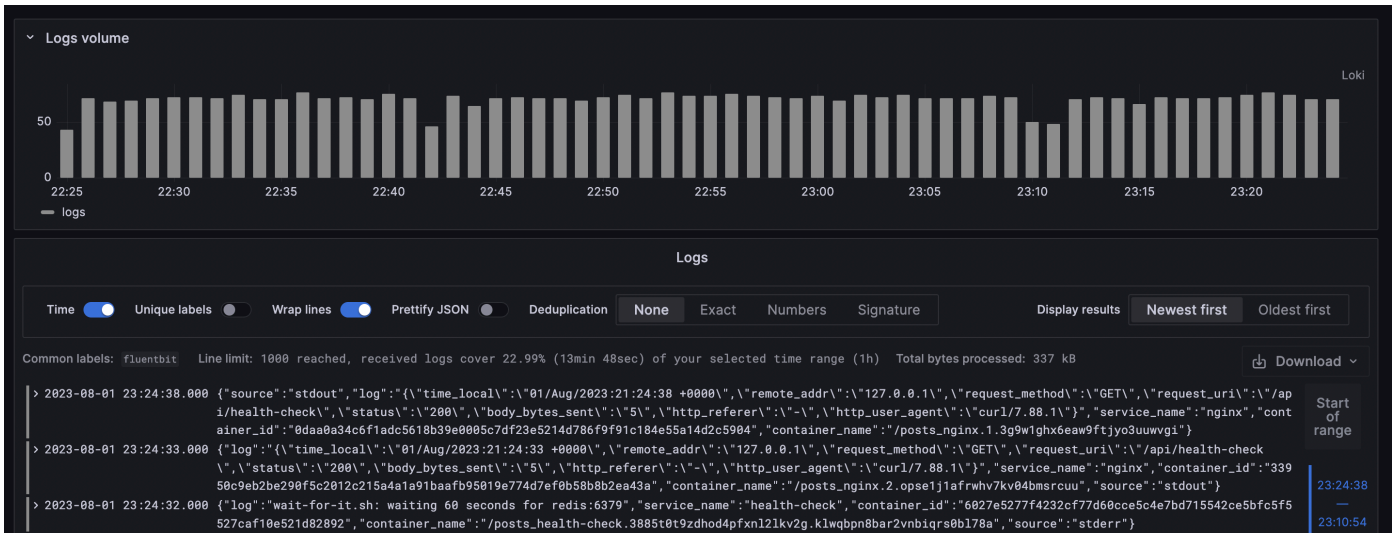


The LogQL query looks like this:

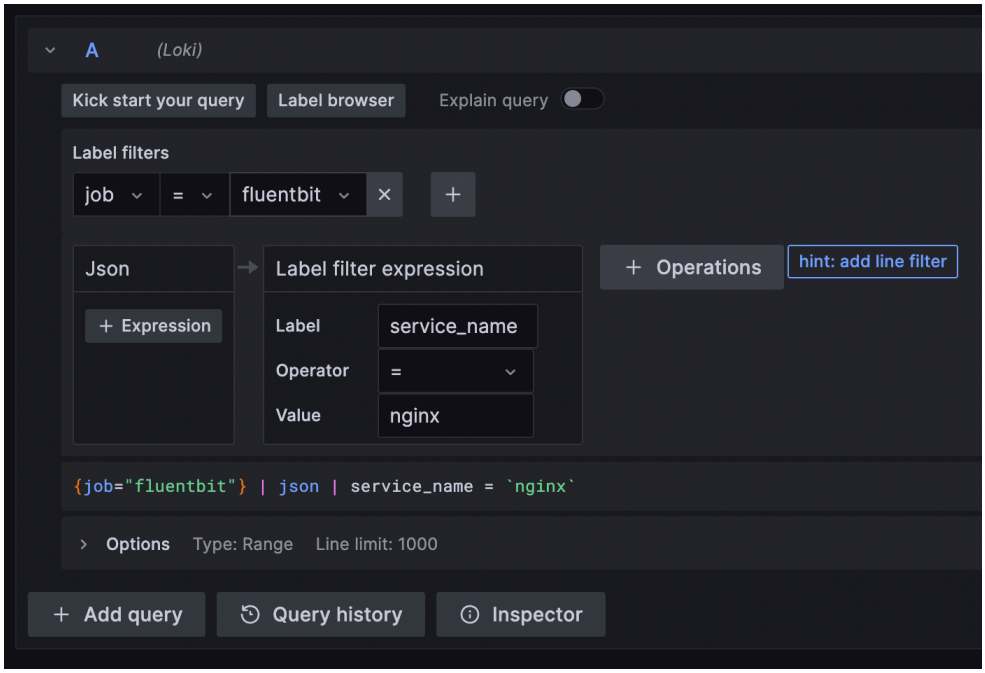
```
{job="fluentbit"} | json
```

These formatter and filters can be piped using the `|` symbol. The `json` formatter doesn't have any argument.

And if you now click on `Run query` it should already give you some results:



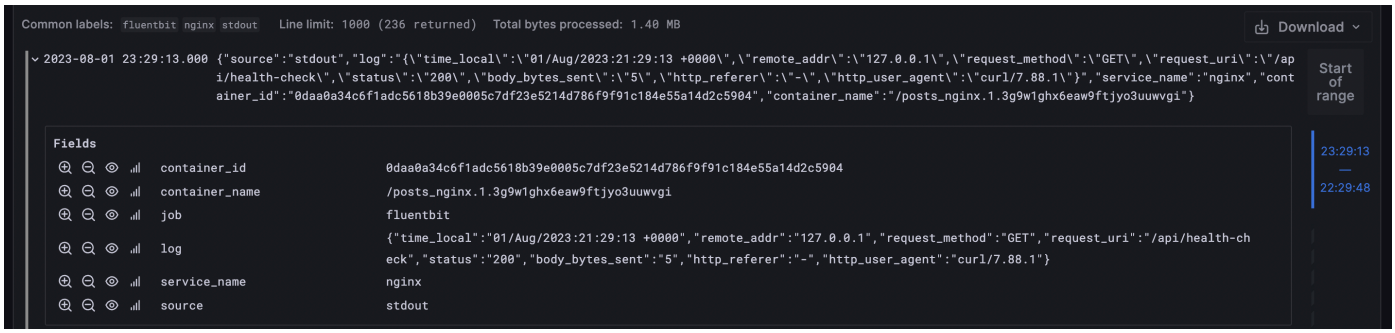
Of course, it returns every log entry now. To get information about the HTTP methods we only need nginx logs. If you remember we added a label in the docker-compose config. In Grafana, we can filter based on these filters. Just choose `Label filter expression`:



The query:

```
{job="fluentbit"} | json | service_name = `nginx`
```

Label filter is like a `where` expression but we need to use the `|` operator again. The query now returns only logs from `nginx` containers. But if you take a look at a log entry it's not quite right:



`log` is the most important property. It contains the actual nginx log with `status` and `request_method` and so on. But it's still just a string. To accomplish this we need to extract it to the top level and then format it as JSON. In PHP we would do something like that:

```php
$logEntry = ['log' => '{"status": "200"}'];

$importantStuff = json_decode($logEntry['log'], true);

[
  'status' => 200,
  ...
]
```

This line:

```php
$logEntry['log']
```

can be accomplished with a `line_filter` in LogQL:

```
line_format `{{.log}}`
```

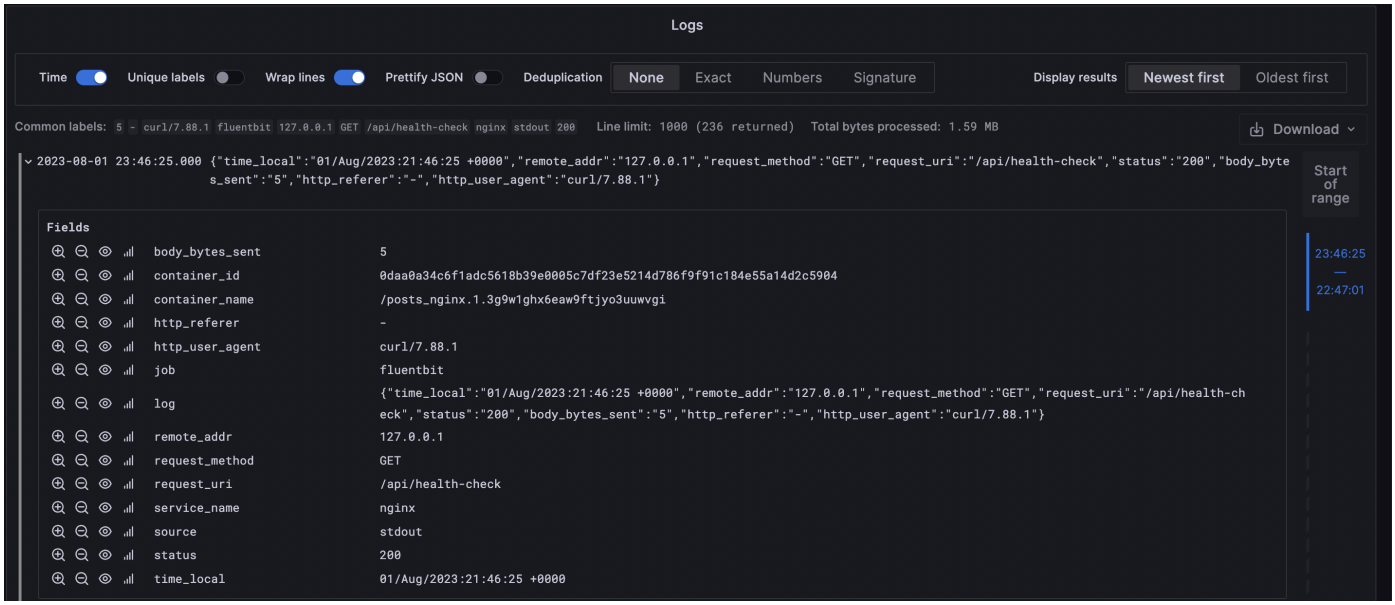
And of course `json_decode` is the equivalent of `json`:

```
line_format `{{.log}}` | json
```

This is the whole query:

```
{job="fluentbit"} | json | service_name = `nginx` | line_format `{{.log}}` | json
```

And now a log entry looks much better:



Now we can query against the actual log entry's properties so let's filter out `OPTIONS` requests:

```
| request_method != `OPTIONS`
```

It's a simple `line_filter`.

The last step is to get a count by `request_method`. A count function looks like this in LogQL:

```
count_over_time(query, [period])
```

It's a function that takes two argument:

- The first one is the query we have so far. So in LogQL `count_over_time` wraps the whole query as if it was something like this in MySQL:

  ```
  count(
    select id from users
  )
  ```

  Instead of:

  ```
  select count(id)
  from users
  ```

- The second one is the time period. We'll use this query in a dashboard where we're going to choose a time range. When you check out a dashboard you rarely ask something like "So let's see what happened in the last 3 years?" but instead "What happened in the last 6 hours?" So it's going to be a variable from a dropdown. There are some special variables we can use in queries. One of the is `$__range` which stores the value of the time period dropdown in the top right corner.

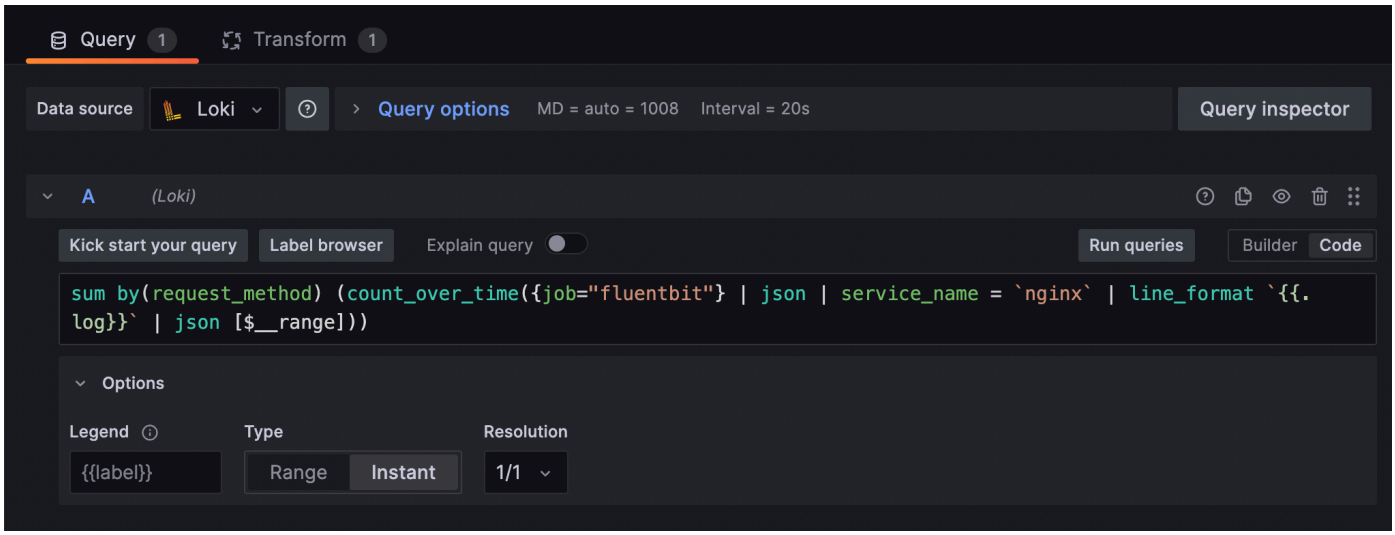
This is the query with `count_over_time`:

```
count_over_time({job="fluentbit"} | json | service_name = `nginx` |
line_format `{{.log}}` | json [$__range])
```

And finally we want the sum by the `request_method` property which can be expressed such as this:

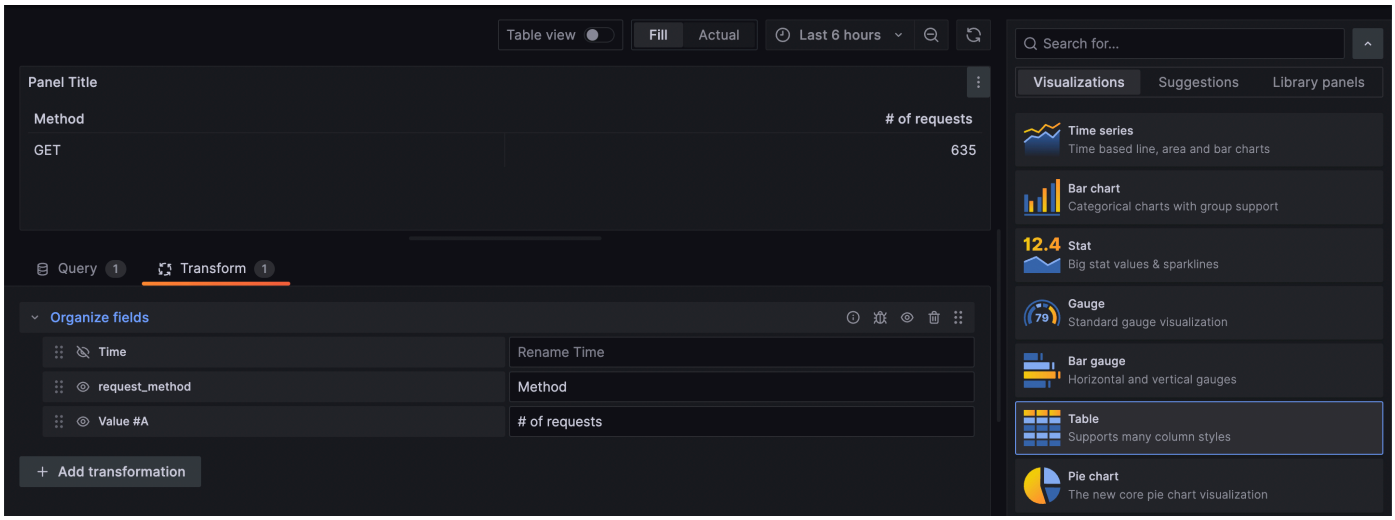
```
sum by(request_method) (count_over_time({job="fluentbit"} | json |
service_name = `nginx` | line_format `{{.log}}` | json [$__range]))
```

Now create a new dashboard and paste the query. Under the actual query there's an `Options` section. Set `Type` to `Instant`:



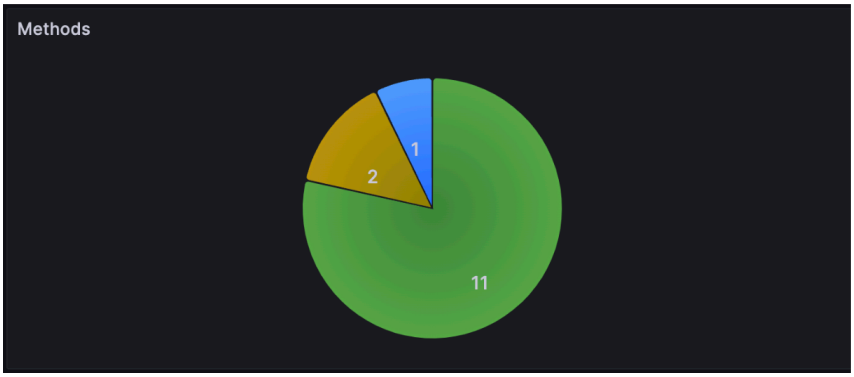
On the right side choose `Table` as visualization type.

And finally in there's a `Transform` tab, select `Organize field` and set the following options:



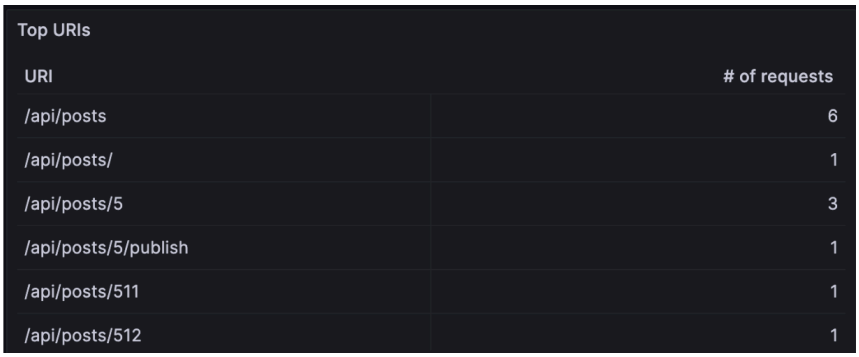
With `Organize fields` you can rename, re-order, or hide columns. As you can see, now we have a nice table showing the request distribution. You can save it to your dashboard as a new panel, in the top right corner.

Method distribution chart



We have the first panel and dashboard. Just copy that and change the visualization type to `Pie chart`. And now you have a pie chart showing the same data.

Top URIs



To get the most popular URIs in the application we almost need to write the same query as before. The query needs the same formats and filters. It also needs a `count_over_time` and a `sum by` but this time is based on the `request_uri` property instead of the `request_method`.

To get only the top X results there's an expression called `topk` and it works like this:

```
topk(n, query)
```

The whole query looks like this:

```
topk(10, sum by(request_uri) (count_over_time({job="fluentbit"} | json | service_name = `nginx` | line_format `{{.log}}` | json [$__range])))
```

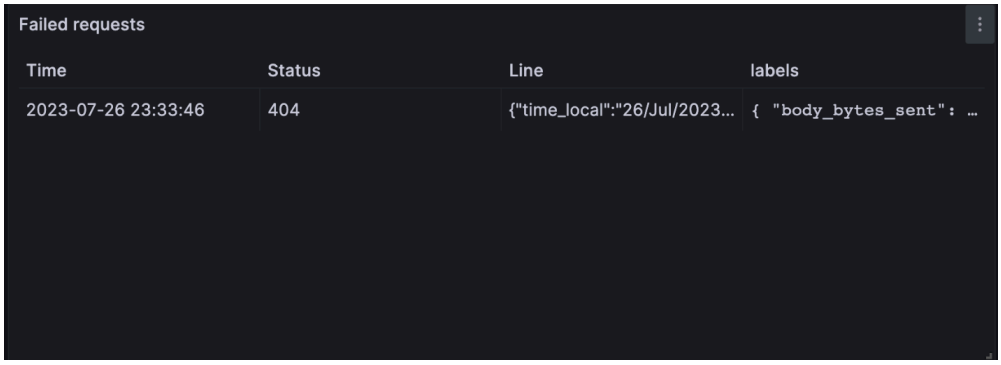
Status code distribution chart

I have good news: you can use the exact same query but with different properties and visualization type:

```
sum by(status) (count_over_time({job="fluentbit"} | json | service_name = `nginx` | line_format `{{.log}}` | json [$__range]))
```

Now set the visualization type to bar chart and you're done.

Failed requests

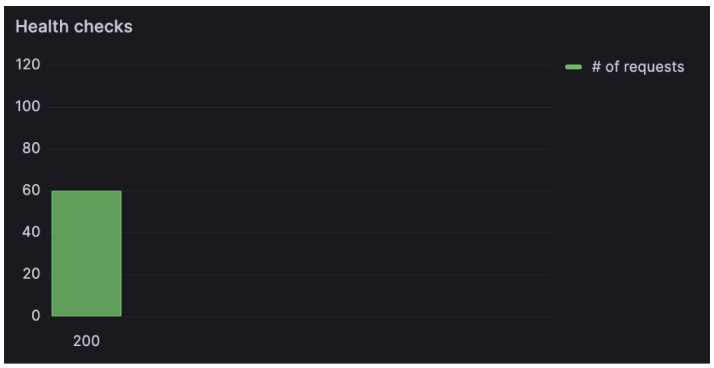


That's an easy one:

```
{job="fluentbit"} | json | service_name = `nginx` | line_format `{{.log}}` | json | request_method != `OPTIONS` | status > 399
```

One more "where" expression to filter out 2xx and 3xx status codes. The visualization type is logs.

Health check status chart



It's also an easy one. The only difference compared to the other queries is that this one has an additional label filter expression because we want only the `/api/health-check` requests:

```
sum by(status) (count_over_time({job="fluentbit"} | json | service_name = `nginx` | line_format `{{.log}}` | json | request_uri = "/api/health-check" [$__range]))
```

As you can see, the number of possibilities is endless so try out Grafana, run these queries, play around with them a little bit, and try to create your own useful charts and dashboards.

Conclusions

As I said in the introduction, learning Docker Swarm is super easy. It's very similar to docker-compose with only a few new commands. The question is: when should you use it?

I think if you want to scale your application, and want to move into a cluster, Swarm is an excellent choice as the first step. Especially if you don't have Kubernetes knowledge in your team. I think if you have a dockerized application, you can move into a cluster in a day or so. And then, you'll have experience in the cluster world. Which is crucial! You'll probably face problems you never knew existed before. If your project grows even further and Swarm starts to be limited for you, just move to Kubernetes. I think it's a valid path. Certainly better than jumping into k8s without any experience. Of course, I'm talking about a production application with reasonable traffic and user base.

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.