DevOps : Docker, CI, monitoring

Purpose of the workshop :

- get used to docker and docker-compose
- setup a symfony app with docker
- write some unit tests
- setup CI
- setup a monitoring stack
- link the symfony app and the monitoring stack
- collect application logs
- collect nginx logs
- visualize logs

To help you accomplish this workshop, you have access to the following resources :

- the slides of the course
- the Internet (documentation of the softwares, tutorials)
- the other students (don't be affraid to ask and share knowledge)
- the teacher

During this workshop we expect you to make some researches. Your work should be posted on github. You should describe what you've managed to accomplish, and what were the difficulties you encountered.

This workshop will matter in your practise notation. We'll consider how far you went, your level of implication, also as your level of understanding.

1. Setup

Start by creating a new directory for this project (eg `docker-symfony`).

The project should have a `docker` directory in which you'll define your Dockerfiles and also place the config files of your docker services.

At the root of your project, create a `docker-compose.yml` file which should expose the following services :

- nginx
- php-fpm
- mysql

This compose file should be used for dev env (ie the project directory should be mounted in each service to reflect the live changes).

You should also create a new symfony application in your project dir. Your docker services will run this application.

2. Simple controller

Create a simple symfony controller which displays an `hello world` page when going on the homepage (`"/"` route) of the application.

3. CI setup

Start by creating a simple `User` entity in your application. Then, write some unit tests for this class (eg to test getters and setters), using `phpunit`.

Finally, run these tests on CircleCI. To do so you should put your project on github and configure a CircleCI workflow.

Tip : Start by signing up on CircleCI using your github account to see all your repositories on CircleCI.

4. Bootstrap monitoring stack

We want to collect and to visualize the application logs and also the nginx logs. To do so, create a new project (eg `docker-monitoring`) alogside your `docker-symfony` project, and create a docker-compose stack with the following services :

- fluentd (to collect logs)
- elasicsearch (to store logs)
- kibana (to visualize logs)

Tip : This stack is similar to the `ELK` stack (Elasticsearch, Logstash, Kibana), except that Logstash is replaced by fluentd (lighter). It's a good idea to start reading about how the ELK stack is working to understand each of these three components purposes. Here is a good introduction to the ELK stack : https://www.youtube.com/watch?v=1r1SOeaDqH4 ("Laisse pas trainer ton log - Olivier Dolbeau, 2014, French").

You will have to expose the `fluentd` service on the symfony's stack network in order to allow the communication between fluentd and php.

5. Ingest application logs

We want to keep track of what's going on with our application. To do so, we will log some messages from our pages. These application logs should reflect how your application behave, for instance :

- `info` : a new user has registered
- `error` : unable to send an email

Update the application to log some test messages (at different levels) for the index page.

The application should send logs to fluentd (the log collector) via syslog using UDP. You can have a look to the existing `monolog` handlers to get some inspiration (a symfony app uses monolog by default). The syslog message should be in the `rfc5424` format. You may have to write your own log handler and formatter. The minimum log level should be `info`.

Tip : Monolog has the ability to use channels. Channels are used to specify which part of the app the log message is related to (eg `users`, `admin`, etc...). You can then filter out the logs by channels to be able to quickly find all the logs for a certain part of your application.

The message payload should be json encoded. You should decode it with fluentd by using a `filter` of type `parser`. Using the json format allows to use complex log payloads, and they are easy to parse.

Fluentd should be able to decode this message and to send the log record to elasticsearch using the logstash format. To do so, the fluentd configuration should have a `source` section for syslog input, and a `match` section to send the logs to elasticsearch.

The log record should contain the log level (eg `info`, `warning`, `error`, `critical`, etc...).

Tip : To see what's going through fluentd, you can print all the records on its stdout with this config :

```
<label @FLUENT_LOG>
  <match fluent.*>
    @type stdout
  </match>
</label>
```

and print the fluentd docker service logs to see the records :

```
$ docker-compose logs -f fluentd
```

6. Visualize application logs

Your logs should now be stored in elasticsearch and you should see them on kibana.

We want to easily see when we have an increase of the error rate on the app. To do so, create a new search on kibana where you're filtering the logs by `level_name` (`ERROR`). Save this search, as we're going to use it for a visualization.

To visualize the error rate, go on kibana's vizualize menu and create a new line diagram from the previously saved search. On the Y-axis, select `count` to count the errors on the application. On the X-axis, select `Date histogram` on the `@timestamp` field.

You should now have a visualization reporting the application error count in function of the time.

7. Ingest nginx logs

We also want to keep track of the nginx responses.

Update the nginx configuratio to send the access and error logs to fluentd via syslog and UDP.

You'll also have to update the fluentd configuration :

- create a new `source` for nginx
- create a new `filter` of type `parser` to be able to parse nginx logs
- reuse the elastcsearch `match` from the previous exercise to send the parsed log record to elasticsearch.

As we now have multiple sources, it's a good idea to tag the sources with a different name to be able to identify them easily.

8. Visualize nginx logs

Once your nginx logs are correctly saved on elasticsearch, create a new search on kibana to get the nginx logs (you can filter them by tag).

Tip : You will certainly have to update your kibana fields list to make the newly indexed fields searchable. To do so, click on the `Management` menu -> `Index pattern` -> `<your_index_name>` -> `Refresh` (icon at the top right hand corner).

Then, create a visualization to count the nginx logs, grouped by status code (200, 404, 500, etc...) :

- Y-axis: COunt
- Buckets:
    - X-axis: Date histogram of @timestamp
    - (sub bucket) Split series : `Term` sub aggregation by `code.keyword`

Finally, add this visualization on the kibana dashboard.

Bonuses

Surveys CRUD

Enhence the application by writting a CRUD section to manage surveys.

This section should have the following features :

- be able to list the created surveys
- be able to create a new survey (eg `"Do you like chocolate ?"`)
- be able to edit an existing survey (eg change the question)

You should also log the performed actions (ie when a survey is created).

Additional tests

- Write some unit tests for the surveys CRUD (eg covering the `Survey` entity).
- These tests should also run on CircleCI.

Visualize

Create a visualization on your kibana dashboard to see the number of created surveys during the last day.

Functional tests

- Write some functional tests for the surveys CRUD (eg be able to create a new survey and to see it on the list).
- These tests should also run on CircleCI.