Docker, helm et Jenkins pour un déploiement continu réussi

Une fois n'est pas coutume, cette belle histoire va commencer par la fin. Et oui comme un vieil épisode d'Inspecteur Colombo, où dès la première scène on connait le coupable, notre équipe DevOps va se pencher sur l'épisode final : **Comment le produit sera exploité ?**

En effet, cette question est essentielle et on doit y apporter des réponses dès le début du projet avant même d'écrire les premières lignes de code. On pourra alors construire des fondations solides sur lesquelles notre code, donc notre produit, prendra forme.

Que le sprint 0 commence !!!

Nous décidons donc de déployer notre produit dans un cluster **Kubernetes** et nous utiliserons **Google Cloud Platform** pour héberger notre solution. On va ensuite travailler avec plusieurs containers où chacun aura un rôle particulier :

- un container pour publier le site web basé sur nginx
- enfin un dernier pour interpréter les pages php de ce site

On délèguera la partie data mysql au service managé Google Cloud SQL. Pour cet exemple, nous préparons une architecture simple mais on pourrait augmenter les performances de notre application en ajoutant un container ElasticSearch et du cache Redis.

Ce choix d'utiliser une technologie autour des containers, est dicté par deux principaux avantages. Notre application, au travers des fonctionnalités Kubernetes, va être disponible en **haute disponibilité** et peut être **scalable** simplement. Le deuxième avantage est que sur le poste de développement je peux **travailler avec un environnement iso-production.** Fini la célèbre réplique du développeur lors d'un déploiement en production : `Non mais sur mon poste ça marchait ...` Si ça compile et si ça fonctionne sur mon poste, ça aura le même comportement en production.

Des containers en local

Une fois l'architecture décrite, passons au coeur du problème : écrire le code pour déployer des containers pour héberger mes applications. Nous avons choisi PHP comme langage de programmation pour notre application. Il faut donc monter à minima un container php ainsi qu'un frontal nginx. Pour nous faciliter la configuration et le déploiement des containers en local, nous allons utiliser le framework Laradock.

```
mat@vm:~/project$ git submodule add
https://github.com/Laradock/laradock.git
```

Ensuite, il faut copier le fichier env-example en .env et modifier les options qui sont nécessaires pour notre projet. Il existe de nombreuses options dans ce framework. Une fois, les options sélectionnées, démarrer les containers

```
mat@vm:~/project$ cd laradock/
mat@vm:~/project/laradock$ cp env-example .env
mat@vm:~/project/laradock$ vi .env
COMPOSE_PROJECT_NAME=bdx_io
PHP_VERSION=7.3
WORKSPACE_INSTALL_MYSQL_CLIENT=true
PHP_FPM_INSTALL_MYSQLI=true
mat@vm:~/project/laradock$ vi nginx/sites/default.conf
    root /var/www/;
mat@vm:~/project/laradock$ docker-compose up -d nginx php-fpm mysql
```

Nous avons utilisé Laradock pour configurer et démarrer des containers locaux sur notre poste. On pourrait très bien faire le même travail avec des containers “faits maisons”. Dans tous les cas, avec ces containers, je peux donc développer mon application sur mon poste de travail avec un environnement équivalent à la production.

Des images “prerelease” prêtes pour la production

Parlons-en de la production. Il est temps de transporter ces images Laradock pour recevoir la production. Par exemple pour le container **php-fpm** nous allons réaliser les actions ci-dessous pour déployer cette image dans la registry de google

```
mat@vm:~/project/laradock$ vi cloudbuild-php-fpm.yml
steps:
- name: 'gcr.io/cloud-builders/docker'
  args: ['build',
       '--build-arg', 'LARADOCK_PHP_VERSION=7.2',
       '--build-arg', 'LARADOCK_PHALCON_VERSION=3.4.1',
       '--build-arg', 'INSTALL_MYSQLI=true',
....
       '--build-arg', 'PHP_FPM_ADDITIONAL_LOCALES="es_ES.UTF-8 fr_FR.UTF-
8"',
       '-t',
       'eu.gcr.io/myProject/myAppli/prerelease/php-fpm',

       '.']
images: [
    'eu.gcr.io/myProject/myAppli/prerelease/php-fpm',
]
mat@vm:~/project/laradock$ cd php-fpm
mat@vm:~/project/laradock/php-fpm$ gcloud builds submit --
config=../cloudbuild-php-fpm.yaml
```

On créé ainsi des images de Laradock dans lesquelles on va pouvoir déployer notre code source. Quelques réflexes sont à acquérir :

- toutes les options sélectionnées dans Laradock, sont à reporter dans l'image
- dès que Laradock est modifié, il faut penser à régénérer ces images

Il faut ensuite faire le même exercice pour chaque container, à savoir ici dans notre exemple nginx et php-fpm

Un sprint de développement

Une fois les premières lignes de code développées, nous allons vouloir **pousser ce code dans notre chaîne d'intégration continue et peupler différents environnements** : intégration, release et au final la production. Nous allons faire la même chose qu'en local sauf qu'on va s'appuyer sur une image prerelease Laradock et on va y placer à l'intérieur le code source.

Préparer les images pour les environnements

Par contre, dans cette image destinée à la production, on ne place pas forcément les mêmes fichiers qu'en développement. Par exemple, le fichier php.ini est très certainement différent entre la production et le développement. Il faut collaborer entre tous les acteurs DevOps du projet pour tenir compte des exigences de chacun et créer ces fichiers.

```
mat@vm:~/project$ mkdir -p build/docker/php-fpm && cd build/docker/php-fpm
mat@vm:~/project/build/docker/php-fpm$ vi Dockerfile
FROM eu.gcr.io/myProject/myAppli/prerelease/php-fpm:latest
ARG BUILD_ENV
RUN curl --silent --show-error https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer
ENV COMPOSER_ALLOW_SUPERUSER=1
COPY build/docker/php-fpm/php7.2.ini /usr/local/etc/php/php.ini
COPY . /var/www/
RUN composer install
RUN php artisan ........
COPY build/docker/php-fpm/docker-entrypoint.sh /usr/local/bin/docker-entrypoint
RUN chmod +x /usr/local/bin/docker-entrypoint
WORKDIR /var/www
ENTRYPOINT ["docker-entrypoint"]
CMD ["php-fpm"]
```

On ajoute donc pour chaque container un builder docker qui sera exécuté par notre pipeline d'intégration continue.

```
mat@vm:~/project/build/docker$ vi cloudbuild-php-fpm.yml
steps:
- name: 'gcr.io/cloud-builders/docker'
  entrypoint: 'bash'
  args:
  - '-c'
  - |
    docker pull eu.gcr.io/myProject/myAppli/release/php-fpm:${_RELEASE_TAG} || exit 0
```

```
- name: 'gcr.io/cloud-builders/docker'
  args: ['build',
         '--build-arg',
         'BUILD_ENV=${_BUILD_ENV}',
         '-t',
         'eu.gcr.io/myProject/myAppli/release/php-fpm:${_RELEASE_TAG}',
         '-f',
         '.build/docker/php-fpm/Dockerfile',
         '--cache-from', 'eu.gcr.io/myProject/myAppli/release/php-
fpm:${_RELEASE_TAG}',
         '.']
images: [
    'eu.gcr.io/myProject/myAppli/release/php-fpm:${_RELEASE_TAG}',

]
```

Et le chef d'orchestre s'appelle Jenkins

Une fois que tout est prêt, c'est à dire le contenant avec les images prerelease et le contenu avec le code source, il est temps de lancer le build depuis notre plateforme de déploiement continu Jenkins.

Pour ce faire, on créé un **Jenkinsfile** pour exécuter les différentes étapes. Premièrement, compiler l'image à partir de l'image prerelease créée précédemment. :

```
stage('Build and push release') {
    parallel {
        stage('PHP-FPM') {
            agent { label "jenkins-slave" }
            steps {
                script {
                    if (BRANCH_NAME == 'develop'){
                        env.RELEASE_TAG='int'
                        env.BUILD_ENV='int'
                    } else if (BRANCH_NAME == 'release'){
                        env.RELEASE_TAG='release'
                        env.BUILD_ENV='release'
                    }
                }
                container('slave') {
                    withCredentials([file(credentialsId: 'gcp-login',
variable: 'GCP_LOGIN')]) {
                        sh "./build/scripts/init.sh"
                        sh "gcloud builds submit --
config=build/docker/cloudbuild-php-fpm.yml \
                            --
substitutions=_BUILD_ENV='$BUILD_ENV',_RELEASE_TAG='$RELEASE_TAG'"
                    }
                }
            }
        }
    }
}
```

Et helm pour ajouter la touche finale

En effet, la dernière étape, celle du déploiement dans le cluster Kubernetes se fera au travers de **helm**. On créé donc pour chaque service un template helm

```
mat@vm:~/project$ mkdir -p build/helm && cd build/helm
mat@vm:~/project/build/helm$ vi values.yaml
....
global:
    releaseTag: <RELEASE_TAG>
php:
    repository: eu.gcr.io/myProject/myAppli/php-fpm/release/php-fpm
    pullPolicy: Always
    replicaCount: 1
....

mat@vm:~/project/build/helm$ vi templates/php-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ template "name" . }}-{{ .Values.global.releaseTag }}-php
  labels:
    app: {{ template "name" . }}-{{ .Values.global.releaseTag }}-php
    chart: {{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_" }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
spec:
  type: ClusterIP
  ports:
    - port: 9000
  selector:
    app: {{ template "name" . }}-{{ .Values.global.releaseTag }}-php
    release: {{ .Release.Name }}

mat@vm:~/project/build/helm$ vi templates/php-deployment.php
apiVersion: apps/v1
kind: Deployment
metadata:
    name: {{ template "name" . }}-{{ .Values.global.releaseTag }}-php
    labels:
        app: {{ template "name" . }}-{{ .Values.global.releaseTag }}-php
        chart: {{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_" }}
        release: {{ .Release.Name }}
        heritage: {{ .Release.Service }}
spec:
    replicas: {{ .Values.php.replicaCount }}
    selector:
        matchLabels:
            app: {{ .Chart.Name }}-{{ .Values.global.releaseTag }}-php
    template:
        metadata:
            labels:
                app: {{ template "name" . }}-{{ .Values.global.releaseTag
}}-php
                release: {{ .Release.Name }}
                date: "{{ .Release.Time.Seconds }}"
        spec:
            containers:
            -   name: {{ .Chart.Name }}-{{ .Values.global.releaseTag }}-php
                image: "{{ .Values.php.repository }}:{{
.Values.global.imageTag }}"
                imagePullPolicy: {{ .Values.php.pullPolicy }}
```

```
                ports:
                -   containerPort: 9000
```

Noter que vous aurez besoin d'un script shell `replace_myvalues.sh` pour permettre de contextualiser le fichier `values.yaml` à partir des informations portées dans le Jenkinsfile. Par exemple le tag de la version en cours de déploiement :

```
sed -i "s/<RELEASE_TAG>/$RELEASE_TAG/g" build/helm/values.yaml
```

Il faut ensuite lancer le déploiement helm dans le Jenkinsfile :

```
stage('Deploy release') {
    agent { label "jenkins-slave" }
    steps {
        script {
            env.REPLICAS=1
            if (BRANCH_NAME == 'develop'){
                env.TAG_APP_NAME='int'
                env.RELEASE_TAG='int'
            } else if (BRANCHNAME == 'release'){
                env.TAG_APP_NAME='release'
                env.RELEASE_TAG='release'
            }
        }
        container('slave') {
            withCredentials([file(credentialsId: 'gcp-login', variable:
'GCP_LOGIN')]) {
                sh "./build/scripts/init.sh"
                sh "./build/scripts/replace_myvalues.sh"
                sh "helm init --client-only"
                sh script: "helm upgrade --force --install myProject-
$TAG_APP_NAME ./build/helm \
                        --namespace=myProject-$TAG_APP_NAME"
            }
        }
    }
}
```

Dans cette dernière étape, via le Jenkinsfile, on peut amener de la variablité qui sera poussée dans le container : par exemple, le mot de passe de connexion de l'admin de l'application. Ce secret est stocké dans le keystore de Jenkins de manière sécurisé et confidentielle.

```
environment {
    ROOT_PASSWORD_INT = credentials('root-password-no-prod')
    ROOT_PASSWORD_RELEASE = credentials('root-password-prod')
}
script {
    if (BRANCH_NAME == 'develop'){
        env.PASSWORD=$ROOT_PASSWORD_INT
    } else if (BRANCH_NAME == 'release'){
        env.PASSWORD=$ROOT_PASSWORD_RELEASE
    }
}
```

Il faut ensuite remonter cette variable `PASSWORD` dans le fichier key:value `values.yaml` du déploiement helm puis dans l'environnement d'exécution du container php en ajoutant un environnement au déploiement helm :

```
mat@vm:~/project/build/helm$ vi templates/php-deployment.php
spec:
    template:
        spec:
            containers:
            -   env:
                -   name: APPLI_PASSWORD
                    valueFrom:
                        secretKeyRef:
                            name: {{ template "name" . }}-{{
.Values.global.releaseTag }}
                            key: appli-password
```

Cette variable d'environnement `APPLI_PASSWORD` sera alors disponible dans fichier `docker-entrypoint.sh` et on pourra faire des manipulations de fichiers post-déploiements:

```
mat@vm:~/project/build/docker/php-fpm$ vi docker-entrypoint.sh
sed -i "s/APPLI_PASSWORD>/$APPLI_PASSWORD/g" /var/www/config.php
```

Conclusion

Au sein d'**Inside Group**, en utilisant les containers Docker, nous pouvons déployer notre application dans tous les environnements en étant représentatif de la production. Pour ce faire, on utilise le pipeline Jenkins pour orchestrer la construction et le déploiement via helm de notre application.

A vous de jouer … Et pourquoi pas venir nous rencontrer pour partager nos expériences.

Mathieu DEFIANAS - Inside Group

Expert DevOps

mathieu.defianas@insidegroup.fr