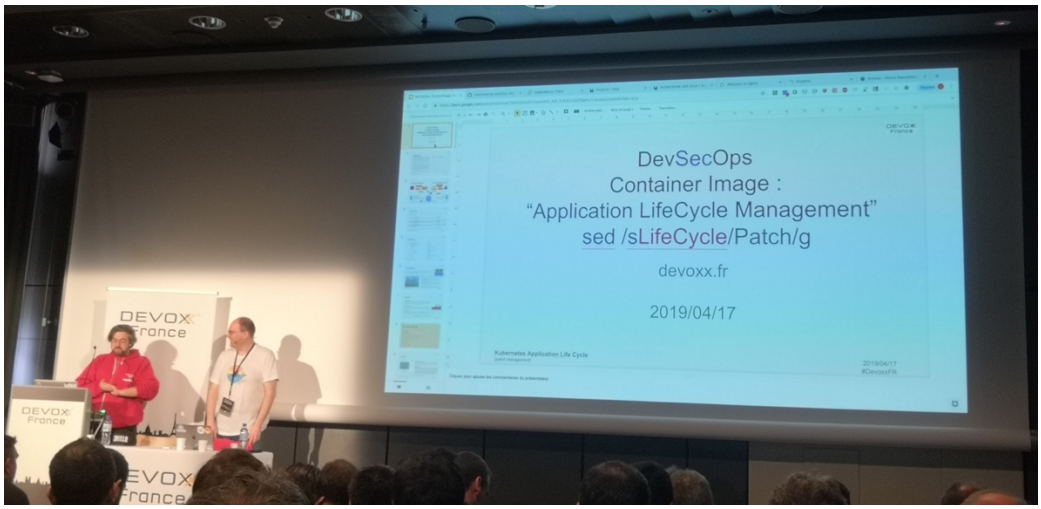
Cycle de vie des applications dans Kubernetes

DevSecOps Container Image : « Application LifeCycle Management »
sed / sLifeCycle/Patch/g

Speakers : Charles Sabourdin (Freelance), Jean-Christophe Sirot (Docker Inc.)
Format : University
Date : 17 avril 2019
Slides : SlideShare de Charles



1. Reminder

1.2 Docker

Charles commence par rappeler que Docker est une technologie venant du kernel Linux basée sur chroot et les **technologies d'isolation**. Les cgroups permettent d'isoler les processus. L'objectif numéro 1 de Docker est **l'hyper-densité** : **sur une machine, mettre plus d'images Docker que de VM**.
Les Union **file systems** permettent de travailler sur des layers qui vont se cumuler les uns aux autres
Sur son laptop, Charles utilise DockerMachine car cela permet de se rapprocher de la Prod, même si cet outil est moins performant que Docker for Mac ou Windows.

Une première démo consiste à démarrer un conteneur nginx puis à utiliser l'outil **dive** pour inspecter les layers d'une image
Exemple des layers de Jenkins : on voit l'image de base puis toutes les lignes de commandes passées. Dive donne la taille de l'image et précise également la taille perdue.
Dans l'image Nginx, on voit l'usage des LABEL :
LABEL Name="ocp-nginx"
Une bonne pratique consiste à chainer les commandes qui ont la même sémantique : ceci afin de ne créer qu'un seul layer dans l'image.
Problématique des outils à intégrer ou non dans l'image : ces outils sont nécessaires pour debugger même s'ils augmentent la taille de l'image.

Glossaire :

- Image Docker : images créées à partir des Dockerfile. Eléments immuables.
- Container : instance d'une image
- Docker client : client en ligne de commande permettant de communiquer avec le démon Docker
- Docker Host : une machine physique ou virtuelle exécutant un démon Docker
- Docker Registry : héberge les images Docker (Docker Hub)

1.2 Kubernetes

Outil d'orchestration des images Docker.
Support des hosts Windows depuis début avril 2019.
Glossaire :

- Master : machine qui orchestre les nœuds.
- Nœuds : machines physiques sur lesquelles sont lancés les images
- Pods : groupe de conteneurs (gousse du haricot) déployés sur un unique nœud : unité atomique d'exécution. Permet de partager l'IP, le hostname et le storage
- Kubelet : service démarrant les nœud et gérant leur état
- Replication controler : contrôle combien de pods s'exécutent sur le cluster
- Service : groupe de pods travaillant ensemble

Le Deployment permet de définir les ReplicatSets qui déclarent les Pods.

Démo de création d'un namespace :
Kubectl create ns sample2

La commande *kubectl run* est un raccourci (déprécié) qui permet de créer un ReplicatSet et un Pod.

Charles utilise un certain nombre d'alias kubectl (scripts Python).
L'outil kubens permet de gérer les namespaces et passer d'un namespace à un autre
On peut ajouter d'autres registry que docker.io

L'option *terminationGracePeriodSeconds* permet de définir le temps de latence avant la suppression du pod. En dév il nous recommande de diminuer sa valeur de 30 secondes à 1 ou 2 secondes.

Commande *kubectl exec* : commande similaire à *docker exec* : permet d'exécuter des commandes (ex : /bin/bash) à l'intérieur du pod.

Le fichier de configuration YAML permet de décrire le déploiement souhaité.
Démo de mise à jour d'un fichier de configuration : v1 -> v2.

Préconisation pour la sécurité : utiliser un Registry avec des images signées.

Plusieurs **stratégies de déploiements** sont possibles :

1. **Rolling deployment** : pendant que les V1 sont arrêtés une à une, les V2 sont démarrés une à une.
2. **Recreate deployment** : on arrête d'un coup toutes les V1, puis on démarre progressivement les V2. Lors de l'upgrade d'une base de données par exemple, on doit arrêter le pod (arrêt de service) pour libérer le volume
3. **Blue Green release** : on instancie les 2 versions d'un coup et on bascule tout
4. **Canary release** : quelques V2 remplacent une petite partie des V1

Stratégie de Probs :

1. **Liveness** : vérifie que le conteneur fonctionne et s'il ne répond pas (pas de code 200) au bout d'un laps de temps, le conteneur est tué puis redémarré
2. **Readyness** : regarde la charge du conteneur. S'il ne répond pas : le pod est retiré du flux et il n'y a plus de charge dessus.

Attention, les Probs peuvent cacher des problèmes : lorsque des Pods redémarrent régulièrement, cela doit cacher un problème et il faut regarder de plus près pourquoi.

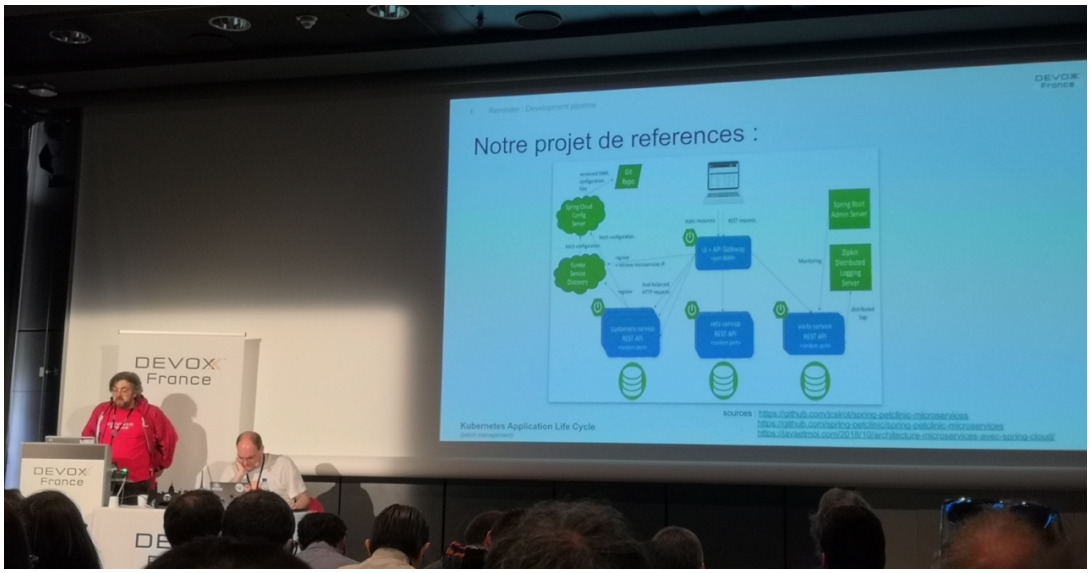
initContainers & Container hook : opérations au démarrage du pod. Exemple d'un script qui regarde la version de la base de données et applique des scripts de mise à jour.

Commande pour accéder à la documentation Kubernetes :
kubetcl explain pods.spec.container
La doc Kubernetes est à jour et relativement complète.

Question sur le temps d'arrêt d'un Pod : Kubernetes est complètement asynchrone et n'est donc pas toujours très réactif. Le Pod va chercher la conf sur le Kublet server. Ce problème de latence peut également venir des volumes. Dans un système statefull, lorsque le Pod 2 qui démarre n'arrive pas à accéder à une ressource partagée (le volume) car elle est bloquée par le Pod 1. Les applications doivent être conçues pour travailler en cluster. Essayer de les penser stateless (ex : session partagée pour une webapp Java).

1.3 Deployment pipeline

A ma grande surprise, le projet de référence utilisé par Charles et Jean-Christophe n'est autre que **Spring Petclinic Microservices**. La PR#126 soumise fugacement la veille aurait dû me mettre la puce à l'oreille.



Les speakers distinguent plusieurs environnements :

- Dev local : Sur son poste de dév, le développeur développe et teste.
- Dev on cluster : développement sur un namespace privé.
- CI on cluster : build automatisé sur Jenkins à chaque commit : construit l'artefact Docker qui sera déployé dans le cluster QA
- QA on cluster : dédiés aux tests automatisés
- Staging on cluster (facultatif) : pour les tests manuels et les tests de charges
- Production on cluster

Pendant la préparation de la démo, les speakers ont trouvé que l'intégration de Spring Petclinic Microservices à Kubernetes était compliquée. J'ai remarqué qu'ils ne s'étaient pas appuyés sur le projet Spring Cloud Kubernetes qui auraient certainement pu leur simplifier la tâche.
Jean-Christophe n'a pas souhaité utiliser le Docker maven plugin de Spotify mis en œuvre sur l'application démo. Sur son poste local, il a préféré avoir le même build que sur la CI. Il a utilisé du build dans Docker. A la place d'un *mvn clean package* : on lance Docker pour que le maven se lance dans l'image. Il a également utilisé du multi-staged build : une première étape « builder » construit le binaire (le JAR) et une 2ne étape « base » utilise le JAR construit par le premier.
Jean-Christophe a utilisé un makefile des années 80 pour enchainer les build docker.
Le build dure très longtemps car il n'y'a pas de volume partagé pour le repo local Maven : les artefacts sont re-téléchargés pour chaque image.
En local, JC utilise docker-compose pour démarrer tous les microservices.

Helm

Helm est un outil permettant de gérer les packages Kubernetes appelés « charts ».
Help permet :

- Créer de nouveaux charts from scratch
- Packager les charts dans une archive tgz

- Installer / désinstaller les charts dans Kubernetes
- Gérer le cycle de vie des charts

Un système de hook permet de déclencher des opérations lors des différentes phases du cycle de vie du déploiement : pre-install, pre-upgrade, crd-install, post-rollback …

Commande : *helm list*
Helm permet de déployer plusieurs releases du même paquet. Privilégier le déploiement dans différents namespaces.
L'ordonnancement des hooks Helm est géré par des poids.

Dans admin-server-deploy.yaml, utilisation de nindent de 12 espaces pour copier du YAML dans un autre.

Helm utilise Tiller pour dialoguer avec le cluster Kubernetes. Tiller utilise le même kubeconfig que helm pour se connecter au cluster.
Le projet Tillerless permet d'utiliser Tiller en local (en dehors du cluster Kubernetes).
Le repo GitHub de Helm fournit des exemples de charts : pour Consul, MongoDB, MySQL, etcd, Jenkins …

Registry

Comme registre Docker, on peut par exemple utiliser le Nexus de Sonatype. Nexus est également un registre pour Java, NPM, Python, C#, NuGet, APT, YUM …

Jenkins

Jenkins construit l'image Docker et le publie dans la Registry.
Exemple d'un Jenkinsfile avec gestion de la sécurité.

SonarQube

SonaQube intervient dans le pipeline de développement.
Le pipeline Jenkins est composé d'une étape lançant l'analyse SonarQube permettant de détecte des défauts de qualité.

ZappProxy

- Man-in-the-middle proxy
- Traditional and AJAX spiders
- Automated scanner
- Fuzzer
- Dynamic SSL certificates

Gestion de la maturité

- Continuous Integration : construction des binaires
- Continuous Delivery (CDE) : on pousse des binaires avec un système de workflow et quelques actions manuelles
- Continuous Deployment (CD) : livraisons automatisées fréquentes

Le passage du CDE au CD nécessite un niveau de maturité certain dans le CDE. Tant qu'on ne met pas en prod le vendredi, c'est qu'on n'est pas prêt pour le CD.

2. Application lifecycle

2.1 Scanning Tools

Les failles de sécurité CVE sont découvertes au fil du temps. Elles nécessitent la mise en place un système permettant de limiter les risques.
Dans les dépôts, les images Docker et les packages Helm peuvent être touchés. Il est donc nécessaire de pouvoir patcher une application en prod en moins de 4h.
Il existe de nombreux outils d'images Docker permettent de prévenir en cas de failles de sécu.
La plupart vont vérifier l'image Docker, mais pas le code applicatif (contrairement à Fortify).
L'outil Dependency Track permet de chercher des failles dans les artefacts Maven ou NPM.
Possibilité de gérer l'acceptabilité des failles directement dans l'outil. 4 failles CVE sur jackson-databind 2.9.7.
L'outil Snyk est un autre outil permettant de détecter des failles dans les dépendances Maven.

Lorsqu'on détecte une faille de sécu, on met à jour les dépendances puis on rebuild l'image. Cela devient vite ingérable par la prod.
Il arrive qu'après un certain temps, il est nécessaire de faire évoluer le process de build afin d'être capable de rebuilder l'image.

Vocabulaire proposé :

- Still image : image Docker utilisée telle quelle (ex : MySQL, MongoDB). On ne les modifie pas.
- Base image : une image de base sur laquelle on construit un applicatif.
- Application image : une application construite à partir de l'image de base et le code applicatif.

3. Proposed Solution

3.1 1,2,3 hosting

Le modèle 1, 2 et 3 :

1. Les Base Image doivent être gérées par la Prod (les Ops) : choix, cycle de vie, maintenance
2. Les projets utilisent leurs propres Dockerfile pour des impératifs projets (ex : dernière version de Node.JS ou de Python pas encore mise à disposition par les Ops) : c'est au projet de gérer son cycle de vie et en est donc responsable. Il faut accepter que les différentes entités d'une grosse société n'aillent pas au même rythme.
3. Les projets utilisent les images Docker signées d'un tiers (ex : Editeur)

3.2 Problèmes classiques

- Absence d'un responsable d'application (project Owner) : il arrive fréquemment qu'il ait changé d'équipe.
- Équilibre à trouver dans l'automatisation des tâches et des outils
- Manque de communication : naturellement, les objectifs des uns et des autres ne sont pas les mêmes (ex : dev team vs hosting team). Les frontières ne sont pas très marquées. Débat : qui écrit le YAML des Helm ? Qui écrit les Dockerfile ? Avec Helm, les ressources (nombre de RAM) peuvent être templatisées et valorisées par les Ops.
- Build to Prod sans test
- Explosion des versions : faut-il pousser dans le Registry Docker toutes les images snapshots ?
- Manque de support