# OPENSHIFT

## Le Paas open source orienté DevOps

SMILE

I.T IS OPEN

# Sommaire

**1 Préambule** **3**
Smile . . . 3
Ce livre blanc . . . 4
Auteurs . . . 6

**2 OpenShift** **7**
Présentation . . . 7
Le PaaS/CaaS . . . 8
Le modèle 3 couches . . . 8
PaaS - Plateform as a Service . . . 10
CaaS - Container as a Service . . . 10
Kubernetes, l'orchestrateur . . . 12
OpenShift, la gestion avancée . . . 13
Un bon compagnon du "continuous delivery" . . . 16
L'application dans un projet . . . 16
L'objet de déploiement - DeploymentConfig . . . 16
Configuration de build - ou BuildConfig . . . 17
Construction des images Docker . . . 19
Rappel sur l'état de l'art . . . 19
Source 2 Image . . . 20
Rappel sur le principe de "Service" et route . . . 23

**3 Didacticiel : Déploiement de ShareLaTeX avec OpenShift** **26**

**4 Didacticiel : Template d'applications** **30**

**5 Retours d'expérience Smile** **36**
Création des images S2I . . . 36
Projet Git et postes de développeurs . . . 37
Projet OpenShift . . . 38
Pré-production . . . 39
Architecture . . . 39
Production . . . 41
Bilan . . . 41

Interview de Sam, développeur dans l'équipe projet Smile . . . . . 42

**6 Ce qu'il faut en retenir 44**

# Préambule

## Smile

Smile est le leader européen de l'intégration et de l'infogérance de solutions open source.

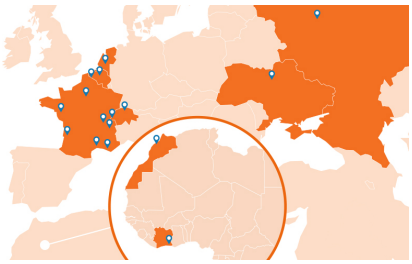
Smile compte 1100 collaborateurs dans le monde, dont plus de 700 en France, ce qui en fait la première société en France et en Europe spécialisée dans l'open source. Depuis 2000, Smile mène une action active de veille technologique qui lui permet de découvrir les produits les plus prometteurs de l'open source, de les qualifier, de les évaluer, puis de les déployer, de manière à proposer à ses clients les produits les plus aboutis, les plus robustes et les plus pérennes. Cette démarche a donné lieu à toute une gamme de livres blancs couvrant différents domaines d'application. La gestion de contenus, les portails, le décisionnel, les frameworks PHP, le cloud, la Gestion Electronique de Documents, les ERP, le big data, le DevOps ...

Chacun de ces ouvrages présente une sélection des meilleures solutions open source dans le domaine considéré, leurs qualités respectives, ainsi que des retours d'expérience opérationnels.


Au fur et à mesure que des solutions open source solides gagnent de nouveaux domaines, Smile est et sera présent pour proposer à ses clients d'en bénéficier sans risque. Smile

apparaît dans le paysage informatique français et européen comme le prestataire intégrateur de choix pour accompagner les plus grandes entreprises dans l'adoption des meilleures solutions open source. Ces dernières années, Smile a également étendu la gamme des services proposés. Depuis 2005, un département consulting accompagne nos clients, tant dans les phases d'avant-projet, en recherche de solutions, qu'en accompagnement de projet. Depuis 2000, Smile dispose d'une Agence Digitale, proposant outre la création graphique, une expertise e-marketing, éditoriale, SEO, et interfaces riches. Smile dispose aussi d'une agence spécialisée dans la Tierce Maintenance Applicative, l'infogérance et l'exploitation des applications.

Enfin, Smile est implanté à Paris, Lyon, Nantes, Bordeaux, Lille, Marseille, Montpellier, Grenoble et Toulouse. Et présent également en Suisse, en Ukraine, aux Pays-Bas, en Belgique, au Maroc ainsi qu'en Côte d'Ivoire.



En parallèle à ces publications, pour bien comprendre la révolution en marche de l'open source, Smile a publié plusieurs livres blancs expliquant les spécificités, les modèles économiques, les sous-jacents ainsi que les atouts de l'open source.






## Ce livre blanc

Les démarches DevOps sont en plein essor. Le rapprochement des développeurs et des administrateurs système permet de réduire drastiquement les temps de mise en production. Nous l'avons vu dans notre précédent ouvrage "DevOps et industrialisation" [1], les

1. http://www.smile.fr/Ressources/Livres-blancs/Systeme-et-infrastructure/Devops-et-industrialisation

outils CI/CD [2] couplés à des plateformes de conteneurs ou de machines virtuelles soulagent fortement la charge de travail nécessaire pour atteindre une automatisation efficace. Parmi ces plateformes, nous ne pouvons pas passer à coté des fameux "PaaS" pour "Platform as a Service", très en vogue et ce pour d'excellentes raisons. Car si certains détracteurs ont parlé d'effet de mode il y a de cela quelques années, force est de constater aujourd'hui que nous sommes bien loin d'un simple effet "buzz". Les PaaS ont apporté du confort, de la méthode et surtout une réduction de coût notable d'infrastructure en plus de la réduction du planning "time-to-market".

Car c'est bien là l'atout principal d'un PaaS efficace : permettre autant la création de POC en quelques clics, que de déployer des dizaines, centaines ou milliers d'applications répliquées dans une infrastructure parfois hétérogène, multi-zones sans en perdre le contrôle.

Les PaaS sont bel et bien des outils d'une efficacité redoutable qui permettent, au-delà de faire tourner des applications scalées sur différents nœuds réseau, de mettre en place des ***méthodologies attachées à l'approche DevOps*** et par conséquent de fournir des services de développement et de mise en production ***industrialisée et fiables*** .

Il existe plusieurs plateformes dont nous avons déjà évoqué les bénéfices.

Nous pouvons parler de Mesos, par exemple, qui propose entre autre une plateforme Docker nommée Marathon. Mais ici, nous allons parler de la solution Red Hat, basée sur l'orchestrateur Kubernetes créé par Google, ***"OpenShift Container Plateform"*** ; que nous désignerons par "OpenShift" dans la suite de cet ouvrage.

Parce que Smile apprécie fortement Kubernetes, le fait qu'un interfaçage plus proche du développeur soit proposé sous licence open source par Red Hat a apporté un nouveau souffle à notre travail !

Là où Kubernetes propose une approche proche des administrateurs systèmes et réseau, OpenShift permet à des non-initiés de s'adapter, déployer et contrôler des applications sur une interface graphique très accessible et réactive. Le principe de template d'application, ou encore l'import de projets "docker-compose" au moyen d'une simple ligne de commande permet de laisser le contrôle à ceux qui produisent le code.

OpenShift est un outil fiable, adapté, qui s'inscrit dans l'univers des outils DevOps et qui peut d'ailleurs très rapidement devenir ***l'atout central de votre chaîne CI/CD et de votre démarche DevOps*** . Nos expériences, notre utilisation interne et nos solutions apportées au travers de OpenShift n'ont de cesse de nous contenter jour après jour. L'outil est fourni, stable, intelligent et très apprécié par nos équipes et nos clients.

Voilà pourquoi il nous faut partager avec la communauté, dans ce livre blanc, ce que vous devez savoir sur cette solution PaaS.

2. Continuous Integration / Continuous Delivery

## Auteurs

Ce livre blanc a été rédigé par Patrice Ferlet (Expert Technique R&D Smile)

Nous remercions Laurent Broudoux de Red Hat pour avoir répondu à nos questions

Cet ouvrage a été rédigé en LaTeX [3] sur l'interface ShareLaTeX [4], deux logiciels open source.

3. http://www.latex-project.org/
4. https://fr.sharelatex.com/

# OpenShift

## Présentation

OpenShift est une solution de ***PaaS (Platform-as-a-Service)*** développée par Red Hat.

OpenShift utilise Kubernetes comme base d'orchestration qui est, lui, développé par Google. ***OpenShift apporte plusieurs fonctionnalités très intéressantes*** , notamment un catalogue d'images à la création de projet, un gestionnaire de construction d'images propre à chaque projet, la gestion d'utilisateurs, etc. Les deux produits sont open source et sont accessibles depuis des dépôts GitHub [1].

OpenShift est un PaaS qui s'appuie sur des conteneurs pour le déploiement de ressources, en l'occurrence sur les technologies apportées par ***Docker*** . Ces technologies sont utilisées à deux niveaux : en tant que format d'image des conteneurs et en tant que moteur d'exécution au travers de la librairie ***runC*** . A ce titre, on peut aussi parler de CaaS (Container As A Service), mais on préférera le terme PaaS, englobant des besoins plus larges, de la conteneurisation au cycle de développement applicatif.

Il existe deux versions de OpenShift :

- ***OpenShift Origin*** est la version communautaire. Cette version propose les dernières évolutions développées. Elle est toutefois moins stable étant donné la jeunesse et la fréquence des avancées. Red Hat ne propose pas de support sur cette version.

- ***OpenShift Container Platform*** est la version stabilisée et supportée. Elle est un peu moins avancée en termes de fonctionnalités nouvelles mais bénéficie de l'expérience des administrateurs, en plus du support de Red Hat, et est donc complètement adaptée à des environnements en production.

1. https ://github.com/kubernetes et https ://github.com/openshift/

# Le PaaS/CaaS

Avant d'aller plus loin, voyons ce qu'est un PaaS, et pourquoi nous parlons aussi parfois de CaaS (Container-as-a-Service).

## Le modèle 3 couches

Dans l'univers des services de Cloud, il existe des niveaux d'abstraction distincts et en relation les uns avec les autres qui permettent d'identifier le rôle des intervenants. Il se dégage 3 couches qui indiquent la visibilité technique des solutions intégrées au Cloud :

- **SaaS** Software as a Service
- **PaaS** Platform as a Service
- **IaaS** Infrastructure as a Service

La nouvelle couche ***CaaS*** n'est qu'une extension du PaaS. Voyons ce que ces couches induisent et comment interpréter la venue de cette couche de conteneurs dans le Cloud.

Le ***IaaS*** , ou "infrastructure-as-a-Service", permet de contrôler une infrastructure à la demande tel que le propose AWS, Google Compute Engine, OVH cloud ou encore dans un environnement interne avec OpenStack. La plateforme propose de gérer le réseau, les machines, les ressources de stockage, et bien d'autres aspects ; et via une plateforme dédiée. En général, les machines sont virtualisées au travers d'un hyperviseur tels que KVM ou VMWare.

La couche ***SaaS*** , ou "Software-as-a-Service", est la couche la plus "abstraite" des trois. Cette fois, la plateforme vous propose une série de logiciels délivrés par la plateforme. Par exemple les services Google Gmail, Microsoft Office 360, ou dans le monde open source OwnCloud, cozyCloud, ... sont de parfaits exemples de SaaS. L'utilisateur ne s'occupe que du paramétrage de l'application si besoin et devient un utilisateur final du service. L'infrastructure ou encore le type d'OS utilisé n'est pas apparent.

La couche qui nous intéresse est le ***PaaS*** , ou "Platform as a Service". Cette couche fournit cette fois-ci une plateforme qui permet de gérer des ressources *applicatives*. Ce niveau d'abstraction permet à des développeurs, chefs de projets, ingénieurs de recherche, etc. d'initialiser des services tels que des services Web, des bases de données ou encore des logiciels de traitement. Le PaaS se trouve entre la couche IaaS et SaaS.

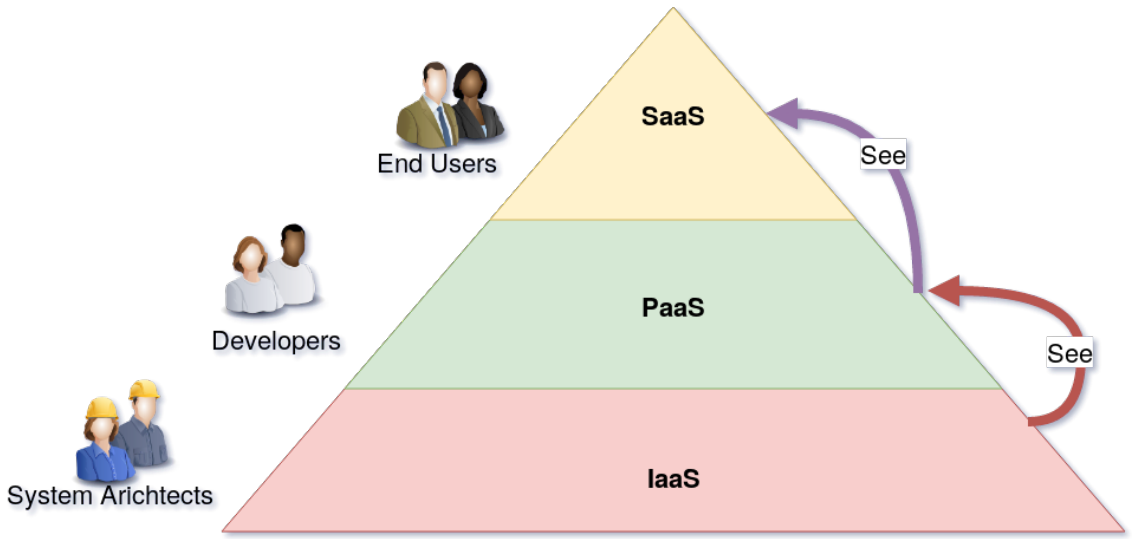



Figure 2.1 – *Couches des structures IaaS, PaaS et SaaS et leurs visibilités l'un-l'autre*

Les 3 couches présentées dans la figure 2.1 permettent de distinguer le niveau d'abstraction et la visibilité qu'un utilisateur de la couche peut avoir par rapport à l'autre.

Nous pouvons clarifier cette vue pour en comprendre ce qu'il s'en dégage : la visibilité des couches ne se fait que de bas en haut, c'est-à-dire depuis l'infrastructure jusqu'à la couche de service logiciel. Rapidement, on comprend que la couche SaaS est surtout proposée pour les utilisateurs finaux alors que la couche IaaS est utilisée par les architectes réseaux et système. Le PaaS étant alors adapté aux développeurs. Effectivement, l'infrastructure "voit" la couche de plateforme (sans pour autant interagir avec), et le développeur "voit" la couche logicielle (puisque finalement, c'est bien ce développeur qui initie le logiciel dans la plateforme).

Dans le modèle 3 couches "as a Service", l'utilisateur final utilise le logiciel (SaaS) qui a été développé et mis en place (PaaS) dans une infrastructure que les architectes système maintiennent (IaaS)

## PaaS - Plateform as a Service

Voyons plus en détail ce qui nous intéresse plus particulièrement, c'est-à-dire ce qu'est un PaaS.

Le PaaS se trouve entre les couches IaaS et SaaS. C'est une couche orientée "développeur" - l'infrastructure n'est pas perçue par le développeur qui l'utilise. Par contre, il peut demander des ressources applicatives telles qu'un serveur Apache HTTPd, une base MySQL, un espace de stockage... - sans pour autant avoir besoin de paramétrer les liens réseaux.

Le PaaS est indiqué dans différents environnements qui nécessitent une visibilité de fonctionnement pour les développeurs afin qu'ils puissent eux-mêmes interagir avec la production sans pour autant impacter l'infrastructure. Mais aussi dans une démarche DevOps en utilisant les principes de Continuous Delivery.

OpenShift officie dans cette couche. Il propose une gestion de namespaces (provenant de Kubernetes) qui isolent les services demandés en tant que "projets". Chaque application est instanciée dans des conteneurs issus d'images qui proviennent d'un registre intégré ou non à la plateforme.

Par extension, nous pouvons admettre une nouvelle couche au sein de la couche "PaaS" que nous appelons "CaaS".

## CaaS - Container as a Service

La notion CaaS, ou "Container as a Service" n'est pas réellement définie dans le modèle 3 couches présenté précédemment. Cela-dit, l'engouement pour les conteneurs intégrés dans les PaaS tels que le proposent OpenShift ou MesOS/Marathon a poussé l'usage de ce terme.

Il se définit par l'utilisation d'un PaaS basé sur la technologie de conteneurs.

OpenShift utilise Docker à ce jour, et RkT [2] pourra aussi être utilisé à l'avenir. L'intérêt du CaaS est d'éviter l'interférence de services en utilisant l'isolation proposée par les technologies de conteneurs.

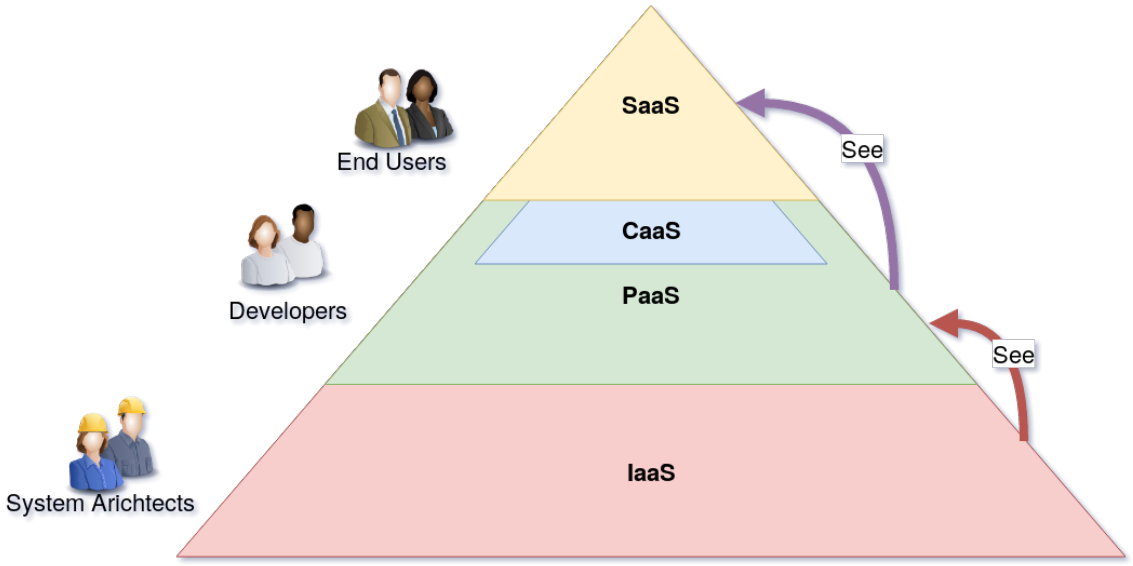



Figure 2.2 – *La couche CaaS se trouve dans la couche PaaS*

Lorsqu'un développeur instancie une application (par exemple un service de base de données ou un serveur Web) alors un conteneur est démarré à partir d'une image. Si le PaaS est installé en cluster alors le conteneur peut être instancié sur une des machines qui compose le nuage. Et par conséquent, si l'application en question est scalable, alors les nouvelles instances peuvent être démarrées sur différents serveurs physiques.

2. "rockit" - développé à l'origine pour la distribution CoreOS

> Remarquez ici l'effort pour ne pas utiliser le terme "service" lorsque nous parlons d'application serveur. En effet, la notion de "service" est un objet spécifique dans un projet OpenShift qui définit comment contacter une application.

En outre, les PaaS gèrent généralement la notion de zones. Ainsi, le PaaS/CaaS peut sélectionner le noeud où doit démarrer le conteneur en fonction de critères plus ou moins fins. On distingue généralement des zones de production et régionales pour permettre une répartition des conteneurs sur différents continents et pour éviter les interactions malencontreuses d'un service en pré-production ou en test avec celui proposé en production.

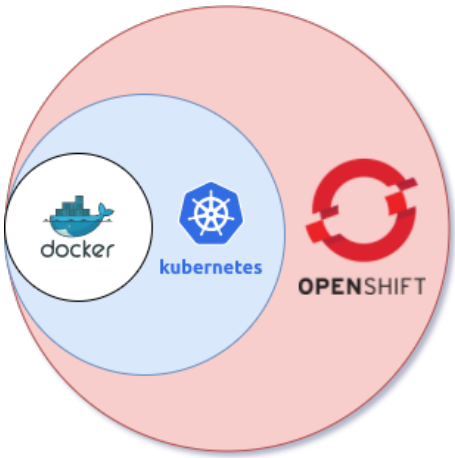



Figure 2.3 – *Encapsulation des technologies, de Docker à Openshift en passant par Kubernetes*

### Kubernetes, l'orchestrateur

Kubernetes, développé par Google, a été libéré en open source et a très rapidement prouvé sa justesse. Son idée est de proposer un certain nombre de types d'objet paramétrables via des fichiers de configuration pour permettre un déploiement et une interconnexion aisée sans pour autant demander à avoir pleinement conscience de l'infrastructure qui se trouve dans le cluster.

Kubernetes va abstraire le concept de "conteneur" pour proposer une vision plus applicative : ***Pod*** , ***Service*** , ***Replication*** ... Docker devient, quant à lui, un outil de conteneuri-

sation encapsulé [3].

Ainsi, proposer un déploiement de service se résume à appeler une API en fournissant des instructions de déploiement d'image Docker. A charge de l'orchestrateur intégré de savoir quelle machine, quel volume et quels ressources seront allouées à ce déploiement.

Mais si Kubernetes permet le déploiement aisé, il ne propose pas de gestion de projet à proprement parler. Il propose bien une gestion d'espaces de noms pour séparer les applications, mais cela ne repose que sur la bonne conduite des déploiements en amont.

Kubernetes est un projet déjà très avancé de Google, développé depuis des années pour centraliser et orchestrer le parc de machines, il a été rendu public et open source depuis l'avènement de la conteneurisation (et notamment Docker).

Kubernetes propose plusieurs services interconnectés tels que :

- un scheduler qui permet de répartir les conteneurs sur les machines en fonction de la charge produite et nécessaire
- un controller qui centralise la configuration, les informations et l'interconnexion
- une API REST qui permet de contrôler, démarrer, stopper, et globalement gérer le cluster
- des addons, tels qu'un dashboard, monitoring, dns interne, etc.

Développé en Go, il est avant tout utilisé par les "Ops" (administrateurs) de par son approche technique et peu contrainte. L'authentification est par exemple généralement proposée sous forme de certificats utilisateurs, les namespaces (qui permettent de regrouper des projets) ne sont pas nativement soumis à des ACL, et il n'y a pas de contrainte en terme de conteneur à démarrer. Une image qui permet l'instanciation d'un conteneur dont l'utilisateur principal est "root" ne pose pas de souci.

En soit, ***Kubernetes est un orchestrateur très puissant, stable et fortement utilisé*** . Sa communauté est très large et très active.

### OpenShift, la gestion avancée

OpenShift est né entre les murs de la firme Red Hat. Il propose de compléter Kubernetes avec des fonctionnalités adaptées pour la gestion de projet.

3. il est d'ailleurs possible d'éxécuter des conteneurs autres que Docker via Kubernetes

OpenShift est un PaaS (Platform as a Service), ce qui signifie qu'il propose une interface de gestion à des services proposés à l'utilisateur. En l'occurence, ici, ***l'utilisateur de OpenShift est un chef de projet, un développeur ou un administrateur de service*** .

Il est une "surcouche" à Kubernetes. L'orchestration, c'est-à-dire la gestion de démarrage et de "scaling" est laissée à la charge de Kubernetes (scheduler et controller) alors que OpenShift en lui-même va proposer des objets de configuration plus poussés.

OpenShift encapsule Kubernetes et propose de nouveaux objets pour :

- construire des images de l'application
- intégrer un registre privé paramétré pour séparer les images par projet
- proposer une interface utilisable par un "non-administrateur"
- proposer des routeurs vers applications
- gestion de droits, restriction, authentification, ...
- ...

Loin de vouloir prendre la place de Kubernetes en l'écrasant de fonctionnalités, il faut bien comprendre que Kubernetes et OpenShift n'ont pas le même public en vue : ***Kubernetes est avant tout un PaaS orienté déploiement, alors que OpenShift est un PaaS orienté développement*** .

En renforçant la séparation des namespaces, nommé "project" pour l'occasion, OpenShift propose aussi une nouvelle liste d'objets qui peuvent, dans beaucoup de cas, supplanter la définition en vigueur dans Kubernetes.

Par exemple, Kubernetes propose ces objets :

- ReplicationController, est très proche d'un objet Deployment ; c'est une configuration qui décrit quelles images sont déployées, le nombre de réplicas en fonction à garantir, comment vérifier l'état du Pod, etc.
- NameSpace, est un espace de nom pur et simple. Il propose simplement de classer des objets de configuration en les groupant au sein d'une entité abstraite.

OpenShift permet ces objets :

- Project - surcharge la notion de Namespace en proposant un ACL
- DeploymentConfig - qui surcharge les Deployments de Kubernetes, il propose de définir quand un déploiement est effectué (eg. une image a été poussée dans le registre), et d'autres options.
- ImageStream - qui est une base d'image pour un projet.

*OpenShift ajoute de plus des objets de contexte de sécurité* (SecurityContext), gestion d'authentification multiple (ex : LDAP), *gestion de "build" d'images à partir de sources sur Git* , etc.

OpenShift intègre aussi tout un système de routeur, qui permet de définir une adresse Web pour un service donné.

Mais l'un des ajouts notables que propose OpenShift est la *gestion de templates d'application* qui permet de démarrer plusieurs applications (reliées entre elles ou non) en passant par un formulaire pour configurer différentes options. Par exemple, le nom des services, les ressources à allouer, l'authentification, la localisation des sources, etc.

En clair, OpenShift permet d'utiliser Kubernetes en termes de gestion de projet pour des équipes diverses et variées.

Les deux équipes de projet Kubernetes et OpenShift communiquent régulièrement dans une optique d'évolution et d'entente. Ainsi, il n'est pas rare de voir les équipes de OpenShift proposer des améliorations à Kubernetes (eg. la gestion de template d'application) et inversement, voir des collaborateurs du projet Kubernetes proposer son aide pour le développement de OpenShift.

Certains développeurs sont même autant actifs dans les deux projets, ce qui démontre une réelle collaboration entre les deux solutions/équipes/sociétés. Tout cela pour dire que depuis 2013, les deux projets sont identifiés comme *pérennes et très actifs* .

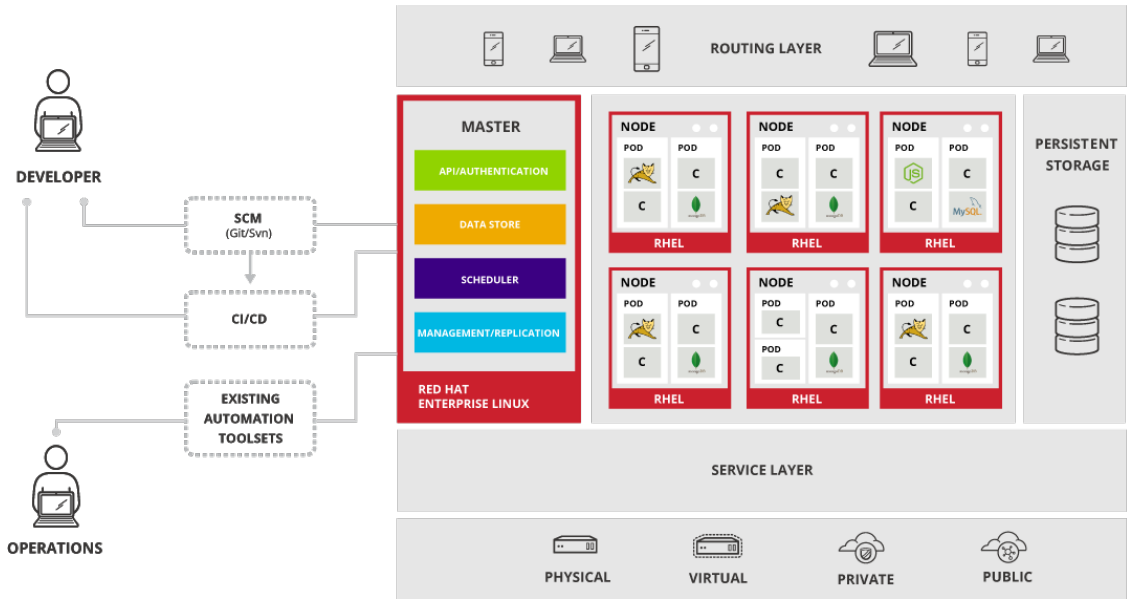



Figure 2.4 – *Architecture OpenShift telle que présentée dans la documentation*

## Un bon compagnon du "continuous delivery"

Si aujourd'hui le "continuous delivery" ou "livraison continue" entre dans les moeurs, il faut bien avouer que Jenkins est très utilisé pour construire ce que nous appelons un "artefact" - l'objet de livraison. Mais par le fait que OpenShift propose la construction d'images Docker, et que l'ensemble d'un "build" s'effectue dans un Dockerfile, OpenShift est bel et bien capable de produire l'artefact pour le mettre en production.

C'est pour cela que OpenShift est un compagnon très intéressant dans une chaine CD (Continuous Delivery).

### L'application dans un projet

Une application est un ensemble de plusieurs objets tels que Deployment, Build, Services, Routes, etc. La commande "oc" permet de créer une application, ce qui a pour effet de générer les objets de configurations nécessaires en une seule commande. Une application n'est pas définie en tant que telle dans OpenShift, c'est une notion abstraite qui n'est valable que lors d'une création.

L'application peut être issue d'un "template", c'est-à-dire le résultat d'un gabarit d'application que propose OpenShift. Encore une fois, l'outil "oc" est capable de faire le process, mais l'interface Web va vous proposer ces templates sous forme de formulaire pour appliquer des valeurs spécifiques telles que l'adresse où se trouve les sources git, les limites de mémoire et CPU à utiliser, variables d'environnements à adapter, etc.

Après application d'un template, ou création d'une application dans un projet, les objets nécessaires sont créés.

### L'objet de déploiement - DeploymentConfig

Cet objet est une configuration de déploiement, en l'occurrence il spécifie quels conteneurs démarrer et comment s'assurer du fonctionnement. C'est un objet de configuration d'un "Deployment" tel qu'il existe dans Kubernetes. D'ailleurs, la syntaxe est très proche de l'objet de Kubernetes.

Ce qui change par rapport à l'objet de Kubernetes, c'est la section qui propose d'automatiser le déploiement en fonction d'évènements précis, par exemple si une nouvelle image est importée dans le registre (via l'ImageStream). L'image est généralement générée par un "Build", configuré lui-même dans un objet "BuildConfig" lié au projet.

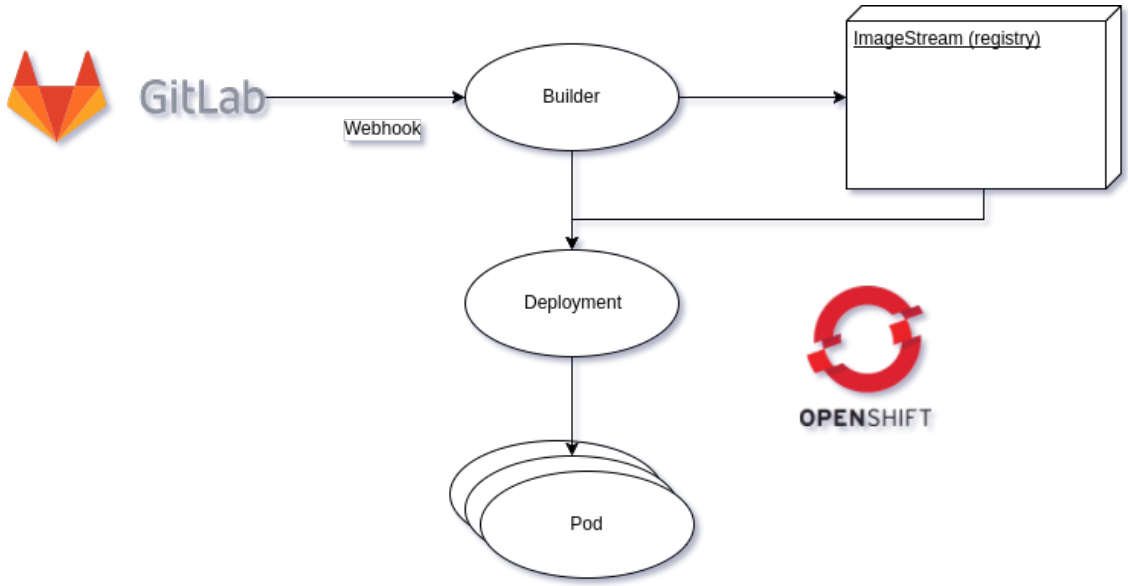



Figure 2.5 – *Principe de déploiement*

### Configuration de build - ou BuildConfig

OpenShift propose un objet de configuration nommé "BuildConfig" qui va permettre de proposer la construction d'une image qui sera utilisée par le biais d'un déploiement. Le principe est assez simple :

Il suffit de définir le point d'accès aux sources, quels évènements vont déclencher un nouveau "build" (triggers) et d'autres options qui peuvent aider au déploiement. Dans l'exemple qui suit, les triggers définis sont des webhooks (de type générique et github), un changement de configuration et un changement d'image de base.

L'image ainsi construite sera alors sauvegardée dans le registre (ImageStream).

Si l'image se base sur une image présente dans OpenShift (surcharge d'image, ou via "source-to-image") alors une mise à jour de l'image de base pourra automatiquement enclencher un nouveau "build" de l'image enfant.

Ci-dessous, un exemple de BuildConfig :

```
apiVersion: v1
kind: BuildConfig
metadata:
  annotations:
    openshift.io/generated-by: OpenShiftNewBuild
  creationTimestamp: null
  labels:
    build: app
  name: app
spec:
  output:
    to:
      kind: ImageStreamTag
      name: api:latest
  postCommit: {}
  resources: {}
  source:
    git:
      uri: ssh://git@git.domain.tld/my-client/app.git
    type: Git
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: golang-s2i:latest
        namespace: openshift
    type: Source
  triggers:
  - github:
      secret: XXXXXXXXXXXXXXXXXXXXXX
    type: GitHub
  - generic:
      secret: XXXXXXXXXXXXXXXXXXXXXX
    type: Generic
  - type: ConfigChange
  - imageChange: {}
    type: ImageChange
status:
  lastVersion: 0
```

Autre option très intéressante, les BuildConfig proposent des ***Webhooks*** [4] pour démarrer une nouvelle construction d'image via un appel HTTP. Ainsi, un outil de gestion de source tels que GitHub, GitLab ou Gogs peuvent faire appel à ce WebHook après un "merge", ou si les tests sont exécutés avec succès.

> Vous le remarquez certainement, les objets de configuration proposés par ***OpenShift*** sont très proches de ceux proposés par ***Kubernetes*** - OpenShift propose une abstraction supérieure de ces objets pour automatiser des procédures. En conjonction avec l'interface Web et l'outil ***oc*** en ligne de commande, la maîtrise d'un projet intégré dans OpenShift est assez aisée.

## Construction des images Docker

### Rappel sur l'état de l'art

La création d'image Docker est relativement simple. Elle passe par une des deux méthodes au choix :

- "commit" d'un conteneur
- utilisation d'un fichier "Dockerfile"

La première méthode n'est pas très recommandée car elle ne permet pas de modifier ou corriger la création. Elle n'est donc pas reproductible.

L'utilisation d'un Dockerfile est quant à elle bien plus évidente, car elle permet de reproduire la création d'une image, mais aussi d'y ajouter des instructions spécifiques, par exemple la directive "ONBUILD" qui indique une opération à effectuer si cette image est "dérivée".

4. un webhook est simplement une adresse Web que peut appeler une application externe pour demander une action. OpenShift propose deux webhooks, l'un compatible GitHub, et l'autre dit "générique" qui peut être utilisé par GitLab, Gogs, Jenkins...

```
# création d'une image à
# partir de alpine
FROM alpine
RUN apk add --no-cache nginx

# Cette ligne est executée lors
# d'une dérivation de l'image
ONBUILD ADD sources /var/www/html
```

Or, lorsque l'on désire utiliser une technologie de clustering telle que OpenShift, il est important de comprendre que ce sont des images qui sont instanciées en conteneurs - cela signifie que les sources de l'application à servir doivent être injectées dans l'image. Ainsi ***chaque image doit être spécifique à un projet*** . Par exemple, nous pouvons prévoir une image "Drupal" qui contient le nécessaire pour exécuter un site, mais les sources dudit site doivent être injectées dans une image à part entière. Nous devons donc prévoir une image PHP comme base de construction, puis une image "mon-site-1" dérivée qui contient les sources du projet pour le premier site.

Jusqu'ici il n'y a rien de très compliqué et la procédure de création est évidente. Mais si une mise à jour de l'image PHP est nécessaire, par exemple après un patch de sécurité, alors il faudra reconstruire l'image PHP, puis modifier toutes les images dérivées pour utiliser cette version d'image de base.

La procédure demande donc un certain nombre de contrôles et de procédures pour maintenir les applications dans une version stable et optimale.

### Source 2 Image

Ce que propose ***source to image*** [5] est de se pencher essentiellement sur les sources de l'application en évitant de maintenir manuellement des images de l'application. Une image de base est nécessaire ; elle doit être construite selon des règles précises afin d'utiliser l'outil "***s2i*** " qui permettra d'ajouter les sources de l'application (ou le binaire compilé) au-dessus de l'image de base.

Au lieu de reconstruire une image à partir d'un Dockerfile, source-to-image demande à minima de fournir deux scripts dans l'image de base :

5. https ://github.com/openshift/source-to-image

- ***assemble*** qui est un script d'assemblage. Il est exécuté au moment de la construction de l'image de l'application.
- ***run*** qui est le script de démarrage de l'application.

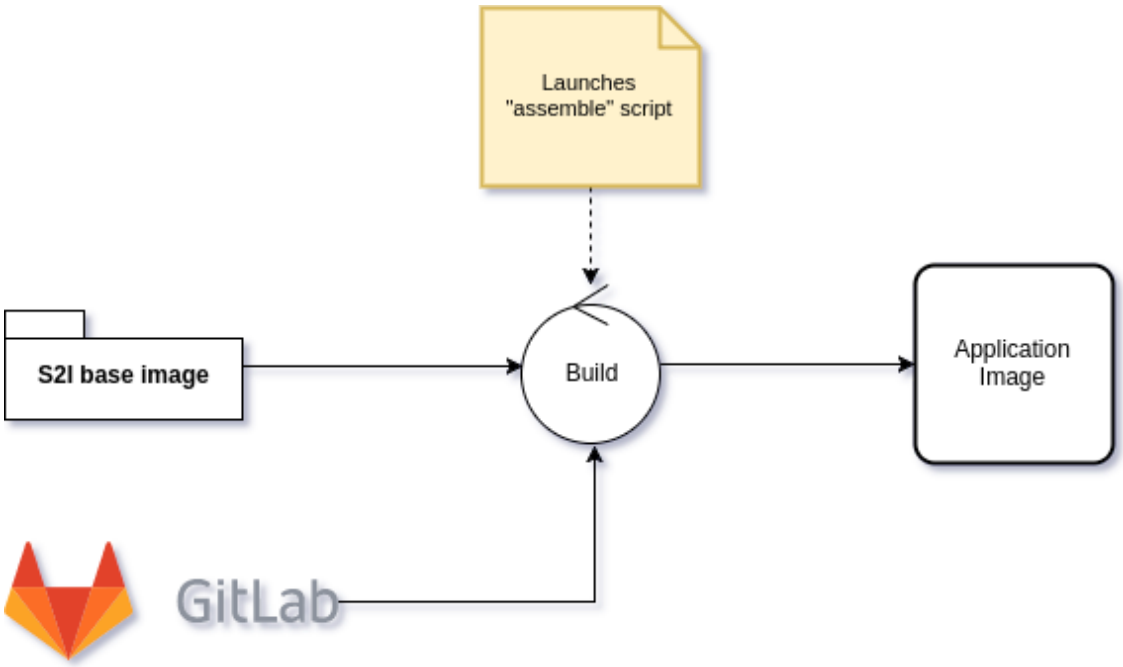



Figure 2.6 – *Logique de construction s2i*

Pour résumer :

- nous créons un Dockerfile de base, cette image contient Composer, Drupal, Apache Httpd, drush, les librairies mysql...
- dans cette image, nous plaçons un script "assemble" qui va appeler "composer" pour installer les librairies de l'application à conteneuriser
- nous y plaçons aussi un script "run" qui démarre Apache Httpd
- l'image est construite et taguée "drupal :8"

Désormais, nous créons un dépôt "site-client-1" qui va recevoir les sources spécifiques à ce client. Pour construire l'image, il suffit de demander à s2i de construire une image "site-client-1" à partir des sources Git (ou autre), et de l'image "`drupal:8`".

À ce moment précis, ***s2i*** va importer les sources du site dans l'image, appeler "assemble" pour installer les librairies nécessaires, puis construire l'image "site-client-1". Cette image, si elle est instanciée en conteneur, va alors appeler le script "run" au démarrage.

Tout autre site basé sur "drupal :8" va suivre la même procédure.

OpenShift sait utiliser s2i nativement. Ce qui veut dire que lors de la création d'un projet dans la plateforme, il est possible de définir quelle image de base est utilisée (ici, par exemple, drupal :8) et de lui indiquer le chemin des sources (svn, git, etc...). En se basant sur les labels injectés dans l'image, OpenShift va découvrir que l'image de base utilise le standard s2i et construire l'image de l'application qu'il importera dans l'ImageStream du projet.

C'est le rôle du "BuildConfig" de prendre en considération les "triggers" qui permettront de relancer une construction d'image. Il en existe deux qui sont les plus usités, à savoir l'appel d'un WebHook depuis le gestionnaire de source (gitlab, gogs, github...) ou la modification d'une image S2I injectée dans le dépôt des images OpenShift.

Si une mise à jour de l'image "`drupal:8`" est injectée dans OpenShift, il sera capable de mettre à jour tous les dérivés. Chaque image sera reconstruite et tentera une migration en demandant un nouveau "build", et par conséquent un nouveau déploiement. Bien entendu, si le déploiement est configuré pour utiliser une version figée de l'image, cela n'a pas de conséquence.

Et dans le même esprit, si les sources du site web utilisant cette image sont mises à jour, une nouvelle image de l'application utilisant "`drupal:8`" sera construite et injecté dans le registre privé, puis un nouveau déploiement sera activé.

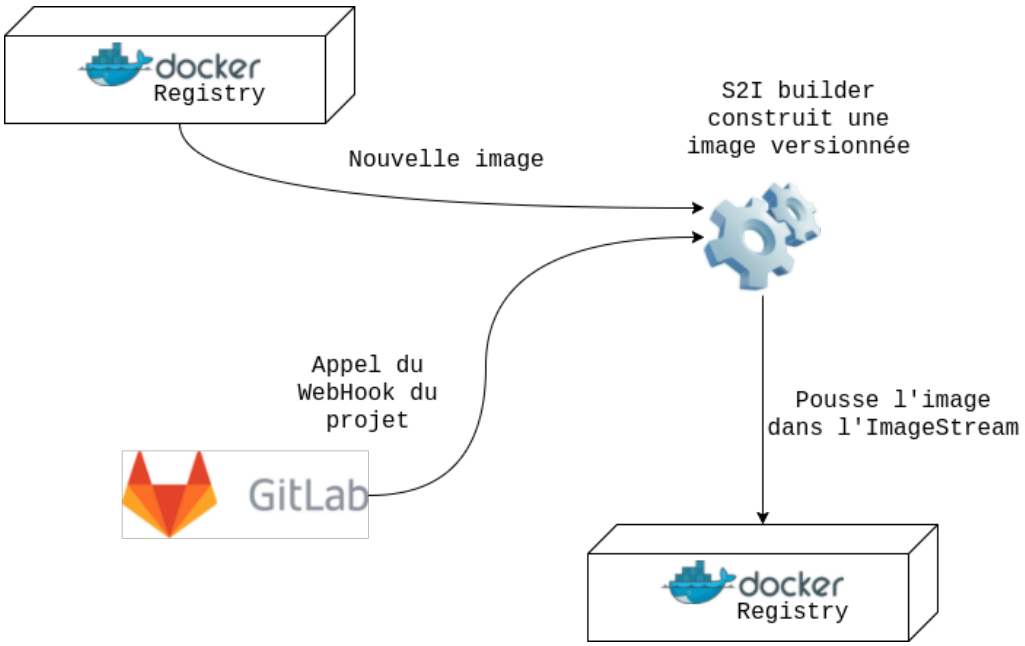



Figure 2.7 – *La modification d'image Docker ou du code source de l'application enclenche une nouvelle construction*

Source-2-image permet d'autant plus d'accélérer le traitement de création d'image, et par la même d'ajouter une logique de mise à niveau (lancer des scripts, injecter des données, etc...) ainsi que de mettre en cache des données lourdes avant traitement de l'assemblage. En d'autres termes, s2i soulage fortement la gestion d'images et de création de descendants. C'est un compagnon extrêmement puissant à OpenShift mais qui peut tout à fait être utilisé dans d'autres chaînes de mise en production, à condition bien entendu de gérer ces procédures soi-même, procédures déjà automatisées dans OpenShift.

La plateforme OpenShift présente également l'avantage de proposer des images de base Source-2-image sur étagère, prêtes à l'emploi et ce pour de nombreuses technologies standards (PHP, Java, NodeJS, Python, Ruby, etc.). Ces images de base concentrent déjà les meilleures pratiques de conteneurisation de ces technologies. Elles vous permettent dans de nombreux cas de ne plus vous préoccuper des problématiques de conteneurisation et représentent donc un accélérateur important. Il vous suffit de déposer votre source dans un Git et le tour est joué !

> ***source 2 image*** est un projet open source de Red Hat qui va fortement augmenter la réutilisabilité de vos images Docker. Elles sont utilisables sur un poste de développement, en conjonction avec Docker-Compose et quelques règles. Ainsi, en créant une base de départ pour créer un projet, les développeurs travaillent dans un environnement ***quasi-iso-prod***

## Rappel sur le principe de "Service" et route

Si vous n'avez pas eut l'occasion de travailler avec Kubernetes, il faut bien garder à l'esprit que OpenShift utilise des concepts fortement ancrés dans cet orchestrateur. Et notamment en le principe de routage. OpenShift redéfini quelque peu certains concepts tout en respectant le principe de base de Kubernetes.

En effet, les principes d'Ingress et Ingress Controller se retrouvent dans l'objet "Route" de OpenShift. Il simplifie la gestion de routage d'un nom de domaine vers un service.

Attention toutefois à la terminilogie Kubernetes/OpenShift. ***Un service est un objet Kubernetes qui permet de router des requêtes vers des Pods*** , ce n'est donc pas le "service à proprement parler" dans les termes "applicatifs". Ainsi, si vous déployez un "serveur

nginx", le service est une objet de configuration qui permet de vous y connecter depuis d'autres Pods, ou depuis l'extérieur en fonction de la configuration apportée [6]

Afin d'exposer un service avec un nom de domaine, il convient de créer un objet "route" qui défini un nom d'hote et une spécification de sélection de service.

Le diagramme 2.8 permet de mieux comprendre le cheminement des connexions entre routes, services et pods.

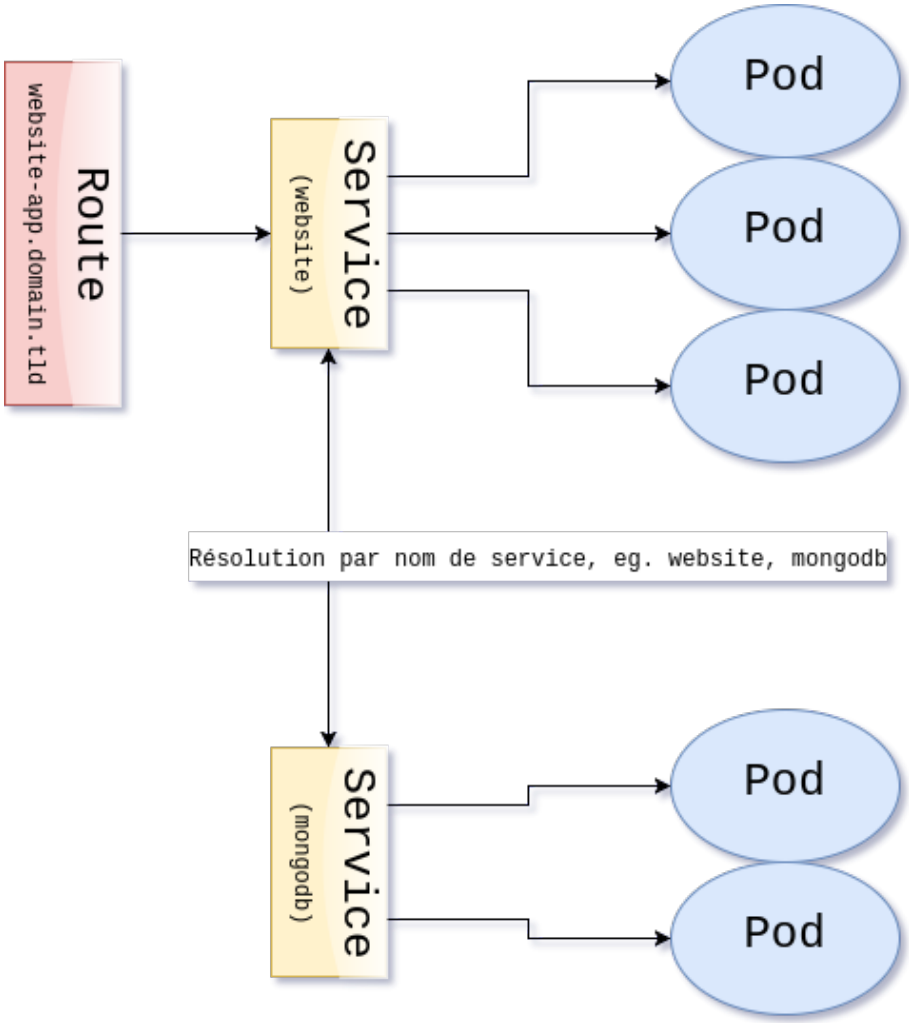



Figure 2.8 – *Le principe de routage, de service et de pods rattachés. Les services servent en interne, quand une route sert en externe.*

6. car un service peut se voir attribuer un port sur tous les noeuds, si le type de service est paramétré en NodePort. Le cas échéant, c'est le type ClusterIP qui est utilisé

Les routes sont en fait gérées par des objets "routeurs" qui sont des serveurs HAProxy déployés sur un ou plusieurs noeuds. Cela permet la répartition de charge depuis un proxy externe et d'assurer une continuité de production.

En règle général :

- un service permet d'accéder à des Pods applicatif, par exemple un service nommé "website" va se connecter aux Pods où tourne nginx, php-fpm, java...
- les Pods peuvent contacter les autres service par leur nom, donc une application PHP peut contacter les Pods MariaDB qui sont servis par un service nommé "database", PHP utilisera alors le nom d'hôte "database" pour s'y connecter
- une route est créé pour permettre à des utilisateurs ou des services externes à OpenShift de se connecter aux application. Par exemple, la route "website.domain.tld" permet à un visiteur de se connecter aux Pods "php" via le service "website"

Toute la configuration s'effectue via des fichier YAML, mais aussi par de simples commandes "oc" qui vont créer la configuration associé. Par exemple :

```
oc expose svc/website --hostname=website-app.domain.tld
```

# Didacticiel : Déploiement de ShareLaTeX avec OpenShift

OpenShift est un excellent outil pour développer des applications dans une démarche DevOps / Continuous Delivery.

Mais il faut aussi insister sur le fait que ***OpenShift permet de servir les applications*** . En d'autres termes, OpenShift est aussi un service de "hosting" qui propose de router des URL vers les services associés.

Chez Smile par exemple, certains outils que nous utilisons sont intégrés dans notre cluster OpenShift. Par exemple, l'outil utilisé pour écrire ce livre se nomme ***ShareLaTeX*** [1]. Il est servi en interne pour rédiger nos livres blancs.

Nous avons besoin de proposer un certain nombre de packages TeX pour le design de ce le livre. Ainsi, nous avons simplement surchargé l'image de base proposée par l'éditeur de la solution pour forcer l'installation de la suite LaTeX entière [2] :

1. https ://fr.sharelatex.com/
2. l'exemple est simplifié, nous avons en réalité eu besoin de manipuler un peu plus notre image pour permettre à certains paquets de fonctionner comme nous le souhaitons, tels que "minted" avec "python-pygmentize", ajouter un logo, changer le template d'affichage, etc.

```
# On part de l'image ShareLaTeX officielle
FROM sharelatex/sharelatex

# Utilisation d'un dépôt proche
ENV TLMGR_REPO http://ftp.oleane.net/pub/CTAN/systems/texlive/tlnet

# Installation de paquets nécessaires
RUN set -xe; \
    apt-get update; \
    apt-get install -y xzdec; \
    rm -rf /var/cache/apt/*; \
    tlmgr init-usertree || :; \
    tlmgr repository remove main; \
    tlmgr repository add $TLMGR_REPO main; \
    tlmgr update --all; \
    tlmgr install scheme-full --no-persistent-downloads || : ;
```

Ce Dockerfile se trouve dans notre GitLab, nous pouvons maintenant créer un projet OpenShift. Dans cet exemple, nous voulons proposer l'image ShareLaTeX par défaut dans le catalogue, donc nous construisons l'image dans le "namespace" (ou "projet") nommé "openshift"

Il est important de créer ces configurations de construction dans le projet "openshift" car c'est l'espace de nom par défaut pour que les déploiements trouvent les images "internes" à la plateforme. Ainsi, l'interface peut lister les images dans le catalogue proposé au développeur.

```
oc project openshift
oc new-build https://git.smile.local/openshift/sharelatex
```

Cette commande crée un "BuildConfig" dans l'espace de nom "openshift" qui est le projet par défaut pour proposer des images, des templates, etc. Lorsque nous mettrons à jour le Dockerfile dans Git, nous pourrons demander une nouvelle construction d'image en utilisant la commande :

```
oc project openshift
oc start-build sharelatex
```

La nouvelle image entrera dans le registre interne, et ***tous les projets utilisant l'image précédente seront mis à jour*** . Donc, si vous avez plusieurs projets ShareLaTeX, les services vont automatiquement utiliser la nouvelle image pour reconstruire une image d'application qui sera instanciée. À moins, évidement, que vous ayez figé la version d'image à utiliser dans votre projet.

Après un certain temps de compilation, l'image est intégrée dans OpenShift.

Ensuite, un projet "ShareLaTeX" est créé.

ShareLaTeX ayant besoin de Redis et MongoDB, nous avons créé deux services comme demandé : un service utilisant une image MongoDB et un service Redis. Lors de la création d'un service MongoDB, il nous est demandé de donner un login et un mot de passe que nous notons. Cela est nécessaire pour pouvoir les fournir au service ShareLaTeX afin qu'il puisse s'y connecter.

Notez aussi que les adresses à fournir à ShareLaTeX sont simplement "mongo" et "redis", qui sont les noms que nous avons donné aux services (et non les noms des images). La résolution de nom sera faite par le DNS interne de OpenShift.

Enfin, nous y ajoutons une application ShareLaTeX à partir de l'image que nous avons créée.

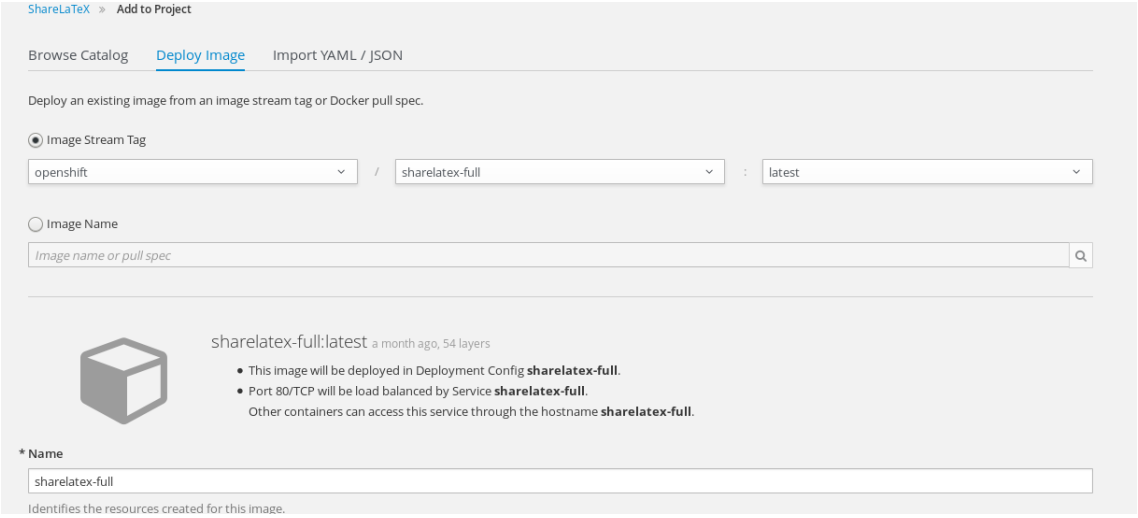



Figure *3.1 – Ajout de l'application ShareLaTeX*

Nous avons simplement ajouté les variables d'environnement décrites dans la documentation de ShareLaTeX pour fournir :

- la configuration SMTP
- le nom de notre service
- les login et mot de passe mongo, ainsi que l'url d'accès à la base
- idem pour la base Redis

Après validation, notre application démarre et nous avons accès à notre shareLaTeX via une adresse exposée du service.

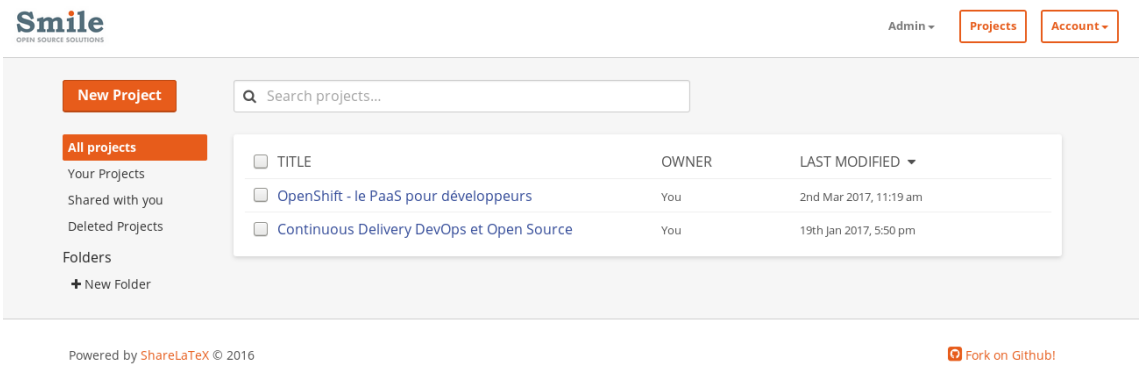



Figure 3.2 – *L'application ShareLaTeX déployée sur OpenShift*

En d'autres termes, mise à part la construction de l'image qui a demandé un peu de temps (le téléchargement de l'ensemble des librairies LaTeX est très conséquent), l'installation d'une application ShareLaTeX n'a pris que quelques minutes. Le temps de déployer la base de données et le service Redis en quelques clics.

OpenShift révèle son potentiel en terme de "PaaS" sitôt que vous avez essayé une première fois de démarrer une application.
L'interface prouve sa valeur en proposant ***un design épuré*** , des explications simples et des formulaires très clairs. OpenShift, outre sa force dans un ***environnement CI/CD*** peut aussi être un outil très pratique pour déployer toutes sortes de services, tels que Redmine, Jenkins, Nuxeo, etc.
Du moment que vous avez une image Docker accessible, tout est possible !

## Didacticiel : Template d'applications

L'installation d'une simple application telle que ShareLaTeX se base sur une image existante et que nous ne modifions pas. Mais pour poursuivre la possibilité de créer une application à partir des sources d'un projet, nous pouvons désormais utiliser un template d'application.

Nous allons voir la capacité de source-2-image à gérer la problématique de création d'image d'application à partir des sources et permettre à un développeur d'initialiser un projet sur OpenShift à partir de ses propres sources.

Ici, nous allons utiliser l'image `php:5.6` proposée par défaut dans le projet "openshift". Cette image permet d'installer l'application php de notre choix et d'exécuter des actions à l'assemblage de l'image. Nous allons donc admettre que nous avons des sources PHP dans un dépôt `https://git.domain.tld/example/php-app`

La création d'un template peut se faire de différentes façons, soit en partant d'un projet existant et en utilisant la commande `oc export`, soit en le créant manuellement.

Le principe d'un template est relativement simple à appréhender, il suffit de créer un objet "Template" et de définir la liste d'objets qu'il sera capable de créer. De plus, nous allons y intégrer des variables dont la syntaxe est `${NOM_VARIABLE}` dont les valeurs sont définies dans la section "parameters" tout en bas du fichier.

Le fichier de template ci-dessous permet de créer un objet de "BuildConfig" et un "DeployementConfig", tous les deux liés par un évènement permettant l'automatisation. Il propose aussi à l'utilisateur de donner un chemin de sources (via Git) qui seront intégrées dans l'image générée.

```
kind: Template
apiVersion: v1
labels:
  template: simple-php
metadata:
  annotations:
```

```
    description: An example from Smile
    iconClass: icon-php
    tags: quickstart,php,simple
  name: smile-php-example
objects:
  # Build Image Configuration, that will create an image
  # named smile-php-example
  - kind: BuildConfig
    apiVersion: v1
    metadata:
      name: smile-php-example
      annotations:
        description: Defines how to build the application
    spec:
      source:
        type: Git
        git:
          uri: "${SOURCE_REPOSITORY_URL}"
          ref: "${SOURCE_REPOSITORY_REF}"
      output:
        to:
          kind: ImageStreamTag
          name: smile-php-example:latest
      strategy:
        sourceStrategy:
          from:
            kind: ImageStreamTag
            name: php:5.6
            namespace: openshift
        type: Source
      triggers:
      - imageChange: {}
        type: ImageChange
      - type: ConfigChange
  # Deployment configuration, the build config
  # will trigger an ImageChange event. So the Deployment will
  # be triggered
  - kind: DeploymentConfig
    apiVersion: v1
    metadata:
```

```
      name: ${NAME}
    spec:
      replicas: "${REPLICA_COUNT}"
      selector:
        name: ${NAME}
      template:
        metadata:
          labels:
            name: ${NAME}
        name: ${NAME}
        spec:
          containers:
            - name: smile-php-example
              image: ' '
              ports:
                - containerPort: 8080
    triggers:
      - imageChangeParams:
          automatic: true
          containerNames:
          - smile-php-example
          from:
            kind: ImageStreamTag
            name: ${NAME}:latest
      - type: ImageChange
      - type: ConfigChange

# Parameters that users could/should fill
# in the provided form or via "oc process" command.
parameters:
  - description: |
        The name assigned to all of the frontend objects
        defined in this template.
    displayName: Name
    name: NAME
    required: true
    value: website
  - name: SOURCE_REPOSITORY_URL
    displayName: Source Repository URL
    description: |
```

```
      The URL of the repository with your application
      source code
  value: ""
  required: true
- name: GITHUB_WEBHOOK_SECRET
  description: |
      A secret string used to configure
      the GitHub webhook
  generate: expression
  from: "[a-zA-Z0-9]{40}"
- name: REPLICA_COUNT
  description: Number of replicas to run
  value: "1"
  required: true
```

Il suffit maintenant d'enregistrer ce template dans l'espace de nom "openshift" :

```
oc create -f template.yaml
```

Ce template apparaît dans la liste des applications :


Figure *4.1 – Le template est désormais injecté dans la liste des applications*

En cliquant sur ce projet, le formulaire adapté à notre template apparaît. L'utilisateur va alors pouvoir configurer l'application avec ses préférences, en entrant l'adresse d'accès aux sources Git de son application.

L'interface Web n'est pas obligatoire, la commande `"oc process ..."` permet de procéder à l'exécution du template en fournissant les valeurs à utiliser en paramètre. Par conséquent, un template peut tout aussi bien être utilisé dans une procédure automatisée, depuis un terminal, etc.

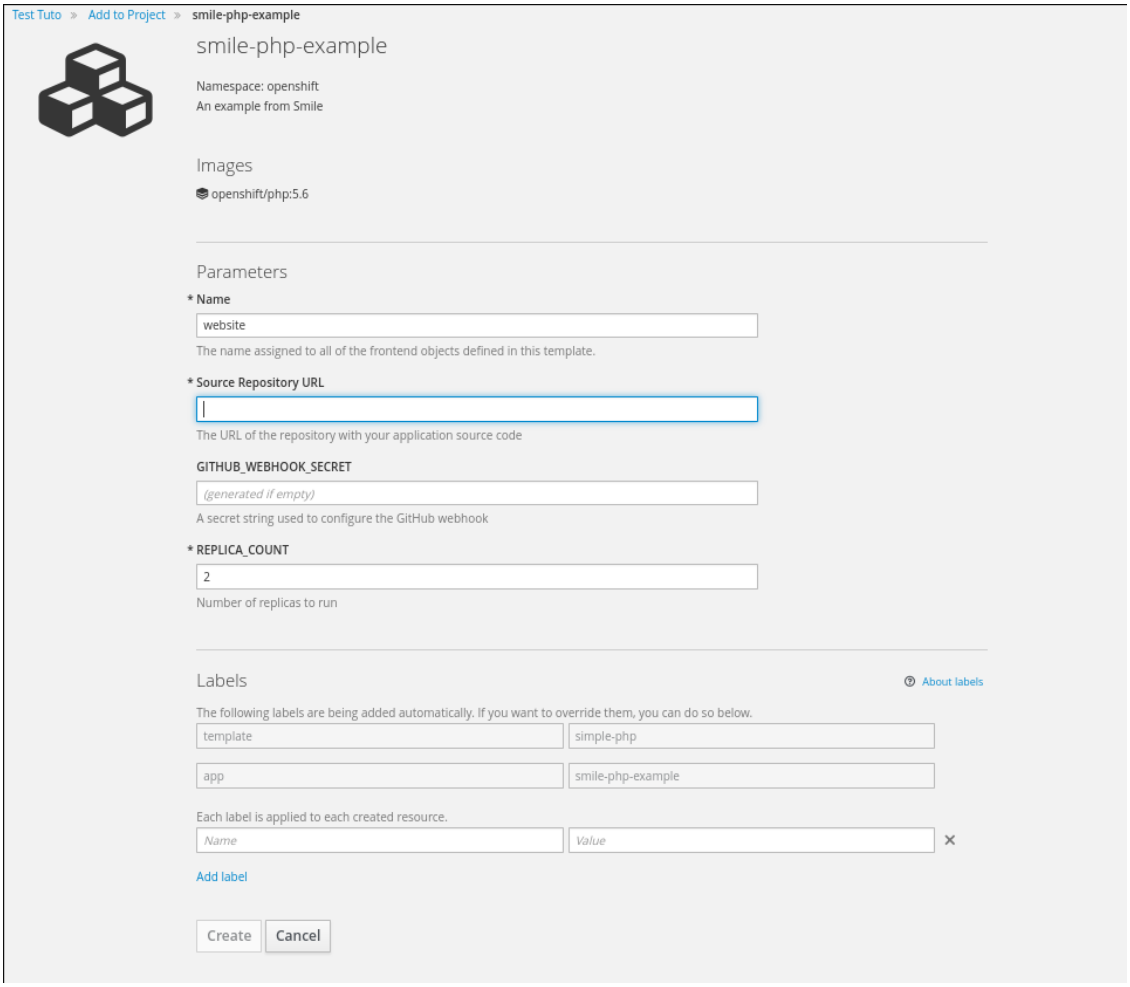



Figure 4.2 – *Le formulaire associé au template est généré par l'interface web*

Bien entendu, ce template n'intègre qu'une image PHP. Il est possible d'ajouter autant d'objets que nécessaire tels que des bases de données, des services de bus, de messagerie, ajouter les routeurs, etc.

Certes, la création d'un template demande un peu de travail, de la recherche et des tests. Mais ce travail fait, les intégrations répétitives de projets se transforment en une simple formalité pour un chef de projet ou un développeur.

Les templates deviennent naturellement la voie à suivre pour simplifier la création de projets complets au sein des équipes. La démarche ressemble beaucoup à l'utilisation de docker-compose - déjà très établie dans les équipes ayant attrait au ***DevOps*** .
Cette approche basée sur l'interaction avec l'utilisateur pour créer un projet fait d'OpenShift l'un des PaaS les plus abouti et les plus adaptés aux développeurs

# Retours d'expérience Smile

Au sein de notre infrastructure, Smile a intégré un cluster de plusieurs machines sur lesquelles la plateforme OpenShift a été déployée. Elle nous permet de proposer aux développeurs un environnement d'intégration et de livraison continue, de tester la scalabilité, de proposer un accès aux APIs ou services Web en test à nos clients, et bien plus encore.

Nous allons décrire ci-après les travaux effectués pour réaliser un projet destiné à l'un de nos clients, en mode 'DevOps' et accompagner les développeurs avec l'outillage adapté, en l'occurence avec OpenShift !

Sans entrer dans les fonctionnalités du projet qui reste confidentiel, le projet se compose techniquement de deux éléments centraux :

- une API REST développée en ***Go***
- une application mobile développée en ***TypeScript*** via ***Ionic2***

L'application interroge l'API qui est connectée à deux bases de données :

- une base de données MariaDB, base technique qui fournit les données à restituer via l'API
- une base de données MongoDB qui stocke des informations statistiques

## Création des images S2I

En premier lieu, nous avions besoin d'avoir des images "builder" pour les deux ***types d'application*** . Ces deux images sont désormais réutilisables pour d'autres projets.

La premier est une image pour Go. Le script d'assemblage réalise deux opérations. D'abord, il récupère les librairies nécessaires à la compilation. Ensuite, elle effectue une compilation paramétrée par des variables d'environnements.

Il est tout à fait possible d'effectuer une exécution de tests unitaires dans le script "assemble", mais nous préférons utiliser Gitlab-CI.

Le script "run" fait simplement appel au binaire généré (eg. `/opt/api.bin`).

En ce qui concerne l'image APK, elle contient Java, Node et une version figée du SDK Android. L'installation de Ionic est effectuée dans l'image S2I afin de figer la version du framework utilisé.

Le script "assemble" effectue ces opérations : il construit la version navigateur, puis l'APK.

Le script "run" va simplement démarrer la commande :

```
$ python -mSimpleHTTPServer
```

Et ce dans le répertoire de l'application. Le "product owner" peut donc voir une version "navigateur" de l'application (ce qui permet de valider des éléments graphiques ou quelques actions, etc.) et peut aussi accéder à l'APK générée. Cet APK est considéré comme un "livrable" de l'application mobile.

NB : Le product owner, ou le client final, a aussi accès aux sources dans GitLab. Il peut à tout moment décider de construire lui-même l'APK, l'API ou utiliser les images Docker S2I pour travailler ou tester de son coté, simplement en appliquant un tag ou en demandant à Gitlab-CI une nouvelle exécution des pipelines.

La gestion "preprod" et "staging" sera discutée plus loin dans ce chapitre.

### Projet Git et postes de développeurs

Les deux applications (mobile et API) se trouvent dans deux projets Gitlab distincts, classés dans une organistation portant le nom de notre client.

Chaque développeur "forke" le projet pour effectuer des pull-requests en temps et en heure.

Une instance ***Gitlab-CI Runner*** tourne sur un projet "infra" dans OpenShift ; ainsi, les demandes de fusion sont testées dans ce conteneur distant. Le paramétrage des tests est défini dans un fichier `.gitlab-ci.yaml` à la racine des projets.

En fonction du projet, la définition demande l'exécution d'un jeu de tests dans un environnement précis.

Si les tests passent, alors le "merge" est autorisé dans la branche "devel" ***et le webhook de build est appelé*** . Ainsi, ***à chaque merge, une nouvelle version de l'API ou de l'APK est générée et servie par OpenShift*** .

À tout moment, un merge réussi enclenche une construction d'image versionnée dans l'ImageStream du projet en question et son déploiement automatique.

Les développeurs ont accès aux images S2I qui sont les mêmes que celle utilisées dans OpenShift. Pour chaque projet, un ficher docker-compose.yml est fourni. Nous avons construit les images de telle manière que lorsqu'elles sont utilisées localement par un développeur, alors un mode "développeur" est utilisé.

Pour l'image Go, c'est le projet ***fresh*** [1] qui permet une reconstruction automatique des sources à chaque modification (Golang permet une compilation rapide ce qui réduit fortement l'impact ressenti par un développeur).

Et en ce qui concerne l'image Ionic, c'est le mode "server" qui est démarré.

Cela a été possible en configurant les fichiers de Docker-Compose de telle sorte que les sources soient montées localement dans les conteneurs, et en modifiant le point d'entrée et la commande utilisé. Une routine pour gérer les problématiques de droits utilisateur (lecture/écriture) a aussi dû être imaginée.

Les développeurs sont donc à même de développer dans un environnement technique très proche de la réalité de production.

### Projet OpenShift

La création d'un projet OpenShift s'est déroulée en plusieurs étapes.

D'abord, la création des deux bases de données avec un stockage persistant. Les variables d'environnement pour les utilisateurs et mots de passe de bases de données sont enregistrées dans un objet "secret" qui peut être lu par différentes instances. De cette manière, les conteneurs MariaDB et API utilisent la même référence secrète et aucun mot de passe n'apparaît dans les fichiers de configuration (DeploymentConfig).

Les deux applications (Go et Ionic) sont démarrées en utilisant respectivement les images S2I produites auparavant, et en indiquant pour chacune d'elles l'emplacement dans Gitlab et quelle branche utiliser (ici, la branche devel).

Après enregistrement, OpenShift initie la première version des applications en clônant le projet, puis en utilisant S2I pour produire l'image. Il l'enregistre dans le registre de projet (ImageStream) et enclenche par la suite un déploiement.

Les deux applications sont aussi "exposées" par un routeur chacune afin de permettre l'accès aux services liés.

1. https ://github.com/pilu/fresh

### Pré-production

Par le fait que nous réalisons souvent des merges dans la branche devel, il y a par conséquent un nombre important de déploiements réguliers. Or, notre client demande une version "stabilisée" pour effectuer des tests sur une période plus longue.

C'est ainsi que nous avons simplement reproduit l'opération de création des applications. C'est-à-dire la création d'une nouvelle base MariaDB, d'une nouvelle base MongoDB, et des deux images d'application API et APK. ***À la différence près que nous demandons cette fois-ci au BuildConfig de prendre la branche "master".*** - cette différence est notable, car les merges dans la branche master sont effectués moins fréquement. En réalité, c'est le chef de projet qui indique quelle version en "devel" est prête à être démontrée au product owner.

C'est généralement en fin de sprint que cela est fait.

### Architecture

Pour simplifier l'illustration, nous omettons volontairement dans le diagramme ci-dessous la duplication des applications et bases de données.

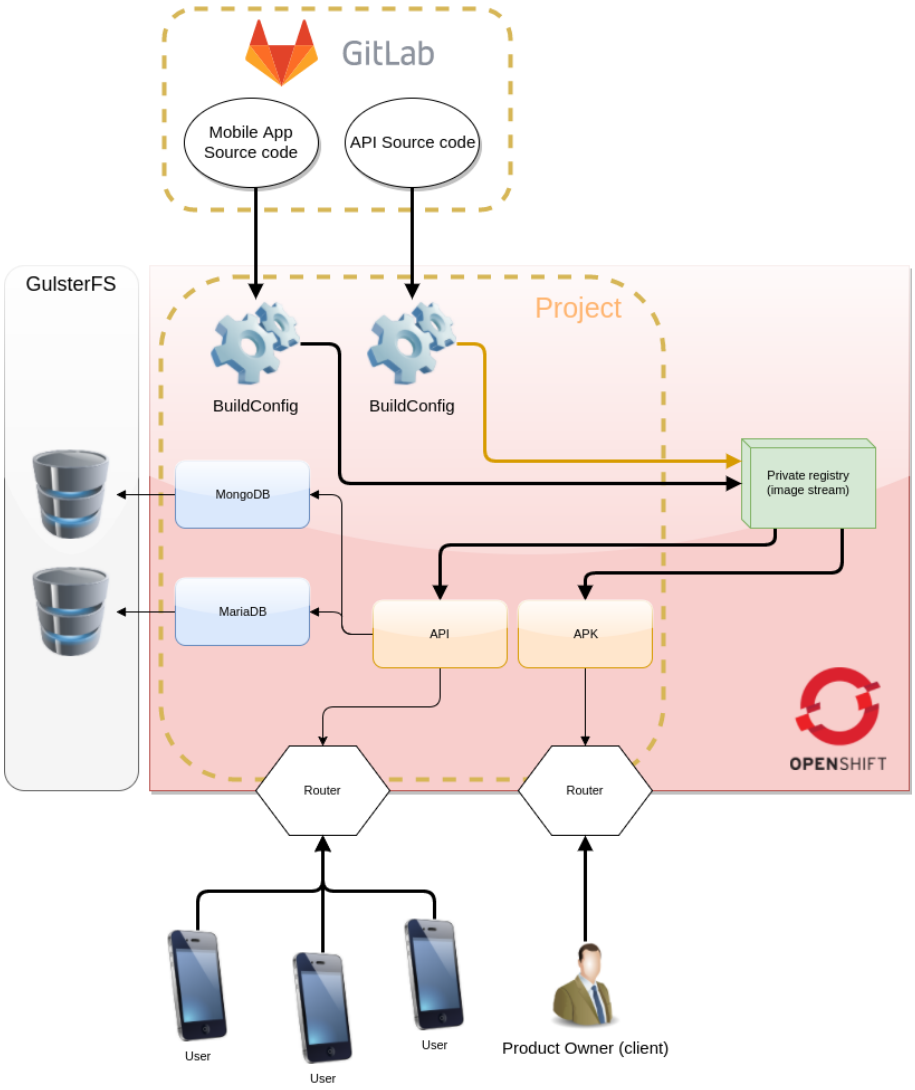



Figure 5.1 – *Architecture du projet API/APK*

Notez bien ici que le bloc "APK" dans la figure 5.1 est un simple serveur web minimaliste qui fournit le fichier au "product owner" en plus d'une vue "Web" de l'application - cela est possible par le fait que l'application est développée sur le framework ***Ionic2*** [2] qui permet de construire deux versions de l'application. Dans un autre processus de déploiement, il est possible par exemple de déployer une vue Web pour proposer une interface au client, et de déposer l'APK sur un espace de partage. Or, dans notre cas, ce n'était pas nécessaire.

Ce qu'il faut retenir ici, c'est de remarquer à quel point les procédures et l'architecture

2. http ://ionicframework.com/

sont liées, mais aussi à quel point elles sont adaptables aux besoins, aux clients ou aux contraintes spécifiques qui peuvent apparaître lors d'un développement de projet. L'architecture présentée ci-dessus est un exemple d'adaptation proposée et acceptée pour des raisons pratiques.

### Production

Dans ce cas précis, bien que OpenShift puisse aussi héberger l'application en production, notre client désire s'occuper de l'exploitation des applications dans une infrastructure Azure, le cloud public de Microsoft. Notez que le cloud Azure peut tout à fait démarrer des images Docker.

Lorsque le client est satisfait d'une version de préprod, on lui fournit l'image versionnée dans l'ImageStream qui correspond à sa demande. Le client n'a alors plus qu'à déployer cette image en paramétrant les variables d'environnement pour se connecter à ses propres bases de données (qu'il ne désire pas conteneuriser pour le moment).

En d'autres termes, l'opération de "livraison au client" se fait de la même manière que pour tout autre artefacts dans d'autres technologies. Ce n'est qu'un livrable, au format "tar".

Il aurait aussi été possible de déployer l'image sur un serveur de production externe (sur AWS via ECS, sur Google Compute Engine, ou même sur un Kubernetes déployé à l'extérieur).

### Bilan

L'intégration a clairement su prouver l'intérêt de OpenShift au sein de nos équipes.

D'abord par l'apport très important de retour visuel sur l'intégration elle-même, que ce soit le démarrage des API ou la compilation de l'APK, mais aussi sur les ressources utilisées, la portée sur le CPU et la mémoire.

OpenShift a aussi apporté un confort non négligeable pour les développeurs en leur apportant l'automatisation de déploiement en fonction de leur étiquetage (tag).

### Interview de Sam, développeur dans l'équipe projet Smile

Sam est un développeur au sein de notre équipe, et ayant participé au sein de Smile à un projet de développement applicatif utilisant OpenShift.

Pour rappel, le projet se base sur plusieurs technologies telles qu'une API en Go, une application mobile Ionic 2, et des services nécessaires au bon fonctionnement comme Redis, MongoDB et MariaDB. Le projet nécessite deux versions : une version accessible en interne pour les développeurs qui désirent tester la scalabilité et les derniers développements, et une version "pré-prod" qui permet de figer l'application à livrer. Voir la figure 5.1 (p. 40) qui illustre l'architecture.

**Depuis quand utilises-tu OpenShift ? Dans quel cadre ?**

*Sam* : Cela fait 6 mois. Nous avions un projet qui demandait à livrer rapidement une API REST développée en Go, de manière continue, ainsi qu'une application mobile développée avec Ionic v2. L'équipe de développement était divisée en deux groupes : un groupe s'occupait de l'API en Go que devait interroger l'application Ionic 2 développé par la seconde équipe.

**Connaissais-tu OpenShift avant ?**

*Sam* : Non, j'en avais entendu parler, mais sans plus. Je ne savais pas exactement ce que ça proposait mis à part le fait que cela pouvait servir à démarrer des applications, de scaler, etc.

**Quels sont les autres outils qui ont été utilisés ?**

*Sam* : Nous utilisons GitLab. Par conséquent Git pour la gestion des sources. Par contre, ce qui a été nouveau pour moi, c'est d'utiliser Docker. À l'instar d'OpenShift, je ne connaissais pas plus que ça cette technologie.

**Comment utilises-tu OpenShift ? Docker ? et Gitlab ?**

*Sam* : En fait ce fut très simple. Le support DevOps de Smile nous a fourni un template de projet contenant un Makefile et des configurations Docker-Compose. Localement, nous n'avions rien à installer de plus que Docker et Docker-Compose.

Nous travaillons chacun sur un "fork" du projet principal, et c'est au moment des "merge requests" que tout se passe. Lors d'une merge-request sur la branche devel, une série de tests unitaires est lancée automatiquement sur GitLab-CI [3] qui tourne dans l'environnement OpenShift. Selon le résultat, en cas de succès, la requête est acceptée et le code est

3. https ://about.gitlab.com/gitlab-ci/

mergé. Sinon un message d'erreur est envoyé par mail nous indiquant le problème. On peut alors vérifier les logs dans GitLab.

À ce moment, OpenShift construit une nouvelle image du projet et sert la version fraîchement compilée. Aucune intervention n'est nécessaire de la part des développeurs et c'est un vrai confort.

Ce qui est intéressant, c'est que OpenShift nous montre les logs de construction de l'image mais aussi ceux du "runtime". On peut donc analyser ce qu'il se passe sur l'application au démarrage et tout au long du fonctionnement. Associé aux graphs qui montre la charge mémoire, CPU, etc. C'est réellement visuel et adapté à un non-Ops.

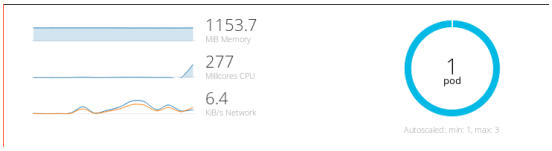



Figure 5.2 – *Graphique de charge du conteneur*

Enfin, quand nous voulons une version livrable, nous appliquons un tag de la version testée, cela enclenche un nouveau build, qui est servi sur une URL distincte. Le tout au sein du même projet OpenShift.

**Quels sont les points qui ont posé le plus de soucis et ceux qui ont donné satisfaction ?**

*Sam* : Le plus difficile a été de comprendre la logique de déploiement (même si elle est automatisée). Cela peut paraître étrange mais le fait de ne pas demander d'action de notre part pour lancer une batterie de tests et déployer une version dans une certaine configuration demande un effort de compréhension. Cela-dit, la logique de cette méthodologie est très vite assimilée.

N'ayant pas eu beaucoup d'expérience avec Docker, les premiers jours m'ont demandé un peu de réflexion et des explications. L'accompagnement d'un ingénieur DevOps est indispensable. En plus de la méthodologie et des modèles à utiliser, c'est lui qui doit amener les informations sur les technologies utilisées et modifier ou corriger ce qui pouvait nous bloquer.

Par contre, j'ai été impressionné par la vitesse à laquelle nous pouvions travailler dans cet environnement. Le fait d'avoir deux environnements iso entre notre poste de développement et ce qui tourne dans OpenShift est un véritable gain de temps.

Et le fait de pouvoir scaler à la volée une application pour tester sa réaction, créer une nouvelle instance pour tester une fonctionnalité développée, l'affichage des métriques pour vérifier que tout se passe bien… et l'accès à un terminal du conteneur directement dans l'interface OpenShift. ***Je ne peux pas lister tout ce qui nous a épaté et rendu service !***

## Ce qu'il faut en retenir

OpenShift est une plateforme PaaS complète qui aide grandement à la phase de Continuous Delivery, mais aussi à aider les équipes à mettre rapidement en place un projet.

Les templates d'applications générant un formulaire permettent de créer l'environnement technique et une liste de services interconnectés sans avoir à taper la moindre ligne de code. La création de POC, de démonstration de produit ou simplement un test de service est absolument possible au sein d'un projet.

L'interaction avec une plateforme d'intégration continue (ou CI) telles que Jenkins ou Gitlab-CI est facilitée via des Webhooks intégrées au "BuildConfig" de OpenShift, ce qui va éviter de devoir maintenir des scripts spécifiques à la plateforme. Et en second lieu l'interface très pratique que propose OpenShift. En effet, même sans connaissance technique préalable, il n'est pas difficile de démarrer une instance d'application, de répliquer et de lire les "metrics" sur l'interface web.

L'installation est fortement simplifiée par des playbooks Ansible [1]. À noter toutefois que selon la cible d'installation du cluster, et selon les spécificités demandées, la gestion de droits à intégrer, ou encore les options de sécurité imposées, il est nécessaire d'avoir une bonne expérience en la matière pour les administrateurs système.

Quoiqu'il en soit, ***OpenShift propose une véritable plateforme de conteneur efficace*** . Les développeurs peuvent contrôler le fonctionnement de leurs projets dans un environnement qualifié et cela leur apporte une vision claire de l'état de l'application à délivrer. Encore une fois, cela est possible grâce à une démarche ***DevOps*** qu'il faut appréhender et à un travail de préparation d'intégration optimisé pour que l'industrialisation soit efficace.

N'oublions pas non plus qu'OpenShift n'est pas seulement une plateforme de rendu d'intégration continu, il est aussi un excellent PaaS de production prévu pour l'hébergement d'applications.

Du point de vue de l'Ops, OpenShift permet de maîtriser des points essentiels de la vie d'une application comme le ***scaling automatique*** , le ***monitoring*** ou encore le schedu-

1. https ://github.com/openshift/openshift-ansible

ling, la résilience... tout cela en plus encore grâce à un outil en ligne de commande très complet, des addons efficaces, une API REST très complète, le tout configurable via des fichiers YAML homogènes.

> ***En d'autres termes OpenShift fait la jonction "Dev-Ops" de manière élégante et efficace.***

Red Hat a permis la flexibilité avancée de Kubernetes et un rapprochement aisé d'un projet intégré sur un poste de développeur et son fonctionnement sur une infrastructure. L'automatisation apportée par des connecteurs WebHook et l'utilisation de "source 2 image" accélère les process perçus habituellement comme des freins (bootstrap de projet, mise en production, contrôle...) - ainsi, OpenShift est un outil DevOps ***dans la lignée du Continuous Delivery*** qui peut éliminer un grand nombre de complexités connues.

C'est pour cela que Smile propose un accompagnement tout au long du processus d'intégration à la plateforme OpenShift. De l'installation du cluster à son paramétrage, et ce jusqu'à la formation des équipes de développeurs, en passant par la méthodologie DevOps.

> Un projet ? Une question ? Une seule adresse : contact@smile.fr