PROGRAMMEZ!
Le magazine des développeurs
AUTOMNE 2021

Compiler. Déployer partout.

Red Hat Developer

Java, Quarkus, NoSQL, API, Kubernetes : du code, du code et du code !

Le seul magazine écrit par et pour les développeurs

M 01642 - 5H - F: 6,99 € - RD

Printed in EU - Imprimé en UE - BELGIQUE 7,50 € - Canada 10,55 $ CAN - SUISSE 14,10 FS - DOM Surf 8,10 € - TOM 1100 XPF - MAROC 59 DH

DISPONIBLE !

Kiosque / Abonnement - Version papier / Version PDF

#HS5REDHAT

Red Hat aime le développeur !

Pour notre numéro thématique d'automne, prenons la direction de Red Hat. Vous savez, l'éditeur au chapeau rouge. Trop longtemps, le développeur cantonnait Red Hat aux distributions Linux, notamment l'édition communautaire, Fedora et à la version entreprise, RHEL. Dépassons cette vision simpliste pour découvrir comment l'éditeur a su aller vers le développeur, notamment en propulsant Java dans l'univers du Cloud et de Kubernetes. Si Red Hat était une carte Magic (ou tout autre RPG), ce serait :

- Sorcier niveau Linus
- Chapeau rouge, openus sourcus Tuxus
- PV : 404
- Tuxus légendaire

Particularité :

aime beaucoup les développeurs

Points forts :

Open Source, IDE, OpenJDK, Cloud Computing, OpenShift

Pouvoirs exclusifs :

Quarkus, Istio, Ansible, OpenShift, Kubernetes

Bonus :

multi-cloud avec OpenShift

Pouvoirs supplémentaires :

serverless, IDE en mode cloud

© Excelsior, L'Heure du dev

Bref, vous l'aurez compris, Red Hat est méconnu du développeur. Et pourtant, comme nous le verrons dans ce numéro, l'éditeur propose énormément de choses pour les développeurs et les équipes techniques. Grâce à lui, Java a su trouver sa place dans les environnements Cloud. Sur Kubernetes, Red Hat propose une plateforme complète que nous avions déjà abordé dans le n°241. Mais ce mois-ci, nous étendons son usage et les outils autour d'OpenShift.

Les meilleurs experts Red Hat ont participé à ce numéro. La diversité des outils et des technologies que vous allez découvrir illustrent toute la richesse du portfolio développeur de l'éditeur.

Plus d'excuses pour ne pas tester les solutions du chapeau rouge.

François Tonic
Le maître du jeu.

Contents

PROCHAIN NUMÉRO

Programmez! n°250
Disponible dès le 7 janvier 2022

Directives de compilation

PROGRAMMEZ!
Programmez! hors-série n°5 - AUTOMNE 2021
Directeur de la publication & rédacteur en chef
François Tonic
ftonic@programmez.com
Contacter la rédaction
redaction@programmez.com
Les contributeurs techniques
Nicolas Massé, Zineb Bendhiba, Katia Aresti, Christophe Laprun, Laurent Broudoux, Fabrice Leray, Vincent Demeester, Damien Grisonnet, Ashish Sardana, Sun Tan, Joël Takvorian, Sébastien Blanc, Yacine Kheddache, Marie-Laetita Baijot-Duchene, Michael Klodawski

Couverture : © RedHat
Maquette : **Pierre Sandré**
Marketing – promotion des ventes
Agence BOCONSEIL - Analyse Media Etude
Directeur : Otto BORSCHA
oborscha@boconseilame.fr
Responsable titre : Terry MATTARD
Téléphone : 09 67 32 09 34
Publicité
Nefer-IT - Tél. : 09 86 73 61 08
ftonic@programmez.com
Impression : **SIB Imprimerie, France**
Dépôt légal : **A parution**
Commission paritaire
1225K78366
ISSN : **2279-5001**

Abonnement
Abonnement (tarifs France) : 49 € pour 1 an, 79 € pour 2 ans. Etudiants : 39 €. Europe et Suisse : 55,82 € - Algérie, Maroc, Tunisie : 59,89 € - Canada : 68,36 € - Tom : 83,65 € - Dom : 66,82 €.
Autres pays : consultez les tarifs sur www.programmez.com.
Pour toute question sur l'abonnement :
abonnements@programmez.com
Abonnement PDF monde entier : 39 € pour 1 an.
Accès aux archives : 19 €.
Nefer-IT
57 rue de Gisors, 95300 Pontoise France
redaction@programmez.com
Tél. : 09 86 73 61 08

Décembre

		1	2	3	4	5
6	7	8	9	10	11	12
GophorCon (en ligne)						
	Meetup Programmez!					
13	14	15	16	17	18	19
		Sortie de Matrix 4	DevCon Programmez : secure by design (EFREI)			
20	21	22	23	24	25	26
27	28	29	30	31		

Janvier 2022

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
				Touraine Tech (Tours)		
24	25	26	27	28	29	30
31						

Février 2022

	1	2	3	4	5	6
		SnowCamp (Grenoble)				
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28						

Les événements Programmez!

Meetups Programmez!

7 décembre : Pulumi

Où : WeWorks, 33 rue Lafayette / Paris
Métros : Notre-Dame de Lorette (l12), Le Peletier (l7)
A partir de 18h30

Informations & inscription : programmez.com

Merci à Aurélie Vache pour la liste 2021, consultable sur son GitHub :
https://github.com/scraly/developers-conferences-agenda/blob/master/README.md

VIENT DE PARAÎTRE

Inside <GIT>

100 % CI/CD
100 % GIT

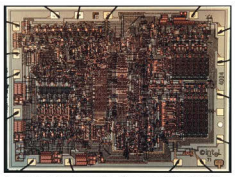
60 PAGES

Disponible sur programmez.com et en impression à la demande sur Amazon.fr

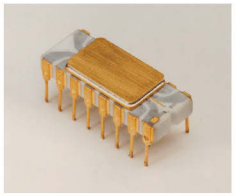
Roadmap des langages

1971 : Intel 4004

Le premier micro-processeur commercial a été introduit sur le marché par Intel en 1971 : le 4004. Le projet remonte à 1969 lorsque Nippon Calculating Machine Corp cherchait un composant totalement intégré pour une future calculatrice. Les ingénieurs vont intégrer 12 composants en 4 dont l'élément central le 4004. Le début d'une longue aventure. C'est finalement les difficultés du Japonais qui va permettre à Intel de commercialiser le 4004 alors qu'il était normalement réservé au constructeur.



A colorized schematic diagram of the 4004



The 4004 microprocessor

Three Intel engineers – Federico Faggin, Stan Mazor and Marcian E. (Ted) Hoff – created the 4004. It was a defining moment in Intel's history and would forever change the way computing technology impacts the world.

From left: Federico Faggin, Stan Mazor and Ted Hoff

novembre
Angular 13 : une grosse version !

Angular 13 est une version importante avec de nombreuses évolutions et un important travail sur Ivy. Ivy est le pipeline de rendu et de compilation d'Angular qui vient remplacer le pipeline des premières versions. View Engine n'est plus disponible à partir de cette version. Cette suppression va réduire la dépendance à ngcc. Angular Package Format (APF) a été modernisé pour mieux servir les développeurs. Google a supprimé les anciens formats de sortie, y compris les métadonnées spécifiques à View Engine.

novembre
TypeScript 4.5

TypeScript propose quelques belles évolutions avec la 4.5 :

- Support ECMAScript dans Node.js
- Support de type dans package.json
- Interopérabilité avec CommJS
- Support lib form node_modules
- Amélioration sur les types awaited et Promise

Des casses de compatibilités sont annoncées : lib.d.ts, inférence, option de compilation. Lisez bien la note de version.

novembre
PHP 8.1

La communauté PHP prépare la 8.1. Cette version introduit plusieurs nouveautés : enums, Fibers, amélioration des performances (opcache), type never, fonction array_is_list. Cette version est considérée comme une version mineure et aucune casse de compatibilité n'est attendue.

novembre
Rust 1.56.1

La 1.56 est sortie en octobre dernier. Les principales nouveautés :

- Support LLVM 13
- Amélioration du support aarch64
- Plusieurs nouvelles API en production dont std ::mem ::transmute, [T] ::first

La 1.56.1 introduit une correction sur Unicode. La 1.57 arrivera en bêta en décembre.

2022
? : Ruby 3.1

Ruby 3.1 est la prochaine version importante du langage. La 3.1 va intégrer YJIT, le nouveau compilateur JIT de Shopify. Il devrait remplacer MJIT, le précédent JIT. Les équipes espèrent un gain important dans les performances entre 20 et 40 % selon les benchs. Pour le moment, il s'agit d'une fonction en développement qu'il faut activer. Dans la preview 1 de Ruby 3.1, YJIT est disponible sur macOS, Linux. On aura droit à un nouveau debugger : debug gem. De nombreux éléments de la librairie standard évolueront.

1er trimestre (?)
GO 1.18

Le support de la programmation générique est une des fonctionnalités linguistiques les plus demandées par la communauté des utilisateurs de ce langage. Les choses ont finalement avancé et Google indique avoir travaillé, depuis deux ans, sur une série de projets de conception qui ont abouti à une conception basée sur des paramètres de type. Google souhaite implémenter totalement les génériques dans la version 1.8 actuellement en développement.

1er trimestre
.Net MAUI

Après avoir retiré à la dernière minute MAUI de .Net 6, Microsoft prévoit une disponibilité dans les prochains mois. MAUI est le nouveau modèle pour les applications multi-plates-formes. On pourrait créer des apps en C# et XAML pour mobile, desktop. Il s'agit d'une évolution de Xamarin.Forms. Le développement a pris du retard durant l'été. La preview 11 est prévue pour décembre.

mars
Java 18

Le prochain Java se remplit peu à peu. À l'heure où nous écrivons, les fonctionnalités annoncées sont :

- UTF-8 par défaut
- Simple Web Server
- Code snippets dans la JAVA API documentation
- Réimplémentation de Core reflection
- Vector API (incubation)
- Internet Address Resolution SPI (incertain pour le moment)

octobre
Python 3.11

Le prochain Python est attendu pour début octobre 2022. Pour le moment, les évolutions ne sont pas totalement fixées. Il y aura des améliorations et nouveautés sur : la gestion des erreurs, sur l'asynchronisme, implémentation de CPython, des améliorations diverses sur les modules fractions, math, operator, os, sqlite3, thread, time.
Attention cette version introduit plusieurs dépréciations et retraits fonctionnels : lib2to3, webbrowser.MacOSX (retrait total dans la 3.13), unittest, binhex.

2022 ou 2023
React 18

Le projet React a lancé officiellement le développement de la v18. La 1ere bêta est apparu début novembre. Actuellement, aucune date de disponibilité n'a été annoncée. Cette version doit apporter de nouvelles API, un nouveau serveur de rendu avec support de React.lazy. Bêta disponible depuis la mi-novembre.

TIOBE n'est pas joué

Pour rappel, l'index TIOBE donne une vue sur les langages les plus recherchés sur différents moteurs. Il ne donne pas réellement la popularité de tel ou tel langage.

Nov 2021	Nov 2020	Évolution	Langage	% dans les recherches
1	2	+1	Python	11.77%
2	1	-1	C	10.72%
3	3	=	Java	10.72%
4	4	=	C++	8.28%
5	5	=	C#	6.06%
6	6	=	VB	5.72%
7	7	=	JavaScript	2.66%
8	16	+8	Assembeur	2.52%
9	10	+1	SQL	2.11%
10	8	-2	PHP	1.81%

Les partenaires 2021 de

PROGRAMMEZ!

Le magazine des développeurs

Niveau maître Jedi

softeluent

LA MANUFACTURE CACD2

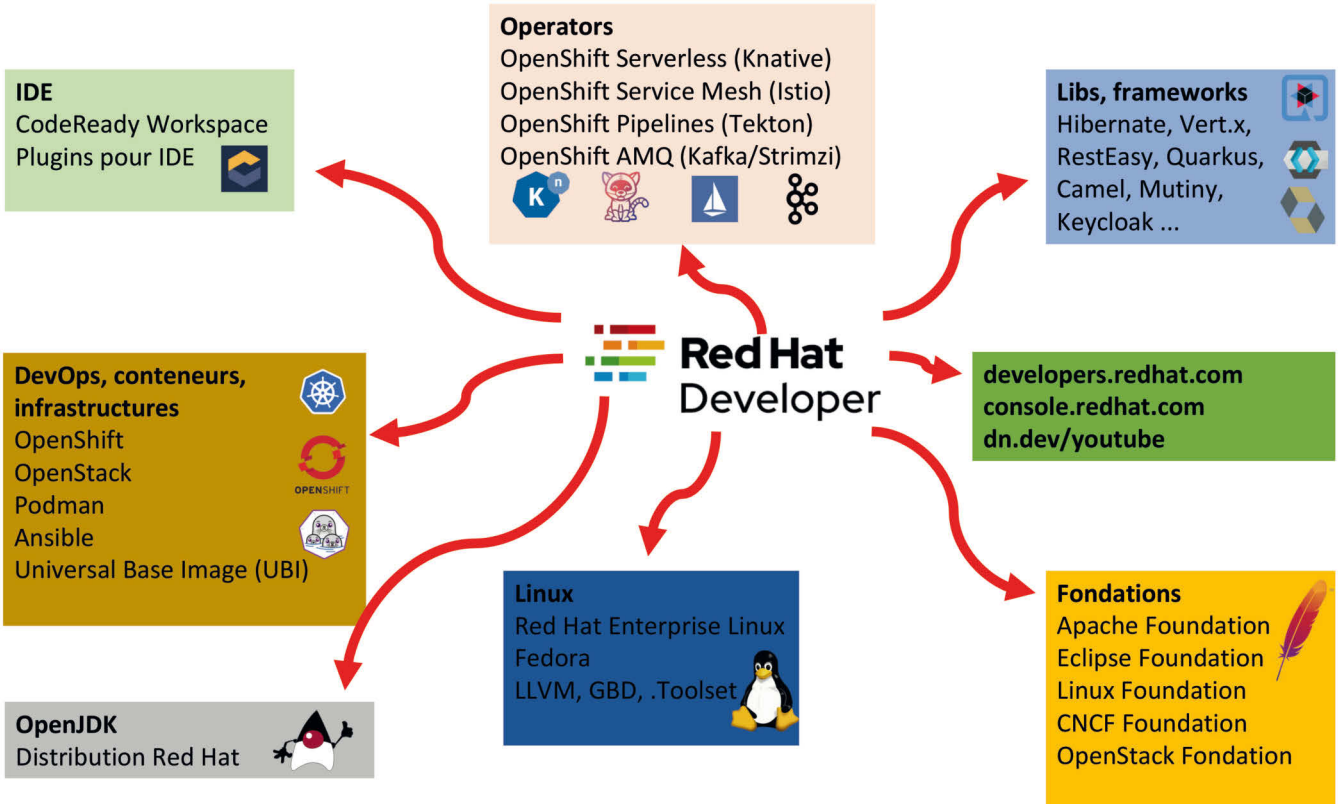
Niveau padawan

OSAXiS

Vous voulez soutenir activement Programmez! ?
Devenir partenaires de nos dossiers en ligne et de nos événements ?

Contactez-nous dès maintenant :
ftonic@programmez.com

Panorama des offres développeurs



Notre mission

Red Hat est une entreprise qui crée des logiciels selon le modèle Open Source. Nous puisons notre inspiration dans les communautés, que ce soit pour les produits que nous créons, la culture que nous adoptons ou les solutions que nous concevons avec les clients.

Dans le cadre de ce modèle de développement ouvert, nos ingénieurs sont en contact avec les communautés Open Source. Les membres de ces communautés travaillent les uns avec les autres pour identifier et concrétiser les idées les plus brillantes. Nous soutenons ces communautés en contribuant au code et en créant des produits à partir des projets en amont.

Nous intervenons au sein de nombreuses communautés Open Source, et nous nous efforçons de créer et de perfectionner les technologies qui constituent les environnements informatiques actuels. Du système d'exploitation aux solutions de middleware, aux conteneurs et aux outils d'automatisation, nous créons sans cesse de nouvelles solutions Open Source adaptées aux entreprises.

Nos moyen d'action

Participer : nous créons des projets communautaires en amont ou rejoignons des projets existants. Nous contribuons à la création du code, nous collaborons aux contenus, nous coordonnons des projets, nous formons des leaders et nous parrainons des événements.

Intégrer : nous intégrons de nombreux projets en amont, ce qui encourage le développement de plateformes communautaires Open Source. De plus, nous travaillons sur des projets qui impliquent différentes entreprises, ce qui nous permet d'intégrer à nos propres projets des technologies standard reconnues par le secteur.

Stabiliser : nous commercialisons ces plateformes et projets en association avec un riche écosystème de services et de certifications, tout en contribuant en retour aux projets en amont dans lesquels nous sommes impliqués.

Les communautés auxquelles nous contribuons

Nous assurons l'intégrité de la communauté pour les projets que nous parrainons et coordonnons, en nous impliquant activement notamment au niveau de l'infrastructure, de la gestion de projet, des communications, de la sensibilisation et de l'intégration.

Exemples de communautés dans lesquelles nous oeuvrons

Nous pouvons citer les Fondations autour de logiciels (Apache Software foundation, The Linux Foundation, la Cloud Native Computing Foundation, Openstack Foundation pour n'en citer que quelques uns), les systèmes d'exploitation (parmi lesquelles nous retrouvons CentOS, Fedora), mais également de nombreuses communautés autour des conteneurs, postes de travail, Middleware et identité, Exploitation, stockage et calcul, outils de développement.

S'agissant des conteneurs, Kubernetes a été initialement développé et conçu par des ingénieurs de Google, l'un des

premiers contributeurs à la technologie des conteneurs Linux, avant d'être donné à la Cloud Native Computing Foundation (CNCF) en 2015. Cela signifie que la CNCF est l'entité responsable de la maintenance de Kubernetes. communauté, tandis que les contributeurs bénévoles et les administrateurs sont responsables du développement, de la maintenance et des versions de Kubernetes.

Red Hat a été l'une des premières entreprises à travailler avec Google sur Kubernetes, avant même son lancement, et est depuis devenue le deuxième contributeur au projet Kubernetes.

Les outils Red Hat pour les développeurs

Red Hat aime les développeurs et les accompagne à toutes les étapes et à tous les niveaux de conception d'un logiciel, d'une librairie ou bien d'une solution complète.

A toutes les étapes signifie que Red Hat peut aider le développeur au moment du développement local sur la machine, au moment de la phase de test, au moment de l'intégration (en continue) mais aussi une fois que le logiciel est mis en production.

Red Hat est présent sur l'ensemble du spectre de l'outillage dont un développeur a besoin :

- Un cadre de développement : Quarkus, par exemple, est une stack Java révolutionnaire qui permet de développer des applications qui répondent à toutes les exigences du monde cloud natif. L'engouement autour de Quarkus est massif et suscite de plus en plus l'intérêt des entreprises qui souhaitent conserver et faire évoluer leur investissement fait sur la technologie Java.
- Des librairies de développement : On ne compte plus les nombres de librairies que Red Hat a développées ou bien auxquelles il contribue activement : Hibernate, Vert.x, RestEasy etc.
- Des environnements de développement : Red Hat est omniprésent au niveau de l'outillage "pur" , il propose son environnement de développement entièrement hébergé dans le cloud et tirant profit de la puissance offerte par Kubernetes : CodeReady Workspaces. Mais pour tous les autres IDE comme Eclipse, IntelliJ ou bien encore VS Code, Red Hat est également présent en proposant un ensemble d'extensions et plugins en support des différentes technologies Red Hat : Quarkus, OpenShift ...

Evidemment l'ensemble de cet outillage est Open Source et les contributions sont toujours les bienvenues !

OpenShift - la plateforme d'innovation pour les Devs

Red Hat s'efforce constamment à briser les silos entre les développeurs et les "ops", condition nécessaire à une bonne transformation numérique au sein de son entreprise. Un exemple concret est l'accent mis sur l'expérience développeur autour de sa plateforme OpenShift. **Figure 1**

OpenShift, la distribution Kubernetes de Red Hat, permet de facilement déployer et gérer ses containers. Or ces containers qui encapsulent les applications doivent être facilement déployables par les développeurs lors des phases de développement, de test et peut-être même lors de la mise en

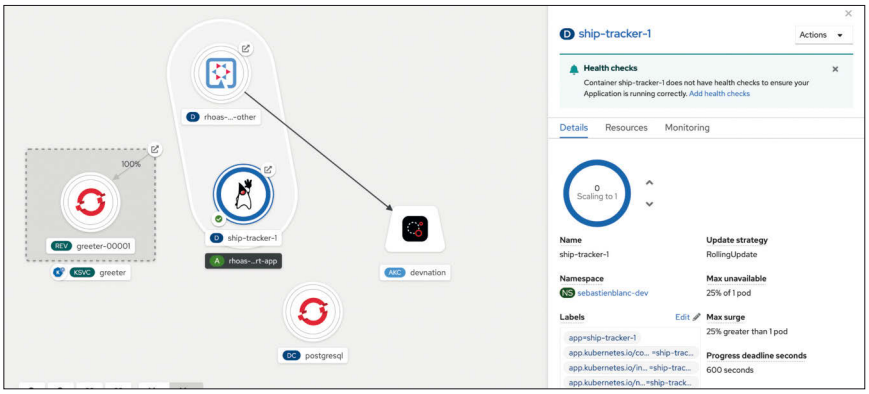


Figure 1

production. OpenShift offre une vue "développeur" qui permet de s'abstraire de toutes les notions Kubernetes que le développeur ne maîtrise pas forcément. Elle met l'accent sur une notion cruciale : déployer son application. Ainsi en quelques clics, le développeur pourra déployer son application parmi ses options différentes :

- En pointant directement vers un dépôt Git
- En pointant directement vers un container hébergé sur une registry de type dockerhub, quay.io …
- En déposant directement son JAR depuis son poste local sur la vue développeur, un peu comme lorsqu'on rajoute une vidéo sur sa chaîne Youtube

Une fois déployé, le développeur peut aisément paramétrer son *workload* comme ajouter des variables d'environnement, le nombre de réplica ou encore rajouter des sondes de santé (health probes).

Ainsi, le développeur peut itérer rapidement sur son projet, tirer profit de la puissance offerte par Openshift et finalement faciliter la tâche des équipes opérationnelles qui peuvent se concentrer sur leurs coeur de métier : mettre en place une infrastructure robuste, performante et extensible.

Pour aller plus loin

La première chose à faire est de se créer un compte sur https://developers.redhat.com/ , c'est rapide et complètement gratuit. Ceci vous donnera accès à une tonne de ressources : ebooks, cheat sheets, tutoriels et des articles.

Mais une fois votre compte créé vous pouvez également bénéficier de la *Developer Sandbox*, un environnement OpenShift complet, rien que pour vous, disponible gratuitement pour une durée de 30 jours avec renouvellement illimité et encore une fois, totalement gratuit. Vous pouvez l'obtenir dès à présent en allant sur developers.redhat.com/developer-sandbox

Voici quelques ressources supplémentaires :

- dn.dev/youtube : La chaine YouTube de Red Hat Developers
- dn.dev/upcoming : Un calendrier des prochains évènements Red Hat Developers (Live stream, Tech Talks etc.)
- console.redhat.com : Avec votre compte Red Hat Developers vous pouvez accéder aux services managés offert par Red Hat comme par exemple Kafka managé.
- https://twitter.com/rhdevelopers : Le compte Twitter de Red Hat Developers

Sun Tan

Senior Software Engineer chez Red Hat, Sun contribue aux projets Eclipse Che, CodeReady Workspaces et désormais JKube. Sun est également JUG Leader du Paris JUG dont l'objectif est de promouvoir Java à travers des présentations au format meetup à Paris.

CodeReady Workspaces: votre IDE dans le Cloud

Email, traitement de texte, calendrier, photos ou encore stockage de fichier, vos outils de travail ont progressivement migré dans le "cloud"… et pourquoi pas votre IDE ?

Construit à partir des projets open source Eclipse Che et Eclipse Theia, CodeReady Workspaces fournit les mêmes fonctionnalités que Eclipse Che avec le support de Red Hat. Il est inclus dans l'offre Red Hat Openshift Container Platform. D'un côté, CodeReady Workspaces inclut toutes les fonctionnalités que l'on peut attendre d'un IDE :

- la complétion de code, la coloration syntaxique, une assistance à la correction,
- le support des langages comme Java, Javascript/Typescript, C++, .NET, PHP, Go ou Python,
- l'intégration de vos outils de débogage et de Git,
- un système d'extension compatible avec les extensions Visual Studio Code.

D'un autre côté, comparé à un IDE traditionnel, CodeReady Workspaces apporte une toute autre expérience utilisateur. Il est à la fois :

- un Cloud IDE
- un Kubernetes-native IDE
- un IDE supportant des fonctionnalités de DEaC (Developer Environment as Code).

Le Cloud IDE

Il y a quelques années j'ai sauté le pas et ai commencé à utiliser exclusivement un Cloud IDE professionnellement. J'y suis toujours !

Quand on parle d'un Cloud IDE, il s'agit :

- **d'un IDE** : un logiciel dédié à la programmation. Il se présente généralement sous la forme d'un éditeur de code amélioré pour assister le développeur lors de ses principales tâches de programmation. Il permet au développeur d'être plus productif. Les IDE sont souvent spécialisés dans un langage de programmation.
- **accessible "as a Service" à partir d'un simple navigateur Web** - À la différence d'une machine dans le cloud sur laquelle vous vous connectez en SSH ou avec VNC. Les Cloud IDE disposent d'une interface web complète pour accéder à votre environnement tout en bénéficiant des fonctionnalités que vous trouvez sur les IDE classiques "desktop".

Mais pourquoi utiliser un Cloud IDE plutôt que d'installer et d'utiliser l'IDE "Desktop" sur son poste ? Voici quelques-uns de mes retours d'expérience après plusieurs années d'utilisation d'un Cloud IDE.

En manque d'espace disque ?

Le premier exemple concerne l'espace disque. Lorsque je développais mes applications depuis mon PC, j'arrivais souvent à court d'espace disque à cause de mes nombreux projets Maven ou Nodejs. Je ne voulais pas prendre le risque de supprimer des projets par peur de perdre des données. Aujourd'hui, je n'ai plus à faire le ménage de mon dossier `git` et aller à la recherche de mes dossiers `target` ou `node_modules` pour gagner quelques Go d'espace disque. Dans le cloud, les ressources disques sont configurables et il est facile d'étendre l'espace disque si besoin.

De plus, avec CodeReady Workspaces, les "workspaces" sont créés à la demande. Supprimer un workspace n'est plus un souci car il est possible à n'importe quel moment de régénérer l'environnement complet juste à partir d'un fichier sauvegardé dans le repo Git.

Internet très haut débit, même à la campagne

Contrairement aux idées reçues, avoir son IDE hébergé dans le cloud n'est pas un frein lorsque l'on dispose d'une connexion internet limitée. En voyage, ma connexion internet ne me permet pas de télécharger facilement les environnements Maven ou Docker. Par exemple, une image docker de quelques Go peut parfois nécessiter jusqu'à 2 jours de téléchargement.

Avec un Cloud IDE les téléchargements importants sont exécutés depuis le data center. Il y a de fortes chances que le débit chez votre hébergeur cloud soit bien plus performant que celui du Wifi ou de l'ADSL de votre maison de campagne. L'expérience utilisateur est également bien plus optimisée avec un Cloud IDE qu'avec un accès VNC car le stricte nécessaire est téléchargé par le navigateur.

Coder avec le même chromebook pendant 10 ans

Après 3 années de longs et loyaux services, il est temps de changer de PC car les 8Go de RAM qui étaient bien suffisants quelques années avant ne suffisent plus pour tester mon application avec Minikube ou encore lancer une compilation native de mon projet Quarkus.

Encore une fois, les ressources sont configurables dans le cloud et je n'ai pas forcément besoin d'une machine très puissante pour développer. Je peux instancier à la demande et de manière mutualisée, mon environnement de développement avec les exigences nécessaires en termes de CPU ou RAM tout en gardant le même PC pendant plusieurs années. Cerise sur le gâteau, je contribue au développement durable.

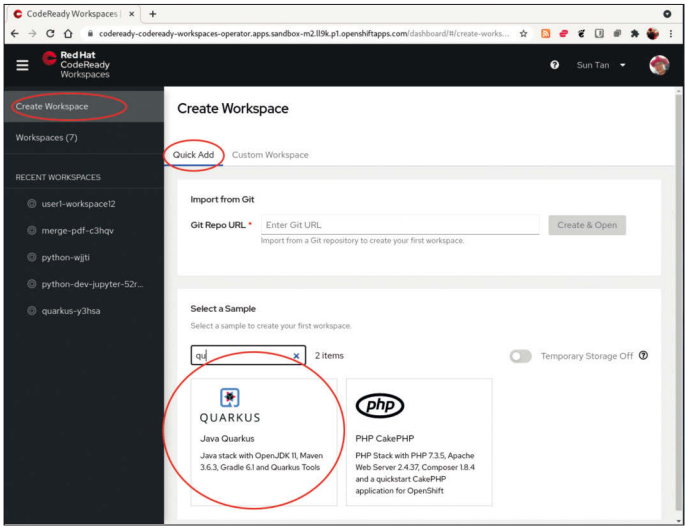
Sécurité

Nous avons tous été contraints au télétravail. Pourtant, au niveau sécurité, les réseaux personnels sont moins protégés que les réseaux d'entreprise. En télétravaillant, nous sommes de plus en plus soumis aux attaques . Pour un minimum de protection il est préférable de laisser les données sensibles dans les datacenters, chose faite avec un Cloud IDE.

Red Hat Developer Sandbox

Pour découvrir Openshift, Red Hat met gratuitement à disposition des clusters Openshift : Red Hat Developer Sandbox [1]. Ces clusters sont limités en ressource et en temps. Mais cela suffit amplement pour se faire une idée ou commencer un petit POC sur Openshift. Les clusters Openshift de la Developer Sandbox incluent également CodeReady Workspaces. Une fois le compte créé sur le site Developer Sandbox [1], vous pouvez directement accéder à une instance de CodeReady Workspaces avec l'URL https://workspaces.openshift.com.

1 Sur https://workpaces.openshift.com, une fois identifié avec votre compte, vous êtes redirigé sur le dashboard de CodeReady Workspace. Cliquez sur `Create Project` et l'onglet `Quick-add`.



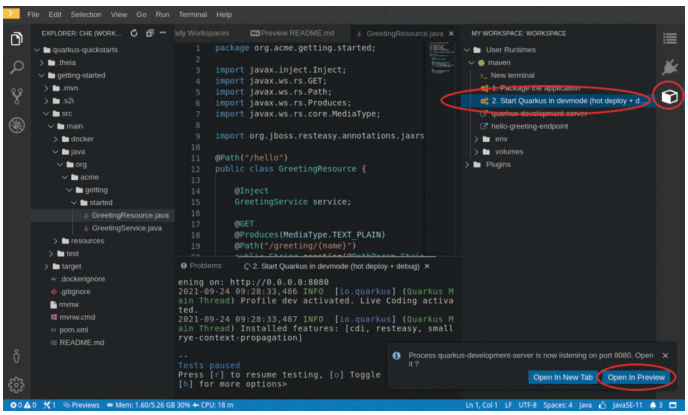
2 Les projets proposés servent d'exemple. Testons le projet d'exemple Quarkus. Cliquez sur `Java Quarkus`.

3 Attendez quelques minutes, vous pouvez regarder les logs.

4 L'IDE est démarré … le projet d'exemple de Quarkus est cloné automatiquement.

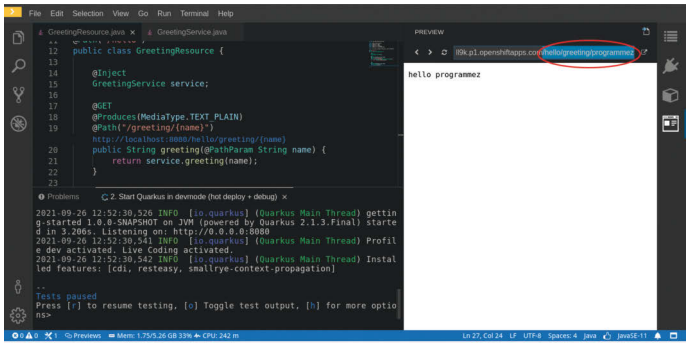
5 Ouvrez le fichier GreetingResource.java à partir de l'explorateur de projet.

6 Disons que vous souhaitez tester l'application avant de faire quelques modifications. Lancez l'application en `devmode` via le panneau de droite `My Workspace` en utilisant la commande `Start Quarkus in devmode (hot deploy + debug)`. Après quelques secondes, l'IDE devrait vous proposer d'ouvrir l'application. Cliquez sur `Open in Preview`.



7 La preview s'affiche.

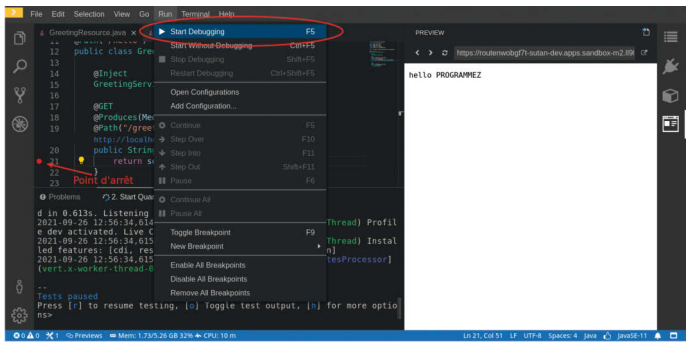
8 Ajouter à l'URL, le chemin vers la ressource `/hello/greeting/programmez`. Vous devriez voir apparaître le texte `programmez`.



9 Au niveau de la méthode `greeting`, utilisez le raccourci `Ctrl-espace` pour compléter et retourner la chaîne `name` en majuscule.

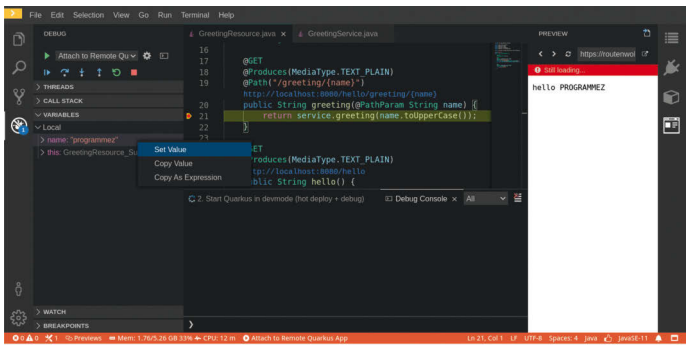
10 Rafraichissez la prévisualisation. Le texte s'affiche bien en majuscule.

11 Ajoutons un point d'arrêt dans le fichier en cliquant sur le côté gauche de l'éditeur à la ligne 24. Puis lancer le débogage.

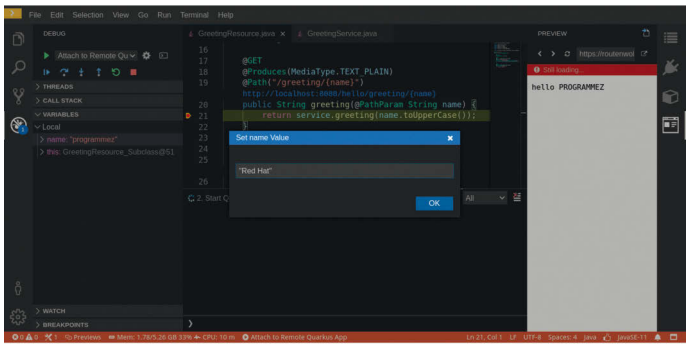


12 Rafraîchissez à nouveau.

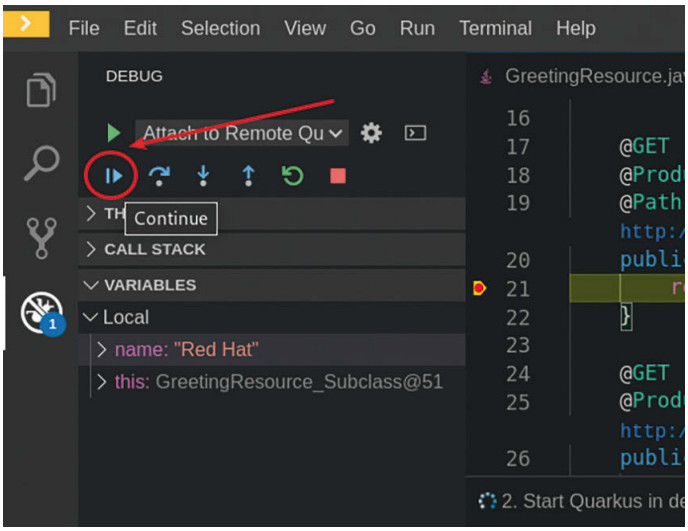
13 Changer la valeur de la variable local `name`.



14 Remplacez la valeur `Programmez` par `Red Hat`.



15 Reprenez l'exécution de l'application.



16 hello RED HAT

En quelques minutes, nous avons :

- démarré un nouveau projet Quarkus,
- lancé l'application en mode de développement,
- utilisé l'intellisense Java et modifié l'application à chaud,
- activé le mode Débogage puis changé la valeur d'une variable locale de la pile d'exécution.

Toute l'exécution s'est faite à distance uniquement au travers d'un navigateur web et pourtant, l'expérience est tout aussi proche d'un IDE lancé localement. Les Cloud IDE fournissent indéniablement beaucoup d'avantages pour les développeurs. CodeReady Workspaces bénéficie de tous ces avantages. Mais pas que !

Il supporte également la fonctionnalité de DEaC (Developer Environment as Code).

Developer Environment as Code

Les principes de DEaC s'inspirent de ceux du IaC (Infrastructure as Code). L'infrastructure as Code permet de déployer une infrastructure en termes de serveurs, stockage et réseau grâce à de simples fichiers texte lisibles par les développeurs. Avec les mécanismes de IaC, il est désormais plus simple, plus rapide et plus sûr pour les administrateurs de déployer et gérer leurs infrastructures.

Pour les projets logiciels, le DEaC étend les principes du IaC. De même qu'avec le IaC, un simple fichier texte est utilisé pour déployer un clone de l'environnement de production sur son environnement de dev.

En complément de ce qu'apporte le IaC, le DEaC ajoute à ce fichier les informations suivantes :

- les dépôts de source code à cloner,
- les plugins et extensions IDE,
- les environnements pour compiler et packager une application,
- les commandes qui seront couramment utilisées lors du développement d'une application.

Pour chacun de vos projets git, CodeReady Workspaces peut utiliser un fichier de description nommé "devfile.yaml". Ce fichier, à stocker à la racine de votre projet Git, inclut toutes ces informations pour générer et démarrer un environnement de développement complet.

Tester les nouveautés de Java 17 sans installer Java 17

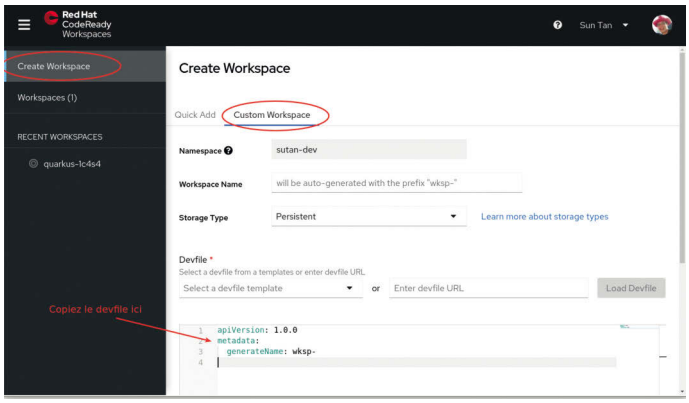
Tester de nouvelles versions de bibliothèques tierces, langages, OS ou un nouveau framework peut être risqué pour le développeur. Il passe une bonne partie du temps à installer les bons outils. Le développeur préférerait minimiser les risques de casser l'installation existante par une mise à jour vers une version plus récente du langage de programmation par exemple. Le 14 septembre dernier est sortie Java 17. Pour découvrir les nouvelles fonctionnalités du langage, iriez-vous jusqu'à installer Java 17 sur votre machine et prendre le risque de compromettre l'installation de l'environnement de développement de votre projet professionnel utilisant Java 8 ? Probablement pas.

Avec les mécanismes de DEaC, il est possible de construire un environnement pour coder en Java 17. Sur CodeReady Workspaces, les workspaces générés étant isolés, nous avons la possibilité d'avoir des projets utilisant Java 8 et d'autres Java 17, ou même des workspaces pour coder dans d'autres langages.

Testez les nouveautés de Java 17 avec la Developer Sandbox

Pour illustrer le dernier exemple, construisons ensemble un devfile.yaml et commençons à développer avec Java 17 sur la Red Hat Developer Sandbox. Le workspace sera lancé à côté du premier workspace Quarkus que vous aviez lancé précédemment. Vous pourrez lancer l'un ou l'autre de manière complètement indépendante : les deux workspaces sont complètement isolés et les changements de l'un n'affectent pas l'autre.

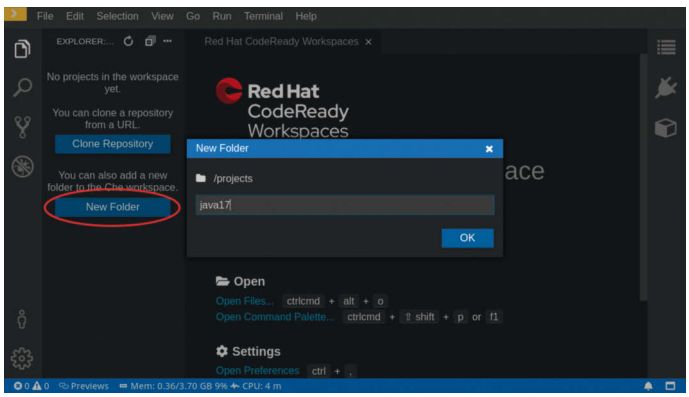
1 Sur https://workpaces.openshift.com , une fois identifié avec votre compte, créez un workspace: `Create Workspace` > `Custom workspace`



2 Collez le devfile suivant et démarrez le avec le bouton `Create and Open`

Code complet sur programmez.com & github

3 Une fois le workspace démarré, créez un nouveau dossier depuis l'explorateur

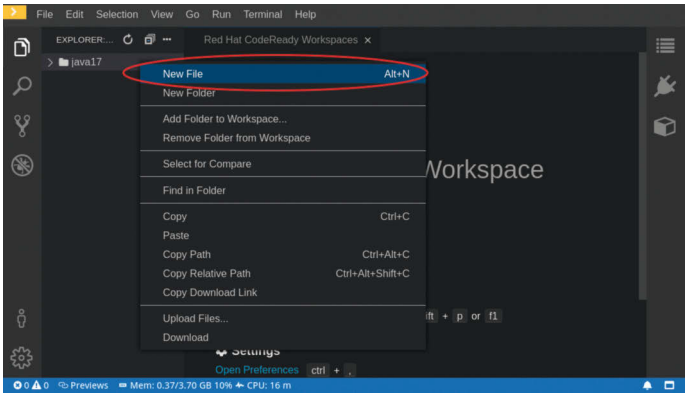


4 Puis un fichier `Main.java`

PROCHAIN NUMÉRO

Programmez! n°250

Disponible dès le 7 janvier 2022



5 Copiez ce contenu qui utilise une fonctionnalité de Java 17 en preview: le Pattern Matching

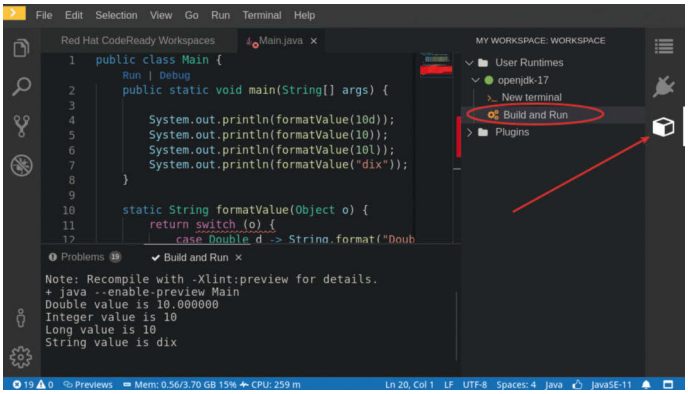
```
public class Main {
  public static void main(String[] args) {

    System.out.println(formatValue(10d));
    System.out.println(formatValue(10));
    System.out.println(formatValue(10l));
    System.out.println(formatValue("dix"));
  }

  static String formatValue(Object o) {
    return switch (o) {
      case Double d -> String.format("Double value is %f", d);
      case Integer i -> String.format("Integer value is %d", i);
      case Long l -> String.format("Long value is %d", l);
      case String s -> String.format("String value is %s", s);
      default -> o.toString();
    };
  }
}
```

6 Sur le panneau de droite, vous disposez d'une commande pour compiler puis exécuter le programme Main avec Java 17 et le mode preview activé. Cliquez sur `Build and Run`. La sortie suivante devrait s'afficher:

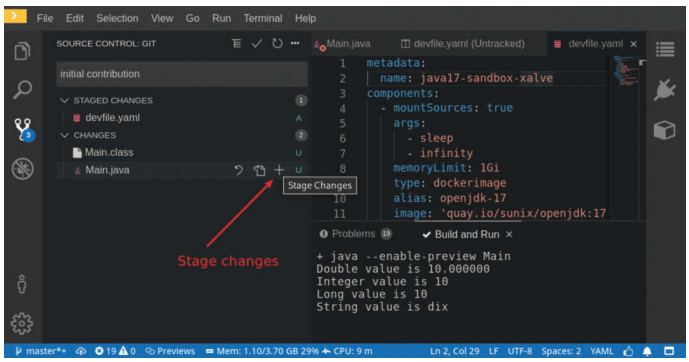
```
+ javac --enable-preview -source 17 Main.java
Note: Main.java uses preview features of Java SE 17.
Note: Recompile with -Xlint:preview for details.
+ java --enable-preview Main
Double value is 10.000000
Integer value is 10
Long value is 10
String value is dix
```



7 Vous pouvez ensuite importer le devfile.yaml dans votre répertoire source `/projects/java17` avec la commande `File -> Save workspace`

8 Initiez le repo git avec la command `Git: Initialize git repo` disponible à partir de la command palette (F1)

9 Ajoutez puis commitez au repo git les fichiers `devfile.yaml` et `Main.java`



10 Ajoutez un remote git avec un nouveau dépôt Github que vous aurez créé préalablement.

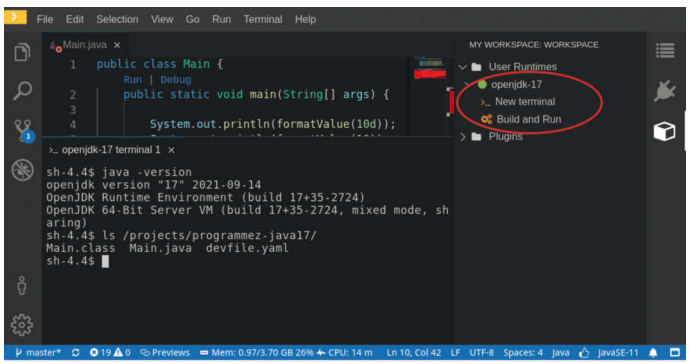
11 Poussez les modifications.

Pour ma part, le résultat se trouve ici : https://github.com/sunix/programmez-java17 . Vous pouvez ensuite partager l'url suivante https://workspaces.openshift.com#https://github.com/sunix/programmez-java17 qui générera en quelques secondes un nouvel environnement de dev pour ce projet là et permettra à n'importe qui ayant un accès à votre instance CodeReady Workspaces de tester les dernières nouveautés de Java 17 sans avoir à l'installer. En utilisant le devfile.yaml à la racine de votre projet Git, les environnements générés seront prêts à l'emploi, disposant des fichiers du projet, du conteneur openjdk-17 ainsi que de la commande `Build and Run`.

Décortiquons le devfile.yaml

Le devfile.yaml déclare 2 composants:

- Le premier composant est `openjdk-17`. Il est de type `dockerimage` et charge une image docker contenant l'openjdk 17 dans le workspace. Dans un workspace, ce conteneur est accessible depuis la vue `My Workspace`. Vous pouvez ouvrir un terminal sur ce conteneur. La déclaration `mountSources: true` permet de monter le volume partagé contenant les fichiers du workspace. Nous remarquons également que la commande `Build and Run` se trouve au niveau du conteneur `openjdk-17`. Cela signifie que la commande sera exécutée dans ce conteneur. Les conteneurs cible des commandes sont indiqués par l'attribut `component` dans la partie `commands` du fichier devfile.yaml



- Le composant `redhat/java/latest` est un plugin. Il référence l'extension VSCode Java et est lui-même exécuté sur son propre conteneur.

Enfin la partie `commands` contient les commandes préconfigurées pour un

projet donné. Ce devfile.yaml ne contient pas de section `projects`. En le démarrant de la partie `Custom Workspace`, aucun projet ne sera cloné. Si vous chargez un projet git contenant un devfile.yaml sans section `projects`, CodeReady Workspaces clonera automatiquement le projet chargé.

"Mais, ça marche sur mon poste"

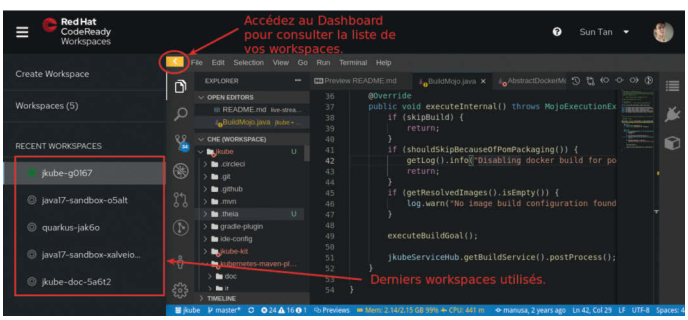
Un autre cas d'utilisation du DEaC concerne la fameuse expression "Ca marche sur mon poste". Que ce soit un bug chez un collègue ou en production, il n'y a rien à faire: "ça marche sur mon poste". Un des aspects du DEaC est de pouvoir répliquer un environnement à l'identique. Il permet de minimiser l'effet "Mais, ça marche sur mon poste" en utilisant, dans le fichier devfile.yaml, exactement les mêmes images Docker qu'en production.

Je n'y connais rien en Javascript

Pour être franc, je ne suis plus vraiment dans ce cas, mais il m'a fallu plusieurs années pour apprivoiser l'écosystème Javascript. Dans la vie d'un développeur, il est impossible de ne pas avoir à toucher un jour ou l'autre à un projet écrit dans un langage différent de celui de prédilection. Mais aujourd'hui encore, lorsque je travaille sur un nouveau projet Javascript, je me pose toujours les mêmes questions : `yarn install` ou `npm install` ? puis `yarn start` ? ou faut-il lancer `npm run package` ou encore une commande `gulp` ? Comment activer le hot reload ? et enfin … quelle version de nodejs dois-je installer ?
Finalement, une fois avoir eu une réponse à toutes ses questions, l'application ne tourne toujours pas sur mon environnement de développement.
CodeReady Workspaces permet d'inclure, dans un devfile.yaml, des containers Docker contenant toutes les dépendances (avec les bonnes versions) nécessaires au développement de vos applications. Le devfile.yaml contiendra également les bonnes commandes réutilisables pour compiler, packager, debugger et lancer vos applications. Ainsi, n'importe quel développeur peut commencer à contribuer sur un projet dès lors qu'un devfile.yaml aura été convenablement inclus à la racine du projet.

Project and Task Switching

Lorsque je travaille sur un projet, il m'arrive de passer d'une tâche à une autre, ou d'un projet à un autre. Par exemple lorsque l'on me sollicite pour une revue de code alors que je suis en plein développement sur une autre tâche. Avec une approche traditionnelle, j'utiliserais `git stash`. Mais cette approche est assez risquée car elle peut causer des problèmes au niveau des fichiers non pris en charge par un git comme les dossier "target" en Java ou "node_modules" en Javascript. CodeReady Workspaces permet de générer et de lancer plusieurs environnements de développement basés sur le même devfile.yaml. Tout est régénéré et donc dupliqué à l'identique. Chaque environnement dispose de volumes dédiés au stockage des fichiers. Les environnements sont isolés et ne communiquent pas entre eux.
Ainsi, je peux travailler sur une tâche d'un projet sur un workspace, être interrompu et lancer un nouvel environnement de développement pour une revue de code sur le même projet, puis bien plus tard revenir sur la première tâche simplement en accédant au premier workspace.



Sur la Developer Sandbox, vous serez limité qu'à un seul workspace en cours d'exécution en parallèle. Cependant, ce paramètre est configurable par l'administrateur du cluster.

Kubernetes-native IDE

Le dernier aspect de CodeReady Workspaces est qu'il est Kubernetes-native. Nous qualifions une application Kubernetes-native lorsqu'elle tourne sur Kubernetes nativement. CodeReady Workspaces tire le meilleur de Openshift et donc de Kubernetes pour vous offrir la meilleure expérience développeur. CodeReady Workspaces est basé sur Eclipse Che. Le moteur de CodeReady Workspaces est instancié et géré par un opérateur Kubernetes : chaque workspace est instancié sous la forme d'un Pod Kubernetes. Le cycle de vie de des workspaces de CodeReady Workspaces est ainsi simplifié.
D'un point de vue développement d'applications Kubernetes, les développeurs sont intéressés par le déploiement et les tests de leurs applications Kubernetes. Dans un schéma classique, les développeurs Openshift utiliseraient les commandes `oc`, `kubectl` ou `odo` pour déployer leurs développements locaux sur un cluster Openshift. Avec CodeReady Workspaces tout cela n'est plus nécessaire. Les environnements de développement de CodeReady Workspaces sont déployés comme des pods Kubernetes. Il est possible de démarrer n'importe quel conteneur Docker dans un workspace Che. Ainsi, nous pouvons instancier des conteneurs de base de données et les conteneurs applicatifs au sein même du workspace.
L'intérêt est de pouvoir utiliser les mêmes conteneurs en production et dans l'environnement de développement.

Touche pas à ma prod

Il y a quelques années, j'ai rencontré un développeur qui accédait directement à la production pour déboguer. Les conséquences peuvent être désastreuses ! Ne faites jamais ça.
Une fois avoir dit ça, il faut avouer qu'il est bien plus pratique de déboguer en production quand on n'est pas en mesure de reproduire un bug dans son environnement de développement.
CodeReady Workspaces vous permet de réduire ces différences et de déboguer presque comme si vous étiez en production. Un workspace étant instancié sous la forme d'un Pod Kubernetes, vous pouvez, sur ce Pod "workspace", soit :

- Instancier un container utilisant la même image Docker utilisée en production,
- Instancier un déploiement Kubernetes utilisant la même définition utilisée en production.

Et ces deux solutions sont possibles juste en déclarant les bons composants dans le devfile.yaml. Vous pourrez toujours personnaliser le workspace une fois instancié.

CodeReady Workspace, votre prochaine chaîne de CI ?

Au niveau des chaînes d'intégration continue, il m'est arrivé de ne pas pouvoir reproduire localement une erreur dans un test unitaire ou encore reproduire localement un problème de compilation. Si seulement nous pouvions accéder à un job de notre CI et y brancher notre IDE.
La plupart des offres de CI intègre désormais la possibilité de lancer les job dans des conteneurs Docker. Il est donc possible avec CodeReady Workspaces de répliquer ces Job en utilisant les mêmes images Docker dans le devfile.yaml et de reproduire à l'identique les différentes erreurs rencontrées dans la CI.
CodeReady Workspaces permet donc d'exécuter, à la demande et dans un workspace isolé, des tâches de build ou même de déploiement. Mais finalement, un workspace de CodeReady Workspaces est-il si différent d'un job Jenkins ? Nous en sommes pas si loin que ça, n'est ce pas ? On pourrait presque en faire une CI interactive ! Les contributions sont les bienvenues.

En action avec la compilation native de Quarkus

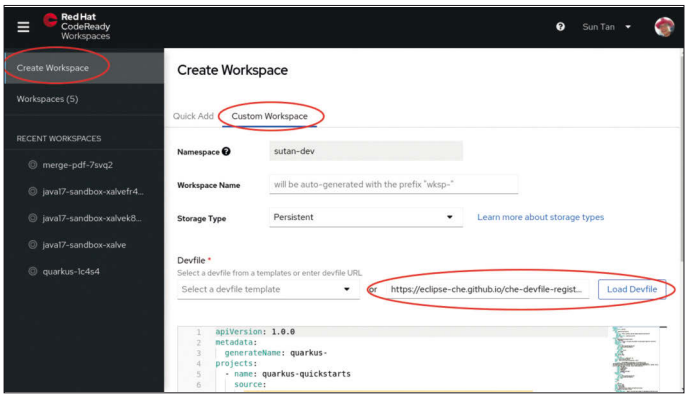
Une fonctionnalité qu'offre le projet Quarkus, est de pouvoir compiler Ahead

of time et nativement vos applications Java. Cela permet d'avoir des chargements plus rapides au démarrage et généralement adaptés aux microservices ou fonctions qui ont besoin de supporter des montées en charge rapide et de courte durée.

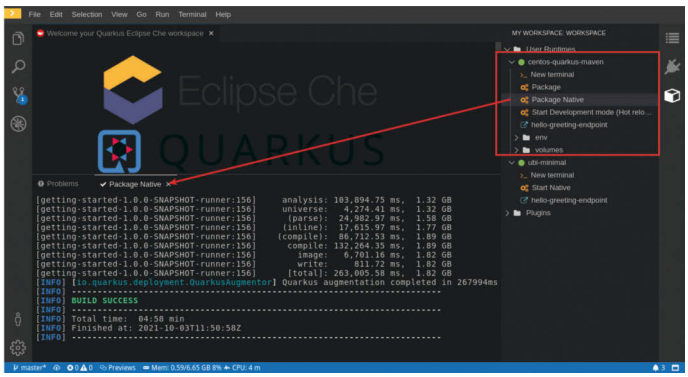
Pour les développeurs, dans un environnement de développement traditionnel, il est difficile de travailler avec ce nouveau mode:

- Les compilations natives peuvent être longues et demander beaucoup de ressources.
- On peut constater des comportements très différents lors de l'exécution de l'application sur un PC par rapport à l'environnement de production. Dans un environnement de production, l'application native sera dans la plupart des cas packagée dans un conteneur disposant du minimum requis pour son exécution.

Sur la Developer Sandbox, importez le devfile suivant: https://eclipse-che.github.io/che-devfile-registry/7.36.0/devfiles/quarkus/devfile.yaml



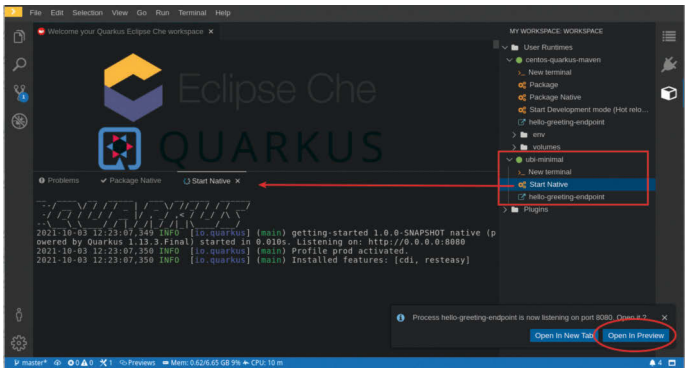
Le workspace généré permet de compiler nativement l'application Quarkus: il dispose d'une commande `Package Native` qui est exécutée dans le conteneur de `centos-quarkus-maven`. Ce conteneur contient toutes les dépendances pour développer les applications Quarkus: Maven, Java et GraalVM. GraalVM est nécessaire pour compiler les applications Quarkus en mode natif.



Utilisez la commande `Package Native` pour lancer la compilation native. La compilation sera bien exécutée sur le conteneur `centos-quarkus-maven` grâce à l'attribut `component` de la commande :

```
components:
 [...]
 - type: dockerimage
   alias: centos-quarkus-maven
   image: quay.io/eclipse/che-quarkus:7.36.0
   [...]
commands:
 [...]
 - name: Package Native
   actions:
     - type: exec
       component: centos-quarkus-maven
       command: ./mvnw package -Dnative -Dmaven.test.skip -Dquarkus.native.native-image-xmx=2G
       workdir: ${CHE_PROJECTS_ROOT}/quarkus-quickstarts/getting-started
```

Une fois l'application compilée en mode native, vous pouvez l'exécuter avec la commande `Start Native`




```
components:
 [...]
 - type: dockerimage
   alias: ubi-minimal
   image: registry.access.redhat.com/ubi8/ubi-minimal
   memoryLimit: 32M
   mountSources: true
   endpoints:
     - name: hello-greeting-endpoint
       port: 8080
       attributes:
         path: /hello/greeting/che-user
   command:
     command: ['tail']
   args: ['-f', '/dev/null']
commands:
 [...]
 - name: Start Native
   actions:
     - type: exec
       component: ubi-minimal
       command: ./getting-started-1.0.0-SNAPSHOT-runner
       workdir: ${CHE_PROJECTS_ROOT}/quarkus-quickstarts/getting-started/target
```

Cette fois-ci, la commande `Start Native` est exécutée dans le conteneur du composant `ubi-minimal`. Ce conteneur est minimal, il ne contient presque aucune dépendance. Il est idéal pour être utilisé en production car la surface d'attaque est limitée avec ce type de conteneur. Les applications natives Quarkus s'exécutent parfaitement sur ce type de conteneur.

Ainsi, avec ce devfile et les workspaces générés avec ce dernier, nous avons pu tester l'application dans quasiment les mêmes conditions qu'en production.

Installer CodeReady Workspaces sur Openshift

L'instance sur Red Hat Developer Sandbox vous permet d'avoir une bonne idée de ce que peut vous apporter CodeReady Workspaces. Cependant, nous serons assez rapidement limité pour une utilisation quotidienne :

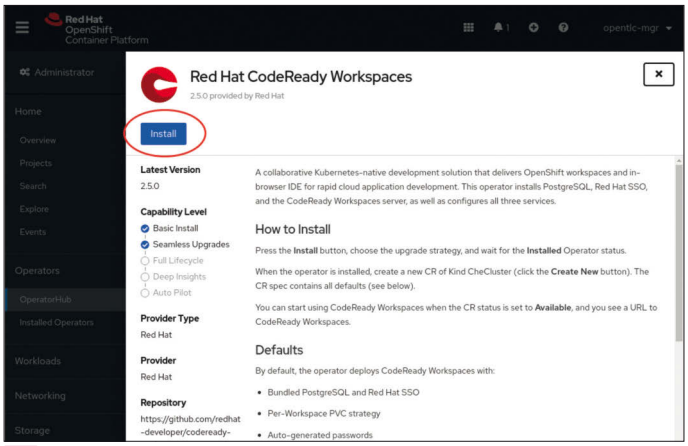
- Quota : Limitation en mémoire et CPU.
- Le compte est réinitialisé après 30 jours.
- Vous ne pourrez lancer qu'un workspace à la fois.
- Pas de possibilité de personnaliser les Projets par défaut sur la page `Create Project` / `Quick-add`.

Utiliser OperatorHub

Si vous avez à disposition une instance d'Openshift 4, vous pouvez y installer CodeReady Workspaces et y effectuer les modifications de votre choix.
La manière la plus simple de l'installer est d'utiliser l'OperatorHub :

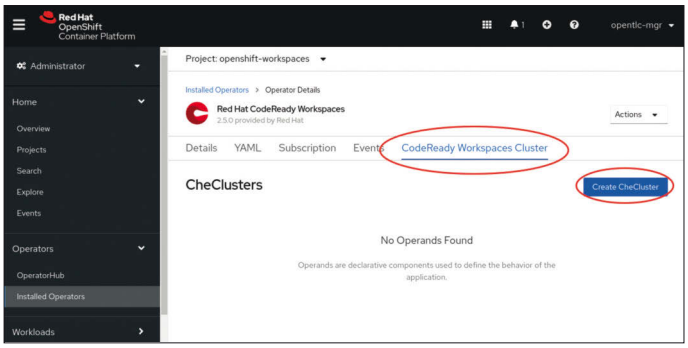
1 Sélectionnez Red Hat CodeReady Workspaces dans la section OperatorHub

2 Cliquez sur Install



3 Sélectionnez les valeurs par défaut puis cliquez sur "Subscribe"

4 Vous pouvez alors installer un cluster CodeReady Workspaces en cliquant sur "Create CheCluster"



5 Une fois le cluster créé, il faudra patienter un peu pour voir apparaître les liens d'accès à CodeReady Workspaces.

Tester les super-pouvoirs de CodeReady Workspaces sans Openshift

Si vous n'êtes pas encore convaincu, sachez que vous pouvez utiliser la version communautaire de CodeReady Workspaces : Eclipse Che. Eclipse Che peut également s'installer sur une autre distribution de Kubernetes que Openshift. Avec Eclipse Che, il faudra garder à l'esprit que vous ne bénéficiez que d'un support communautaire et que l'installation et la maintenance seront plus complexes. Suivez le guide [2] dans la documentation [3] d'Eclipse Che.

Conclusion

Nous arrivons au bout de cet article sur CodeReady Workspaces et n'avons évidemment pas pu couvrir la totalité des fonctionnalités du produit. A l'utilisation :

1 CodeReady Workspaces ressemble à VSCode. Il y a deux principales raisons : CodeReady Workspaces est basé sur Eclipse Che pour la gestion des workspaces.
2 Sur chaque workspace, un IDE web est mis à disposition. Par défaut, il s'agit d'Eclipse Theia

3 Eclipse Theia intègre l'éditeur de code JavaScript "Monaco" qui est celui utilisé par VSCode.
4 Eclipse Theia supporte les extensions VSCode.
CodeReady Workspaces intègre par défaut certaines extensions. Ces extensions sont testées et supportées par les équipes Red Hat de manière à ce qu'elles fonctionnent "out of the box". Vous pouvez utiliser d'autres extensions VSCode, mais nous ne garantissons pas le bon fonctionnement.

What's next?

Attention, une nouvelle version de devfile arrive bientôt : plus rapide, plus interopérable et chargée par une nouveau moteur de workspaces: le DevWorkspace. Pour en savoir plus, un blog post [4] a été publié concernant les travaux en cours.

#CheJoy

Enfin, vous trouverez sur Twitter avec le hashtag #chejoy [5], une série de petites vidéos courtes montrant des fonctionnalités sympa de Eclipse Che et de CodeReady Workspaces. Pour résumer, CodeReady Workspaces est à la fois un Cloud IDE et un kubernetes-native IDE supportant les fonctionnalités de DEaC. J'espère que cet article vous aura donné l'envie de sauter le pas et d'utiliser CodeReady Workspaces.

Références

[1] Developer Sandbox for Red Hat OpenShift :
https://developers.redhat.com/developer-sandbox
[2] Guide d'installation de Eclipse Che sur Google Cloud Platform :
https://www.eclipse.org/che/docs/che-7/installation-guide/installing-che-on-google-cloud-platform/
[3] Documentation d'Eclipse Che :
https://www.eclipse.org/che/docs
[4] Blog devfile v2
https://che.eclipseprojects.io/2021/06/23/@florent.benoit-devfile-v2-and-ide-plug-ins.html
[5] Videos #CheJoy :
https://twitter.com/search?q=%40eclipse_che%20chejoy

Développer et déployer un service d'analyse du trafic routier basé sur l'IA avec les solutions Nvidia et Openshift

La nécessité d'automatiser et d'améliorer l'efficacité opérationnelle et la sécurité dans nos espaces physiques n'a jamais été aussi grande. Qu'il s'agisse de rationaliser l'expérience client dans les points de ventes, de s'attaquer aux embouteillages dans nos villes en expansion ou d'améliorer la productivité dans nos usines, la puissance de l'IA et de l'edge computing est essentielle.

Ashish Sardana

Working as a Deep Learning Engineer for NVIDIA, Ashish is skilled in applying research to develop deep learning (DL) applications in conversational and recommender systems. In addition to this, he works closely with the DL Product Management group to increase product adoption. His current interests include – multimodal representation and capsule networks.

Environ 1 milliard de caméras IoT déployées dans le monde génèrent une mine de données qui, lorsqu'elles sont associées à la perception et au traitement basés sur l'IA, sont essentielles pour transformer des zones ordinaires en espaces intelligents.

Développement d'une application d'analyse du trafic

Nous allons vous guider dans la création d'un service d'IA qui analyse le trafic routier avec les outils logiciels (gratuits) Nvidia, intégrés avec Red Hat Openshift, sur sur une instance AWS équipée de GPU. Cette application basée sur une approche IA Deep-Learning analysera les flux de caméras de surveillance pour suivre les différents véhicules et les classer par type…

Nous guiderons l'installation et la configuration de l'instance AWS GPU muni de la dernière génération de GPU Nvidia Ampère A100, de l'orchestrateur Openshift et du Nvidia GPU operateur pour une intégration des GPU avec Openshift.

Parmi les outils logiciels, nous couvrirons l'utilisation de modèles Deep-Learning pré entrainés disponibles sous le catalogue logiciel Nvidia NGC, l'utilisation du kit de développement logiciel (SDK) Nvidia Metropolis intégrant notamment Deepstream pour une gestion facilitée et performante des flux vidéos, l'utilisation du SDK Nvidia TAO intégrant TensorRT pour l'optimisation de ces modèles Deep Learning, avant qu'ils ne soient déployés en production à grande échelle grâce au serveur d'inférence Nvidia Triton intégré avec Openshift pour robustesse et agilité.

NVIDIA METROPOLIS

NVIDIA Metropolis est un framework qui simplifie le développement, le déploiement et le passage à l'échelle d'applications d'analyse vidéo basées sur l'IA, traitées en périphérie et/ou dans le cloud.

Avec l'augmentation du nombre d'automobiles, vient la nécessité de surveiller le flux de circulation sur les voies urbaines, péri-urbaines et autoroutes, dans l'objectif d'une amélioration du trafic. Le problème fondamental qui se pose est d'analyser trois composantes principales : le volume du trafic, la vitesse de déplacement et le type de véhicules. **Figure 1**

Ces 3 composantes s'appuient fortement sur deux techniques de base : la détection d'objets et leur suivi.

La détection d'objets est utilisée pour localiser les véhicules sur la route. Nous utilisons ensuite un deuxième algorithme pour classer les véhicules en différentes catégories telles que "voitures", "camions", "SUV", etc., ce qui nous aide à comprendre la véritable nature du trafic. Enfin, nous suivons les véhicules sur différentes images pour éviter un comptage redondant, ce qui peut être une tâche compliquée si plusieurs caméras ont des champs de vision qui se chevauchent.

Pour résoudre ce problème, nous aurons besoin de la mise en œuvre efficace d'un pipeline d'analyse vidéo, depuis l'ingestion des flux vidéo jusqu'à l'analyse temps réel des flux à l'aide d'algorithmes d'apprentissage profond, tout en considérant le passage à l'échelle de la solution pour des millions de caméras. **Figure 2**

Figure 1

Figure 2

Pas à pas du développement du service d'analyse du trafic

Le flux de travail pour construire ce cas d'utilisation est le suivant :

1. Déployer Red Hat OpenShift sur des instances p4 AWS avec l'opérateur GPU NVIDIA.
   - Télécharger et Installer les conteneurs (pull container) d'application
   - Télécharger et Installer les modèles pré-entraînés depuis Nvidia NGC.
2. Optimiser ce modèle en utilisant Nvidia TensorRT.
3. Configurer l'application Nvidia DeepStream pour cet environnement.
4. Assurer le passage à l'échelle de l'inférence du modèle sur les 8 GPU A100 en utilisant Red Hat OpenShift.

Figure 3

Étape 1 : Configurer l'environnement sur AWS avec OpenShift

Vous pouvez déployer OpenShift en tant que service géré chez votre fournisseur de cloud préféré pour une expérience transparente sur Azure, AWS, IBM Cloud ou Google Cloud.

Le GPU NVIDIA A100 est doté d'un GPU multi-instance (MIG) qui divise le GPU en sept instances, chacune étant totalement isolée avec sa propre mémoire à large bande passante, son cache et ses cœurs de calcul. Cela permet de prendre en charge plusieurs traitements en parallèle pour optimiser l'utilisation du GPU. Nous utiliserons une instance AWS p4 qui fournit 8 GPU A100, chacun d'entre eux disposant de 40 Go de mémoire GPU HBM2 haute performance.

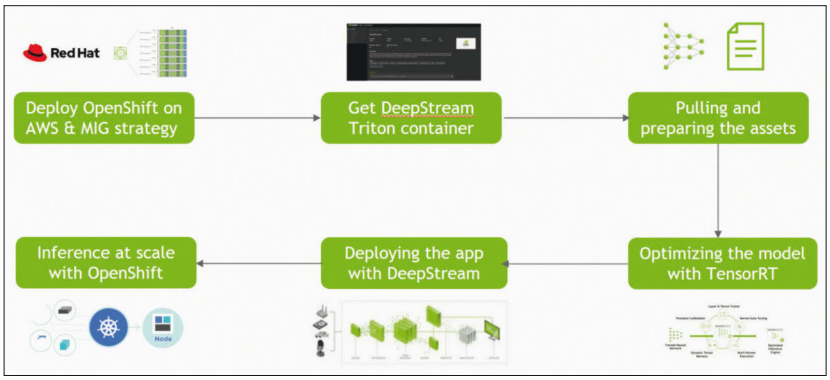


Figure 3

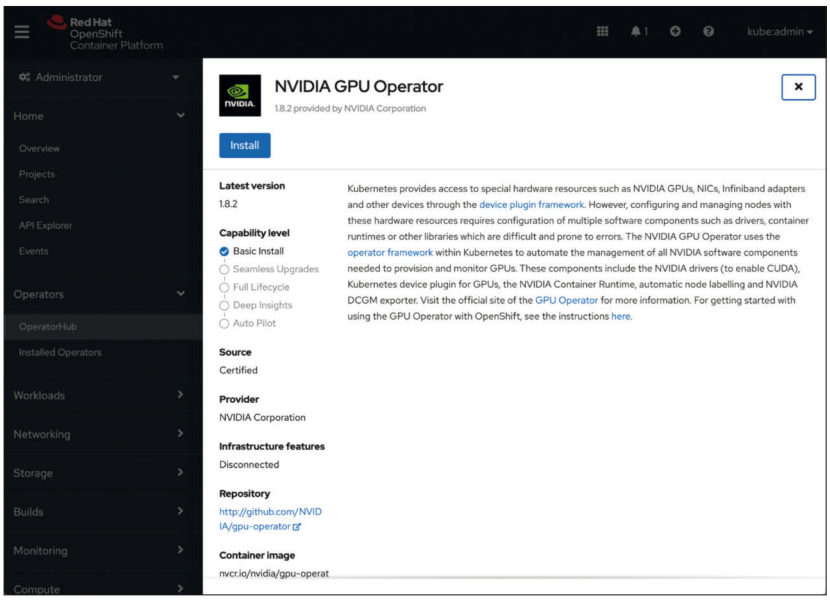


Figure 4

Pour utiliser les GPU NVIDIA avec OpenShift, vous devez installer l'opérateur NVIDIA GPU. Il expose les GPU à Kubernetes en tant que ressources étendues visibles et accessibles dans les pods et les conteneurs. L'opérateur GPU permet à l'administrateur du cluster OpenShift de décider de la géométrie MIG à appliquer aux GPU compatibles, dans un nœud, d'appliquer une étiquette spécifique à ce nœud et d'attendre que l'opérateur GPU reconfigure les GPU et annonce la nouvelle configuration MIG comme ressources à Kubernetes.

Pour installer l'opérateur, vous pouvez utiliser la commande CLI 'oc' ou faire un clic sur "Installer" dans la console OpenShift : **Figure 4**

Toute la documentation sur l'opérateur GPU NVIDIA d'OpenShift est disponible sur https://docs.nvidia.com/datacenter/cloud-native/

La configuration MIG proprement dite est effectuée par un contrôleur Kubernetes exécuté sur chacun des nœuds : le *mig-manager*. Ce contrôleur surveille l'étiquette *nvidia.com/gpu.deploy.mig-manager* du nœud, et lorsqu'il détecte que la valeur a changé, il déclenche l'outil système *nvidia-mig-parted* pour effectuer la reconfiguration réelle du GPU.

Les configurations MIG valides sont stockées comme un objet config-map. Nous avons constaté que l'application 'Traffic Analysis' offre les meilleures performances sur un profil MIG 2g.10gb, et nous allons donc utiliser les commandes suivantes pour configurer l'A100 avec ce même profil.

Voici les étapes de la configuration MIG à appliquer avec OpenShift CLI :

1. Décidez de la configuration à appliquer au nœud à partir de la liste préremplie, ou ajoutez une configuration personnalisée :

```
$ oc edit ConfigMap/mig-parted-config -n gpu-operator-resources
```

2. Appliquez la configuration MIG souhaitée

```
$ MIG_CONFIGURATION=all-2g.10gb
$ oc label node/$NODE_NAME nvidia.com/mig.config=$MIG_CONFIGURATION --overwrite
```

3. Confirmez que les ressources MIG ont été exposées

```
$ oc get nodes/$NODE_NAME -ojsonpath={.status.allocatable}
    | jq -c 'with_entries(select (.key | startswith("nvidia.com")))' | jq
```

Nvidia NGC - Hub for GPU-optimized AI software

Le catalogue NGC est une librairie en ligne gratuite, indispensable pour l'optimisation GPU des applications AI/ML et

leur déploiement sur site, dans le cloud ou en périphérie ('edge'). En plus des frameworks d'apprentissage profond qui sont continuellement optimisés et publiés chaque mois, NGC propose également des modèles pré-entraînés et des scripts de modèles pour divers cas d'utilisation, notamment la vision par ordinateur, le traitement du langage naturel (NLP) et les moteurs de recommandation.

Étape 2 : Obtenir Nvidia DeepStream

DeepStream - SDK pour l'analyse intelligente de la vidéo (IVA)

DeepStream propose des d'outils d'analyse en continu pour le traitement multi capteur basé sur l'IA et la compréhension de la vidéo, de l'audio et des images. En utilisant le SDK DeepStream, les développeurs peuvent construire des pipelines optimisés qui prennent la vidéo en continu comme entrée et produisent des informations issues de traitement IA.

DeepStream supporte C/C++ et Python et propose une intégration clé en main des modèles entrainés avec la boîte à outils Nvidia TAO (Train, Adapt and Optimize).
Pour obtenir le conteneur Docker v5.1 de DeepStream, connectez-vous sur ngc.nvidia.com, allez dans la section conteneurs et recherchez DeepStream. **Figure 5**

Étape 3 : Extraire des modèles pré-entraînés pour la détection et la classification d'objets

La création de modèles nécessite beaucoup de données d'entraînement et une expertise approfondie en IA, et peut prendre plusieurs mois. Pour surmonter ces obstacles, les développeurs peuvent utiliser la plate-forme TAO (Train, Adapt and Optimize) de NVIDIA pour accélérer le processus d'adaptation de leur modèle.

TAO utilise l'apprentissage par transfert - le processus consistant à prendre un modèle pré-entraîné (un modèle qui a été entraîné sur un ensemble de données similaire, pour la même tâche) et à en affiner quelques couches sur vos données personnalisées.

Pour l'application d'analyse du trafic, nous allons utiliser deux modèles pré-entraînés de NGC - TrafficCamNet et VehicleTypeNet et les affiner à l'aide du TAO Toolkit, une solution Jupyter notebook basée sur CLI et composant central de la plateforme TAO. Le premier modèle est un détecteur d'objets qui localise les voitures, les personnes, les motos, etc. dans les rues et le second modèle classe les véhicules en SUV, camions, berlines et autres.
Ces modèles peuvent être extraits à l'aide des commandes données sur leur page Web respective du MBAC :

```
wget https://api.ngc.nvidia.com/v2/models/nvidia/tlt_trafficcamnet/versions/pruned_v1.0/files/resnet18_trafficcamnet_pruned.etlt
wget https://api.ngc.nvidia.com/v2/models/nvidia/tlt_vehicletypenet/versions/pruned_v1.0/files/resnet18_vehicletypenet_pruned.etlt
```

Étape 4 : Optimisation du modèle pour le déploiement :

TensorRT - SDK pour optimiser les modèles

Nvidia TensorRT contient un optimiseur d'inférence pour les modèles d'apprentissage profond entraînés et un runtime pour l'inférence à haut débit et faible latence.

Afin d'optimiser les deux modèles pré-entraînés qui sont utilisés pour notre application, nous utiliserons *trt*exec, un outil qui optimise le réseau en combinant les couches et en optimisant la sélection des noyaux pour améliorer la latence, le débit, l'efficacité énergétique et la consommation de mémoire.

Cet outil est appelé en interne par DeepStream avant d'exécuter l'application. Ainsi, nous spécifions la précision (float16 ou integer8) et la taille du lot (utilisée pour mieux gérer la mémoire), et laissons DeepStream générer un fichier moteur optimisé. Pour cette application, nous avons trouvé que la taille de lot de 35 et 140, pour les modèles TrafficCamNet et VehicleTypeNet respectivement, offrait les meilleures performances de bout en bout.

Étape 5 : Déploiement de l'application à l'aide de DeepStream

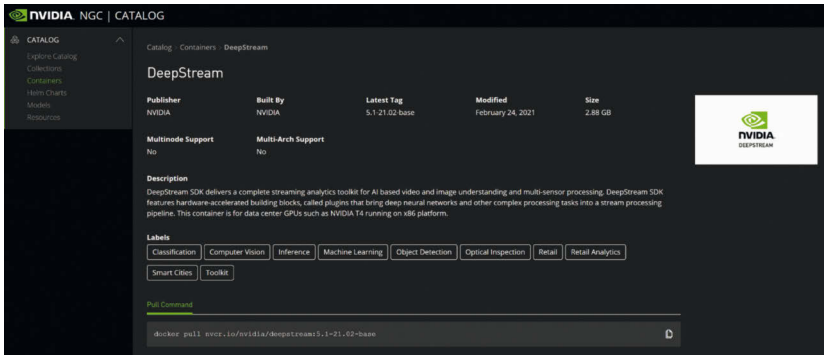
Nous allons utiliser DeepStream pour capturer les flux de caméra, décoder les images vidéo, les prétraiter, les regrouper pour plus d'efficacité et effectuer l'inférence DL, suivre les objets et enregistrer ou afficher le résultat de la boîte englobante.

Pour exécuter l'application, DeepStream a besoin des fichiers de configuration suivants :

- Application config (1) - spécifie le flux d'entrée/sortie, l'architecture du pipeline, le tracker, etc.
- Model config (2) - spécifie les paramètres liés au modèle, les étapes de pré/post-traitement, etc.
- Triton config (2) - spécifie les instances du modèle, le format du modèle, les couches d'entrée/sortie, etc.

Le serveur d'inférence NVIDIA Triton est le backend de DeepStream. Triton permet aux équipes de déployer des modèles d'IA formés à partir de n'importe quel framework (TensorFlow, NVIDIA TensorRT, PyTorch, ONNX Runtime ou personnalisé) à partir d'un stockage local ou d'une plateforme en nuage sur n'importe quelle infrastructure basée sur un GPU ou un CPU (nuage, centre de données ou périphérie).

Figure 5



Au total, nous devons écrire 5 fichiers de configuration. Plusieurs paramètres peuvent être modifiés dans ces fichiers, mais les plus importants sont :

1. Configuration du modèle TrafficCamNet :
   - maximum batch size : la taille du lot à préparer à partir des flux d'entrée - 35
   - operate on gie id : le modèle opère directement sur l'image d'entrée - -1
2. Configuration du modèle VehicleTypeNet :
   - maximum batch size : la taille du lot à préparer à partir du flux d'entrée - 140
   - operate on gie id : le modèle opère sur la sortie du premier modèle - 1
   - operate on class id : le modèle opérerait sur la boîte de délimitation de sortie sélectionnée (objets classés comme voitures) - 0
3. Configuration de l'application Traffic Analyser :
   - number of sources : le nombre de flux d'entrée alimentant l'application - 35
   - streammux batch size : la taille du lot à préparer (à partir de l'entrée) pour alimenter les modèles – 35

Étape 6 : Orchestrer l'inférence avec OpenShift :

Arrivé à cette étape, notre application DeepStream est configurée et les modèles sont optimisés via TensorRT. Afin d'exécuter cette application sur un cluster Red Hat OpenShift avec un nœud GPU 8 x A100, nous utiliserons un fichier de configuration *yaml* de déploiement qui contiendra les détails des GPU et exécutera le script d'automatisation qui effectuera les étapes ci-dessus sur chaque *pod* individuel.

Puisque chaque *pod* exécute l'application séparément, nous n'avons pas besoin de configurer un réseau ou un stockage pour ces *pods*.

La configuration de déploiement pour OpenShift que nous utiliserons est ci-dessous et sera planifiée en utilisant la commande :

```
$ oc create -f deployment.yaml
```

Code complet sur programmez.com & github

Le notebook jupyter démontrant les étapes de configuration d'OpenShift et de déploiement de ce cas d'utilisation est disponible sur https://github.com/AshishSardana/ds_triton/blob/master/traffic-analyser-openshift.ipynb

Conclusion

Si le développement d'une solution basée sur l'IA est complexe, NVIDIA simplifie et accélère ce flux de travail grâce au framework Metropolis. Les scientifiques des données et les développeurs peuvent prendre une longueur d'avance dans la création de modèles personnalisés en affinant les modèles pré-entraînés. En outre, Metropolis prend également en charge l'optimisation et l'inférence des modèles afin que les entreprises puissent commercialiser leurs solutions plus rapidement.

Cet article ne vous a présenté qu'un seul exemple de création d'un service d'IA, mais vous pouvez vous en inspirer pour créer d'autres applications de vision par ordinateur. En fait, le catalogue NGC contient de nombreux SDK, frameworks et Jupyter Notebooks qui vous aideront à démarrer votre développement d'IA pour d'autres cas d'utilisation, notamment l'IA conversationnelle.

NVIDIA AI est conçu pour fonctionner avec OpenShift, ce qui vous donne la flexibilité de développer, déployer et passer à l'échelle vos solutions dans le centre de données, sur le cloud et à la périphérie.

Monitoring in OpenShift

A travers cet article nous allons découvrir ce qu'est le monitoring de systèmes d'information, son intérêt et comment le mettre en place dans OpenShift pour exposer les outils et informations nécessaires aux administrateurs systèmes.

Damien Grisonnet

Damien a rejoint Red Hat en mars 2020 en tant que stagiaire au sein de l'équipe Monitoring d'OpenShift et a depuis transitionné vers une position de Software Engineer toujours au sein de la même équipe. Il s'intéresse principalement à l'instrumentation de Kubernetes et fait partie des leaders dans ce domaine en maintenant des projets tel que kube-prometheus, kube-state-metrics, metrics-server et prometheus-adapter qui sont très populaire dans cet écosystème.

Dans le domaine de l'observabilité de systèmes d'information, on fait référence à trois piliers: monitoring, logging et tracing. Ils sont rarement tous les trois présents au sein d'une même infrastructure. Les plus populaires étant le monitoring et le logging puisqu'ils permettent à eux deux de couvrir une très grande partie des besoins des administrateurs systèmes et des développeurs lors d'une réponse à incident. Le tracing peut également s'avérer primordial pour du debugging même si celui-ci est plus récent et complexe à mettre en place, le rendant un peu moins populaire.

Ces différents vecteurs d'observabilité vont permettent de détecter des problèmes en amont et d'en trouver rapidement les causes. Quand ces causes sont applicatives, il est généralement du ressort des équipes de développement de poursuivre l'investigation afin d'éviter que le problème persiste. Pour ce faire, les équipes ont besoin d'informations précisent quant au fonctionnement de l'application. Celles-ci se retrouvent généralement sous la forme de métriques, logs ou traces. C'est pour cela, qu'il est très important que les développeurs monitorent leurs applications en y ajoutant les informations qui leur seront utiles pour investiguer des bugs.

Dans cet article, nous allons principalement nous intéresser au monitoring puisqu'il s'agit du seul pilier de l'observabilité installé par défaut dans OpenShift. Nous allons notamment essayer de comprendre les raisons de ce choix ainsi que les spécificités de la solution proposée dans OpenShift. Nous allons également détailler comment les développeurs peuvent étendre la couverture monitoring de leurs clusters en monitorant leurs propres applications, les rendant ainsi plus facilement débuguables et observables.

Monitoring

Les solutions de monitoring collectent les métriques exposées par les différents services du système, les stocker et les mettre à disposition des utilisateurs. Cela peut se faire par le biais de langages de requêtage, d'interfaces graphiques et d'alertes émises par la solution. Les métriques sont collectées par la solution de monitoring en continu et vont être utiles pour analyser le système et ses composants en temps réel. Cela va se faire soit en analysant les métriques directement, soit par le biais de dashboards et de graphs qui vont permettre de facilement suivre l'évolution d'une donnée en fonction du temps. Des alertes vont également pouvoir être mise en place à partir des métriques collectées afin de prévenir les personnes en charge du système en cas de comportement inhabituel nécessitant une action de leur part. Le monitoring est un aspect primordial de l'administration système puisqu'il permet de détecter des problèmes, fournir les informations nécessaires à l'investigation de sa cause et enfin sa résolution. Cela permet aux administrateurs d'avoir une vue globale de leurs systèmes tout en s'assurant qu'ils seront en mesure de gérer tout imprévu dans les plus brefs délais.

Parmi les données collectées par le monitoring, les plus importantes pour un administrateur système vont être celles relatives à la santé du système et de ses différents composants puisque ce sont ces informations qui vont lui permettre de mieux comprendre son système. Le monitoring ne se limite pas qu'à la collection de ce type de métriques. Par exemple, une entreprise qui propose un site de commerce en ligne sera potentiellement intéressée par obtenir des informations relatives au nombre de visites des utilisateurs sur leur plateforme ainsi qu'à leurs habitudes de consommation. Dans ce cas, les équipes techniques pourront mettre en place du monitoring pour obtenir ses informations et les transmettre aux personnes qui en auront besoin mais ces informations ne seront d'aucune utilité aux administrateurs système qui seront plus intéressés par la disponibilité du site et de ses différents services.

Dans les systèmes de monitoring moderne, ces données sont collectées sous forme de métriques qui représente une compilation de mesures issues des propriétés techniques ou fonctionnelles d'un logiciel. Par exemple, dans le cas d'un service HTTP, il est intéressant de connaître le nombre de requêtes reçues par l'application. Pour récupérer cette information, elle devra être exposée par l'application qui la mettra à jour à chaque fois qu'elle reçoit une nouvelle requête. Le système de monitoring sera ensuite responsable de collecter cette information à intervalles réguliers. Cependant, connaître uniquement le nombre de requêtes reçues par une application est rarement suffisant, il serait préférable de connaître le nombre de requêtes pour chaque endpoint de l'application accompagné des codes HTTP renvoyés par le serveur pour pouvoir différencier les succès des erreurs. Pour cela, les métriques vont être composés de labels qui vont correspondre à un ensemble de clés et de valeurs apportant davantage d'informations quant à la métrique. En reprenant l'exemple du serveur HTTP, on aurait une métrique correspondant au nombre de requêtes reçues par le serveur par endpoint et codes de retour. Ces métriques une fois exposées par les services et collectées vont être stockées sous forme de timeseries. Chaque timeserie est identifiée par un nom et un ensemble unique de labels. A cette timeserie sera associée la valeur de la métrique et la date à laquelle cette valeur à été mesurée afin de pouvoir par la suite la requêter en fonction du temps.

Alerting

L'alerting vise à faire usage des métriques obtenues sur le système, les évaluer pour obtenir une information précise par

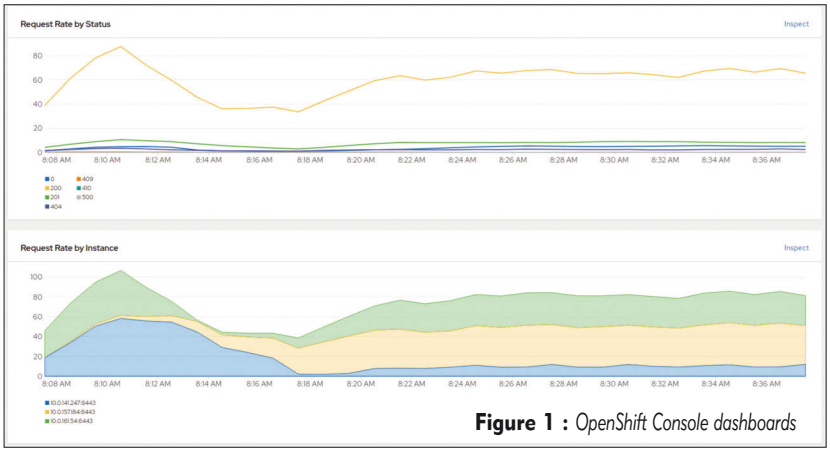


Figure 1 : *OpenShift Console dashboards*

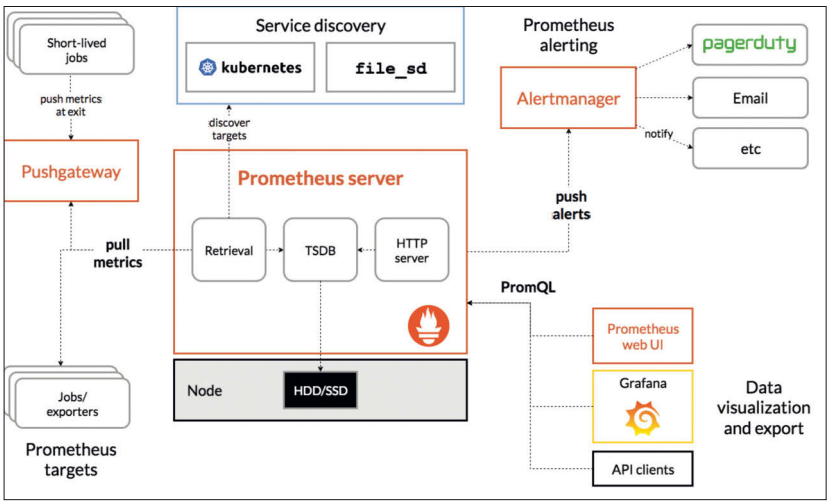


Figure 2 : *Architecture de Prometheus*

rapport au système et définir des seuils permettant de détecter des disruptions ou autres problèmes pouvant arriver à tout moment. Une fois un problème identifié, la solution d'alerting va transmettre l'information aux administrateurs afin qu'ils puissent le résoudre. Cela est essentiel notamment pour les équipes d'astreinte afin d'être informé du moindre problème et de réagir au plus vite en fonction de leurs sévérités. Les canaux de communications utilisés pour transmettre les notifications émises lors de l'apparition d'une alerte sont nombreux, les mails, les messages instantanés sur les applications de communications tel que slack et pour les plus complexe PagerDuty qui permet plus de traitement vis-à-vis des alertes.

Les alertes se présentent sous la forme d'expressions qui vont agréger des métriques et définir des seuils permettant de détecter d'éventuels problèmes. Ces alertes vont être associées à une sévérité qui détermine son impact sur le système. Les équipes d'astreinte se basent notamment sur cette information pour définir s'il est nécessaire de résoudre le problème immédiatement ou d'attendre jusqu'au lendemain. Dans l'idéal les alertes sont associées à des runbooks qui sont des documents expliquant comment résoudre les alertes afin de faciliter la tâche des administrateurs et améliorer leur efficacité. Comme exemple d'alerte critique, l'interruption d'un service très important peut impacter de nombreux utilisateurs et doit donc être résolu dans les plus brefs délais tandis que le fait qu'une machine du système ait atteint plus de 80% d'utilisation mémoire est moins important étant donné que cela n'aura pas un impact immédiat sur les utilisateurs et pourra donc être traité ultérieurement.

Dashboards **Figure 1**

Certaines solutions de monitoring proposent des interfaces utilisateurs et des langages de requêtage permettant de visualiser les métriques de leurs infrastructures. Cependant, cela s'avère souvent primaire et ne permet pas nécessairement de visualiser l'évolution des données en continu. De plus, si un utilisateur veut suivre l'évolution de plusieurs informations en même temps dans une interface unique regroupant toutes les informations utiles sans avoir à changer de fenêtre, cela n'est pas possible par défaut. C'est pourquoi des solutions de visualisation de métriques ont vu le jour et font pleinement partie du monitoring d'un système puisqu'elles permettent aux utilisateurs de suivre en continu ce qu'il se passe dans leurs systèmes.

Prenons l'exemple d'une équipe voulant suivre en continu le fonctionnement d'un de ses services en production. Cette équipe aura besoin d'avoir accès à des informations tel que: la disponibilité du service, son utilisation CPU, mémoire, réseau… Afin d'avoir toutes ces informations à portée de main, cela va nécessiter la mise en place d'une solution de visualisation sur laquelle les utilisateurs pourront créer des dashboards contenant toutes les informations qu'ils ont besoin de suivre en continu. Les informations en elles-même pourront être visibles par le biais de graphiques, nombres, statistiques, ...

Prometheus

Prometheus est un projet open source dans l'instrumentation d'applications qui à été créé par SoundCloud en 2012. De nombreuses entreprises ont adopté Prometheus et le projet dispose d'une communauté de développeurs et d'utilisateurs très active. En 2016, Prometheus a notamment rejoint la Cloud Native Computing Foundation en tant que projet hébergé. **Figure 2**

Prometheus comporte trois composants majeurs :

- **Prometheus server** : responsable de la collecte et du stockage des métriques.
- **Prometheus web UI** : interface utilisateur permettant de visualiser et tracer des courbes à partir des métriques stockées par le serveur.
- **Alertmanager** : permet d'envoyer des alertes provenant de clients tel que le serveur prometheus vers différent receveurs.

Prometheus a pour rôles de collecter des métriques provenant de différents services, évaluer les différentes règles et expressions qui sont configurées en se basant sur les métriques qu'il a stocké et tout ça à des intervalles de temps régulier. A travers l'évaluation des règles d'alerting, Prometheus sera également responsable d'envoyer des alertes à un plateforme capable de les traiter et les transmettre tel que Alertmanager. Les données stockées par Prometheus sont également mise à disposition des utilisateurs via une interface à travers laquelle il est possible de requêter et tracer les données.

Prometheus se distingue par son unique structure de données multidimensionnel qui lui permet de stocker les données de monitoring même si ce format est de plus en plus adopté par les solutions de monitoring. En effet Prometheus traite les données sous forme de timeseries. Celles-ci sont composées d'un

nom de métrique et d'un set de clé/valeur ajoutant des dimensions aux métriques. Prometheus a également son propre langage de requêtage, le Prometheus Query Language, PromQL, qui se base sur la dimensionnalité des timeseries. C'est notamment cette spécificité de la structure de données utilisée par Prometheus qui lui permet d'être puissant et flexible. C'est notamment par le biais de ce langage que les utilisateurs de Prometheus peuvent définir des recording rule responsable d'associer des noms à des expressions et des alerting rules associant des alertes à des expressions. Prometheus est que la collection de timeseries se fait à travers le protocole HTTP, ce qui facilite l'instrumentation de micro-services et leur intégration dans la solution de monitoring. De plus, afin de faciliter la visualisation des métriques et de permettre aux utilisateurs de tracer leurs données de monitoring, Prometheus à sa propre interface qui peut être supplémenter par des solutions de visualisation tel que Grafana qui permettent encore plus de possibilités graphiques à partir des donnés de monitoring. Le format de Prometheus est supporté par une grande majorité des solutions, ce qui laisse beaucoup de choix.

Les métriques n'ont pas de types à proprement parler. Cependant, les bibliothèques permettant de communiquer avec Prometheus définissent quatre type de métriques afin de faciliter leurs utilisations :

- **Counter**: métrique cumulative qui représente un seul compteur à croissance monotone dont la valeur ne peut qu'augmenter ou être remise à zéro au redémarrage du service. Par exemple, vous pouvez utiliser un compteur pour représenter le nombre de requêtes servies, de tâches exécutées ou d'erreurs apparues.
- **Gauge**: métrique qui représente une valeur numérique pouvant varier arbitrairement à la hausse ou à la baisse. Par exemple, le nombre d'utilisateurs en ligne.
- **Histogram**: échantillonne les observations et les compte dans des compartiments configurables. Ce type offre également la somme de toutes les valeurs observées. Par exemple, il va être utilisé pour monitorer la durée des requêtes et ainsi pouvoir déterminer la latence d'unservice.
- **Summary**: similaire à un histogramme, il fournit un résumé des échantillons d'observations. Bien qu'il fournisse également un nombre total d'observations et une somme de toutes les valeurs observées, il calcule des quantiles configurables sur une fenêtre de temps glissante.

Même si cela est transparent pour le serveur Prometheus car les métriques seront stockées sans être typées, cela est vraiment utile lors de l'utilisation des clients. Chaque type de métrique a sa propre API qui facilite la mise à jour de la métrique si elle est correctement typée. De plus, en interrogant l'API HTTP de Prometheus, les utilisateurs peuvent voir le type de chaque métrique, ce qui peut être utile afin de comprendre une métrique.

Alertmanager

Alertmanager est la solution par défaut qui à été choisie dans OpenShift pour faire de l'alerting en raison de sa proximité avec Prometheus. Alertmanager à été développé par la communauté de Prometheus dans l'unique but de s'occuper de l'alerting de Prometheus. De ce fait, c'est la solution par défaut quand il s'agit de faire de l'alerting avec Prometheus. Alertmanager reçoit des alertes provenant de Prometheus et son rôle va être de les transmettre à des receivers. Ces receivers vont être différentes plateformes capables de recevoir les alertes provenant d'alertmanager et de notifier les utilisateurs. Il en existe de nombreux, mais on peut notamment noter, les emails, Slack et PagerDuty dans les principaux. En configurant les receivers, les utilisateurs d'Alertmanager vont pouvoir sélectionner quelles alertes devront être routées vers quel canal de communication. Cela permet notamment de router les alertes vers différentes mailing list ou canal Slack afin de rediriger les alertes vers les canaux les plus adaptés.

Grafana

Grafana est un des projets open source les plus populaires pour visualiser des métriques. Il offre de nombreuses fonctionnalités permettant de visualiser les métriques sur des dashboards pouvant accueillir de multiples interfaces. Il est notamment possible de créer des graphiques, obtenir des statistiques, afficher des valeurs, des jauges et autres. Tout ça regroupant sur une même interface à travers laquelle les utilisateurs peuvent obtenir les informations qui les intéressent en continu. Cela s'ajoute très bien par-dessus Prometheus étant donné que Grafana supporte Prometheus en tant que source de données et que l'interface de base de Prometheus ne permet pas de visualiser correctement nombre d'informations en continu. Cependant, l'interface de Prometheus reste toujours utile si les utilisateurs ont besoin de requêter des métriques particulières. Grafana a longtemps été utilisé dans OpenShift mais il est déprécié dans les versions les plus récentes du produit et est voué à être complètement retiré en faveur de la Console. Cependant même si Grafana ne sera plus déployé dans OpenShift, le projet va toujours être utilisé puisque la Console d'OpenShift se base sur le format de Grafana pour configurer des dashboards.

OpenShift monitoring

OpenShift a fait le choix du monitoring par défaut. Cela veut dire que tout cluster OpenShift comprendra une stack monitoring. Cette stack va fournir les outils pour être informés en cas de problèmes sur leur cluster OpenShift ainsi que les informations dont ils auront besoin pour investiguer et résoudre les éventuels problèmes que leur cluster pourra rencontrer. Il est également possible de visualiser en temps réel de multiples informations relatives à la santé du cluster OpenShift et de ses principaux composants via des dashboards directement intégré à la Console d'OpenShift.

La stack monitoring d'OpenShift se base sur des technologies open source telles que Prometheus, Alertmanager, Thanos et Grafana. **Figure 3**

Depuis OpenShift 4.6, la stack monitoring d'OpenShift se sépare en deux parties. Il y a tout d'abord la stack appelée "Platform Monitoring" installée par défaut sur tous les clusters OpenShift. Elle est responsable de monitorer les applications au cœur du cluster, notamment les services de Kubernetes et de récolter des données de telemetry qui

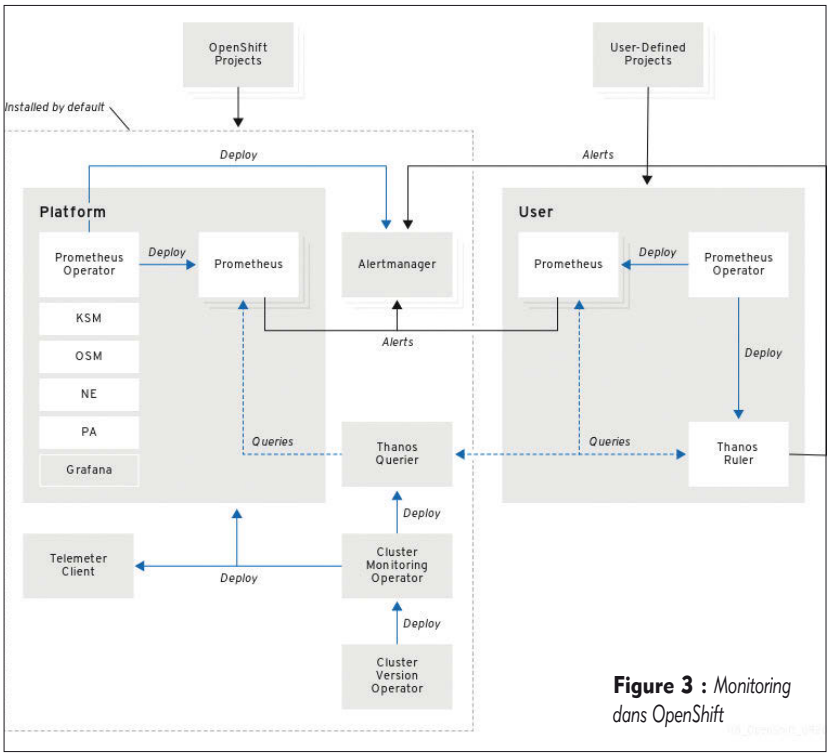


Figure 3 : *Monitoring dans OpenShift*

seront ensuite envoyé à Red Hat. Celle-ci est immuable afin de la protéger contre des mauvaises configurations utilisateur pouvant impacter le monitoring du cluster, cependant quelques aspects de la stack sont configurables par les administrateurs du cluster, notamment pour mettre en place du stockage persistent et d'autres fonctionnalité qui sont décrite dans la documentation officielle. La seconde stack qui n'est pas installée par défaut mais qui peut être activée par les utilisateurs est "User Workload Monitoring" qui comme son nom l'indique va être responsable de monitorer toutes les applications créées par les utilisateurs d'OpenShift. Le but de cette fonctionnalité qui est relativement récent dans OpenShift est de permettre aux utilisateurs de pouvoir intégrer le monitoring de leurs applications aux solutions mises en place dans le cluster et de pouvoir accéder aux différentes informations émises par leurs applications depuis la Console. Cela permet notamment aux administrateurs de pouvoir gérer leurs clusters dans l'intégralité depuis une seule interface, la console d'OpenShift.

Afin de déployer ces fonctionnalités de monitoring dans OpenShift, des opérateurs sont installés dans les clusters pour mettre en place et manager les différents composants de la stack monitoring. En effet, le monitoring ne fait pas exception à la politique générale d'OpenShift de tout installer et manager par le biais d'opérateurs. L'opérateur qui va gérer toute la partie monitoring d'OpenShift est le cluster-monitoring-operator qui est déployé à l'intérieur du namespace "openshift-monitoring" par l'opérateur cluster-version-operator, responsable de l'installation de tous les composants installés par défaut dans OpenShift. C'est cluster-monitoring-operator qui va être responsable de déployer tous les composants nécessaires à la mise en place de la stack monitoring. Il est aussi responsable de synchroniser en permanence les ressources Kubernetes qu'il manage afin de les rendre immuables. En plus de synchroniser les ressources dont il est responsable, il va synchroniser la ConfigMap qui permet aux administrateurs de clusters OpenShift de configurer quelques aspects de la stack monitoring. Cependant, les administrateurs auront uniquement la possibilité de modifier la ConfigMap, tous les autres changements sont appliqués par cluster-monitoring-operator afin de prévenir d'éventuelles erreurs humaines pouvant altérer le bon fonctionnement de la partie monitoring d'OpenShift.

L'opérateur cluster-monitoring-operator installe les deux stacks monitoring dans des namespaces différents afin de les isoler. Tous les composants nécessaires à la mise en place de la partie platform vont être installé dans "openshift-monitoring" tandis que tous les composants permettant de monitorer les applications utilisateurs vont se trouver dans le namespace "openshift-user-workload-monitoring". A noter que ces namespaces étant préfixé par "openshift", ils sont donc uniquement accessibles aux administrateur, ce qui permet d'éviter que des utilisateurs aient accès à de la donnée pour laquelle ils ne sont pas accrédité.

Platform monitoring

Afin de pouvoir monitorer les applications au coeur d'OpenShift dont les services de Kubernetes, la stack platform monitoring va être composée des applications suivantes :

- **Prometheus-operator**: opérateur servant à créer, configurer et manager les instances de Prometheus et d'Alertmanager par le biais de CustomResource Kubernetes. Il va également permettre la gestion des targets de monitoring.
- **Prometheus**
- **Alertmanager**
- **Grafana**
- **Node-exporter**: application qui est installée comme daemonset sur chaque nœud Linux du cluster OpenShift. Elle expose des métriques sur l'OS et le hardware de la machine.
- **Kube-state-metrics**: exporter Prometheus qui expose des métriques sur l'état des objets Kubernetes présent dans le cluster OpenShift. Par exemple des infos sur les pods qui sont présents dans le cluster.
- **Prometheus-adapter**: application qui permet de faire de l'autoscaling dans Kubernetes en se basant sur des métriques Prometheus. Cela est rendu possible via la création de ressource de type HorizontalPodAutoscaling qui vont permettre d'autoscale des applications en se basant sur leurs utilisations CPU/mémoire.
- **Thanos-querier**: interface utilisateur permettant de requêter les métriques présentées sur plusieurs instances de Prometheus. C'est notamment le composant qui sert de backend à la partie Monitoring de la Console OpenShift.
- **Telemeter-client**: client permettant d'envoyer un set défini de métriques vers un serveur de Red Hat. Ces métriques anonymisées sont ensuite traitées par les équipes de Red Hat afin d'établir des statistiques et d'aider dans la résolution de tickets de support. Le but principal étant pour les équipes de RedHat d'améliorer l'expérience des utilisateurs en se basant sur ces métriques.

En plus de ces composants, la stack monitoring par défaut va aussi être responsable de monitorer les applications suivantes :

- CoreDNS
- Elasticsearch (si logging est installé)
- etcd
- Fluentd (si logging est installé)
- HAProxy
- Image registry
- Kubelets
- Kubernetes apiserver
- Kubernetes controller manager
- Kubernetes scheduler
- Metering (si metering est installé)
- OpenShift apiserver
- OpenShift controller manager
- Operator Lifecycle Manager (OLM)

Toutes ces targets du monitoring dans OpenShift exposant des métriques au format Prometheus vont être collectées à intervalles réguliers par les instances de Prometheus se trouvant dans le namespace "openshift-monitoring". Cependant tous ces projets exposent des métriques sensibles et il serait dangereux de laisser ce type de données transiter dans le cluster sans les protéger. Pour ce faire, il est tout d'abord fait en sorte que chaque targets soit collecté via HTTPS afin de ne pas avoir de métrique qui circule en clair dans le cluster. Cependant, cela ne nous assure pas que seules les applications autorisées peuvent collectées les métriques des différents services du cluster. Afin de s'assurer que ce soit le cas, des proxys de type kube-rbac-proxy sont mis en place devant les services pour vérifier que le ServiceAccount qui a émis la requête à les permissions nécessaires pour accéder aux endpoints /metrics des différentes applications. Avec ce système mis en place, on peut s'assurer que les données sensibles de monitoring qui transitent dans le cluster OpenShift sont protégées et uniquement accessibles par les utilisateurs et applications autorisées. Cependant qu'en est t'il des interfaces permettant de visualiser ces métriques ? Les seules applications qui le permettent sont la Console d'OpenShift, l'interface de Prometheus et celle de Thanos Querier. Chacune d'entre elles à un système d'authentification par OAuth proxy qui restreint l'accès aux métriques aux administrateurs du cluster.

Par le biais de cette stack, OpenShift va pouvoir fournir de nombreux outils de monitoring qui vont permettre aux administrateurs de manager les clusters et pouvoir répondre aux éventuels incidents qu'ils rencontrent. Les fonctionnalités de monitoring installées par défaut dans un cluster OpenShift couvre tous les composants qui forment le cœur d'OpenShift et de Kubernetes. Des informations telles que l'utilisation de ressources des différents pods du cluster pourront également être utiles pour obtenir des informations sur les applications installées par les utilisateurs mais cela est insuffisant pour instrumenter et monitorer les applications utilisateurs du cluster. Afin de résoudre ce problème, OpenShift à introduit la fonctionnalité user-workload monitoring qui permet de collecter les métriques exposées par ces applications.

OpenShift User Workload Monitoring

La stack responsable de monitorer les applications utilisateurs est beaucoup plus minimale étant donné qu'elle ne contient pas tous les exporters Prometheus. Elle se compose des services suivants:

- **Prometheus-operator**
- **Prometheus**
- **Thanos Ruler**

L'instance de prometheus-operator déployée par cette stack va permettre aux utilisateurs de créer des custom resources Kubernetes de type ServiceMonitor ou PodMonitor dans le namespace de leur application afin de pouvoir enregistrer leurs targets dans l'instance de Prometheus qui se trouve dans le namespace openshift-user-workload-monitoring. Dès lors, il sera de la responsabilité de l'utilisateur de sécuriser l'accès aux métriques de son application si nécessaire. En plus de pouvoir monitorer leurs applications, les utilisateurs peuvent également créer leurs propres recording et alerting rules via la ressource PrometheusRule. Notons que cette stack met en place des mesures de soft-tenancy pour que les utilisateurs ne puissent pas accéder aux données des autres namespaces pour lesquelles ils ne sont potentiellement pas autorisés. Notamment, lors de l'évaluation des règles, seulement les métriques locales au namespace dans lequel la règle a été créée sont considérées. Cela est forcé directement par la stack monitoring pour contrôler l'accès aux données.

Comment utiliser User Workload Monitoring

Afin de bénéficier de la fonctionnalité user workload monitoring dans OpenShift, il faut l'activer manuellement étant donné qu'elle n'est pas activé par défaut. Cette fonctionnalité demandée et attendue par beaucoup d'utilisateurs d'OpenShift est disponible de manière expérimentale depuis OpenShift 4.3 et est devenue officiellement supporté depuis 4.6. Afin de l'activer, il suffit d'exécuter la commande suivante pour éditer la configmap qui permet de customizer la stack monitoring et y ajouter la configuration ci-dessous:

```
$ oc -n openshift-monitoring edit configmap cluster-monitoring-config

apiVersion: v1
kind: ConfigMap
metadata:
 name: cluster-monitoring-config
 namespace: openshift-monitoring
data:
 config.yaml: |
  enableUserWorkload: true
```

Une fois cette ConfigMap mise à jour, cluster-monitoring-operator va s'occuper de déployer la stack user-workload monitoring dans le namespace openshift-user-workload-monitoring.

Avec cette fonctionnalité plusieurs possibilités s'offrent aux utilisateurs d'OpenShift qui étaient jusqu'à lors impossible avec la stack monitoring de base:

- Monitorer vos propres services
- Créer des alertes et recording rules

Nous allons dans un premier temps voir comment monitorer son propre service à l'intérieur d'OpenShift. Si celui-ci n'expose pas encore de metrics, il va falloir en ajouter. Pour cela, de nombreux clients Prometheus sont disponibles: https://prometheus.io/docs/instrumenting/clientlibs/. Ces clients facilitent considérablement la création, le management et l'exposition de metrics. Il est donc fortement conseillé de les utiliser pour ajouter des métriques et les exposer. Il est également conseillé de suivre les bonnes pratiques définies par Prometheus afin de maintenir des metrics contrainte d'un service à l'autre. Ces bonnes pratiques peuvent être trouver sur https://prometheus.io/docs/practices/naming/.

Les bases de l'instrumentation étant posées, il faut désormais comprendre quelles métriques vont être utiles. Pour cela, il est conseillé de se baser sur trois méthodes :

- The four golden signals - latency, traffic, errors, and saturation. Utile pour micro-services
- The USE method - utilization, saturation, and errors. Utile pour monitorer un système
- The RED method - rate, errors, and duration. Evolution des four golden signals pour micro-services

En se basant sur ces méthodes on obtient les bases qui seront utiles pour connaitre la santé du service ou du système. Pour les services, on peut notamment se servir de ces métriques pour mettre en place des SLO, Service Level Objective et alerter en fonction de la vitesse à laquelle le budget d'erreur est consommé.

Ces méthodes sont génériques et se baser uniquement dessus ne permet pas de couvrir l'entièreté des besoins en monitoring d'une l'application. Il est donc nécessaire de ne pas se limiter qu'à ces bases et à également monitorer les spécificités des applications. Afin de choisir de bonnes métriques il faut d'en un premier temps se demander comment celles-ci vont être utilisées. Elles doivent soit servir à obtenir des informations pertinentes sur l'application tout en s'assurant que celle-ci seront utilisées, soit à permettre de créer des alertes visant à informer les administrateurs de quelconques problèmes. Toute métrique n'ayant pas de scope d'utilisation défini est candidate à devenir inutile et donc, il faut la remettre en question avant même de l'ajouter.
Un autre aspect qu'il faut garder en mémoire avant d'ajouter une nouvelle métrique est ce qu'on appelle sa cardinalité. En effet, l'un des problèmes fréquents qui peut intervenir avec Prometheus lorsque l'on ne fait pas bien attention aux métriques qui sont collecté est que Prometheus va consommer beaucoup trop de mémoire. Cela est souvent causé par des métriques qui génèrent beaucoup trop de séries en mémoire. Précédemment, nous avons évoqué que Prometheus utilise un modèle de données multidimensionnel et stocke les métriques sous forme de timeseries correspondant à un nom de métrique et un set de labels. Cela veut dire que le nombre théorique de timeseries stockées par Prometheus dépend du nombre d'unique combinaison de labels. Par exemple si l'on prend une métrique qui a comme labels le nom d'un pod et son namespace, le nombre théorique de timeseries en mémoire seraient de O(pods)*O(namespaces). Sachant que les valeurs potentielles que peuvent prendre ces labels ne sont pas limitées, il peut y avoir un nombre non borné de timeseries en mémoire. Afin d'éviter cela et se retrouver avec un phénomène d'explosion de cardinalité pouvant engendrer le fait que Prometheus soit OOMKilled, on va toujours chercher à ajouter des labels pouvant prendre un nombre borné de valeurs.
Maintenant que l'on sait comment créer de bonnes métriques, il faudrait créer des alertes pour les utiliser. Pour cela, il existe de nombreuses bonnes pratiques à suivre mais la plus importante concerne ce sur quoi l'alerte se base. On va distinguer deux catégories, les alertes qui se basent sur des causes et celles qui se basent sur des symptômes. Les alertes qui se basent sur des causes sont celles les plus instinctives mais celles qu'il faut éviter au maximum. Un exemple serait une alerte se basant sur le fait qu'une application ne puisse pas se connecter à sa base de données. Un administrateur recevant cette alerte n'aura pas suffisamment d'informations pour comprendre le problème apparent, le forçant donc à chercher de lui-même et perdre un temps précieux. Tandis qu'une alerte se basant sur des symptômes serait par exemple le fait de servir de nombreuses requêtes 4xx et 5xx. En alertant là dessus, l'administrateur serait tout de suite capable d'affiner son investigation en essayant de comprendre pourquoi il y a ces erreurs HTTP. La règle à suivre pour créer de bonnes alertes est donc de se poser la question de qu'est ce qui peut engendrer des problèmes dans mon application plutôt que de se demander pourquoi il y a un problème.

Votre CI/CD dans Kubernetes avec Tetkon

Faisons un petit rappel sur ce que l'on met derrière les termes CI/CD, comment et pourquoi il est intéressant d'intégrer ceux-ci dans Kubernetes.

Vincent Demeester

Vincent est Senior Principal Software Engineer chez Red Hat, et est l'architecte du produit Red Hat OpenShift Pipelines. Présent depuis la création du projet Tekton, il est l'un des « gardiens de temple » de la communauté ainsi qu'un de ces principaux contributeurs. Avant de rejoindre Red Hat, il a notamment travaillé à Docker ainsi qu'à Zenika en tant que consultant et formateur. Il est également un fervent adepte de Linux — et principalement de NixOS dont il maintient un ensemble de packages — et des logiciels libres.

CI/CD désigne des pratiques d'intégration continue (Continuous Integration) et de déploiement continu (Continuous Deployment / Continuous Delivery). En quelques mots, ces pratiques permettent d'augmenter la vitesse de production d'application et la confiance en ce qu'on livre à nos utilisateurs.

Kubernetes est devenu la plateforme de choix pour le déploiement d'applications sur le cloud. Cependant, l'intégration continue (CI) et le déploiement continu (CD) reposent sur des outils largement créés avant l'apparition de nouvelles technologies comme Kubernetes, qui ont changé la donne pour le développement d'applications cloud-native. Le meilleur exemple, *au hasard*, est Jenkins qui est apparu bien avant la création de Kubernetes, Docker et des standards autour des conteneurs. Ces outils nécessitent souvent beaucoup d'adaptations pour être capables de tourner dans Kubernetes et ne sont pas optimisés.

C'est suite à cette constatation que le projet Tekton a été créé, avec comme but d'apporter une intégration optimale dans le monde Kubernetes.

Tekton c'est quoi ?

Tekton est un framework Open Source Kubernetes natif servant à la création de systèmes d'intégration et de livraison continues (CI/CD). Il est composé d'un ensemble de composants dont le principal est Tekton Pipeline. Nous allons nous focaliser sur ce composant pendant la majorité de l'article, et nous introduirons le reste des composants dans la dernière partie de l'article.

Tekton Pipeline est le résultat d'une expérimentation, dans le projet Knative, composant Build, de fournir un système avancé de construction d'image de conteneur. Tekton Pipeline repose sur les principes suivants :

Containers : les conteneurs sont le noyau de tout cluster Kubernetes, et de manière générale de toutes approches « cloud-native ». Tekton est optimisé pour la création, le test et le déploiement d'applications conteneurisées.

Serverless : comme mentionné précédemment, Tekton faisait à l'origine partie de Knative et anciennement connu sous le nom de Knative Build. En tant que *progéniture* Knative, Tekton a hérité d'un ADN intéressant : il fonctionne comme une solution CI/CD sans serveur sans avoir besoin d'un service/moteur central qui nécessite une gestion et une maintenance continues. Le "maître" Tekton est Kubernetes lui-même, car il est implémenté sous la forme d'une série de CRD et de leurs contrôleurs de support.

DevOps : CI/CD a besoin de DevOps, Tekton est conçu pour des équipes distribuées, mais collaboratives et pour des microservices ou même une architecture d'application monolithique plus traditionnelle.

Déclaratif : À l'image de Kubernetes, Tekton apporte une approche déclarative à la définition de pipeline CI/CD.

Composable Tekton insiste sur l'aspect « composable » dans la définition d'une Pipeline comme nous le verrons ci-dessous.

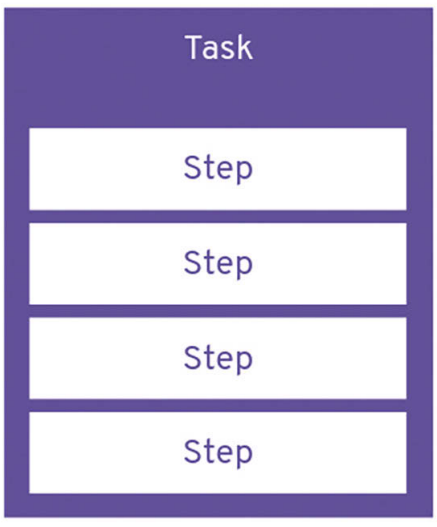
Tekton est une « application » Kubernetes. L'un des avantages de cette approche est que l'on peut utiliser les outils existants et continuer avec Tekton.

Principaux concepts

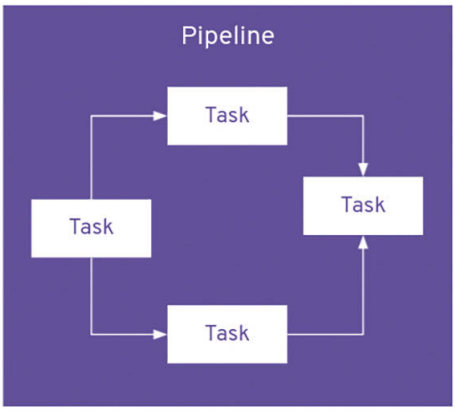
Tekton Pipeline définit un ensemble de concept, et d'objets Kubernetes (CRD) dont les principaux sont les suivants :

Step : C'est la plus petite unité d'exécution présente dans Tekton Pipeline, et correspond à un conteneur. C'est ici que nous allons pouvoir définir quelle commande doit être exécutée, avec quelle image de conteneur, ainsi que l'ensemble des options possible de passer à un conteneur dans Kubernetes. Elle fait partie de l'objet Task.

Task (CRD) : Une « tâche » est un ensemble de Step qui s'exécute de manière séquentielle (l'une après l'autre). Il est possible de définir des paramètres d'entrée, des « workspace » permettant de partager des données (entre tâches…) ou d'y attacher des secrets ou configuration. Il est également possible de définir des résultats, que les Step sont responsables de fournir, et qui peuvent être ensuite réutilisés (en tant que paramètres) par d'autres « tâches ».



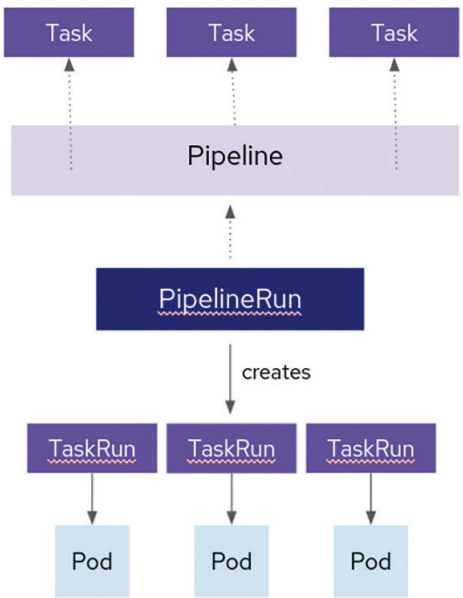
Pipeline (CRD) : Un Pipeline est un ensemble de Tasks qui seront exécutées selon un graph d'exécution. Il est possible de définir des dépendances entre les différentes Task, qui résultent en ce graph d'exécution. À l'instar des Task, il est possible d'y définir des paramètres d'entrée, des « workspace » et des résultats.



Les Task sont voués à être partagées si elles sont assez génériques. Par exemple, une Task qui compile un projet Go sera la même quel que soit le projet. Le projet Tekton propose un catalogue de Task qui est disponible et installable dans n'importe quel cluster Kubernetes où Tekton est installé. Le projet a également un service, Tekton Hub, permettant de naviguer et chercher les Task. Jusqu'ici nous n'avons vu que les objets dit de « définition ». Lorsque l'on crée une Task ou un Pipeline, ils ne sont pas exécutés par Tetkon. Nous avons besoin de créer des objets différents pour démarrer une instance de ces derniers (Task ou Pipeline).

TaskRun (CRD) : Il s'agit de l'exécution d'une Task. C'est sur cet objet que l'on définit les paramètres et les données à utiliser. L'exécution va créer un Pod et monitorer son statut.

PipelineRun (CRD) : Il s'agit de l'exécution d'un Pipeline. À l'instar d'un TaskRun, c'est sur cet objet que l'on définit les paramètres et les données à utiliser.



L'un des avantages de cette approche, est qu'il est possible de définir, des Task ou Pipeline, qui peuvent être relativement générique et configurable à partir des paramètres. Nous allons illustrer ce point lors des prochaines parties de cet article, en construisant un Pipeline qui nous permettra de déployer notre application, constituée de 2 composants — un *frontend* et un *backend*.

Et OpenShift Pipelines dans tout ça ?

Nous n'avons pas parlé uniquement de Tekton depuis le début de l'article, faisons un rapide aparté sur OpenShift Pipelines.

OpenShift Pipelines est une version optimisée pour OpenShift de Tekton. La principale différence est l'installation, facilitée grâce aux opérateurs accessibles directement dans OpenShift et son intégration dans la console d'OpenShift. OpenShift Pipeline ne se contente pas d'intégrer Tekton Pipeline dans OpenShift, mais également le reste des composants fourni par Tekton. Hormis l'installation, il est possible d'intervertir Tekton Pipeline et OpenShift Pipelines dans le reste de l'article.

Déployer notre application avec Tekton Pipeline

Installation

L'installation de Tekton Pipeline est, comme la plupart des projets augmentant l'API de Kubernetes, relativement simple. Il s'agit d'appliquer un fichier yaml à votre cluster.

```
kubectl apply --filename https://storage.googleapis.com/tekton-releases
/pipeline/previous/v0.27.3/release.yaml
```

Cette commande va créer un ensemble de CRD (objets Kubernetes), créer un nouveau namespace nommé tekton-pipelines et y déployer une instance de Tekton, un controller, responsable de *garder un œil* sur les objets Tekton, et un *admission* controller, responsable de valider les objets au moment de leur création.

Tekton nécessite une version minimum de Kubernetes. Par exemple, la version actuelle, 0.27.3 nécessite Kubernetes en version 1.19 pour fonctionner de manière optimale.

Il est possible de configurer certains comportements de l'installation de Tekton grâce aux différentes configurations (configmap) présentes dans le namespace tekton-pipelines.

1 OpenShift Pipelines

Pour installer Tekton Pipeline pour OpenShift, le plus simple est de passer par les Opérateurs via la console. **Figure 1**

2 tkn, la ligne de commande (CLI)

Il est possible d'interagir avec Tekton Pipeline (et les autres composants) avec les outils Kubernetes existants. Cependant, cela peut s'avérer parfois compliqué, notamment pour, par exemple, récupérer les logs d'un Pipeline ou d'une Task. C'est pourquoi Tekton fournit une ligne de commande, nommée tkn, disponible sur GitHub (https://github.com/tektoncd/cli). Elle est disponible sous Linux, Windows et macOS.

Pour plus de détails sur l'installation de cette ligne de commande, rendez-vous sur la documentation. Voici cependant quelques exemples :

```
# Mac OS X
browse-wrapped install tektoncd-cli
# Windows avec chocolatey
choco install tektoncd-cli --confirm
# ... ou avec Scoop
```

```
scoop install tektoncd-cli
# Linux RPMs (Fedora,…)
dnf copr enable chmouel/tektoncd-cli
dnf install tektoncd-cli
```

Pré-requis

Pour la suite de cet article nous partons du principe que Tekton Pipeline est déployé sur Kubernetes. Cela dit, il devrait fonctionner de manière similaire sur OpenShitf Pipelines.

Nos Task

Avant de pouvoir construire notre application, nous allons devoir être capables de récupérer le code source de celle-ci. Pour ce faire, nous avons besoin d'une Task qui fasse un git clone. Cette Task est disponible au sein du hub et est installable de la façon suivante :

```
kubectl apply -f https://raw.githubusercontent.com/tektoncd/catalog/main/task/git-clone/0.4/git-clone.yaml
```

Regardons plus en détail certains aspects de cette Task:

```
apiVersion: tekton.dev/v1beta1
kind: Task (1)
metadata: (2)
 name: git-clone
 labels:
  app.kubernetes.io/version: "0.4"
 annotation:
  tekton.dev/pipelines.minVersion: "0.21.0"
  tekton.dev/categories: Git
  tekton.dev/tags: git
  tekton.dev/displayName: "git clone"
spec:
 # […]
 workspaces: (3)
  - name: output
    description: The git repo will be cloned onto the volume backing this Workspace.
  # […]
 params: (4)
  - name: url
    description: Repository URL to clone from.
    type: string
  - name: revision
    description: Revision to checkout. (branch, tag, sha, ref, etc...)
    type: string
    default: ""
  # […]
 results: (5)
  - name: commit
    description: The precise commit SHA that was fetched by this Task.
  - name: url
    description: The precise URL that was fetched by this Task.
 steps: (6)
  - name: clone
   image: gcr.io/tekton-releases/github.com/tektoncd/pipeline/cmd/git-init:v0.27.3
    env: (7)
    # […]
    - name: PARAM_URL
      value: $(params.url)
    - name: PARAM_REVISION
      value: $(params.revision)
    # […]
    script: | (8)
     #!/usr/bin/env sh
     # […]
     /ko-app/git-init \
      -url="${params.url}" \
      -revision="${PARAM_REVISION}" \
      -refspec="${PARAM_REFSPEC}" \
      -path="${CHECKOUT_DIR}" \
      -sslVerify="${PARAM_SSL_VERIFY}" \
      -submodules="${PARAM_SUBMODULES}" \
      -depth="${PARAM_DEPTH}" \
      -sparseCheckoutDirectories="${PARAM_SPARSE_CHECKOUT_DIRECTORIES}"
     cd "${CHECKOUT_DIR}"
     RESULT_SHA="$(git rev-parse HEAD)"
     # […] (9)
     printf "%s" "${RESULT_SHA}" > "$(results.commit.path)"
     printf "%s" "${PARAM_URL}" > "$(results.url.path)"
```

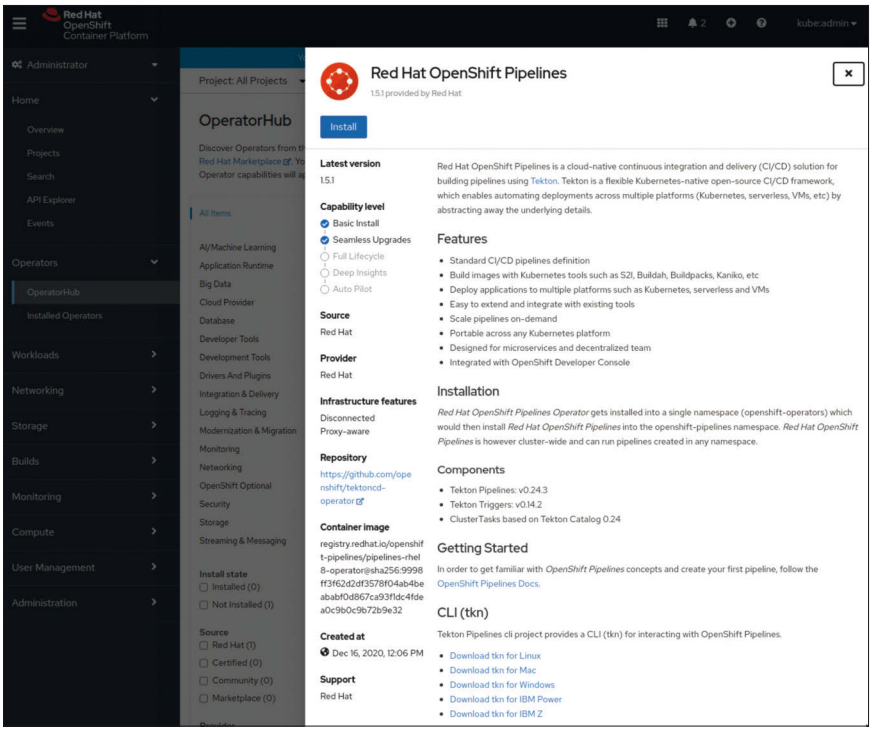


Figure 1

1 Comme tout objet kubernetes, nous avons une apiVersion — ici tekton.dev/v1beta1 — et un kind — ici Task.

2 Il est possible d'attacher un certain nombre de métadonnées à notre Task. Ici, le nom (name) est obligatoire. Dans le cas de cette Task qui est disponible sur le catalogue communautaire de Tekton, nous utilisons des labels et annotations standard, comme par exemple app.kubernetes.io/version pour dénoter la version de notre objet, ou tekton.dev/pipelines.minVersion pour documenter avec quelle version minimum de Tekton Pipeline cette Task est compatible.

3 Cette Task définit un workspace nommé output. Il est possible d'attacher une description pour donner plus d'information de la Task.

4 La liste des paramètres dont cette Task a besoin. Dans notre cas, il s'agit de l'URL à utiliser pour effectuer un git

clone ainsi que la révision (branche, tag, commit,…). Il est possible de définir une valeur par défaut pour les paramètres, qui devient de fait optionnelle.

5 Cette Task définit deux résultats, qui pourront être utilisés pas d'autre Task dans un pipeline par exemple. Ici, il s'agit du commit cloner — utile si l'on utilise une branche comme révision, pour savoir quelle commit de cette branche a été utilisée — et de l'URL « précise » utilisée lors du clone.

6 La liste des Step de notre Task. Ici, nous n'avons qu'une seule Step : clone, responsable de faire un git clone. On peut voir que cette Step utilise une image fournie par Tekton Pipeline.

7 Il est possible de passer des variables d'environnement au processus exécuté dans notre Step. À noter que nous utilisons ici les paramètres comme valeur.

8 Cette Task définit un script, c'est-à-dire un ensemble de commandes à exécuter. On peut noter l'usage de variable d'environnement, ainsi que des variables de paramètres ($(params.url)) et résultats ($(results.commit.path))

9 Ces deux lignes permettent, depuis notre Step d'écrire le contenu d'un résultat, que Tekton Pipeline et d'autres Task utiliseront par la suite.

Pour construire nos images, nous allons utiliser buildah. Cette Task est disponible au sein du hub et est installable de la façon suivante :

```
kubectl apply -f https://raw.githubusercontent.com/tektoncd/catalog/main/task/buildah/0.2/buildah.yaml
```

Regardons plus en détail certains aspects de cette Task:

```
---
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
 name: buildah
 # […]
spec:
 # […]
 params:
 - name: IMAGE
  description: Reference of the image buildah will produce.
 - name: BUILDER_IMAGE
  description: The location of the buildah builder image.
  default: quay.io/buildah/stable:v1.18.0
 # […]
 - name: DOCKERFILE
  description: Path to the Dockerfile to build.
  default: ./Dockerfile
 - name: CONTEXT
  description: Path to the directory to use as context.
  default: .
 # […]
 workspaces: (1)
 - name: source
 - name: sslcertdir
  optional: true
 results:
 - name: IMAGE_DIGEST
  description: Digest of the image just built.
 steps: (2)
 - name: build (3)
  image: $(params.BUILDER_IMAGE)
  workingDir: $(workspaces.source.path)
  script: |
   [[ "$(workspaces.sslcertdir.bound)" == "true" ]] && CERT_DIR_FLAG="--cert-dir $(workspaces.sslcertdir.path)"
   buildah ${CERT_DIR_FLAG} --storage-driver=$(params.STORAGE_DRIVER) bud \
    $(params.BUILD_EXTRA_ARGS) --format=$(params.FORMAT) \
    --tls-verify=$(params.TLSVERIFY) --no-cache \
    -f $(params.DOCKERFILE) -t $(params.IMAGE) $(params.CONTEXT)
  # […]
 - name: push
  image: $(params.BUILDER_IMAGE)
  workingDir: $(workspaces.source.path)
  script: |
   [[ "$(workspaces.sslcertdir.bound)" == "true" ]] && CERT_DIR_FLAG="--cert-dir $(workspaces.sslcertdir.path)"
   buildah ${CERT_DIR_FLAG} --storage-driver=$(params.STORAGE_DRIVER) push \
    $(params.PUSH_EXTRA_ARGS) --tls-verify=$(params.TLSVERIFY) \
    --digestfile $(workspaces.source.path)/image-digest $(params.IMAGE) \
    docker://$(params.IMAGE)
 # […]
 - name: digest-to-results
  image: $(params.BUILDER_IMAGE)
  script: cat $(workspaces.source.path)/image-digest | tee /tekton/results/IMAGE_DIGEST
 # […]
```

1 Tout comme git-clone, la Task buildah accepte des workspaces. Dans ce cas, il s'agit des sources (source), ainsi que d'un workspace optionnel, qui peuvent contenir, dans ce cas, les certificats du registry d'image que l'on souhaite utiliser par exemple.

2 La Task buildah a trois Step : build de l'image, push de l'image (sur un registry d'image), écriture du résultat, ici le *digest* de l'image (permettant d'identifier de manière unique et fiable cette version précise)

3 Il est possible d'utiliser les valeurs des paramètres dans l'image à utiliser. Cela permet, par exemple, de donner la possibilité à l'utilisateur de changer l'image à utiliser, par la sienne.

Une fois que nous avons construit notre image de conteneur, nous devons la déployer. Nous allons avoir besoin d'être capable d'appliquer des fichiers yaml, contenant les objets de service et de déploiement kubernetes. Nous allons utiliser la Task kubernetes-actions disponible dans le catalogue.

```
kubectl apply -f https://raw.githubusercontent.com/tektoncd/catalog/main/task/kubernetes-actions/0.2/kubernetes-actions.yaml
# […]
spec:
 # […]
 workspaces:
  - name: manifest-dir (1)
   optional: true
  - name: kubeconfig-dir (2)
   optional: true
 results:
  - name: output-result
   description: some result can be emitted if someone wants to.
```

```
params:
 - name: script (4)
   description: The Kubernetes CLI script to run
   type: string
   default: "kubectl $@"
 - name: args (5)
   description: The Kubernetes CLI arguments to run
   type: array
   default:
     - "help"
 # [...]
steps:
 # [...]
```

1. Le dossier de base dans lequel se trouve les *manifest* (les fichiers YAML contenant nos déploiements Kubernetes,...).
2. La configuration Kubernetes à utiliser pour déployer nos *manifest*. Cette option est là pour permettre aux utilisateurs d'utiliser la même tâche pour déployer des manifests kubernetes sur n'importe quel cluster Kubernetes ou le cluster Kubernetes courant si aucun workspace n'est utilisé. On utilise, la plupart du temps, les secrets Kubernetes pour ce workspace.
3. Cette Task étant très générique, il est donné à l'utilisateur la possibilité de passer un script comme paramètres (comme nous le verrons dans la définition de notre pipeline)
4. Le paramètre arg a pour valeur help par défaut. L'idée est que si l'on run cette Task sans paramètres script ou args, cette Task exécutera kubectl help.

Pipeline de déploiement

Nous avons à disposition, dans notre cluster Kubernetes, des Task de base qui vont nous permettre de déployer notre application. L'étape suivante consiste à écrire un Pipeline qui va utiliser ces Task pour construire et déployer notre application dans notre cluster.

Le flux de notre Pipeline est relativement simple :

Récupérer les sources (git-clone)

Construire une image (buildah)

Déployer cette image via un manifest (kubectl-action)

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline (1)
metadata:
 name: build-and-deploy
spec:
 workspaces:
 - name: shared-workspace (2)
 params: (3)
 - name: git-url
   type: string
 - name: git-revision
   type: string
   default: "main"
 - name: image
   type: string
 tasks: (4)
 - name: fetch-repository
   taskRef:
     name: git-clone
   workspaces:
   - name: output
     workspace: shared-workspace
   params:
   - name: url
     value: $(params.git-url)
 - name: build-image
   taskRef:
     name: buildah
   params:
   - name: IMAGE
     value: $(params.image)
   workspaces:
   - name: source
     workspace: shared-workspace
   runAfter: (5)
   - fetch-repository
 - name: deploy-app
   taskRef:
     name: kubernetes-actions
   workspaces:
   - name: manifest-dir
     workspace: shared-workspace
   params:
   - name: script
     value: |
       kubectl apply -f manifests/
   runAfter:
   - build-image
```

1. Comme pour une Task, mais cette fois l'objet est Pipeline
2. Le workspace qui sera utilisé pour partager les données — ici les sources de notre application — entre les différents Task. Nous allons l'attacher au différentes Task, qui ont définis différent nom pour celui-ci — par exemple git-clone utilise output, et buildah utilise source.
3. Notre Pipeline définit plusieurs paramètres : git-url pour récupérer nos sources, git-revision pour définir la révision à utiliser — par défaut la branche main sera utilisée — et image pour la référence de l'image de conteneur à utiliser.
4. Notre Pipeline définit trois Task : fetch-repository, build-image et deploy-app.
5. Notre Task build-image a besoin d'être exécutée après avoir récupéré les sources, c'est pourquoi nous utilisons runAfter qui nous permet de dire, exécute-toi une fois les sources récupérées et accessibles dans le workspace partagé.

Exécution du pipeline et déploiement

Nous avons maintenant tous les composantes nécessaires pour déployer notre application dans notre cluster. Il ne nous reste plus qu'à exécuter notre Pipeline en créant un PipelineRun.

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun (1)
metadata:
 generateName: build-deploy-pipelinerun- (2)
spec:
 pipelineRef: (3)
   name: build-and-deploy
```

```
params: (4)
- name: git-url
  value: https://github.com/vdemeester/go-helloworld-app.git
- name: image
  value: quay.io/vdemeest/go-helloworld
serviceAccountName: sa-hub-vdemeester-foo (5)
workspaces:
- name: shared-workspace (6)
  volumeClaimTemplate:
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 500Mi
```

1. Comme pour une Task ou un Pipeline, mais cette fois l'objet est PipelineRun.
2. Il est possible d'utiliser generateName à la place de name pour les PipelineRun et TaskRun ; cela permet de laisser Kubernetes générer un nom avec le prefix souhaité, ce qui s'avère très utile pour démarrer plusieurs fois un PipelineRun.
3. Un PipelineRun référence (aka utilise) un Pipeline, ici nous allons utiliser le Pipeline build-and-deploy.
4. Nous passons les valeurs souhaitées aux paramètres définis par notre Pipeline. Ici nous sources sont disponibles sur le repository https://github.com/vdemeester/go-helloworld-app.git et nous souhaitons publier l'image sur quay.io/vdemeest/go-helloworld
5. Nous souhaitons utiliser un ServiceAccount différent de celui par défaut, pour pouvoir y attacher nos *credential* d'authentification à Quay. Pour plus d'information, il est possible de regarder la documentation de Tekton (https://github.com/tektoncd/pipeline/blob/main/docs/auth.md).
6. L'instance de notre workspace. Ici, nous utilisons un PersistentVolumeClaim d'une taille de 500Mi, que Tekton Pipeline aura la responsabilité de créer au préalable.

Pour créer ce PipelineRun, il suffit de l'appliquer à son cluster:

```
$ kubectl create -f build-and-deploy-pr.yaml
pipelinerun.tekton.dev/build-deploy-pipelinerun-jj59z created
```

Pour suivre l'exécution du PipelineRun, nous pouvons encore une fois utiliser kubectl:

```
$ kubectl get pipelinerun -w
NAME                        SUCCEEDED  REASON   STARTTIME  COMPLETIONTIME
# [...]
build-deploy-pipelinerun-jj59z  Unknown    Running  44s
build-deploy-pipelinerun-jj59z Unknown    Running  45s
build-deploy-pipelinerun-jj59z Unknown    Running  84s
build-deploy-pipelinerun-jj59z True       Succeeded  86s      0s
```

Si vous avez installé tkn, il est possible de suivre, en direct, l'exécution du PipelineRun

```
$ tkn pipelinerun logs -f build-deploy-pipelinerun-jj59z
# [...]
[fetch-repository : clone] + /ko-app/git-init '-url=https://github.com/vdemeester/go-helloworld-app.git' '-revision=' '-refspec=' '-path=/workspace/output/' '-sslVerify=true' '-submodules=true' '-depth=1' '-sparseCheckoutDirectories='
[fetch-repository : clone] {"level":"info","ts":1631694832.8834538,"caller":"git/git.go:169","msg":"Successfully cloned https://github.com/vdemeester/go-helloworld-app.git @ dd9473987aa399172626520ed6e837e0de7b4569 (grafted, HEAD) in path /workspace/output/"}
[fetch-repository : clone] {"level":"info","ts":1631694832.900327,"caller":"git/git.go:207","msg":"Successfully initialized and updated submodules in path /workspace/output/"}
# [...]
[build-image : build] + buildah --storage-driver=overlay bud --format=oci --tls-verify=true --no-cache -f ./Dockerfile -t quay.io/vdemeest/go-helloworld .
[build-image : build] STEP 1: FROM docker.io/library/golang:1.16 AS builder
[build-image : build] Getting image source signatures
[build-image : build] Copying blob sha256:911ea9f2bd51e53a455297e0631e18a72a86d7e2c8e1807176e80f991bde5d64
# [...]
[build-image : build] STEP 7: ENTRYPOINT ["/helloworld"]
[build-image : build] STEP 8: COMMIT quay.io/vdemeest/go-helloworld
[build-image : build] --> aade6b0f8ed
[build-image : build] aade6b0f8ed34111cab39f981440b9fcd6cdc658bddb15c8407780e67c4d5d09
# [...]
[build-image : push] + buildah --storage-driver=overlay push --tls-verify=true --digestfile /workspace/source/image-digest quay.io/vdemeest/go-helloworld docker://quay.io/vdemeest/go-helloworld
# [...]
[build-image : push] Storing signatures
[build-image : digest-to-results] + cat /workspace/source/image-digest
[build-image : digest-to-results] + tee /tekton/results/IMAGE_DIGEST
[build-image : digest-to-results] sha256:1c43f2938af866b9caa1c56cbedaa49847be71392a4b1b498cc8f902d69ace77
[deploy-app : kubectl] service/helloworld created
[deploy-app : kubectl] deployment.apps/helloworld created
```

Et voilà, nous avons construit une image, envoyé cette image sur un registry d'images de conteneurs et nous avons déployé notre application.

Aller plus loin avec Tekton Pipelines

Nous avons à peine effleuré Tekton. Nous avons défini et exécuté un Pipeline, qui nous a permis de construire et déployer notre application, le tout dans Kubernetes. Grâce aux différents composants mis à disposition par Tekton, il est possible d'aller beaucoup plus loin dans la mise en place de notre système CI/CD. Listons quelques exemples de composants et leurs usages.

Triggers (https://github.com/tektoncd/triggers)
Chains (https://github.com/tektoncd/chains)
Results (https://github.com/tektoncd/results)
Dashboard (https://github.com/tektoncd/dashboard)
Workflows (https://github.com/tektoncd/community/issues/464)
Pipeline as Code (https://github.com/openshift-pipelines/pipelines-as-code/)

Ainsi se termine notre brève introduction à Tekton. Pour plus d'information sur le sujet :

- le site web : https://tekton.dev
- le project GitHub : https://github.com/tektoncd

Opérateur autonome Couchbase : comment déployer et gérer le cycle de vie d'une base de données NoSQL sur OpenShift

Couchbase Server est une base de données distribuée NoSQL orientée documents. Elle permet le stockage de documents JSON ou binaires sous forme clé/valeur. L'entité logique de stockage de ces documents est la collection (équivalent de la table pour un SGBD Relationnel) au sein d'un bucket (équivalent d'une base de données pour un SGBDR).

Fabrice Leray

Architecte solutions chez Couchbase France, j'accompagne mes clients dans la définition de l'architecture et le déploiement de bases de données applicatives pérennes. En complément des déploiements sur Kubernetes/OpenShift, les problématiques IoT et de synchronisation Mobile sont au cœur des projets Couchbase retenus par mes clients.

Background : développeur ayant versé dans la cartographie numérique une dizaine d'années, puis le NoSQL.

Le requêtage des documents JSON peut également se faire via les moteurs N1QL (SQL pour JSON) ou FTS (Full Text Search). D'autres services intégrés tels l'Eventing (qu'on peut rapprocher à la notion de trigger) et Analytics (utilisé pour la BI et le reporting) sont également fournis nativement avec le composant Couchbase Server.

En complément, Couchbase offre une solution de déploiement et de gestion d'un cluster Couchbase Server Enterprise Edition sur Kubernetes/Openshift via l'opérateur autonome *Couchbase Autonomous Operator* (ou CAO).

Cet article présente la mise en œuvre du CAO pour le déploiement associé d'un cluster Couchbase Server, le tout hébergé sur OpenShift.

La notion d'opérateur pour Kubernetes/OpenShift est présenté dans un autre article dans ce numéro. Nous verrons dans cet article comment :

- Déployer les composants nécessaires à la gestion du cluster et de ses buckets
- Instancier un cluster Couchbase sur OpenShift et options avancées (haute disponibilité, stockage persistent, montée de version, autoscaling)
- Instancier un pod "side-car" respectivement pour le monitoring et log forwarding

Pré-requis :

- Avoir un compte Red Hat afin d'accéder au catalogue Red Hat (https://catalog.redhat.com/) contenant les images instanciées par les containers
- Créer un cluster OpenShift, de préférence multi-zones (multi-AZ) dans une région donnée. Un minimum de 3 AZ est conseillé afin de tirer partie des fonctionnalités de haute disponibilité au niveau Server Group de Couchbase Server (voir le paragraphe Mise en oeuvre de la haute disponibilité).
- Assurez-vous de la compatibilité de la version de l'opérateur deployé et de Couchbase Server ainsi que celle de la version d'OpenShift (cf. https://docs.couchbase.com/operator/current/prerequisite-and-setup.html)
- Avoir un fournisseur de stockage dynamique (*dynamic storage provisioner*) couplé à OpenShift. A noter que ce sera automatiquement le cas si vous utilisez OpenShift avec les cloud provider AWS, Azure ou encore Google Cloud.

Note 1 : *les différents fichiers YAML utilisés dans cet article sont disponibles sur :*https://github.com/couchbaselabs/programmez-magazine

Note 2 : *il existe un chart Helm pour faciliter le déploiement du cluster Couchbase Server et intégrant le déploiement du composant serveur additionnel Sync Gateway (pour l'usage du Mobile).Il utilise lui aussi le CAO Couchbase et est disponible à cette URL :*https://github.com/couchbase-partners/helm-charts

Installation de la boîte à outils

Le package "Cloud-Native Database", est téléchargeable sur https://www.couchbase.com/downloads et contient tout l'outillage nécessaire à l'instanciation et à la gestion du cycle de vie d'un cluster Couchbase sur OpenShift, à savoir :

- Le fichier `crd.yaml` qui définit ensemble des *Custom Resource Definitions (CRD)* nécessaires à la création des ressources Couchbase
- Le binaire `cbopcfg` contenu dans le répertoire `bin` du package, responsable de l'instanciation des `deployments` suivants :
  - Dynamic Admission Controller
  - Couchbase Autonomous Operator
- Le rôle nécessaire à la création du cluster Couchbase.

Notes : *Le package est disponible pour les plateformes Kubernetes et OpenShift. Dans notre cas, on veillera donc à se positionner sur l'onglet "Couchbase Autonomous Operator (Red Hat OpenShift)" de la page de téléchargements,Veillez ensuite à sélectionner le package correspondant à la version la plus récente (2.2.1) et votre OS d'administration du cluster OpenShift (macOS, macOS ARM 64, Windows ou Linux).*

CRD Couchbase

Définition

Le fichier `crd.yaml` définit l'ensemble de *Custom Resource Definitions (CRD)* des types de ressources Couchbase telles que `CouchbaseCluster`, `CouchbaseBucket`, `CouchbaseBackup`…

Ce fichier de définition est lié à une version donnée de l'opérateur.

Une fois installées, ces ressources Kubernetes/Openshift personnalisées sont alors instanciables au sein de Kubernetes (tout comme l'est un `deployment`, `pod`, `service` etc…) via l'opérateur autonome Couchbase (CAO).

Installation

```
oc create -f crd.yaml
```

En sortie, pour la version 2.2.1 du package, 11 ressources Couchbase sont créées

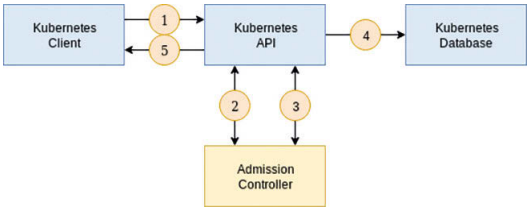
```
customresourcedefinition.apiextensions.k8s.io/couchbaseautoscalers.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbasebackuprestores.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbasebackups.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbasebuckets.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbaseclusters.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbaseephemeralbuckets.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbasegroups.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbasememcachedbuckets.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbasereplications.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbaserolebindings.couchbase.com created
customresourcedefinition.apiextensions.k8s.io/couchbaseusers.couchbase.com created
```

Dynamic Admission Controller

Définition

Le package contient également le binaire cbopcfg responsable de l'instanciation du deployment du *Dynamic Admission Controller* (DAC) : ce composant (obligatoire) peut être vu comme un simple serveur web. Il agit comme un garde-fou et permet la validation de la configuration (fichier YAML) de votre cluster et autres ressources Couchbase, tâche préalable avant délégation de leur création/modification via l'opérateur autonome. Il intercepte tout changement réalisé sur les ressources Couchbase.

Ainsi 2 *webhooks* sont en écoute au niveau de l'API Kubernetes :



1. Un client se connecte à l'API Kubernetes et crée une ressource issue des CRD Couchbase (CouchbaseCluster, CouchbaseBucket…).
2. Un premier *mutating webhook* est alors déclenché : il est chargé de l'**affectation** des valeurs par défaut et complémente donc la ressource soumise.
3. Le second *validating webhook* réalise quant à lui la **validation** des valeurs soumise par le client (au regard du schéma JSON contenu dans les CRD)
4. Une fois les contrôles passés avec succès, la ressource peut alors être créée dans la base de données interne à Kubernetes/OpenShift (etcd).
5. L'API Kubernetes renvoie un statut OK suite à la création de la ressource.

En cas d'erreur de configuration, un message descriptif est alors émis par ce composant dans la sortie standard.

Tout changement de configuration d'un cluster Couchbase Server via l'API Kubernetes (réalisés par les commandes kubectl create, kubectl edit, and kubectl apply) passe donc par une revalidation.

Déploiement du DAC

Il est recommandé de déployer 1 instance unique du DAC par cluster Kubernetes/OpenShift. Dans notre cas, nous l'isolons dans le *project* default.

La connexion au catalogue Red Hat requiert, pour un utilisateur OpenShift non cluster-admin, que les déploiements aient les droits d'accès permettant d'accéder aux images des containers (dont les images Couchbase Server, Couchbase Exporter, OpenShift FluentBit par exemple). Ces login/mot de passe sont stockés sous la forme de *Secret* OpenShift :

```
oc create secret docker-registry rh-catalog --docker-server=registry.connect.redhat.com \
 --docker-username=<rhel-username> --docker-password=<rhel-password> --
docker-email=<docker-email>
```

```
bin/cbopcfg create admission -n default --image-pull-secret rh-catalog
```

```
serviceaccounts/couchbase-operator-admission created
clusterroles/couchbase-operator-admission created
clusterrolebindings/couchbase-operator-admission created
secrets/couchbase-operator-admission created
deployments/couchbase-operator-admission created
services/couchbase-operator-admission created
validatingwebhookconfigurations/couchbase-operator-admission created
mutatingwebhookconfigurations/couchbase-operator-admission created
```

Vérifier que le pod du DAC est bien présent :

```
oc get pod -n default | grep admission
```

```
couchbase-operator-admission-5fbff9c8d-db87h  1/1   Running  0       69m
```

Couchbase Autonomous Operator

Définition

Le même binaire cbopcfg permet la création du *Couchbase Autonomous Operator (CAO)*, responsable du déploiement et de la gestion du cycle de vie d'un cluster Couchbase au sein d'un cluster Kubertenets/Openshift.

Déploiement du CAO

Contrairement au DAC, il est recommandé (mais pas obligatoire) de déployer 1 CAO au sein d'un *namespace* Kubernetes (*project* OpenShift) dédié.

Ainsi on a donc 1 CAO pour un nombre limité de clusters Couchbase, afin que, lors d'une montée de version de l'opérateur, seuls les pods des clusters Couchbase gérés par cet opérateur soient mis à jour.

Note : *un pattern courant de déploiement consiste en 1 project = 1 instance CAO = 1 cluster Couchbase. Le découpage par projects peut alors correspondre aux plateformes de déploiements classiques : DEV, RECETTE, PRE-PROD, PROD.Pour aller plus loin : https://docs.couchbase.com/operator/current/concept-operator.html*

```
oc new-project programmez
bin/cbopcfg create operator -n programmez --image-pull-secret rh-catalog
```

Vérifier que le pod du CAO est bien présent :

```
oc get pod -n programmez | grep operator
```

```
couchbase-operator-6648df9b7f-wlnzg  1/1   Running  0       56m
```

Installation des permissions utilisateurs

Par défaut sur les plateformes OpenShift, les utilisateurs "non-admin" du cluster Openshift n'ont pas le droit de créer/modifier le CRD Couchbase. Le rôle spécifique couchbase-cluster, contenu dans le fichier cluster-role-user.yaml permet aux utilisateurs en héritant d'accéder aux CRD Couchbase.
Pour se faire, créer le rôle couchbasecluster dans OpenShift :

```
oc create -f cluster-role-user.yaml
```

Pour associer ce rôle avec l'utilisateur *foo* dans le namespace *programmez*, exécuter la commande suivante :

```
oc create clusterrolebinding foo-couchbasecluster --namespace programmez
--user foo --clusterrole couchbasecluster
```

Synthèse de l'installation des composants

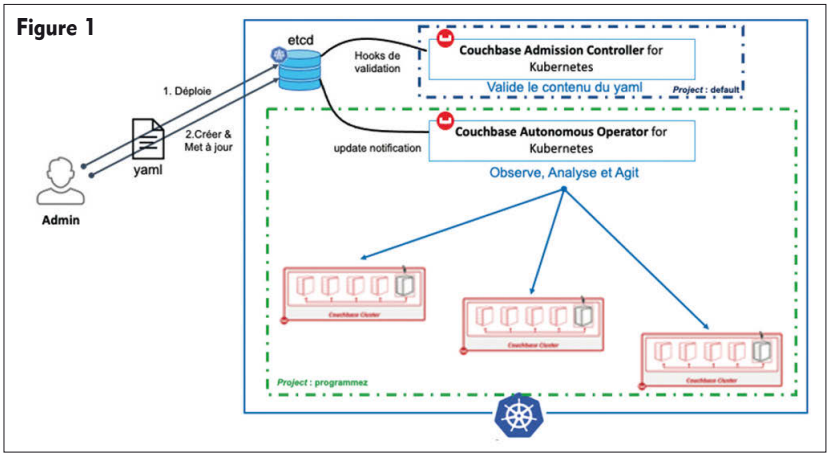
Le cluster OpenShift est prêt à accueillir l'hébergement de vos clusters Couchbase Server. Dans le diagramme ci-dessous, les CRD, le DAC et le CAO sont installés. Les clusters Couchbase (composants rouges) restent à déployer. **Figure 1**

Déploiement du cluster Couchbase "Hello Couchbase Cluster"

Également fourni dans le package, le fichier couchbase-cluster.yaml décrit le déploiement de notre premier cluster Couchbase. Ce fichier est concis et contient une configuration volontairement basique :

```
1.  apiVersion: v1
2.  kind: Secret
3.  Metadata:
4.   name: cb-example-auth
5.  type: Opaque
6.  Data:
7.   username: QWRtaW5pc3RyYXRvcg== # Administrator
8.   password: cGFzc3dvcmQ=       # password
9.  ---
10. apiVersion: couchbase.com/v2
11. kind: CouchbaseBucket
12. Metadata:
13.  name: default
14. ---
15. apiVersion: couchbase.com/v2
16. kind: CouchbaseCluster
17. Metadata:
18. name: cb-example
19. Spec:
20.  image: registry.connect.redhat.com/couchbase/server:6.6.2-1
21.  Security:
22.   adminSecret: cb-example-auth
23.  Buckets:
24.   managed: true
25.  Servers:
26.  - size: 3
27.   name: all_services
28.   Services:
29.   - data
30.   - index
31.   - query
32.   - search
```

Figure 1




```
33.  - eventing
34.  - analytics
```

Lignes 1 à 8 la définition d'un *Secret* OpenShift cb-example-auth, référencé ligne 22 et servant de compte/mot de passe au serveur Couchbase Server, composé de 3 nœuds identiques exposant les 6 services Couchbase (lignes 26 à 34).
La gestion des *buckets* est également déléguée à l'opérateur autonome (ligne 23 et 24), permettant la création d'un bucket default (lignes 10 à 13).
Vous pouvez déployer ce cluster ainsi :

```
oc create -f couchbase-cluster.yaml -n programmez
```

Et superviser le déploiement des pods du cluster Couchbase :

```
oc get pods -n programmez -w
```

```
NAME                                  READY   STATUS    RESTARTS   AGE
cb-example-0000                       1/1     Running   0          36m
cb-example-0001                       1/1     Running   0          35m
cb-example-0002                       1/1     Running   0          34m
couchbase-operator-6648df9b7f-wlnzg   1/1     Running   0          71m
```

Pour accéder rapidement à la console via *port-forwarding* du port 8091, exécuter dans un terminal séparé :

```
oc port-forward cb-example-0000 8091:8091 -n programmez
```

Puis se connecter sur http://localhost:8091/ui/index.html (credentials Administrator/password) : **Figure 2**

```
oc get service -n programmez
```

```
NAME                  TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
cb-example            ClusterIP   None            <none>        4369/TCP,8091/TCP,8092/TCP,8093/TCP,8094/TCP,8095/TCP,8096/TCP,
9100/TCP,9101/TCP,9102/TCP,9103/TCP,9104/TCP,9105/TCP,9110/TCP,
9111/TCP,9112/TCP,9113/TCP,9114/TCP,9115/TCP,9116/TCP,9117/TCP,
9118/TCP,9120/TCP,9121/TCP,9122/TCP,9130/TCP,9140/TCP,9999/TCP,
11207/TCP,11209/TCP,11210/TCP,18091/TCP,18092/TCP,18093/TCP,18
094/TCP,18095/TCP,18096/TCP,19130/TCP,21100/TCP,21150/TCP   6m52s
cb-example-srv        ClusterIP   None            <none>        11210/TCP,11207/TC P   6m52s
couchbase-operator    ClusterIP   172.30.24.239   <none>        8080/TCP,
8383/TCP
```

Afin de simuler une charge sur le serveur, on instancie le *Job* Pillowfight fourni dans le package (fichier *pillowfight-data-loader-openshift.yaml*). Pillowfight est un utilitaire fourni avec le binaire Couchbase Server qui permet de simuler une char-

ge de requêtage en lecture/écriture de type clé/valeur (https://docs.couchbase.com/sdk-api/couchbase-c-client/md_doc_cbc-pillowfight.html).

Le service cb-example-srv exposé par OpenShift est accessible depuis les autres *pods* du clusters, le port 11210 étant le port du service Data (11207 également, pour la version chiffrée).

```
oc create -f pillowfight-data-loader-openshift.yaml -n programmez
```

Après quelques secondes, le nombre de documents dans le buckets s'accroît (jusqu'à 10000 comme paramétré dans le script) :

```
apiVersion: batch/v1
kind: Job
metadata:
 name: pillowfight
spec:
 template:
  metadata:
   name: pillowfight
  spec:
   containers:
   - name: pillowfight
    image: couchbaseutils/pillowfight:v2.9.3
    command: ["cbc-pillowfight",
          "-U", "couchbase://cb-example-srv/default?select_bucket=true",
          "-I", "10000", "-B", "1000", "-c", "10", "-t", "1", "-u", "Administrator",
"-P", "password"]
   restartPolicy: Never
```

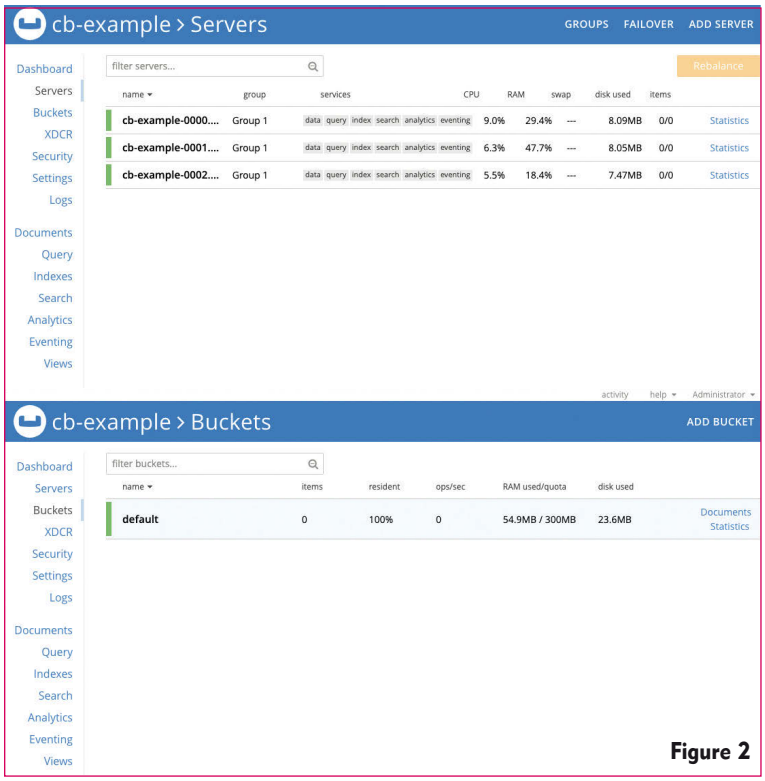


Figure 2

Figure 3

Note : *en changeant la valeur "-c","10" à "-c","-1" vous pouvez simuler une charge infinie dans le temps (limité à 10000 documents car "I", "10000"). Pour accroître la charge, on peut jouer sur le paramètre "-t" qui commande le nombre de threads à instancier.*

Scénario de la perte d'un pod

Avant de poursuivre, nous allons changer le paramètre de durée avant déclenchement de l'auto-failover (propriété autoFailoverTimeout positonnée à 120s étant valeur par défaut) pour le passer à 10s (plus réaliste) :

```
spec: ... cluster:   autoFailoverTimeout: 10s
oc apply -f couchbase-cluster.yaml -n programmez
```

Nous allons alors simuler la perte d'un pod du cluster en supprimant brutalement le pod cb-example-0001 (par exemple).

```
oc delete pod cb-example-0001 -n programmez
```

Couchbase Server détecte la perte d'un nœud (*pod*) du cluster. L'auto-failover est déclenché après autoFailoverTimeout secondes : **Figure 3**

L'opérateur autonome Couchbase :

1. Détecte la rupture de contrat (décrit dans le YAML)
2. Ré-instancie, en réponse, le *pod* manquant et lui fait rejoindre le cluster existant.
3. Pour finalement déclencher l'opération de *rebalance* (pour le service *data*, cela consiste en un rééquilibrage/redistribution des documents/réplicas sur les 3 noeuds du clusters).

Bonnes pratiques de déploiement et haute disponibilité

Isolation des pods et des clusters Couchbase

Les problématiques de consommation partagée de ressources CPU/RAM pour un *worker node* donné nécessite de *scheduler* intelligemment les pods Couchbase.

Ainsi en PROD, le paramètre couchbaseclusters.spec.antiAffinity doit-il être positionné à true afin de garantir que 2 pods Couchbase ne soient pas co-localisés sur le même *worker node*.

Afin d'assurer un déploiement strict des pods Couchbase (et des autres pods), on peut également, dans le même ordre d'idée :

- Définir des pods *tolerations* (couchbaseclusters.spec.servers.pod.spec.tolerations) en positionnant des *taints* (https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/) sur les nœuds Kubernetes/OpenShift dédiées au clusters Couchbase Server
- Référencer ces dernières dans le fichier de configuration du cluster via le *nodeSelector* (couchbaseclusters.spec.servers.pod.spec.nodeselector)

Plus d'informations sur :
https://docs.couchbase.com/operator/current/concept-scheduling.html

Respect du *multidimensional scaling* (MDS)

Il est recommandé d'isoler les différents types de services Couchbase par *pod*. Un pattern de déploiement classique peut donc se composer de 3 *pods* data, 2 *pods* query/index et 2 *pods* FTS.

Note : *comme vous l'aurez remarqué, il existe une exception pour les services query et index qui peuvent, dans la majorité des cas, être co-localisés sur un même pod car ils n'accèdent pas aux mêmes ressources. Le service query (moteur des requêtes N1QL) sollicitera principalement la CPU du pod alors que les index seront consommateurs de sa RAM.*

Haute disponibilité des pods

Couchbase Server supporte nativement la perte d'un nœud ou d'un groupe de nœuds. Le mécanisme de haute disponibilité s'appuie sur :

- La notion de réplicas : pour chaque document d'un bucket, 0 à 3 réplicas peuvent être créés. Ceux-ci sont naturellement hébergés sur un/plusieurs nœud(s) annexe(s) au nœud contenant le document actif.
- La détection d'une panne et le déclenchement de l'auto-failover :
  - À la perte d'un noeud (node auto-failover)
  - À la perte d'un groupe de nœuds, groupe qu'on associe à un rack, data center ou encore Availability Zone dans le Cloud. On parle alors de "*auto-failover Server Group*".

Dans les 2 cas et afin de pallier toute problématique de split-brain, les conditions de déclenchement de l'auto-failover sont directement liées au nombre de participants : il faut a minima 3 noeuds dans le cluster et 3 groupes de noeuds pour respectivement déclencher l'auto-failover *node* et *server group*. Pour plus de détails sur le sujet et notamment les autres conditions de déclenchement de l'auto-failover, se référer à cette page : https://docs.couchbase.com/server/current/learn/clusters-and-availability/automatic-failover.html#auto-failover-constraints

Le CAO va plus loin **en automatisant la reprise après incident.** En effet, si l'auto-failover (*node* ou *server group*) est natif Couchbase, les actions de résilience telles que la création d'un nouveau *pod* (remplaçant le *pod* défaillant), son intégration au cluster et le déclenchement du ré-équilibrage (*rebalance*) sont des tâches qui incombent à l'opérateur Couchbase.

Ces 2 points se traduisent ainsi dans la configuration de notre cluster Couchbase Server :

```
spec :
 ...
 servers:
 - size: 3
   name: data
   services:
   - data
   serverGroups:
    - eu-west-1a
    - eu-west-1b
    - eu-west-1c
   volumeMounts:
     default: pvc-default
 - size: 2
   name: query_index_ensemble
   services:
   - index
   - query
   serverGroups:
    - eu-west-1b
    - eu-west-1c
   volumeMounts:
     default: pvc-default
 - size: 2
   name: fts_seul
   services:
   - search
   serverGroups:
    - eu-west-1a
    - eu-west-1c
```

Note : *le nom des serverGroups doit correspondre aux zones de votre fournisseur cloud. Pour les obtenir :*

```
oc describe nodes | grep -e "Name:" -e "failure-domain.beta.kubernetes
.io/zone"
```

Dans le cas où votre plateforme ne fournit pas de zone pour les worker nodes, il est également possible de les labelliser. Voir la documentation: https://docs.couchbase.com/operator/current/howto-server-groups.html

Haute disponibilité du stockage

Avant d'évoquer le scénario de perte d'un nœud du cluster Couchbase, il convient préalablement de présenter ce que sont les PV (Persistent Volume) dans Kubernetes/OpenShift et pourquoi ils jouent un rôle déterminant pour Couchbase Server.

Couchbase Server est une base de données et donc un élément logiciel qu'on peut qualifier de *Stateful*. Kubernetes/OpenShift ont été initialement conçus pour héberger des applications *Stateless* (backend server). Or depuis quelques années, la notion de PV est venue assouplir cette contrainte : les données ne sont plus stockées sur le *pod* même et peuvent être contenues dans des espaces de stockage persistant (*Persistent Volume*). Ce découplage fournit un haut degré de résilience car, en cas de perte d'un pod, le PV est toujours joignable et, mieux encore, ré-attachable sur un nouveau *pod*.

Les PV offrent à Couchbase Server :

- Récupération des données stockées sur disque : forte diminution de la durée de rebalance des données et de la reconstruction des index
- Récupération des logs après crash d'un pod
- Provisionnement dynamique : le CAO peut créer des PV dynamiquement, à la demande en fonction de la nécessité de mise à l'échelle. Plus besoin alors de pré-provisionner un espace de stockage.
- Choix du type de classe de stockage : plus ou moins rapide, plus ou moins chère en fonction de l'usage qui en est fait. Ainsi, les PV dédiés au stockage des données data et index (GSI et FTS) reposent potentiellement sur des *Storage Class* (SC) plus rapide/coûteuse que les PV hébergeant le répertoire d'installation de Couchbase et de ses logs.

Point d'attention : pour la plupart des Cloud Providers, les PV sont *schédulés* (créés) sur différentes AZ. Or un *pod* doit résider dans la même AZ que son stockage (son *PV*).De fait, l'opérateur Couchbase requiert un couplage lâche (*lazy binding*) afin que les *PV* ne soient créés *qu'après* la création du *pod*.Cela se traduit par la propriété VolumeBindingMode de la Storage Class utilisée qui doit être être positionnée à WaitForFirstConsumer

```
oc get sc
```

```
NAME            PROVISIONER             RECLAIMPOLICY  VOLUMEBINDINGMODE
    ALLOWVOLUMEEXPANSION  AGE
gp2 (default)  kubernetes.io/aws-ebs  Delete      WaitForFirstConsumer
  true              7d9h
gp2-csi        ebs.csi.aws.com        Delete      WaitForFirstConsumer  true
            7d9h
```

Si tel n'est pas le cas, pas de panique il suffit alors juste de dériver la SC de voiture choix et de créer votre propre SC personnalisée avec le bon paramétrage (cf. https://docs.couchbase.com/operator/current/howto-storage-class.html#creating-a-lazily-bound-storage-class)

Pour aller plus loin sur le sujet :
https://docs.couchbase.com/operator/current/concept-persistent-volumes.html

Les PV adhoc (pv-default/data/index) sont instanciés à partir de PVC (Persistent Volume Claim), eux-mêmes créés à partir de *templates*. Ces templates sont référencés dans dans les *volumeMounts* associés au service

```
spec:
 servers:
 ...
  - size: 3
   name: data
   services:
   - data
   ...
   volumeMounts:
      default: pvc-default
      data: pvc-data
  - size: 2
   name: query_index_ensemble
   services:
   - index
   - query
   ...
   volumeMounts:
      default: pvc-default
      index: pvc-index
  - size: 2
   name: fts_seul
   services:
   - search
   ...
   volumeMounts:
      default: pvc-default
      index: pvc-index
 ..
 volumeClaimTemplates:
   - metadata:
      name: pvc-default
    spec:
     storageClassName: gp2
     resources:
      requests:
       storage: 1Gi
   - metadata:
      name: pvc-data
    spec:
     storageClassName: gp2
     resources:
      requests:
       storage: 20Gi
   - metadata:
      name: pvc-index
    spec:
     storageClassName: gp2
     resources:
      requests:
       storage: 5Gi
```

Suppression du précédent cluster :

```
oc delete -f couchbase-cluster.yaml -n programmez
```

Application de la configuration du nouveau cluster :

```
oc create -f couchbase-mds-servergroups-pv.yaml -n programmez
```

Re-connexion au cluster :

```
oc port-forward cb-example-0000 8091:8091 -n programmez
```

Montées de versions

La stratégie d'*upgrade* du cluster Couchbase Server est (par défaut) de type *RollingUpgrade*, c'est-à-dire que tous les pods du cluster sont mis à jour 1 à la fois. Pour chaque noeud du cluster, la cinématique est alors la suivante :

- Un nouveau pod entre (en version N+1) remplaçant le pod sortant (version N).
- L'opérateur déclenche une opération de *rebalance* entre ces 2 pods uniquement afin d'*échanger ("swapper")* les configurations de services ainsi que leurs données associées (notamment pour les services *stateful* Data/Index/FTS/Analytics).

L'ensemble de ces opérations se dénomme *Rolling Online Upgrade* avec *Swap Rebalance*.

Cette stratégie présente donc l'avantage d'éviter toute période d'indisponibilité du cluster.

La montée de version de l'opérateur autonome AO (et du DAC associé) engendre également une re création des pods du cluster existant géré par l'AO. La cinématique d'upgrade est, par défaut, la même que celle décrite précédemment.

La mise en place d'un upgrade consiste alors à modifier le numéro de version de l'image (DAC/AO ou Couchbase Server) contenu dans le YAML et d'appliquer les changements via la commande oc apply .

Containers utilitaires : monitoring, transfert de logs

Les versions Couchbase Server <7.x n'offrent pas d'*exporter* des données statistiques au format Prometheus. Toutefois

c'est le rôle de l'image Docker couchbase/exporter fournie sur Docker Hub (et sur le catalogue RedHat). Il s'agit d'un container *side-car* : hébergé dans le même *pod* que le container applicatif (Couchbase Server) il partage le même réseau et stockage. Le *log forwarding* est également disponible via ce même type de déploiement. Là aussi un container dédié (couchbase/fluent-bit) est instancié dans le *pod* contenant le container applicatif afin d'accéder au stockage des logs.
Ajoutons ces 2 capacités à notre configuration de cluster. Le bloc spec s'enrichit alors des structures logging et monitoring :

```
spec:
 ...
 logging:
  server:
   enabled: true
   manageConfiguration: true
   configurationName: "fluent-bit-config"
   sidecar:
    image: registry.connect.redhat.com/couchbase/fluent-bit:1.1.0-1
  audit:
   enabled: true
   garbageCollection:
    sidecar:
     enabled: true
 monitoring:
  prometheus:
   enabled: true
   image: registry.connect.redhat.com/couchbase/exporter:1.0.5
   resources:
    requests:
     cpu: 100m
     memory: 100Mi
```

Au final, dans l'outil **Lens** (https://k8slens.dev/) par exemple, on peut rapidement vérifier que les containers ont bien été tous instanciés.
Ces "sorties" issues de ces containers side-car peuvent alors servir d'entrées respectivement à un container annexe Prometheus ou Fluent Bit.

Pour aller plus loin :
Intégration de CB Server avec Prometheus + Grafana :
https://blog.couchbase.com/step-by-step-guide-for-running-couchbase-autonomous-operator-2-0-with-prometheus-part-1/
https://blog.couchbase.com/step-by-step-guide-for-running-couchbase-autonomous-operator-2-0-with-prometheus-part-2/
Intégration avec Fluent Bit :
https://blog.couchbase.com/using-fluent-bit-for-log-forwarding-processing-with-couchbase-server/

Autoscaling de la base de données

La version 2.2.1 de l'opérateur autonome permet de s'appuyer sur les capacités de mise à l'échelle horizontale offerte nativement par Kubernetes/OpenShift, via la ressource *Horizontal Pod Autoscaler* (HPA). Suite à son déploiement, la plateforme commence alors à requêter les ressources CPU/RAM des *pods*.

Note : *via la notion de label et de selector, il est bien sûr possible de ne monitorer/mettre à l'échelle qu'un sous-ensemble de pods.*

Lorsque ces métriques sont disponibles, le HPA calcule le ratio entre la métrique d'utilisation courante et celle désirée (et spécifiée dans la configuration du HPA) et effectue une mise à l'échelle à la hausse comme à la baisse (*scale up / down*).
Ce mécanisme permet d'allouer au bon moment le nombre de *pods* nécessaires et ainsi d'absorber les pics de charge temporaires. Tous les services Couchbase peuvent bénéficier de l'autoscaling.
L'auto-scaling OpenShift peut reposer sur les métriques de type :

- *Ressource* telles que la CPU ou mémoire des *pods* ou *nodes* du cluster OpenShift. Ces métriques sont collectées par *Metrics Server* et exposées via la *Metric API* d'OpenShift.
- *Couchbase* telles que la latence des requêtes N1QL ou le quota mémoire des buckets, etc... L'auto-scaler se connecte alors à la *custom metrics API* d'OpenShift pour récupérer ces valeurs, préalablement exposées par le container side-car *Couchbase Prometheus Exporter* (vu à la section précédente).

Ci-après, nous mettons en œuvre l'auto-scaling basé sur la détection de métriques de type *Ressource* en s'appuyant sur la supervision de la CPU des pods.
Afin de vérifier que le *Metric Server* est correctement installé, exécuter la commande suivante :

```
oc get --raw /apis/metrics.k8s.io/v1beta1
```


```
{"kind":"APIResourceList","apiVersion":"v1","groupVersion":"metrics.k8s.io/v1beta1","resources":[{"name":"nodes","singularName":"","namespaced":false,"kind":"NodeMetrics","verbs":["get","list"]},{"name":"pods","singularName":"","namespaced":true,"kind":"PodMetrics","verbs":["get","list"]}]}
```


La réponse doit contenir une APIResourceList. Si tel n'est pas le cas, il faut alors installer *Metric Server* (cf. https://github.com/kubernetes-sigs/metrics-server#installation).

Côté configuration YAML, cela se traduit par l'ajout de la ressource HorizontalPodAutoscaler. Ci-après un exemple d'implémentation d'un HPA pour le service data où le seuil est volontairement faible (fixé à 10% de la CPU du *pod*) afin de déclencher plus rapidement l'auto-scaling et illustrer notre propos :
Code complet sur programmez.com & github

Puis, au niveau de la configuration du cluster Couchbase Server, on ajoute les propriétés autoscaleStabilizationPeriod et autoscaleEnabled afin de spécifier quelle(s) classe(s) de service (*data* ici) peut bénéficier de l'auto-scaling :

```
apiVersion: couchbase.com/v2
kind: CouchbaseCluster
metadata:
 name: cb-example
spec:
 autoscaleStabilizationPeriod: 120s
 ...
```

```
servers:
- size: 3
  autoscaleEnabled: true
  name: data
  services:
  - data      ...
- size: 2
  name: query_index_ensemble      ...
```

```
oc apply -f couchbase-autoscaler.yaml -n programmez
oc describe couchbasecluster cb-example | grep 'Autoscaler'
```

```
Normal  EventAutoscalerCreated  9m19s      Autoscaler for config `data` added
```

La propriété autoscaleStabilizationPeriod spécifie la durée (en secondes) pendant laquelle le HPA associé au cluster Couchbase Server est désactivé, dès lors que le cluster a commencé une opération de rééquilibrage (*rebalance*).
Afin de simuler une charge accrue (>10% CPU), on augmente le nombre de threads instanciées par le *Job* pillowfight (passage de 1 à 5), cf. fichier *pillowfight-data-loader-5-threads-openshift.yaml* :

```
…
command: ["cbc-pillowfight",
          "-U", "couchbase://cb-example-srv/default?select_bucket=true",
          "-I", "100000", "-B", "1000", "-c", "-1", "-t", "5", "-u", "Administrator", "-P",
"password"]
…
```

```
oc apply -f pillowfight-data-loader-5-threads-openshift.yaml -n programmez
```

En réponse à cette hausse charge (CPU > 10%), l'opérateur commande l'ajout d'un noeud additionnel :

```
oc get logs couchbase-operator-6648df9b7f-knwtk -f
```

```
...
{"level":"info","ts":1632217361.3430882,"logger":"cluster","msg":"Resource updated",
"cluster":"programmez/cb-example","diff":" (\n \t\"\"\"\n \t... // 39 identical lines\
n \t status: \"False\"\n \t type: AutoscaleReady\n+ \t- lastTransitionTime: \"2021-09
-21T09:42:41Z\"\n+ \t lastUpdateTime: \"2021-09-21T09:42:41Z\"\n+ \t message:
'Current cluster size: 3, desired cluster size: 4'\n+ \t reason: ScalingUp\n+ \t
status: \"True\"\n+ \t type: Scaling\n \tcurrentVersion: 6.6.2-1\n \tmembers:\n
\t... // 6 identical lines\n \t\"\"\"\n )\n"}
{"level":"info","ts":1632217361.6256902,"logger":"cluster","msg":"Creating pod",
"cluster":"programmez/cb-example","name":"cb-example-0003","image":
"registry.connect.redhat.com/couchbase/server:6.6.2-1"}
...
```

```
oc get pods -n programmez
```

NAME	READY	STATUS	RESTARTS	AGE
cb-example-0000	1/1	Running	0	58m
cb-example-0001	1/1	Running	0	57m
cb-example-0002	1/1	Running	0	57m
cb-example-0003	1/1	Running	0	52m
couchbase-operator-6648df9b7f-knwtk	1/1	Running	0	11d
pillowfight-6kv66	1/1	Running	0	55m

Si la charge est amoindrie ou arrêter, le HPA entre à nouveau en action et réduit le nombre de *pod* du cluster Couchbase à la valeur minimale spécifiée dans la règle (minReplicas: 3).

```
oc delete -f pillowfight-data-loader-5-threads-openshift.yaml -n programmez
```

```
oc get logs couchbase-operator-6648df9b7f-knwtk -f
```

```
{"level":"info","ts":1632220962.6843483,"logger":"cluster","msg":"Resource updated",
"cluster":"programmez/cb-example","diff":" (\n \t\"\"\"\n \t... // 39 identical lines\
n \t status: \"False\"\n \t type: AutoscaleReady\n+ \t- lastTransitionTime: \"2021-09
-21T10:42:42Z\"\n+ \t lastUpdateTime: \"2021-09-21T10:42:42Z\"\n+ \t message:
'Current cluster size: 4, desired cluster size: 3'\n+ \t reason: ScalingDown\n+ \t
status: \"True\"\n+ \t type: Scaling\n \tcurrentVersion: 6.6.2-1\n \tmembers:\n
\t... // 7 identical lines\n \t\"\"\"\n )\n"}
...
{"level":"info","ts":1632220964.5545125,"logger":"cluster","msg":"Rebalancing","cluster":
"programmez/cb-example","progress":0}
{"level":"info","ts":1632220968.5770028,"logger":"cluster","msg":"Rebalancing","cluster":
"programmez/cb-example","progress":8.576605758946451}
...
{"level":"info","ts":1632221009.1143816,"logger":"cluster","msg":"Pod deleted",
"cluster":"programmez/cb-example","name":"cb-example-0003"}
{"level":"info","ts":1632221009.1264546,"logger":"cluster","msg":"Resource updated",
"cluster":"programmez/cb-example","diff":" (\n \t\"\"\"\n \t... // 51 identical lines\
n \t - cb-example-0001\n \t - cb-example-0002\n- \t - cb-example-0003\n- \tsize:
4\n+ \tsize: 3\n \t\"\"\"\n )\n"}
```

```
oc get pods -n programmez
```

NAME	READY	STATUS	RESTARTS	AGE
cb-example-0000	1/1	Running	0	67m
cb-example-0001	1/1	Running	0	67m
cb-example-0002	1/1	Running	0	66m
couchbase-operator-6648df9b7f-knwtk	1/1	Running	0	11d

Pour aller plus loin sur ce sujet :
https://docs.couchbase.com/operator/current/concept-couchbase-autoscaling-best-practices.html#recommendations-2
https://docs.couchbase.com/operator/current/resource/couchbaseautoscaler.html
https://docs.couchbase.com/operator/current/concept-couchbase-autoscaling.html#resource-metrics
https://docs.couchbase.com/operator/current/concept-couchbase-autoscaling.html#couchbase-metrics

Conclusion

Les capacités natives de Couchbase Server - accès rapide aux données (architecture *memory first*) via clé/valeur ou N1QL, haute disponibilité via les réplicas, résilience à la panne (auto-failover) - couplées à la gestion autonome de la base de données via l'opérateur CAO et les mécanismes standard offerts par Kubernetes/OpenShift (PV, HPA, zonage des pods…) permettent de délivrer une solution d'hébergement robuste de vos données dans un environnement *conteneurisé* aisé à déployer et à maintenir.
C'est d'ailleurs la solution retenue par Couchbase pour son offre d'hébergement des données en mode DBaaS "Couchbase Capella".

Accélérer vos développements d'API grâce à une démarche Full Lifecycle !

Les API sont au cœur de la transformation de nombreuses organisations. Déployer une solution d'API Management est en revanche loin d'être suffisant pour obtenir toute la flexibilité et l'efficience requises pour la mise à disposition de nouvelles API ! Ces solutions doivent être complétées d'un nouveau workflow et d'une vision claire sur le cycle de vie des API. Cet article illustre cela via un exemple concret où nous allons designer, simuler, développer et tester une nouvelle API. Nous finirons par expliquer comment ces étapes peuvent s'intégrer de façon automatisée dans votre workflow et pipeline DevOps.

Laurent Broudoux

Laurent est Architecte Solution chez Red Hat France, où il est spécialisé dans le développement des applications cloud-native et la transformation des applications existantes. Avant de rejoindre Red Hat il y a 5 ans, il a été Architecte SOA pendant plus de 10 ans dans les Services financiers où il a défini des stratégies de transformation via l'usage des API et de l'Intégration. Il est le fondateur et lead developer du projet Open Source Microcks.io : un outil Kubernetes-native pour le mocking et testing d'API. @lbroudoux

Vous avez dit "API" ?

Les *Application Programming Interfaces* (ou API) sont la pierre angulaire d'un nombre incroyable d'innovations récentes: des applications mobiles à l'internet des objets au cloud-computing. Elles sont devenues essentielles pour tous les secteurs de marché : de l'Open Banking à la voiture connectée en passant par la santé avec des la construction de nouveaux services à valeur ajoutée en des temps records. TousAntiCovid, ViteMaDose et CovidTracking en sont les meilleurs exemples et montrent bien que les API apportent de l'innovation, plus facilement, tout en favorisant de nouveaux modèles de collaboration.

Aujourd'hui, lorsqu'on emploie le terme API on pense surtout aux API Web en désignant des fonctionnalités appelables à distance en utilisant des protocoles standards tels que HTTP, WebSocket, MQTT ou AMQP. Elles sont notamment indispensables pour mettre en place des Architectures Microservices. Leur utilisation procure des avantages énormes en termes de découplage (technique, organisationnel, temporel) mais présente aussi de nombreux challenges auxquels se heurtent les organisations : la sécurité bien sûr, la communication et la gouvernance mais aussi l'impact de ce nouveau modèle d'architecture sur les développements.

En effet, dans un monde où tout le monde est consommateur et fournisseur d'API : **comment délivrer efficacement** - sans avoir à attendre sans cesse les autres - **avec une haute qualité** - en garantissant le respect de l'interface - **les implémentations des API** de mon application ?

Full API Lifecycle: pourquoi ? Comment ?

Afin de répondre aux challenges de sécurité, de communication et de gouvernance ont fleuri cette dernière décennie des solutions dites d'API Management. Red Hat propose notamment la solution Red Hat 3scale API Management présentant l'intérêt d'être intégrée aux écosystèmes OpenShift et Fuse / Apache Camel. Ces solutions sont insuffisantes pour couvrir l'ensemble des activités entrant en jeu entre l'inception d'une API et sa consommation. Le Cycle de vie complet des API (ou Full API Lifecycle) décrit un ensemble d'étapes qui peuvent vous guider depuis l'idée jusqu'à la production. Ce cycle est souvent représenté en utilisant deux volets :

- Un volet gauche représentant les activités de *BUILD* du fournisseur,
- Un volet droit représentant les activités au *RUN* et liées aux consommateurs. **Figure 1**

Ce schéma représente la couverture typique des solutions d'API Management. Est-ce à dire que ces solutions doivent gagner en périmètre dans le futur ? Pas nécessairement. La fourniture d'une API de bout en bout est un processus complexe qui implique différents acteurs, rôles, préoccupations et filières techniques d'implémentation. Nous pensons chez Red Hat que la séparation des préoccupations, la modularité et l'ouverture sont les fondations de la véritable agilité et encourageons plutôt nos clients à considérer des outils de la communauté, spécialisés chacun sur un sous-ensemble de problématiques.

Nous détaillerons justement les phases non couvertes par l'API Management :

- La phase de **Design** permet via une approche dite *Contract-first* de définir l'interface de l'API. L'objectif est de

RIEN DE NEUF ?

Les puristes vous diront que le concept d'API n'est pas nouveau : ce concept se trouve depuis des décennies au cœur du noyau Linux et de nos applications. Il s'agit essentiellement de dissocier la description de la fonction (le quoi : l'interface) de sa réalisation (le comment : l'implémentation). On pourra alors au fil du temps améliorer l'implémentation d'une fonction de façon transparente sans affecter l'interface : pointe de l'iceberg vue par les utilisateurs.

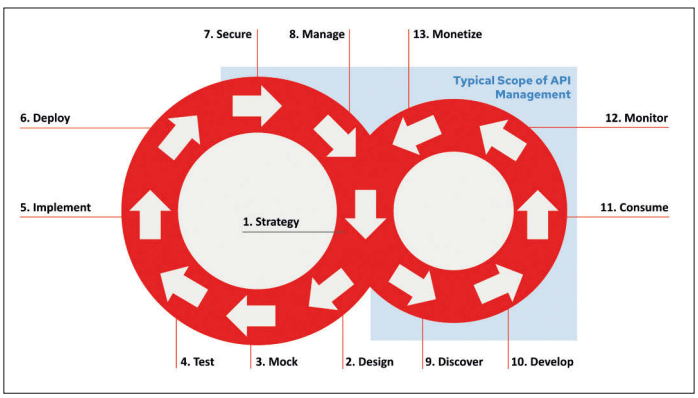


Figure 1

produire ce fameux contrat sur lequel producteur et consommateur vont s'entendre pour activer des fonctions, échanger des messages,

- La phase de **Mock** - ou de simulation - doit permettre à un futur consommateur de l'API de commencer à travailler immédiatement avant le développement de l'implémentation. Elle vise donc à paralléliser le travail et collecter rapidement du feed-back de la part des utilisateurs : l'API apporte-t-elle de la valeur ? Est-elle facilement utilisable ? Pour la mettre en œuvre, on spécifie souvent des exemples visant à illustrer le comportement attendu,
- La phase de **Test** vise à définir les tests d'acceptation de l'API. Une partie de ces tests est déduite directement du contrat mais un complément est naturellement apporté lors de la fourniture d'exemples. Cette phase est donc très liée à la phase de simulation : le même ensemble d'exemples pouvant être utilisés pour valider un comportement. Si vous êtes un vendeur d'alcool en ligne, votre API doit retourner une erreur si un mineur tente de faire un achat. Cette règle est certainement difficile à formaliser dans un contrat d'API mais simple à translater en termes de simulation et de tests,
- La phase de **Développement** qui vise finalement à produire le composant technique implémentant l'API tout en respectant son contrat. Ce composant devra être soumis aux tests lors du **Déploiement** pour s'assurer de sa qualité.

Notre cas d'usage

Nous allons utiliser une API fictive de gestion de notre catalogue de fruits ! Pour les lecteurs qui nous suivent, cette API pourrait correspondre au dossier paru dans Programmez #241. Cette fois, l'été est derrière nous mais il est toujours temps de faire le plein de vitamines.

Nous allons endosser le rôle du fournisseur d'API qui doit la mettre à disposition de divers partenaires utilisateurs. Certains utilisateurs ont décidé de fournir une application Web ou Mobile aux clients ; d'autres utiliseront directement l'API depuis leurs scripts. **Figure 2**

Notre mission - au-delà du développement de l'API - consiste à permettre aux utilisateurs de pouvoir travailler sur l'intégration de notre API le plus rapidement possible sans qu'ils aient à attendre la phase finale de recette de notre API. Nous utiliserons pour cela une description d'interface et de la simulation. Vitesse ne signifie pas précipitation ! Nous devrons garantir que le produit final tient bien les promesses et est conforme à l'interface définie. Prêt ?

Outils et prérequis

Afin de mener à bien cette mission, nous utiliserons les outils et frameworks communautaires suivants :

- Apiurio Studio (https://www.apicur.io/studio/) nous permettra de *designer* notre API et d'apporter les exemples qui nous serviront pour les mocks et tests,
- Microcks (https://microcks.io) nous permettra de simuler notre API avant son implémentation et de réaliser des tests de conformité au contrat (ou *contract-testing*),
- Quarkus (https://quarkus.io) nous permettra de développer notre implémentation en Java Supersonic et Subatomic ! **Figure 3**

L'ensemble des sources et ressources utilisées dans cet article sont sur le référentiel GitHub : https://github.com/lbroudoux/programmez-article-api-lifecycle. Pour disséquer chez vous ces éléments, vous n'aurez besoin que d'un navigateur Web, un éditeur de code type VS Code et une installation d'un moteur de containers (Docker et Docker-compose ou Podman et Podman-compose).

CODE-FIRST OU CONTRACT-FIRST ?

La remarque revient souvent: une approche code-first *- où l'on commence par écrire le code duquel on extrait ensuite le contrat - est plus simple à mettre en œuvre ! Et cela est vrai… si l'on considère uniquement le point de vue du producteur d'API, lors de la première itération de développement. D'autres considérations deviennent extrêmement complexes à mettre en place sans une approche Contract-First :*

- *Comment permettre à d'autres de commencer à travailler et raccourcir le temps global du projet ?*
- *Comment capturer et documenter la connaissance fonctionnelle attachée au contrat d'API ?*
- *Comment s'assurer de non-régressions et du maintien de compatibilité ascendante lors des releases successives ?*

Élargir le point de vue aux enjeux des consommateurs et de l'organisation entière milite pour une adoption du Contract-First. *D'autant que nous le verrons dans cet article : l'outillage fait en sorte que cela ne soit pas synonyme de perte de vélocité.*

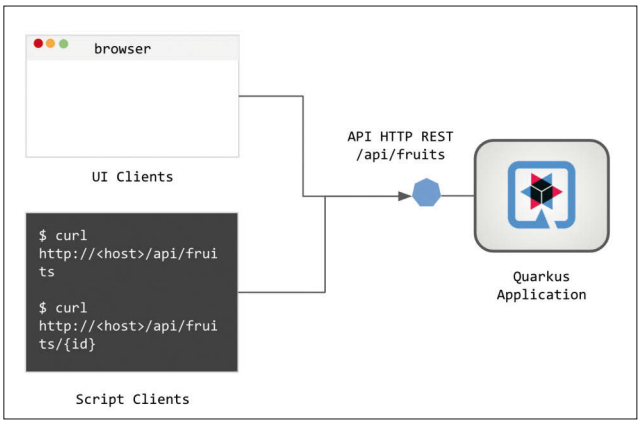


Figure 2

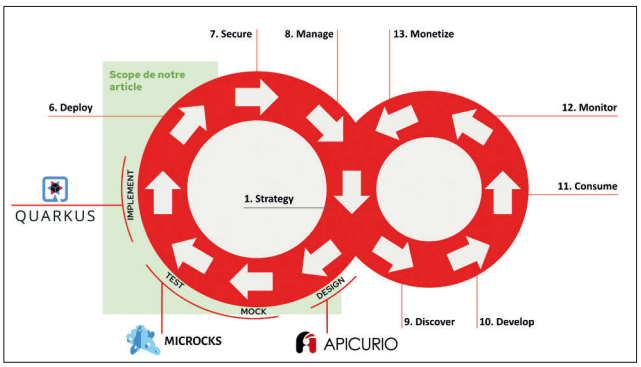


Figure 3

Design d'API avec Apicurio

Nous allons réaliser la conception de notre Fruits Catalog API et produire un contrat en respectant le standard OpenAPI v3. Nous allons utiliser la version en ligne de Apicurio Studio qui se trouve sur https://studio.apicur.io. Il vous sera nécessaire de créer un compte avant d'accéder à l'outil.

L'écran d'accueil d'Apicurio Studio est un tableau de bord permettant de voir vos dernières activités sur le site. Il vous permet également de créer une nouvelle API (**Create New**

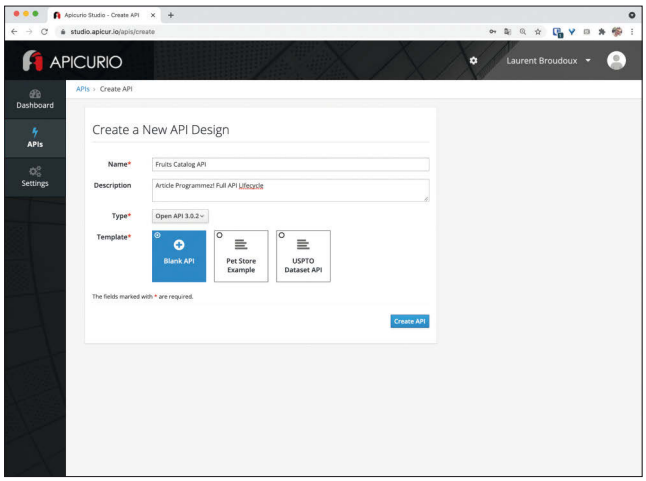


Figure 4

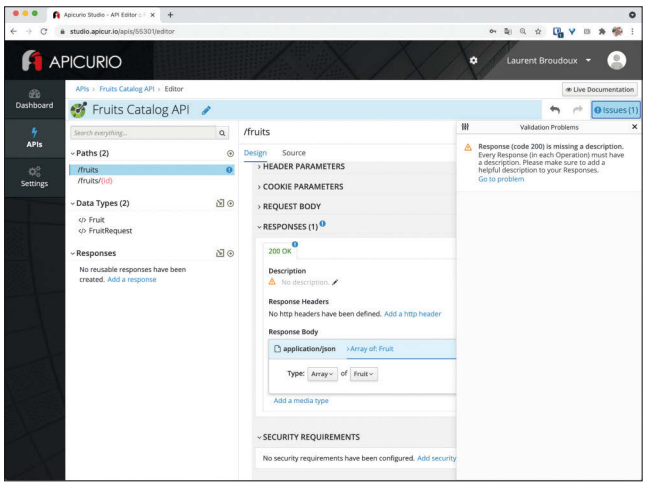


Figure 5

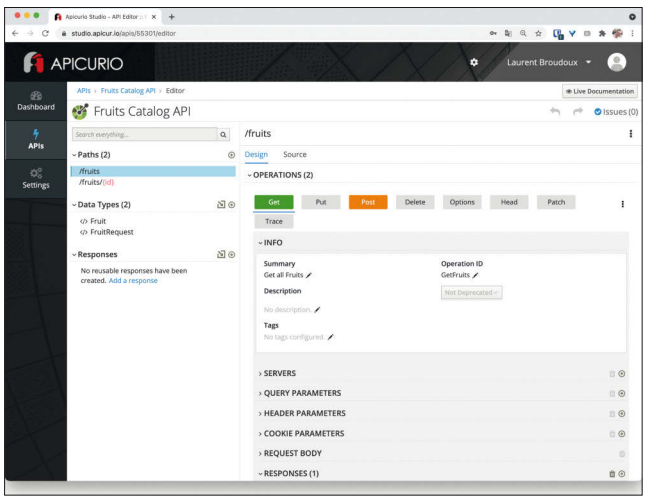


Figure 6

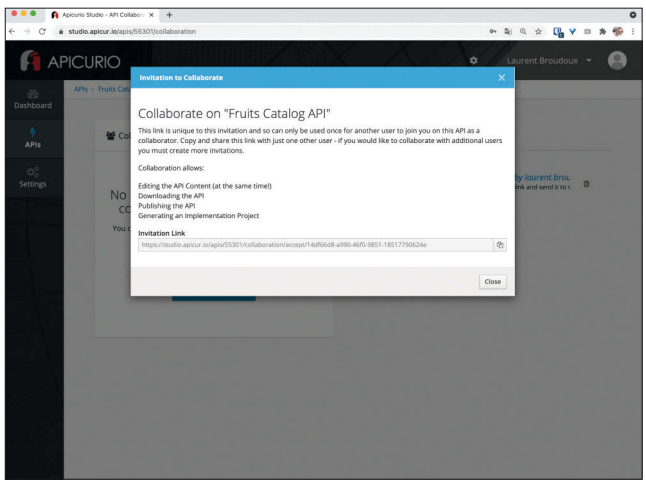


Figure 7

API) ou d'importer un design d'API existant (**Import API**). La création d'un nouveau design est très simple : un nom, le type d'API et un template de création sont suffisants : **Figure 4**

Vous avez ensuite la possibilité d'éditer votre design d'API en mode graphique, de façon guidée par l'outil ; ce qui permet de ne pas avoir à connaître la spécification et les syntaxes JSON et YAML en cas d'oubli ou d'erreur, l'outil vous affiche les violations détectées par une validation en tâche de fond de vos éditions au sein d'un volet **Issues**. Pour les intrépides, il y a également la possibilité de passer en mode **Source** afin d'éditer directement le document OpenAPI. **Figure 5**

Pour la conception de notre Fruits Catalog API en version 1.0.0, nous avons défini deux chemins (/fruits et /fruits/{id}) ainsi que cinq opérations différentes :

- Créer un fruit : POST /fruits,
- Lister les fruits : GET /fruits,
- Récupérer un fruit avec son identifiant : GET /fruits/{id},
- Mettre à jour un fruit : UPDATE /fruits/{id},
- Supprimer un fruit : DELETE /fruits/{id}

Vous pouvez recréer ce design par vous-même ou alors importer la définition qui se trouve à https://github.com/lbroudoux/programmez-article-api-lifecycle/blob/main/contracts/fruits-catalog-openapi-1.0.0.yaml pour visualiser le résultat : **Figure 6.**

Collaborer et publier

Studio Apicurio est également fait pour travailler à plusieurs. Vous pouvez inviter des collaborateurs sur un *design*. À l'instar d'une Google Suite, vous pourrez voir en temps réel les modifications apportées par les différents participants et ainsi réaliser des revues de façon efficace. **Figure 7**

Finalement, Apicurio vous permet de publier le résultat de votre conception. Soit sous forme de fichier JSON ou YAML ou plus directement en poussant directement dans un référentiel GitHub, GitLab ou BitBucket managé ou *on-premise*. **Figure 8**

Simulation d'API avec Microcks

Nous avons maintenant la v1 de notre contrat d'API, voyons comment l'exploiter pour lancer des simulations avec Microcks. Microcks est un outil permettant la simulation et le *contract-testing* de tous types d'API : REST ou asynchrones ou encore basées sur des WebServices SOAP. Pour des questions de simplicité, nous allons utiliser la méthode d'installation utilisant Docker-compose comme détaillée dans le Getting Started du projet.

```
$ git clone https://github.com/microcks/microcks.git
$ cd microcks/install/docker-compose
$ docker-compose up -d
```

Après quelques secondes, vous pouvez ouvrir un navigateur et pointer sur http://localhost:8080, vous connecter en utilisant admin / microcks123 comme utilisateur et mot de passe.

Via la section Importers, il est possible soit de charger directement le fichier OpenAPI précédemment récupéré depuis Apicurio Studio ; soit de créer un nouvel importer job pointant directement sur le référentiel Git à l'adresse : https://raw.githubusercontent.com/lbroudoux/programmez-article-api-lifecycle/main/contracts/fruits-catalog-openapi-1.0.0.yaml. L'API est alors découverte par Microcks et se charge dans le référentiel : **Figure 9**

Les opérations sont bien présentes mais Microcks ne peut pas exposer de simulation, car nous n'avons pas défini d'exemples ! Nous allons en ajouter en utilisant Apicurio Studio

Ajout d'exemples

Quoi de mieux qu'un exemple de la vraie vie pour illustrer le comportement attendu d'une API ? Et c'est encore mieux si l'exemple est attaché directement au contrat de cet API ! C'est ce que permet de faire OpenAPI. Nous avons ajouté des exemples de paramètres, de corps de requête et de corps de réponses aux différentes opérations de notre Fruits Catalog API.

Vous pouvez le vérifier en important https://github.com/lbroudoux/programmez-article-api-lifecycle/blob/main/contracts/fruits-catalog-openapi-samples-1.0.0.yaml au sein du Studio Apicurio et en explorant ces fragments d'exemple. **Figure 10**

DE LA COHÉRENCE DES EXEMPLES

Même si l'objectif est de donner une teinte métier aux exemples, il est important de garder en tête les éléments techniques assurant la cohérence de l'ensemble. On veillera ainsi que les différentes opérations de lecture / modification utilisent les mêmes identifiants afin de permettre le déroulement d'un scénario.

Une fois notre design OpenAPI complété et sauvegardé, nous pouvons importer depuis Microcks cette nouvelle mouture en pointant sur https://raw.githubusercontent.com/lbroudoux/programmez-article-api-lifecycle/main/contracts/fruits-catalog-openapi-samples-1.0.0.yaml. La définition de notre API s'en trouve modifiée et Microcks s'est chargé de mettre en place des points d'accès REST pour les différents exemples. Regardons l'opération GET /fruits. Nous disposons maintenant d'une **Mock URL** : **Figure 11**

Nous pouvons utiliser directement cette Mock URL pour interagir avec l'API comme si elle existait déjà :

```
$ curl http://localhost:8080/rest/Fruits+Catalog+API/1.0.0/fruits -s | jq .
[
 {
  "id": "31e1c6c8-4d75-4753-918f-d026749e3743",
  "name": "Apple",
  "origin": "France"
 },
 {
  "id": "cbc41087-40ff-44b6-a3bb-b88a47b49ffd",
  "name": "Grape",
  "origin": "Italy"
 }
]
```

Nous pouvons également utiliser l'opération GET /fruits/{id} pour récupérer les détails d'un fruit particulier : **Figure 12**

Le moteur de mocking de Microcks analyse l'identifiant utilisé (le critère id) et va dispatcher la bonne réponse :

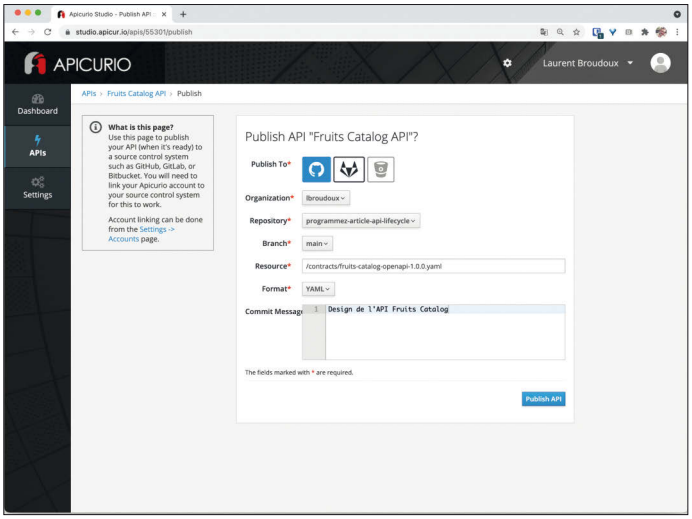


Figure 8

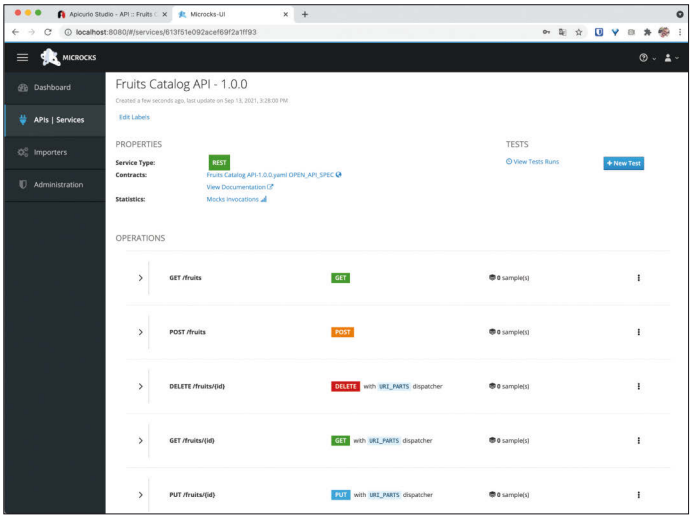


Figure 9

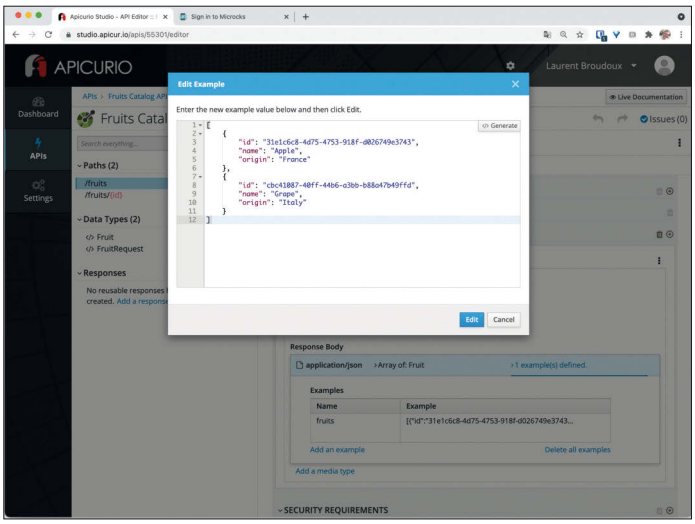


Figure 10

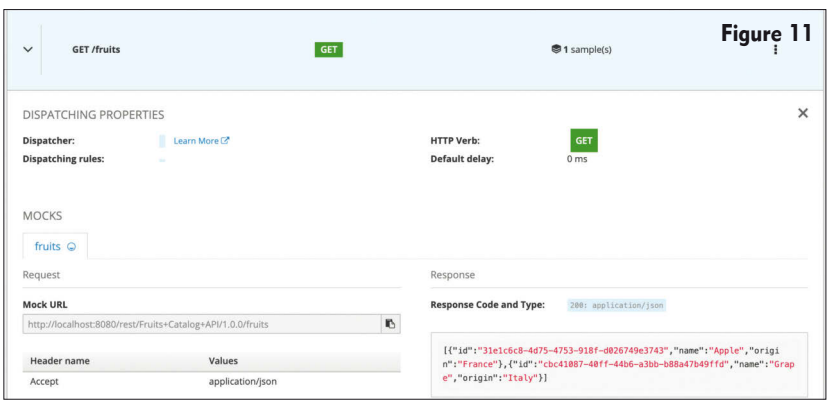


Figure 11

```
$ curl http://localhost:8080/rest/Fruits+Catalog+API/1.0.0/fruits/cbc41087-40ff-44b6-a3bb-b88a47b49ffd -s | jq .
{
 "id": "cbc41087-40ff-44b6-a3bb-b88a47b49ffd",
 "name": "Grape",
 "origin": "Italy"
}
```

Microcks permet la mise en place de simulations statiques mais avec une certaine dynamique. Regardons de plus près l'opération POST /fruits qui vise à créer de nouveaux fruits. La création d'un nouvel objet doit théoriquement entraîné l'assignation d'un nouvel identifiant et nous pouvons faire cela avec la notation suivante : **Figure 13**

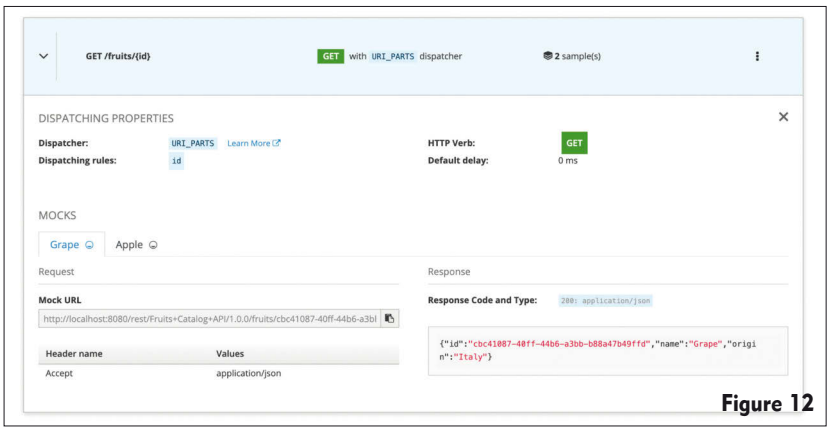


Figure 12

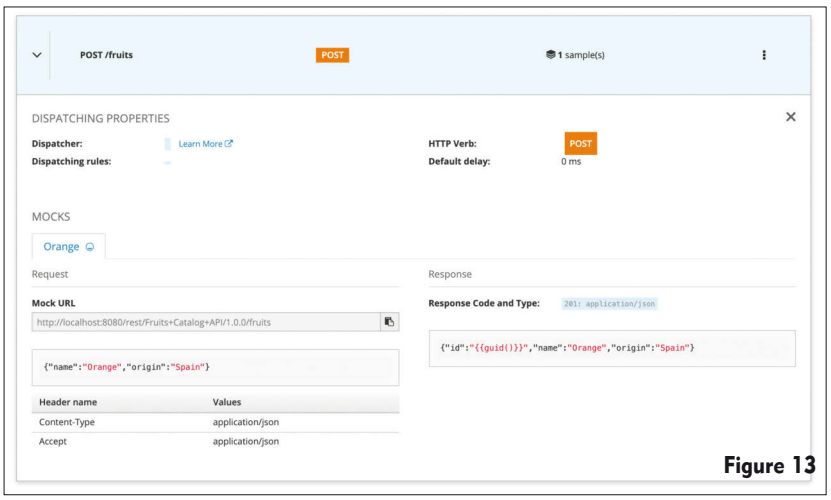


Figure 13

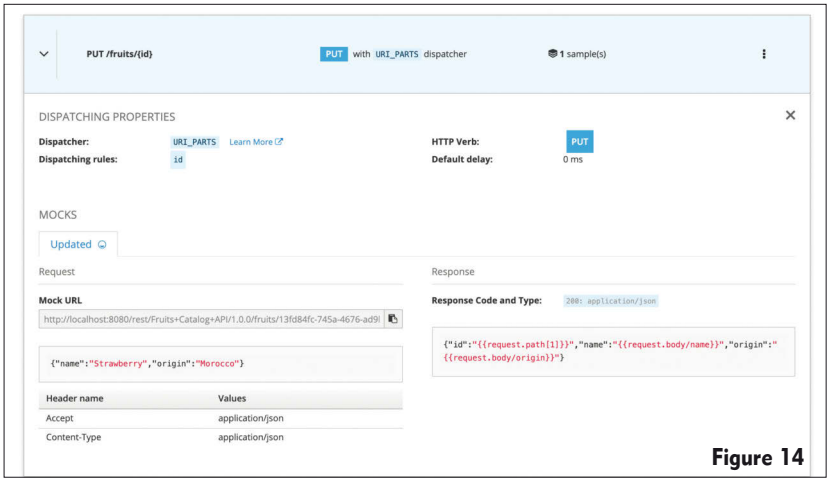


Figure 14

La notation {{guid()}} permet de demander à Microcks la génération d'un nouvel UUID. Vérifions cela avec une nouvelle requête de création :

```
$ curl -XPOST http://localhost:8080/rest/Fruits+Catalog+API/1.0.0/fruits -H 'Content-type: application/json' -d '{"name":"orange","origin":"Spain"}' -s | jq .
{
 "id": "caa3f3cc-1428-4f40-a142-459ffb7f83b2",
 "name": "Orange",
 "origin": "Spain"
}
```

Ça fonctionne ! Chaque requête successive entraînera la génération d'un nouvel identifiant unique suivant le format UUID.

Templating de réponse

Les lecteurs attentifs auront peut-être noté dans l'exemple précédent que nous avions demandé la création d'une orange mais avons finalement reçu une Orange. C'est particulièrement fâcheux si l'on souhaite décrire la logique d'une opération de mise à jour... Fort heureusement, Microcks possède des fonctionnalités de *templating* de réponse puissantes, décrites sur cette page https://microcks.io/documentation/using/advanced/templates/. Avec ces fonctionnalités, on peut notamment décrire que le champ d'une réponse doit être composé d'éléments de la question comme le montre ci-dessous l'opération PUT /fruits/{id} : **Figure 14**

Je peux alors émettre n'importe quelle demande de modification via les propriétés name et origin, celles-ci seront prises en compte lors de la génération d'une réponse par Microcks :

```
$ curl -XPUT http://localhost:8080/rest/Fruits+Catalog+API/1.0.0/fruits/13fd84fc-745a-4676-ad9b-185082000652 -H 'Content-type: application/json' -d '{"name":"Pear","origin":"France"}' -s | jq .
{
 "id": "fruits",
 "name": "Pear",
 "origin": "France"
}
```

Développement d'API avec Quarkus

Nous devons maintenant développer l'implémentation de notre Fruits Catalog API et pour ce faire nous avons choisi le Java "next-gen" supersonic et subatomique : Quarkus !

Le code est disponible sur le référentiel GitHub suivant : https://github.com/lbroudoux/programmez-article-api-lifecycle.

Génération OpenAPI

Nous avons précédemment construit une magnifique spécification OpenAPI grâce à Apicurio Studio : il serait idiot de ne pas nous en servir ! Nous avons pour cela utilisé le générateur OpenAPI branché en tant que plugin Maven. Nous l'avons configuré pour utiliser le contrat présent dans notre référentiel et ne générer que les éléments du modèle (c'est-à-dire les structures de données) :

```
<plugin>
 <groupId>org.openapitools</groupId>
 <artifactId>openapi-generator-maven-plugin</artifactId>
 <version>5.1.0</version>
 <executions>
  <execution>
   <goals>
    <goal>generate</goal>
   </goals>
   <configuration>
    <inputSpec>${project.basedir}/contracts/fruits-catalog-openapi-1.0.0.yaml</inputSpec>
    <generatorName>java</generatorName>
    <modelPackage>com.redhat.fruits.catalog.dto</modelPackage>
    <modelNameSuffix>DTO</modelNameSuffix>
    <generateApis>false</generateApis>
    <generateSupportingFiles>false</generateSupportingFiles>
    <configOptions>
     <sourceFolder>src/gen/main/java</sourceFolder>
     <library>resteasy</library>
     <serializationLibrary>jackson</serializationLibrary>
    </configOptions>
   </configuration>
  </execution>
 </executions>
</plugin>
```

Cette configuration nous permettra de générer les objets dans un package spécifique dto avec un suffixe DTO afin d'éviter les collisions avec les objets persistants de l'application qui se trouve dans un package domain. La génération sera faite à chaque nouveau build et permettra de refléter directement les évolutions de contrat. Vous pouvez trouver la documentation complète de ce plugin sur ce site : **https://openapi-generator.tech/docs/plugins/#maven**
Nous disposons de deux jeux de structure de données (les dto et les domain) sur lesquels nous avons besoin de faire des mappings. Et comme nous sommes de bon développeurs paresseux, nous allons utiliser pour cela la librairie MapStruct. MapStruct va nous permettre de générer simplement des mappers efficaces et personnalisables. Dans notre cas, les propriétés des classes dto et domain portent les mêmes noms donc il suffit de créer une simple interface portant la définition des méthodes de transformation :

```
@Mapper(componentModel = "cdi")
public interface FruitMapper {

 Fruit fromResource(FruitRequestDTO fruit);

 FruitDTO toResource(Fruit fruit);

 List<FruitDTO> toResources(List<Fruit> fruits);

 default Origin fromResource(FruitDTO.OriginEnum origin) {
  return Origin.fromValue(origin.getValue());
 }

 default FruitDTO.OriginEnum toResource(Origin origin) {
  return FruitDTO.OriginEnum.fromValue(origin.getValue());
 }
}
```

MapStruct permet bien sûr d'aller plus loin en définissant des renommages, des recompositions ou tout autre opération sur les propriétés d'objet ; la plupart du temps au travers de simples annotations ! Nous avons maintenant tout ce qu'il nous faut pour débuter le développement de la logique de l'API.

Définition des endpoints REST

Le plugin de génération OpenAPI est capable de générer des squelettes de endpoints REST mais le résultat n'est pas très exploitable pour Quarkus. Il est plus simple de le faire directement.
Le principe est simple : nous définissons une fonction portant les annotations JAX-RS et correspondant donc à une opération du contrat d'API puis nous utilisons le fruitMapper MapStruct défini précédemment pour translater nos objets domain en dto et réciproquement. Voici un exemple sur l'opération destinée à lire les fruits du catalogue :

```
@Inject
FruitMapper fruitMapper;

@GET
@Produces(MediaType.APPLICATION_JSON)
public List<FruitDTO> listFruits() {
 logger.info("Listing fruits...");
 return fruitMapper.toResources(Fruit.listAll());
}
```

Ci-dessous le résultat pour nos différentes opérations correspondant à la spécification de notre Fruits Catalog API :

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Transactional
public Response createFruit(FruitRequestDTO fruitRequest) { [...] }

@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public FruitDTO getFruit(@PathParam("id") String id) { [...] }

@PUT
@Path("/{id}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public FruitDTO updateFruit(@PathParam("id") String id,
  FruitRequestDTO fruitRequest) { [...] }

@DELETE
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
@Transactional
public Response deleteFruit(@PathParam("id") String id) { [...] }
```

Un dernier point d'importance dans notre implémentation est le jeu de tests. Nous pouvons le déduire directement depuis notre spécification OpenAPI ; il nous suffit de le translater en données persistantes qui seront intégrées dans la base de

données lors de chaque démarrage de mode dev ou test. Dans Quarkus, il suffit de fournir un fichier /src/main/resources/import.sql comme ci-dessous :

```
INSERT INTO fruits (id, name, origin) VALUES ('31e1c6c8-4d75-4753-918f-d026749e3743', 'Apple', 0);
INSERT INTO fruits (id, name, origin) VALUES ('cbc41087-40ff-44b6-a3bb-b88a47b49ffd', 'Grape', 1);
INSERT INTO fruits (id, name, origin) VALUES ('caa3f3cc-1428-4f40-a142-459ffb7f83b2', 'Orange', 2);
INSERT INTO fruits (id, name, origin) VALUES ('13fd84fc-745a-4676-ad9b-185082000652', 'Strawberry', 3);
```

JEU DE DONNÉES DE TEST

La construction des jeux de données de tests est toujours un exercice compliqué et c'est un autre avantage de l'approche Contract-first / Acceptance Test-driven *que nous décrivons ici : le contrat d'API contient au travers des exemples la définition du jeu de tests. Il revient au développeur de l'exprimer selon la forme la plus utile pour son implémentation (structuré, objet, graph,…). Ce jeu de données pourra alors être utilisé sur les environnements de tests - chargé via un pipeline CI/CD par exemple - pour permettre les tests automatisés.*

Exécution

Deux possibilités pour exécuter l'application : recompiler et exécuter OU lancer un container !

1 Via le wrapper Maven, nous devons forcer la phase de compile pour générer les objets dto et préférer le port 8082 si vous utilisez toujours Microcks :

```
$ ./mvnw compile quarkus:dev -Dquarkus.http.port=8082
[...]
Listening for transport dt_socket at address: 5005
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2021-09-14 09:26:50,414 INFO  [io.quarkus] (Quarkus Main Thread) fruits-catalog 1.0.0-SNAPSHOT on JVM (powered by Quarkus 2.2.2.Final) started in 2.844s. Listening on: http://localhost:8082
2021-09-14 09:26:50,418 INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
2021-09-14 09:26:50,419 INFO  [io.quarkus] (Quarkus Main Thread) Installed features: [agroal, cdi, hibernate-orm, hibernate-orm-panache, jdbc-h2, micrometer, narayana-jta, resteasy, resteasy-jackson, smallrye-context-propagation, smallrye-health]
```

2 Via docker en spécifiant l'attachement au port 8082 et le profile dev (autrement, vous n'aurez pas de données dans la base) :

```
$ docker run -i —rm -p 8082:8080 -e QUARKUS_PROFILE=dev quay.io/lbroudoux/fruits-catalog-programmez:latest
[...]
Status: Downloaded newer image for quay.io/lbroudoux/fruits-catalog-programmez:latest
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2021-09-14 08:18:06,423 INFO  [io.quarkus] (main) fruits-catalog 1.0.0-SNAPSHOT on JVM (powered by Quarkus 2.2.2.Final) started in 3.028s. Listening on: http://0.0.0.0:8080
2021-09-14 08:18:06,427 INFO  [io.quarkus] (main) Profile dev activated.
2021-09-14 08:18:06,427 INFO  [io.quarkus] (main) Installed features: [agroal, cdi, hibernate-orm, hibernate-orm-panache, jdbc-h2, micrometer, narayana-jta, resteasy, resteasy-jackson, smallrye-context-propagation, smallrye-health]
```

L'application est donc disponible sur http://localhost:8082 et vous pouvez rejouer les requêtes curl suivantes :

```
$ curl localhost:8082/api/fruits

$ curl localhost:8082/api/fruits/31e1c6c8-4d75-4753-918f-d026749e3743

$ curl localhost:8082/api/fruits/31e1c6c8-4d75-4753-918f-d026749e3743 -XPUT -H 'Content-type: application/json' -d '{"name":"Pear", "origin":"France"}'

$ curl localhost:8082/api/fruits -XPOST -H 'Content-type: application/json' -d '{"name": "Lemon", "origin":"Spain"}'
```

Contract testing d'API avec Microcks

Et nous voici presque à la dernière étape de notre Cycle de vie : nous allons valider que l'implémentation développée en Quarkus est bien conforme à la spécification OpenAPI de notre Fruits Catalog API. Pour ce faire, nous allons réutiliser Microcks et sa fonction de *contract-testing*.

Depuis la page de description de Fruits Catalog API - 1.0.0, repérez et cliquez sur le bouton + **New Test**. Vous avez maintenant accès à un formulaire permettant de vérifier un *Endpoint* REST de test. Vous devez saisir ici la valeur http://host.docker.internal:8082/api qui permet au container Microcks d'avoir accès au réseau de votre machine sur le port 8082 où s'exécute l'application Quarkus. Vous devez également choisir un *Runner* - c'est-à-dire un exécutant d'une stratégie de test. Nous choisissons **OPEN API SCHEMA** pour demander à Microcks de valider les réponses obtenues d'après les schémas de données présents dans le contrat OpenAPI. **Figure 15**

Vous pouvez maintenant lancer l'exécution de tests et vous pouvez alors suivre en temps réel l'avancement de celui-ci. Microcks va réutiliser les données des exemples pour interagir avec le *endpoint* indiqué et va ensuite tenter de valider la conformité des données. **Figure 16**

Une fois le test achevé, vous avez la possibilité de cliquer sur **Full results** pour accéder aux détails. Vous aurez alors accès aux codes réponses, aux flux échangés ainsi qu'aux en-têtes HTTP et ceci pour chaque exemple faisant partie du jeu de données du contrat. **Figure 17**

Pratique pour débugger et notamment comparer des résultats différents entre deux environnements !

Et tout çà dans mon workflow et mon pipeline CI/CD !

Nous avons déroulé un cycle complet et sommes prêts à attaquer l'itération suivante mais se pose une question : Comment automatiser et intégrer cette démarche dans mon flux de travail ? Comment faire pour piloter tout cela depuis un pipeline CI/CD ? C'est l'objectif de cette dernière partie.

Tout d'abord, il convient de dissocier trois flux de travail différent correspondant à différents moments et différentes populations : la partie *design*, la partie développement du consommateur et la partie développement du fournisseur d'API. C'est ce que nous avons représenté en **Figure 18.**

Flux de design

Comme nous l'avons vu, ce flux se concentre essentiellement sur Apicurio Studio. Il permet à vos architectes, concepteurs ou experts de travailler de façon collaborative sur la conception d'un contrat d'API. Lorsque le résultat est stable, deux possibilités :

- Pousser directement sa spécification dans Microcks (**1a**) grâce à l'utilisation des API Microcks. Un connecteur existe dans le studio, il est documenté sur cette page : https://www.apicur.io/studio/docs/integrate-microcks-for-mocking-your-api. En fonction de votre organisation, il peut également être intéressant de considérer une synchronisation (**1b**) entre des instances de travail et une instance globale,
- Publier plutôt sa spécification dans un référentiel Git (**2a**) comme nous l'avons montré un peu plus tôt dans l'article. Dans ce cas, on pourra mettre en place une automatisation de pipeline (**2b**) pour pousser cette spécification directement (**2c**) dans un Microcks partagé. Microcks propose différents plugins pour faire cela (Jenkins, Tekton, GitHub Action, CLI), voir https://microcks.io/documentation/automating/.

À titre d'exemple, nous avons implémenté une GitHub Action sur notre référentiel. Le code complet est ici : https://github.com/lbroudoux/programmez-article-api-lifecycle/blob/main/.github/workflows/import-api-specifications.yml et voici les éléments importants :

```
name: import-api-specifications.yml
on: [push]
jobs:
 import:
  runs-on: ubuntu-latest
  environment: Development
  steps:
   # Checkout repository content
   - name: Checkout
    uses: actions/checkout@v2

   - name: Filter artifacts
    uses: dorny/paths-filter@v2
    [...]

   - name: Microcks Import GitHub action
    if: steps.changes.outputs.apiSpecifications == 'true'
   uses: microcks/import-github-action@main
   with:
    specificationFiles: 'contracts/fruits-catalog-openapi-samples-1.0.0.yaml:true'
    microcksURL: ${{ secrets. MICROCKS_URL }}
    keycloakClientId: ${{ secrets. MICROCKS_SERVICE_ACCOUNT }}
   keycloakClientSecret: ${{ secrets. MICROCKS_SERVICE_ACCOUNT_CREDENTIALS }}
```

Flux de développement consommateur

Ce flux est très simple, car le seul changement pour le développeur du consommateur est de pointer sur l'instance Microcks (**3**) proposant des *endpoints* simulant l'API. Ces *endpoints* sont versionnés et toujours en phase avec la source de vérité : la dernière version du contrat d'API présent.

L'impact sur la vitesse d'itération et le temps de livraison totale est énorme : le développeur peut fournir des *feed-backs* immédiatement et n'a pas à attendre de *backend* pour commencer à travailler. Il s'assure également de travailler avec la source de vérité et pas son interprétation de spécifications reçues.

Figure 15

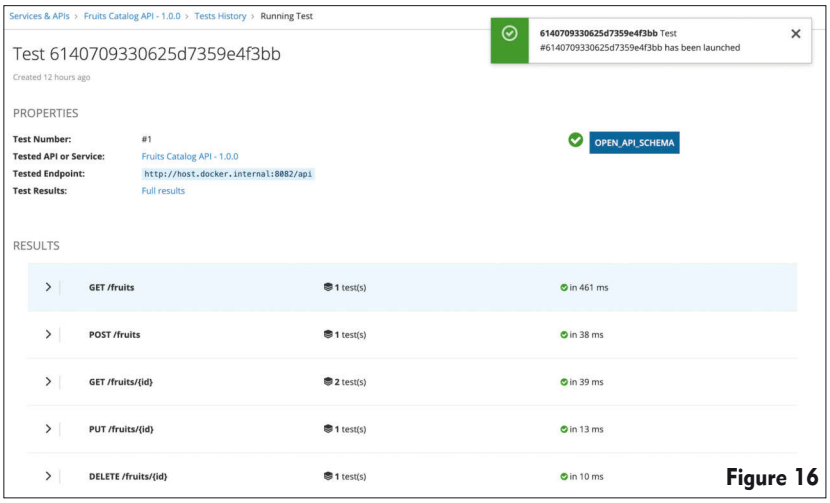


Figure 16

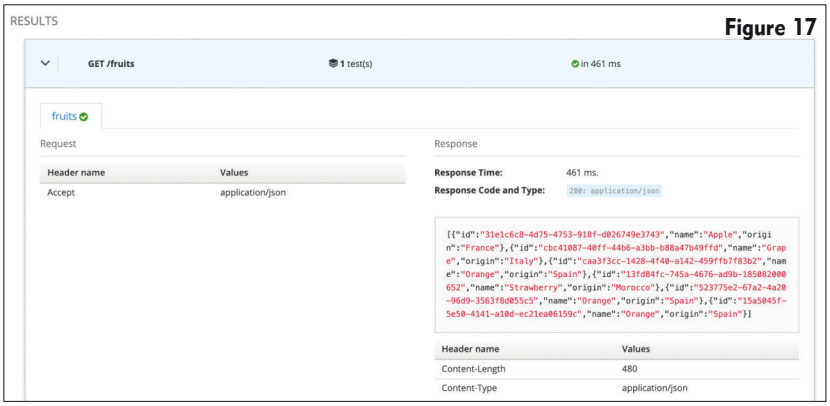


Figure 17

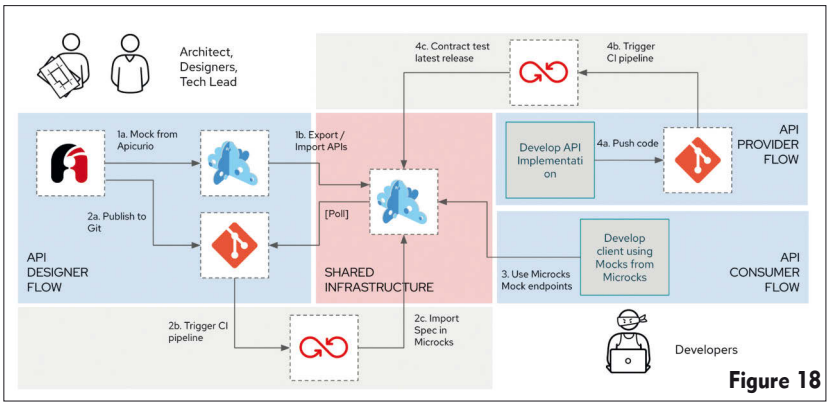


Figure 18

Flux de développement fournisseur

Le flux de développement fournisseur débute avec l'écriture de code, idéalement partiellement généré depuis la spécification OpenAPI comme vu plus haut dans l'article. Lorsque le résultat est stabilisé, le code est poussé dans un référentiel Git (**4a**). On peut alors déclencher un pipeline automatique (**4b**) pour construire, tester unitairement et déployer la nou-

velle version de l'application. Il est également souhaitable de mettre en place du *contract-testing* sur notre API (**4c**). Microcks propose différents plugins pour faire cela (Jenkins, Tekton, GitHub Action, CLI), voir https://microcks.io/documentation/automating/. À titre d'exemple, nous avons implémenté une GitHub Action sur notre référentiel. Le code complet est ici : https://github.com/lbroudoux/programmez-article-api-lifecycle/blob/main/.github/workflows/build-deploy-test-implementation.yml et voici les éléments importants ci-après :

```
name: build-deploy-test-implementation.yml
on: [push]
jobs:
 test:
  runs-on: ubuntu-latest
  environment: Development
  steps:
   # Checkout repository content
   - name: Checkout
     uses: actions/checkout@v2

   - name: Install OpenShift client
     uses: redhat-actions/openshift-tools-installer@v1
     [...]

   - name: Authenticate to OpenShift and set context
     uses: redhat-actions/oc-login@v1
     [...]

   - name: Create/update resources on OpenShift cluster
     [...]

   - name: Filter artifacts
     uses: dorny/paths-filter@v2
     [...]

   - name: Start build on OpenShift cluster
     if: steps.changes.outputs.sources == 'true'
     [...]

   - name: Check during max 3 minutes to allow deployment to be ready
     if: steps.changes.outputs.sources == 'true'
     timeout-minutes: 3
     [...]

   - name: Microcks Test GitHub action
     if: steps.changes.outputs.sources == 'true'
     uses: microcks/test-github-action@main
     with:
      apiNameAndVersion: 'Fruits Catalog API:1.0.0'
      testEndpoint: ${{ secrets. FRUITS_CATALOG_URL }}/api
      runner: OPEN_API_SCHEMA
      microcksURL: ${{ secrets. MICROCKS_URL }}
      keycloakClientId: ${{ secrets. MICROCKS_SERVICE_ACCOUNT }}
      keycloakClientSecret: ${{ secrets. MICROCKS_SERVICE_ACCOUNT_CREDENTIALS }}
      waitFor: 10sec
```

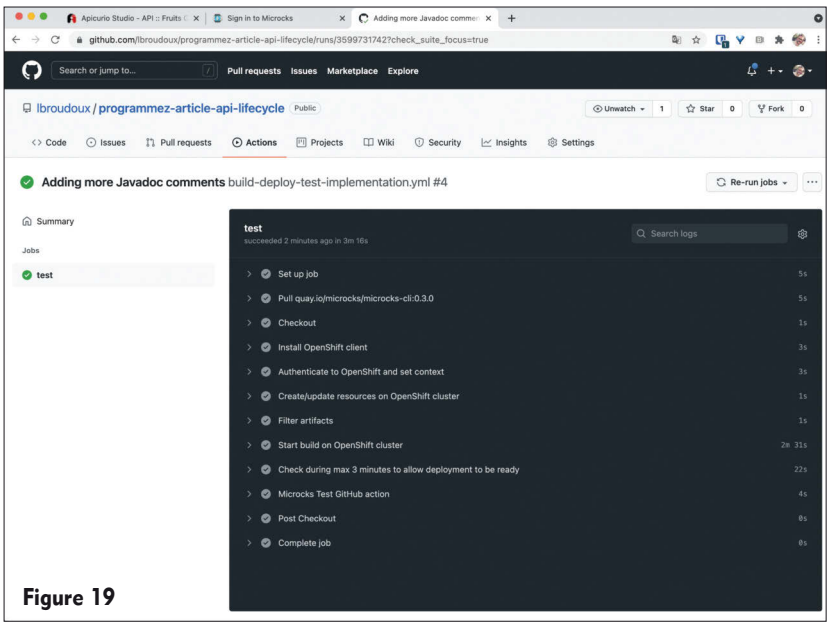


Figure 19

Nous avons exécuté ce pipeline sur un cluster OpenShift. À chaque modification du code, nous réalisons les étapes suivantes :

- Mise à jour des ressources Kubernetes sur le cluster,
- Compilation, tests unitaires et packaging de l'application,
- Déploiement de la nouvelle version d'image de container,
- *Contract-testing* en passant par les *Ingress* et l'ensemble des couches réseaux.

Notre implémentation est *rock-solid*, conforme au contrat et prête à être utilisée par les consommateurs de cette API sur les environnements de *QA* et *staging*. **Figure 19**

Conclusion

Dans cet article, nous avons disséqué un nouveau workflow de production d'une API au travers d'un exemple concret.

Nous avons tout d'abord vu comment on pouvait *designer* une API simplement et rapidement avec l'aide de Apicurio Studio. Nous avons ensuite vu comment Microcks pouvait partir de cette conception et simuler cette API avant qu'elle ne soit développée. La simulation permet aux développeurs ayant besoin de l'API de commencer immédiatement à travailler. Puis, nous avons exploré comment Quarkus pouvait lui aussi réutiliser ce *design* d'API pour accélérer le développement.

Dernière étape de notre Cycle de vie : nous avons validé avec Microcks que notre développement était conforme à la spécification de départ. Point d'orgue de l'article, nous avons passé en revue les différentes façon d'automatiser toute cette démarche au travers de pipelines CI/CD. *What a hell of a ride !* Merci beaucoup aux lecteurs attentifs qui sont restés jusqu'au bout ! En guise de perspectives, je vous propose de poursuivre la lecture de deux blogs posts intéressants :

- La prolongation de ce Full API Lifecycle à la solution 3scale API Management de Red Hat : https://developers.redhat.com/blog/2019/07/26/5-principles-for-deploying-your-api-from-a-ci-cd-pipeline
- Une introduction aux API asynchrones - fondamentales pour l'adoption de l'Event Driven Architecture - et à leur support dans Microcks : https://microcks.io/blog/continuous-testing-all-your-apis/

Bonne API !

Sécurisez vos déploiements dans Kubernetes avec Kube Linter !

Nombreux sont les entreprises et projets Open Source à packager leurs applications pour Kubernetes. Avec la multiplication des contributeurs, il devient critique de s'assurer que ces contributions atteignent un certain niveau de qualité et n'introduisent pas de problèmes de sécurité et de fiabilité. Kube Linter est un projet Open Source qui s'attaque à ce problème avec une approche intéressante : les ressources Kubernetes sont vérifiées depuis leur repository Git dans la chaîne CI/CD.

Nicolas Massé

Solution Architect chez Red Hat. Il aide ses clients à adopter OpenShift ainsi que les outils de sécurité de Red Hat (API Management, Single Sign On, sécurité des conteneurs, etc.). En dehors de Red Hat, il écrit aussi à propos de l'Open Source sur son blog (itix.fr).

Kube Linter se positionne comme une étape automatisée de la chaîne CI/CD, passage obligé pour arriver en production. A chaque commit, il vérifie que les ressources Kubernetes sont cohérentes, fiables et sécurisées.

La cohérence permet de gagner du temps : avant même d'arriver dans les environnements de recette on peut épingler les déploiements qui vont échouer. La fiabilité vise à améliorer la disponibilité de l'applicatif déployé en s'assurant du respect des bonnes pratiques d'exploitation. La sécurité, essentielle à tout applicatif, est assurée en passant en revue toutes les recommandations de sécurité et en s'assurant que l'applicatif déployé applique le principe du moindre privilège.

Kube Linter sait manipuler des ressources Kubernetes nues (fichiers YAML en vrac dans un répertoire) mais également les Charts Helm. Voilà qui le rend adapté aux développeurs (packaging d'applications) mais aussi aux administrateurs qui pratiquent le GitOps !

Installez Kube Linter !

Kube Linter est développé en Go et packagé sous la forme d'un binaire statique pour Linux, macOS et Windows. Vous pourrez le télécharger sur la page GitHub du projet (github.com/stackrox/kube-linter). Sur un système Linux, un simple curl suivi d'un tar suffit à l'installer.

```
curl -sfL https://github.com/stackrox/kube-linter/releases/download/0.2.3/kube-linter-linux.tar.gz | tar -zxv -C /usr/local/bin
```

Sous macOS (et Windows), remplacez "linux" par "darwin" (ou "windows"). Lors de l'écriture de cet article, Kube Linter n'était fourni que pour les architectures x86_64. Si vous êtes sur ARM par exemple, vous devrez passer par une classique compilation des sources.

```
git clone git@github.com:stackrox/kube-linter.git
cd kube-linter
go generate ./...
CGO_ENABLED=0 GOOS=linux GOARCH=arm64 scripts/go-build.sh ./cmd/kube-linter
cp bin/linux/kube-linter /usr/local/bin/
```

L'écosystème Go et Kubernetes évoluant rapidement, j'ai rencontré des difficultés pour la compilation des sources. En particulier j'ai dû remplacer une version de dépendance obsolète.

```
go mod edit -droprequire github.com/openshift/api
go get -d github.com/openshift/api@release-3.9
```

Premiers pas

Prenons un exemple tout simple : un Pod qui utilise l'image busybox pour lancer un processus "sleep".

```
apiVersion: v1
kind: Pod
metadata:
 name: unsecure
spec:
 containers:
 - name: demo
  image: busybox
  resources:
   requests:
    memory: "64Mi"
    cpu: "250m"
  command: [ "sh", "-c", "sleep 1h" ]
```

Vous vous demandez comment un exemple aussi simple pourrait-il présenter un problème, n'est-ce pas ?

Enregistrez ce fichier YAML dans le répertoire courant et appelez la commande kube-linter avec la sous-commande "lint". Notez la présence du point à la fin de la ligne qui indique à Kube Linter de scanner le répertoire courant.

```
kube-linter lint .
```

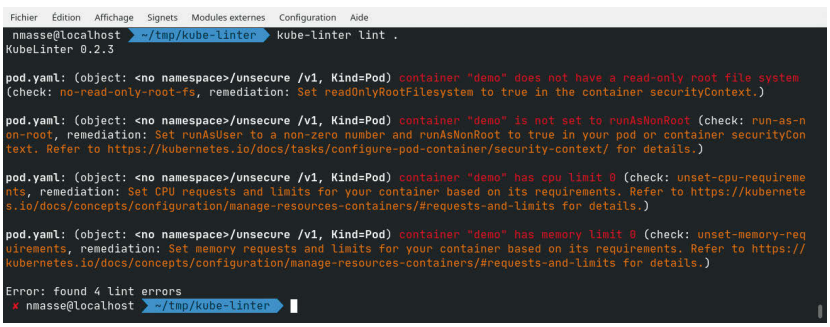
Normalement, vous devriez obtenir quatre erreurs. **Figure 1**

Ces quatres erreurs font partie d'un ensemble de vérifications appliquées par défaut par Kube Linter. Elles peuvent se classer en trois catégories : contrôles de surface, bonnes pratiques et sécurité.

Les contrôles de surface vérifient qu'il n'y a pas d'incohérence dans les ressources scannées, telles que :

- Un service n'a pas de déploiement associé.
- Un champ déprécié (serviceAccount) est utilisé.

Figure 1



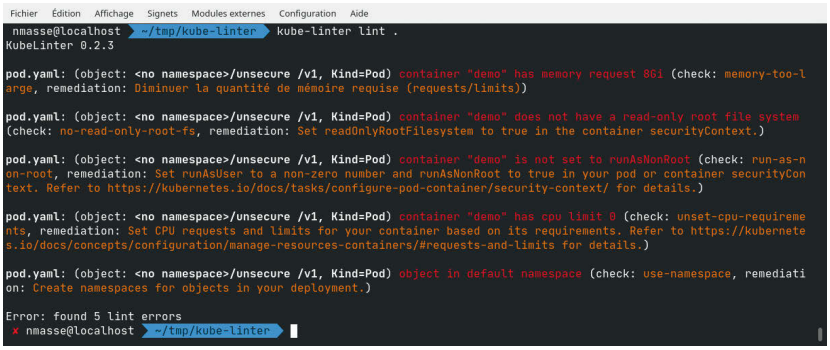


Figure 2

- Les étiquettes d'un déploiement font que les pods créés ne seront pas rattachés au déploiement.
- Le compte de service du déploiement n'existe pas.

La vérification des bonnes pratiques permet de s'assurer que le déploiement sera fiable une fois en production. Cela comprend les point d'attention suivants :

- L'image utilisée pointe sur le tag "latest". Il sera alors difficile de savoir quelle version sera déployée en production.
- Les règles d'anti-affinité n'ont pas été spécifiées. La disponibilité de l'application pourrait être à risque.
- L'utilisation de ressources Kubernetes en version v1beta. Ces APIs ne sont pas encore stabilisées.
- Les ressources CPU et RAM ne sont pas spécifiées. Le déploiement aura une qualité de service de type "best effort".
- Des paramètres sysctl non sûrs sont utilisés.

Et enfin, Kube Linter vérifie que les recommandations de sécurité ont bien été respectées. Il va en particulier détecter si :

- La socket Docker est montée dans le Pod.
- Le Pod conserve la *capability* Linux "NET_RAW".
- Une variable d'environnement contient un secret brut.
- Le Pod partage le namespace *PID*, *IPC* ou *network* de l'hôte.
- Le système de fichier racine n'est pas monté en lecture seule.
- Les paramètres de sécurité permettent une escalade des privilèges.
- Le Pod est privilégié.
- Le Pod tourne en root.
- Certains points de montage sensibles de l'hôte (/proc, /sys, etc.) sont montés dans le Pod.
- Le Pod expose le port 22 (SSH).

Chacune de ces vérifications de sécurité permet de verrouiller un peu plus le déploiement et contenir une éventuelle intrusion. Toutes ces vérifications sont activées par défaut et font partie d'un ensemble plus large. Vous pouvez obtenir la liste complète des vérifications directement depuis la ligne de commande.

```
kube-linter checks list
```

Activez/désactivez des vérifications

Si certaines de ces vérifications créent de fausses alertes, vous pouvez les désactiver en passant le paramètre "exclude" à la sous-commande "lint".

```
kube-linter lint --exclude unset-memory-requirements .
```

À l'inverse, si des vérifications complémentaires vous intéressent, vous pouvez les activer via le paramètre "include".

```
kube-linter lint --include use-namespace .
```

Deux paramètres sont également présents pour activer toutes les vérifications (--add-all-built-in) ou toutes les désactiver (--do-not-auto-add-defaults). À combiner avec les paramètres --include et --exclude pour créer un jeu de vérifications adapté à votre projet !

Pour permettre à tous les membres d'un projet de partager le même jeu de vérifications, créez un fichier .kube-linter.yaml dans le répertoire courant. Les champs exclude, include, doNotAutoAddDefaults et addAllBuiltIn correspondent aux paramètres éponymes de la ligne de commande.

Par exemple, pour inclure la vérification du champ "namespace" et exclude celle des *requests* et *limits* de la mémoire, le fichier de configuration ressemblera à ceci :

```
checks:
 exclude:
 - unset-memory-requirements
 include:
 - use-namespace
```

Configuration avancée

Kube Linter peut être configuré pour réaliser des vérifications spécifiques à votre projet. On instancie pour cela des vérifications personnalisées (*checks*) qui s'appuient sur des gabarits standards (*templates*). Ces derniers sont décrits extensivement dans la documentation en-ligne du projet.

Par exemple, vous pourriez vouloir vous assurer qu'aucun déploiement Kubernetes ne spécifie de pré-requis mémoire trop importants (ce qui pourrait rendre les Pods difficile à ordonnancer s'il n'y a pas de nœud assez large pour les accueillir). Il vous suffirait alors d'ajouter au fichier de configuration .kube-linter.yaml :

```
customChecks:
 - name: memory-too-large
   template: memory-requirements
   params:
     requirementsType: any
     lowerBoundMB: 4096
   scope:
     objectKinds:
       - DeploymentLike
   remediation: Diminuer la quantité de mémoire requise (requests/limits)
```

D'après la documentation, le template memory-requirements épingle les déploiements dont les demandes (ou limites) mémoire sont entre les bornes hautes et basses spécifiées. Pour épingler les déploiements qui utilisent plus de 4 gigaoctets de mémoire, il suffit alors de spécifier une borne basse (lowerBoundMB) à 4 Go.

Editez le fichier pod.yaml pour passer le Pod à plus de 4 Go de RAM.

```
spec:
 containers:
 - name: demo
   resources:
     requests:
       memory: "8Gi"
```

Et vérifiez avec un "kube-linter lint" que Kube Linter épingle bien notre Pod ! C'est la première erreur dans la sortie suivante : **Figure 2**

Repassez le Pod à 64 Mo de RAM et vérifiez que l'erreur disparaît !

Utiliser les annotations kubernetes

Jusqu'à présent, nous avons vu comment adapter Kube Linter aux spécificités de notre projet mais pas comment l'adapter aux multiples spécificités de chaque composant logiciel qui compose ledit projet. Cette adaptation passe par les annotations Kubernetes.

En effet, si un composant logiciel de votre projet nécessite de tourner en root ou qu'il déclenche une fausse alerte sur une des vérifications, vous pourrez légitimement ajouter une exception pour ce composant plutôt que de désactiver globalement la vérification en question.

Dans l'exemple ci-dessus, nous pouvons désactiver la vérification "run-as-non-root" juste pour notre Pod en lui ajoutant l'annotation suivante:

```
ignore-check.kube-linter.io/run-as-non-root: "Ce pod interagit avec le système. Il a besoin d'être root."
```

Il est aussi possible de désactiver toutes les vérifications via l'annotation "kube-linter.io/ignore-all".

Utilisez Kube Linter dans votre pipeline de CI/CD !

Jusqu'à présent nous avons utilisé Kube Linter sur notre station de travail mais pour passer à l'échelle, il va falloir l'insérer dans notre pipeline CI/CD ! Cela permettra de systématiser le contrôle et de s'assurer que toutes les contributions sont conformes au corpus de vérifications choisi.

En particulier, nous montrerons comment Kube Linter s'insère dans un Pipeline Tekton ou un workflow GitHub Action.

Pour illustrer tout ça, nous allons utiliser un repository Git déjà préparé. Pour cela, téléchargez le contenu du repository github.com/nmasse-itix/programmez-kubelinter :

```
git pull https://github.com/nmasse-itix/programmez-kubelinter.git
cd programmez-kubelinter
```

Dans ce repository, vous trouverez :

- Le fichier de configuration Kube Linter utilisé ci-dessus.
- Un répertoire "kube" contenant le fichier "pod.yaml" utilisé ci-dessus.
- Un répertoire ".github" contenant un workflow GitHub Action.
- Un répertoire "tekton" contenant un pipeline Tekton.

Kube Linter dans un Pipeline Tekton

Avez-vous remarqué que votre cluster OpenShift contient un opérateur nommé "OpenShift Pipelines" ? OpenShift Pipelines n'est autre que la distribution Tekton de Red Hat ! Vous pouvez l'installer pour bénéficier d'une CI/CD moderne et légère. Pensez aussi à installer la ligne de commande Tekton (tkn) sur votre station de travail ! Je vous invite à lire l'article de Vincent Demeester dans ce même numéro pour approfondir le sujet. Une fois OpenShift Pipelines installé, créez les ressources Tekton :

```
oc apply -f ./tekton/
```

Cette commande va instancier un namespace "kubelinter" et plusieurs ressources Tekton. Les deux ressources qui nous intéressent ici sont le pipeline et la tâche.

La tâche est une ressource générique et réutilisable. Elle définit comment Kube Linter est exécuté (via l'image stackrox/kube-linter) et quelles sont les ressources nécessaires à fournir en entrée (un repository de type git dans /workspace/src).

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
 name: kubelinter
 namespace: kubelinter
spec:
 resources:
  inputs:
  - name: src
    type: git
 steps:
 - name: kubelinter
  image: stackrox/kube-linter:0.2.3-alpine
  workingDir: /workspace/src
  command: [ "/kube-linter", "lint", "./kube" ]
```

Le pipeline fourni ici à titre d'exemple ne comprend qu'une étape qui exécute la tâche ci-dessus en lui fournissant un repository en paramètre.

Code complet sur programmez.com & github

Exécutez le pipeline via la CLI Tekton :

```
tkn pipeline start kubelinter-sample -n kubelinter --showlog
```

Ou bien via l'interface graphique si vous préférez ce mode. Ouvrez la console OpenShift, naviguez dans **Developer** > **Pipelines**. Ouvrez le pipeline "kubelinter-sample", dépliez le menu **Actions** et sélectionnez **Start**.

Après quelques secondes, vous devriez observer (via la CLI ou l'interface graphique) que le pipeline échoue car Kube Linter identifie des erreurs dans notre fichier pod.yaml.

Kube Linter dans un workflow GitHub Action

Si vous êtes adepte du CI/CD de GitHub, Kube Linter a ce qu'il vous faut : une GitHub Action prête à l'emploi !

Dans le repository téléchargé précédemment, ouvrez le fichier .github/workflows/kubelinter-sample.yaml.

Code complet sur programmez.com & github

Ce sont les cinq dernières lignes qui nous intéressent ici. Elles référencent la GitHub Action du projet Kube Linter ainsi que le répertoire à scanner ("kube") et le fichier de configuration (".kube-linter.yaml"). Attention, le numéro de version de la GitHub Action n'est pas alignée avec le numéro de version du projet Kube Linter !

Forkez le repository via l'interface GitHub, poussez une modification mineure et observez le résultat.

Comme dans l'exemple précédent, le pipeline échoue car Kube Linter identifie des erreurs dans notre fichier pod.yaml.

Conclusion

Dans cet article nous avons découvert comment Kube Linter peut aider à sécuriser et fiabiliser la mise à disposition de ressources Kubernetes. Nous avons également intégré Kube Linter à deux chaînes CI/CD populaires du marché : GitHub Actions et Tekton.

Joël Takvorian

Ingénieur logiciel chez Red Hat, Joël a commencé à contribuer sur Hawkular, une timeseries database utilisé notamment sur OpenShift 3.x, avant de participer à la création de Kiali, la console graphique d'Istio, et parallèlement actif en tant que mainteneur du module de métriques de Vert.x, la toolbox réactive sur JVM. Il travaille désormais sur l'observabilité réseau d'OpenShift au sens large. Retrouvez-le sur Twitter et Medium : @jotak

Istio et Kiali : un mesh pour les gouverner tous

Nous abordons ici le sujet du service mesh, qui se situe à la frontière du Dev et de l'Ops. Le mesh désigne une technologie qui permet de gérer, observer, sécuriser les communications entre services ou workloads par le biais de proxys hautement configurables, interceptant toutes les communications. C'est une grande boîte à outils dans laquelle on pourra piocher les fonctionnalités qui nous intéressent. Pour en citer quelques-unes : rate limiting, traffic shifting, circuit breaking, mutual TLS, distributed tracing, sans oublier tout un ensemble de métriques utiles pour le monitoring, et bien d'autres choses encore.

Avant l'essor du service mesh, certaines de ces fonctionnalités avaient déjà été regroupées au sein de bibliothèques telles que Netflix OSS via des outils comme Hystrix, connus notamment dans le monde Java. Le service mesh offre une approche très différente, dont l'un des grands intérêts est de proposer une couche en grande partie découplée du code et du runtime applicatif : cette couche viendra "enrober" à peu près n'importe quel workload, quel que soit le langage de programmation utilisé. Cela signifie aussi qu'elle sera actionable et configurable à chaud, sans nécessiter de redémarrage de l'application à chaque intervention. Mais par quelle magie est-ce possible ?

Le service mesh pousse à son paroxysme le concept de proxy, ou plus précisément, dans le cadre de Kubernetes, de "sidecar proxy". Dans la suite de cet article, nous allons nous focaliser sur l'environnement Kubernetes qui est le principal contexte d'usage d'un service mesh aujourd'hui, même si rien n'interdit en théorie de transposer le concept dans d'autres environnements. Dans Kubernetes, un sidecar est un conteneur qui accompagne le conteneur principal (l'application) exécuté au sein d'un Pod. Cela lui offre une situation privilégiée pour accéder aux ressources partagées au sein du Pod. Cela garantit aussi une parité, 1 Pod / 1 Proxy. En effet, chaque Pod de l'application composant le mesh embarquera un proxy. Dans le cas d'Istio, il s'agit de proxies Envoy, réputés pour leurs performances, l'étendue de leurs fonctionnalités et leur extensibilité. Ces proxies auront à leur charge d'intercepter le trafic entrant et sortant vers et depuis le Pod, et d'appliquer divers traitements en fonction des configurations que l'on fournit à Istio. Chaque proxy a connaissance de la topologie complète ou partielle du mesh de façon à être autonome pour accomplir certaines tâches comme par exemple de la répartition de charge.

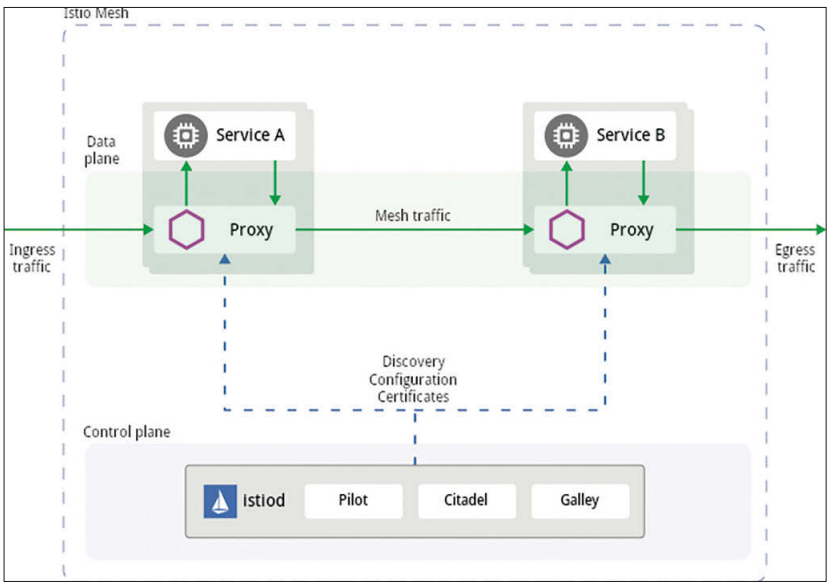


Figure 1. *Architecture d'Istio*

L'ensemble de ces proxies est ce qu'on appelle le data plane d'Istio. Par opposition, le control plane est constitué d'un petit nombre de composants qui maintiennent à jour la topologie du mesh, propagent les configurations vers le data plane, gèrent les certificats, etc. Ces composants sont aujourd'hui réunis au sein d'un unique déploiement nommé Istiod (**Figure 1**).

C'est par le biais de *Custom Resources* que l'on va pouvoir interagir avec Istio pour piloter certains comportements de nos services. Istio définit un grand nombre de CRD, ce qui peut être intimidant au premier abord, mais nul besoin de les maîtriser d'un coup. Comme je l'indiquais en introduction, on peut voir Istio comme une boîte à outil dans laquelle on pioche au cas par cas les fonctionnalités que l'on recherche. Parmi tous les CRD disponibles, si certains sont réservés à des usages avancés que l'on peut ignorer pour commencer (comme pour personnaliser les filtres des proxies Envoy), d'autres représentent le socle de métier d'Istio, par exemple les ressources de type *VirtualService* et *DestinationRule* constituent les briques de base pour aborder la gestion du trafic.

Nous aborderons par la suite certaines des fonctionnalités proposées par Istio. Mais avant cela, un peu d'histoire...

Istio a été créé en 2017 par Google, IBM et Lyft - ce dernier fournissant la brique Envoy responsable du data plane. Le concept de service mesh existait déjà : notamment le concurrent Linkerd apportait un maillage de proxies similaire, mais

à un coût plutôt conséquent pour de grandes flottes de microservices; le tout était alors sur JVM, codé en Scala (choix revisité par la suite avec un mix de Go et Rust). Finalement, le concept était peu exploité, assez anecdotique (quoique prometteur), et l'approche privilégiée, y compris par les géants Netflix / Twitter / Google, restait l'utilisation de bibliothèques dédiées, intégrées aux applications. En introduisant un proxy plus performant pour le cas d'utilisation microservices / conteneurs, et une configurabilité plus poussée, Istio ambitionnait de rebattre les cartes (on peut sans doute affirmer aujourd'hui qu'il y est parvenu).

Et Red Hat ?

Avec son produit phare OpenShift, Red Hat a soutenu et a contribué à l'essor des architectures microservices, conscient aussi bien des opportunités que des challenges qui en résultent. Le service mesh se positionne comme une évolution, un apport à ces architectures. Au-delà de leur simple adoption et intégration avec OpenShift, Red Hat a pris une part active dans leur développement. Dès début 2018 alors qu'Istio n'est qu'en version 0.6, deux initiatives voient le jour : Maistra, dont le but est de proposer une expérience optimale d'Istio avec OpenShift, et Kiali, qui offre une console graphique pour gérer et observer un mesh Istio, que ce soit avec OpenShift ou toute autre distribution de Kubernetes.

Maistra donnera le jour notamment à l'un des tous premiers opérateurs Istio, multi-tenant, ajoutant le support des routes OpenShift ou encore un plugin CNI mieux adapté à la sécurité renforcée d'OpenShift, entre autres choses. Dans sa version "produit", Maistra se nomme Red Hat OpenShift Service Mesh.

Quant à Kiali, il n'y a pas de filiation directe avec OpenShift, le projet est conçu pour Istio lui-même, "upstream". Il en est vite devenu la console de référence, remplaçant l'austère Service Graph d'origine (voir **Figures 2 et 3**). La plupart des captures d'écran suivantes montreront Kiali, afin de donner un bon aperçu global.
Figures 2 et 3

Red Hat s'est hissé parmi les principaux contributeurs au code d'Istio, après Google et IBM (**Figure 4**). En termes de gouvernance, Red Hat fait son entrée dans le comité directeur à la mi-2021 (indépendamment de la maison-mère IBM).

Que trouve-t-on dans la boîte à outils ?

Pour prendre un exemple simple, supposons que nous ayons une application de réservation de voyages faite de trois microservices : un portail web, le service de réservation et un service annexe gérant les codes promotionnels. On ajoute à cela une base de données SQL. Du très classique.

À ce stade, vous vous dîtes peut-être qu'il n'y a guère d'intérêt à déployer un service mesh ? Et pourtant nous pouvons déjà déployer Istio et en tirer deux bénéfices immédiats : le *mTLS* et l'observabilité.

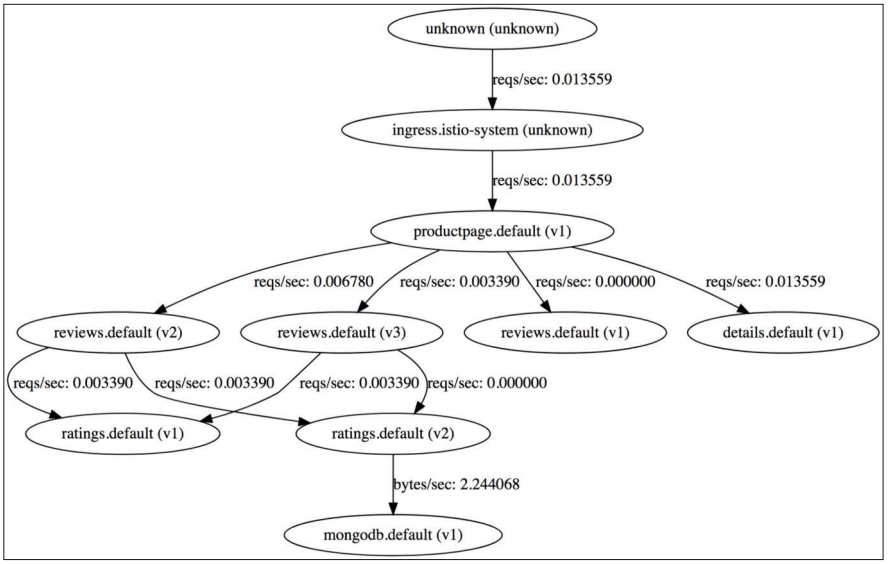


Figure 2. *Le Service Graph d'origine intégré à Istio*

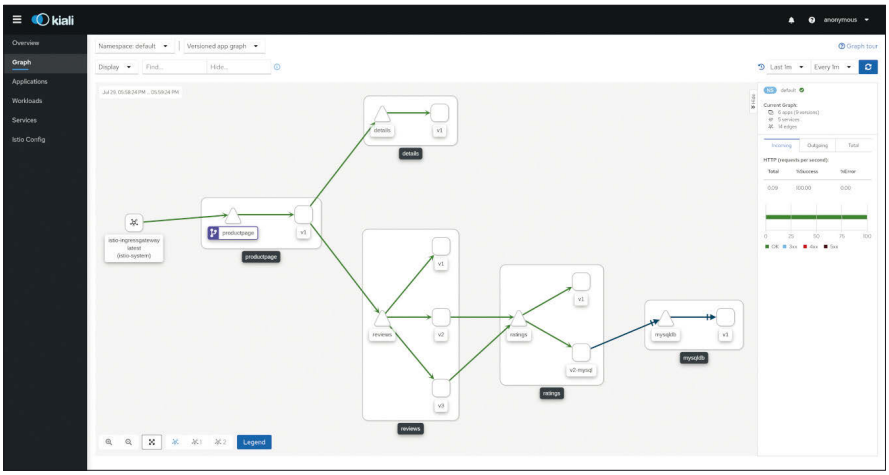


Figure 3. *Même graphe, revisité par Kiali*

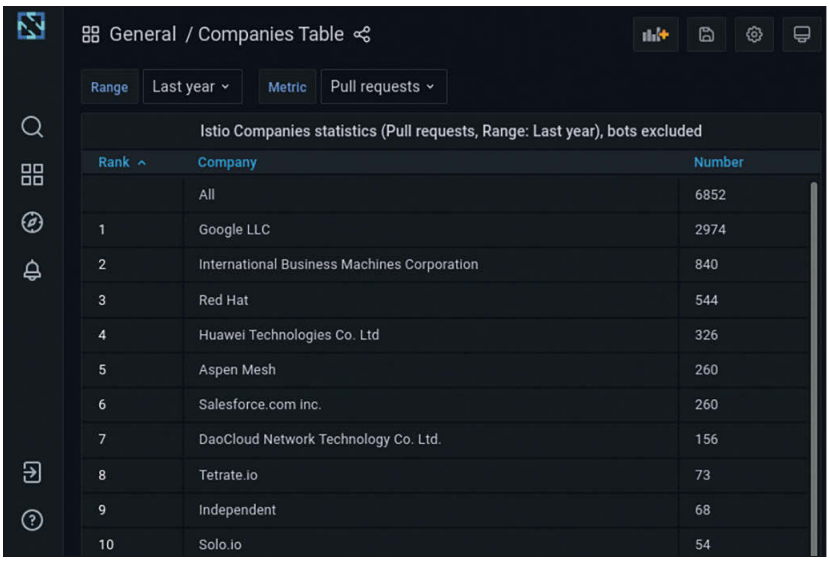


Figure 4. *Entreprises contribuant à Istio (hors Envoy) par nombre de pull-requests, au 1/09/2021, sur l'année écoulée (sans conclusion hâtive : toutes les pull-requests ne se valent pas).*

Depuis Istio 1.5, le *mTLS* automatique est activé par défaut, c'est-à-dire que chaque sidecar chiffre et déchiffre le trafic lorsqu'il communique avec un autre sidecar du mesh. Il est possible de renforcer la sécurité en activant le mode *STRICT*, qui interdit tout trafic non chiffré ; ou à l'inverse d'assouplir ces contraintes en activant le *mTLS* de façon plus granulaire, par namespace ou par workload. Tout cela se configure via une ressource *PeerAuthentication* (groupe API *security.istio.io*).

Note : *le terme "workload" utilisé dans cet article suit la terminologie de Kubernetes, il fait donc référence à un ensemble de Pods gérés par un même contrôleur. Typiquement, il s'agira d'un Deployment, un DaemonSet, un StatefulSet, etc.*

En ce qui concerne l'observabilité, il faudra installer a minima *Prometheus* pour en bénéficier (pour la collecte et le stockage des métriques, plus une interface graphique minimaliste mais souvent utile). Le profil d'installation *demo* propose également *Grafana* (pour les dashboards de métriques), *Jaeger* (pour le *distributed tracing*, nous y reviendrons) et la console graphique d'Istio, *Kiali*.

Une fois le *control plane* installé et fonctionnel, reste à définir le *data plane* : qu'allons-nous intégrer à notre mesh, autrement dit quels pods doivent se voir associer un sidecar proxy ? Deux méthodes sont proposées, l'injection manuelle - via la commande *istioctl kube-inject*, les conteneurs relatifs à Envoy seront ajoutés à la déclaration de vos déploiements - ou l'injection automatique, via le label *istio-injection=enabled* appliqué à un namespace, chaque pod créé au sein de ce namespace sera intégré au mesh.

Les différentes distribution d'Istio peuvent proposer d'autres méthodes d'injection, par exemple Maistra requiert d'annoter non pas un namespace, mais les *Deployments*, afin de ne pas interférer avec d'autres fonctionnalités d'OpenShift telles

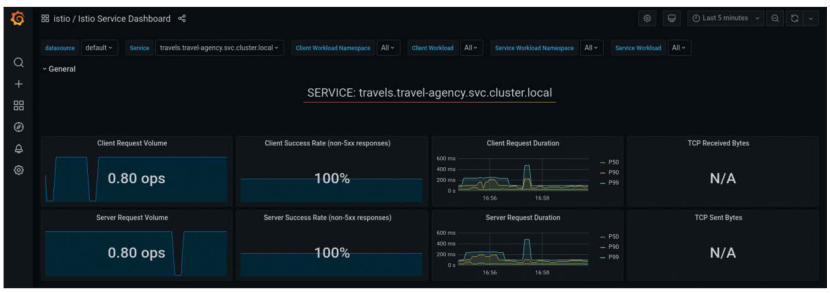


Figure 5. *Aperçu des dashboards Grafana*

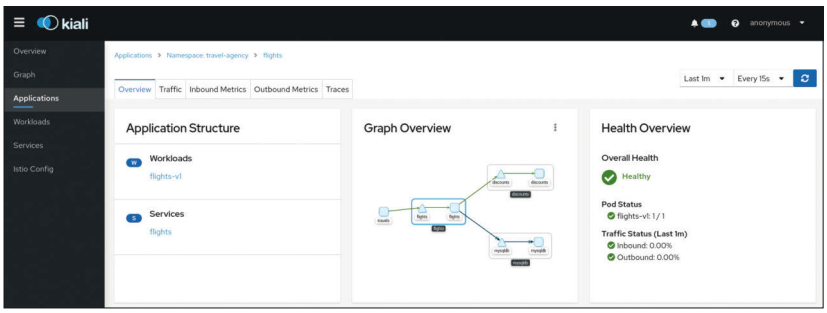


Figure 6. *Aperçu d'un graphe de topologie et des statuts de santé dans Kiali*

que les pipelines de build, qui n'ont pas vocation à être intégrées au mesh.

Lorsque Istio est installé et le mesh défini, les différents dashboards commencent à se remplir et rendent compte du trafic en cours. Chaque requête HTTP incrémente un compteur, incluant des informations telles que la source et la destination, le code HTTP, éventuellement le status GRPC, etc. La taille et le temps de réponse de chaque requête sont également mesurés, ainsi que la volumétrie pour les autres flux TCP.
Figures 5 et 6

Magicien des routes

Supposons maintenant que l'on veuille déployer une nouvelle version du portail web, avec une stratégie Blue/Green (une seule version active à la fois) ou Canari (montée en charge progressive vers la nouvelle version). Cette nouvelle version se présente sous la forme d'un nouveau déploiement ayant le même label d'application (par exemple, *app=travels*) et un label de version différent (*version=v2*). Si nous le déployons tel quel, le Service associé jouera son rôle de load balancer en partageant le trafic à parts égales entre les deux versions, le plus souvent via un algorithme round-robin. Pour éviter cela, nous utiliserons deux ressources : une *DestinationRule* nous permettant d'identifier les destinations candidates, et un *VirtualService* dans lequel nous écrirons une règle de pondération.

La *DestinationRule* :
Code complet sur programmez.com & github

Le *VirtualService* :
Code complet sur programmez.com & github

Ces deux ressources fonctionnent de façon complémentaire, les *subsets* visibles dans le *VirtualService* étant définis dans la *DestinationRule* via une sélection par labels. La distribution de charge se règle ici par l'attribut *weight* (poids), exprimé en pourcentage. **Figures 7 et 8**

Ici, toutes les requêtes suivront cette logique de pondération. On pourra graduellement augmenter la part des requêtes redirigées vers notre nouvelle version, en modifiant le poids. Il existe aussi des outils permettant d'automatiser un déploiement Canari avec Istio, comme Flagger, qui est capable de surveiller le déploiement de nouvelles versions et automatiquement appliquer un scénario de déploiement Canari, augmentant petit à petit la charge tout en surveillant les métriques de santé et éventuellement déclencher un rollback si nécessaire.

Pour s'assurer que la distribution des requêtes reste consistante vis à vis des sessions ou d'une origine, Istio propose de configurer l'algorithme de load-balancing via un mécanisme de *sticky sessions* dans la *DestinationRule*, basé sur un header, un cookie, un query param ou encore l'IP source. Par exemple :

```
# ...
spec:
  host: travels.travel-agency.svc.cluster.local
  trafficPolicy:
```

```
loadBalancer:
  consistentHash:
    httpHeaderName: x-user-id
```

Ainsi, chaque requête provenant d'une même source (dans l'exemple : d'un même "*x-user-id*") sera dirigée vers le même pod.

Plutôt que de traiter toutes les requêtes entrantes suivant le même schéma, il est aussi possible de discriminer ces requêtes selon plusieurs critères. Des règles de *matching* peuvent être ajoutées aux *VirtualServices*. Par exemple, nous pouvons router toutes les requêtes provenant d'un groupe d'utilisateurs donné vers la v2, et le reste vers la v1 : il s'agit de A/B testing. Istio propose plusieurs façons de définir ces groupes : en matchant un ou plusieurs headers, l'URI, le port, les labels du pod source, etc. Le tout avec regexp ou égalité stricte.

Par exemple, supposant que les requêtes soient assorties d'un header *x-country* indiquant le pays de provenance, nous pouvons rediriger tout ou partie des requêtes venant de France vers la v2 :

```
# ...
http:
 - match:
    - headers:
       x-country:
         exact: fr
   route:
    - destination:
       host: travels.travel-agency.svc.cluster.local
       subset: v1
      weight: 0
    - destination:
       host: travels.travel-agency.svc.cluster.local
       subset: v2
      weight: 100
```

Les possibilités sont immenses en termes de routage. On peut imaginer déployer conjointement plusieurs releases candidates pour un nombre restreint de requêtes, et utiliser les métriques et dashboards pour comparer les performances. Ou encore, se livrer à l'énigmatique pratique du traffic shadowing...

Maître des ombres

Le *traffic shadowing*, également connu sous le nom de *mirroring*, consiste à dupliquer des requêtes HTTP vers une destination supplémentaire. Cette technique peut être utilisée dans des stratégies de déploiement pour réduire les risques, notamment comme une amélioration des déploiements Blue/Green. Contrairement à l'exemple du Canari abordé précédemment, l'idée ici sera de déployer une nouvelle version d'un service, mais sans le releaser, et tout en ayant un aperçu du comportement de notre candidat en quasi-production.

Dans un *VirtualService*, le *shadowing* s'active de la façon suivante :

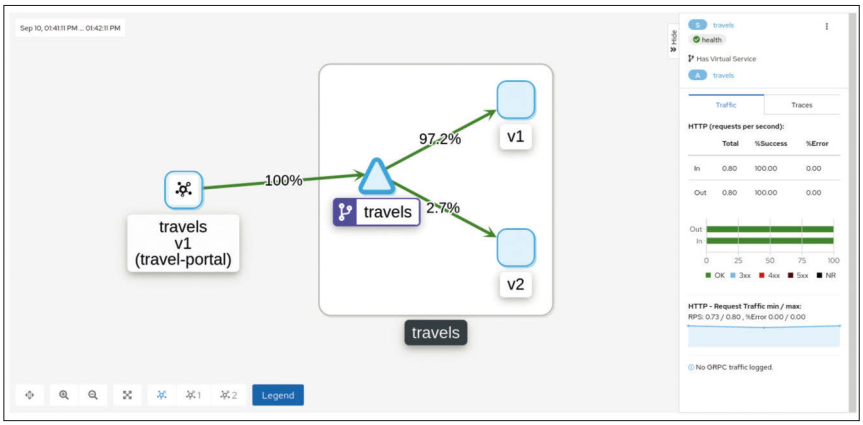


Figure 7. *Distribution des requêtes lors d'un déploiement Canari*

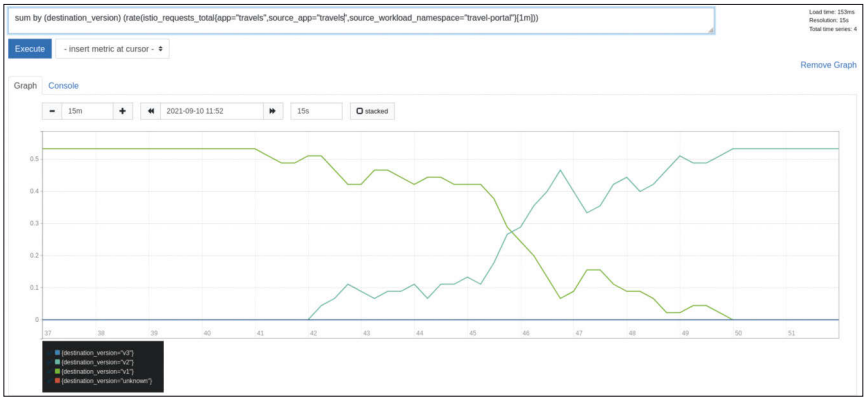


Figure 8. *Volumétrie par version au cours du déploiement Canari*

```
http:
 - route:
    - destination:
       host: travels.travel-agency.svc.cluster.local
       subset: v1
      weight: 100
   mirror:
    host: travels.travel-agency.svc.cluster.local
    subset: v2
   mirrorPercentage:
    value: 100
```

Lorsqu'une requête est émise vers le service *travels* (v1), le proxy Envoy génère une copie de cette requête, adressée à la version v2. L'émetteur n'a pas connaissance du shadowing : de son point de vue, il n'a effectué qu'une seule requête, et ne recevra qu'une seule réponse. En revanche, le workload v2 recevra bien du trafic, Istio génèrera les métriques habituelles qui pourront permettre par exemple de qualifier, ou non, ce déploiement. On minimise ainsi les risques en production avant de releaser.

... Enfin, à condition de bien faire les choses ! Car attention aux pièges, si la v2 est par exemple branchée sur la base de données de production, il y aura bien des conséquences réelles. L'activation du *traffic shadowing* ne se fait pas à la légère. Le trafic dupliqué est annoté par l'ajout du suffixe "*-shadow*" aux headers Host/Authority, permettant aux workloads candidats d'agir en conséquence pour éviter tout effet de bord indésirable.

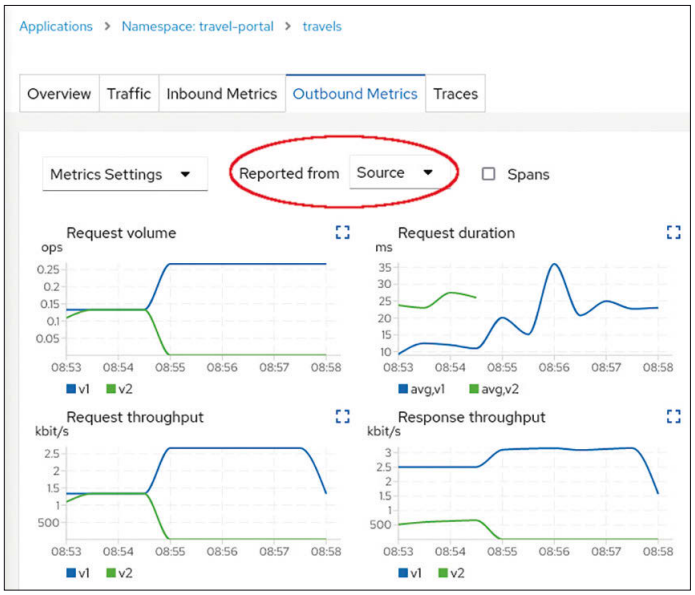


Figure 9. *Traffic shadowing vu depuis la source: le trafic vers la v2 semble s'être interrompu vers 8:55 (passage d'un LB 50/50 au shadowing)*

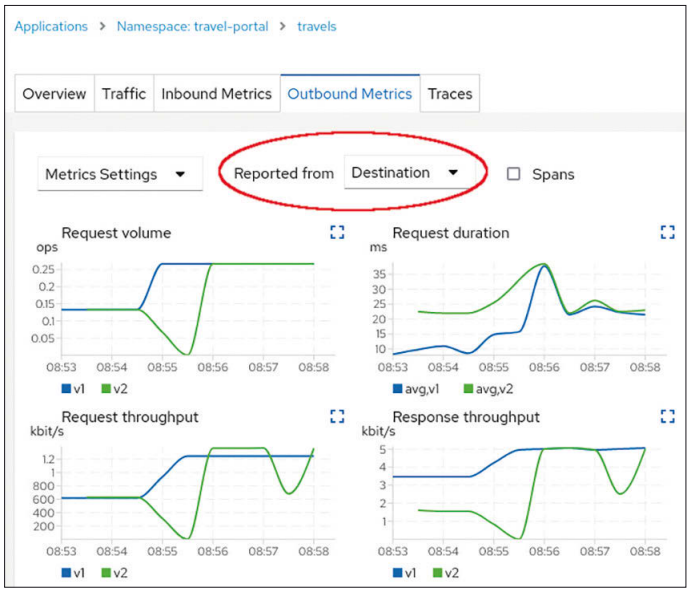


Figure 10. *Traffic shadowing vu depuis la destination: vers 8:55, la totalité du trafic atteint la v1… mais aussi la v2!*

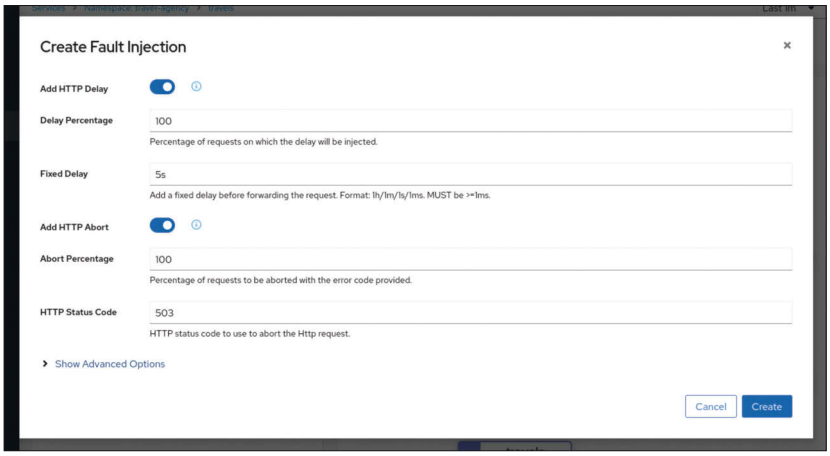


Figure 11. *Injection de chaos via Kiali*

Pour approfondir les différents patterns relatifs au *traffic shadowing*, je recommande la lecture du blog de Christian Posta, "*Advanced Traffic-shadowing Patterns for Microservices With Istio Service Mesh*" [1].

Les **figures 9 et 10** montrent comment le trafic shadowing peut être mis en évidence dans Kiali, en changeant le point de vue ("Reported from Source / Destination").

Cavalier du chaos

Nous avons vu plusieurs patterns permettant globalement de réduire les risques lors des déploiements et des releases. Et si pour réduire encore les risques, il fallait en prendre? La résilience ne vient-elle pas de l'exposition aux risques? C'est le principe du chaos engineering : détruire des pods, générer des erreurs, ajouter de la latence ; bref, provoquer volontairement du chaos. Si l'idée de bousculer un peu sa production peut paraître effrayante de prime abord, elle est en fait un moyen de s'assurer de la résilience des systèmes et ainsi de gagner en confiance.

Le chaos engineering n'est certes pas le cœur de métier d'Istio, et d'autres outils sont dédiés à cela, notamment le vénérable pionnier Chaos Monkey développé par Netflix dès 2011, ou Chaos Mesh, une plateforme dédiée à Kubernetes offrant une grande variété de possibilités. En comparaison, Istio n'est pas la solution la plus complète dans ce domaine. Par exemple, il n'a pas vocation à détruire aléatoirement des pods pour créer du chaos. En revanche, il peut agir sur la partie réseau, qui correspond bien à son champ d'action. Pour qui utilise Istio et s'intéresse au chaos engineering, il s'agit d'une option intéressante.

En éditant nos *VirtualServices*, ou en trois clics dans Kiali, nous pouvons configurer les proxies Envoy pour qu'ils génèrent des erreurs aléatoirement, ou qu'ils ajoutent une latence factice. L'intérêt réside dans la surveillance des autres services liés, de près ou de loin : continueront-ils à fonctionner relativement bien, ou aura-t-on droit à des dysfonctionnements en cascade? **Figure 11**

Chasseur de traces

Au début de cet article, j'ai brièvement évoqué *Jaeger* et le distributed tracing. En deux mots, le distributed tracing consiste à tracer au runtime les transactions logiques ou métier sous forme d'une chaîne (d'un arbre) d'événements liés par une relation de cause à conséquence, qui peuvent avoir lieu au sein d'un processus, d'une machine ou d'un réseau, le tout étant mesuré dans le temps. Les données sont stockées dans une base dédiée pour permettre des recherches ou analyses ultérieures, notamment pour du *troubleshooting*. Il est souvent considéré comme l'un des piliers du monitoring moderne. *Zipkin*, *Jaeger* ou *Apache SkyWalking* sont des logiciels de distributed tracing.

Le rapport avec Istio ? Habituellement avec le distributed tracing, il revient au développeur d'instrumenter son code pour initier une *trace* (c'est-à-dire une transaction logique ou métier), pour y ajouter des *spans* (c'est-à-dire un événement constitutif de la trace) et propager le contexte des traces lors des communications sur un réseau. De par le maillage de

proxies fourni par Istio, celui-ci est capable de mâcher une partie de ce travail.

Lorsqu'un service A appelle un service B, le proxy de A crée une trace constituée d'un span racine et ajoute un contexte de tracing dans les headers de la requête (suivant le standard OpenTracing). À la réception, le proxy de B lit ces headers, en déduit le contexte de tracing, et ajoute un nouveau span à la trace. Ainsi, Istio crée automatiquement des traces minimalistes de deux spans chacune pour chaque appel sur le réseau.

À noter que seuls les appels HTTP sont concernés, puisque la propagation du contexte de tracing implique de transmettre des métadonnées selon un procédé normé (en l'occurrence les headers HTTP), chose impossible avec seulement le protocole TCP. Cela ne signifie pas qu'on ne puisse pas tracer les connexions TCP, mais cela se gère au cas par cas et Istio ne pourra pas le faire à votre place. Si l'on prend l'exemple de Kafka, un message dispose également de headers, mais Istio ne distinguant pas les flux Kafka de tout autre flux TCP, il vous reviendra d'implémenter la propagation de contexte avec les headers Kafka (ou d'utiliser une bibliothèque supportant cela).

Ces mini-traces d'Istio ne sont pas franchement très utiles en l'état. Lorsque le service B a reçu une requête provenant de A, peut-être a-t-il dû contacter C en conséquence. Cela, Istio ne va pas le deviner, car il s'agit de la logique du code applicatif, sur lequel Istio n'a aucun regard. Istio voit bien que le service C a été contacté, mais ne peut pas établir la causalité. Pour obtenir cette corrélation A -> B -> C, il va falloir intervenir dans le code : il faut récupérer le contexte de tracing des headers de la requête A -> B, et le réinjecter dans les headers de B -> C. Les headers en question sont ceux préfixés par "*x-b3-*" ainsi que le "*x-request-id*" créé par Envoy. Voici un exemple simple, en go, pour accomplir cette propagation :

```
func propagateHeaders(in *http.Request, out *http.Request) {
    headers := []string{
        "x-request-id",
        "x-b3-traceid",
        "x-b3-spanid",
        "x-b3-parentspanid",
        "x-b3-sampled",
        "x-b3-flags",
    }
    for _, header := range headers {
        out.Header.Add(header, in.Header.Get(header))
    }
}
```

Alternativement, il existe de nombreuses bibliothèques dédiées au tracing, notamment hébergées sur le github de Jaeger (*jaegertracing*), permettant non seulement de gérer la propagation de contexte, mais aussi pour créer de nouveaux *spans* que l'on pourra intégrer aux traces générées par Envoy, par exemple pour rendre compte de décisions prises dans le code, mesurer la durée de tâches de calculs, etc.

En terme d'observabilité, le distributed tracing se marie particulièrement bien avec le service mesh, car ce dernier apporte une consistance qui va permettre de plus facilement corréler traces, logs et métriques, et ainsi faciliter le travail d'investigation et de *troubleshooting*. La corrélation entre traces et métriques permet non seulement d'affiner et de progresser au cours d'une investigation (les métriques offrant un aperçu agrégé, "grand angle" tandis que les traces fournissent une vue plus précise, "zoomée"), mais au-delà de ça, cette corrélation permet aussi d'afficher des traces sur un graphe de micro-services, ou encore d'évaluer la performance de traces individuelles au regard d'indicateurs plus généraux et ainsi faciliter la localisation d'éventuels problèmes méritant une analyse approfondie. C'est ce que l'on s'efforce de proposer dans Kiali.

Pour cela, Kiali génère des *heatmaps* qui permettent d'apprécier en un coup d'œil la qualité des traces, ou celle des spans au sein des traces, *qualité* s'entendant ici en termes de temps de réponse comparativement aux statistiques tirées des métriques - prises sur une durée allant des dernières 10 minutes à la dernière heure - et incluant la comparaison à la moyenne, au p50, p90 et p99. Les traces contenant des erreurs (codes 5xx) sont aussi directement identifiables en rouge.

Ces corrélations visuelles permettent de raccourcir les temps d'investigation. Pour prendre un exemple, la **figure 12** montre comment corréler une augmentation des temps de réponse et une augmentation de la mémoire utilisée, résultant d'un upscaling.

The end has no end

Il y aurait encore à dire non seulement sur Istio mais également Envoy, qui à lui seul est une pièce puissante mais assez complexe : configurable et extensible grâce au support de filtres WASM, cela permet de brancher vos propres filtres personnalisés en entrée ou sortie de toute connexion ou requête. Là encore, Kiali s'efforce de rendre la tâche plus aisée, notamment dans ses dernières versions qui mettent l'accent sur une meilleure visibilité des informations extraites d'Envoy (logs, configs, métriques, etc.).

Vous trouverez sur le site https://kiali.io, rubrique *Tutorial*, des exemples approfondis basés sur cette même démo d'application de réservation.

Notes

[1] https://blog.christianposta.com/microservices/advanced-traffic-shadowing-patterns-for-microservices-with-istio-service-mesh/

Figure 12. *Corrélation nette avec une augmentation de la mémoire, et accroissement marqué de l'uptime, révélateurs d'un upscaling*

Programmez un opérateur en Java avec Quarkus et Java Operator SDK!

Java Operator SDK. un projet open source initié par Container Solutions et auquel Red Hat contribue, propose une architecture simplifiant la création d'opérateurs Kubernetes en Java. Dans cet article, nous rappellerons ce que sont les opérateurs et pourquoi il peut être intéressant d'en créer en Java avant d'examiner un exemple concrêt utilisant JOSDK et son extension `quarkus-operator-sdk` pour Quarkus.

Christophe Laprun

Ingénieur logiciel passionné, il allie expertise technique et communication pour comprendre les besoins de ses utilisateurs, avec un soin particulier mis sur l'utilisabilité. Son sujet de prédilection ces derniers temps est l'expérience développeur ciblant Kubernetes et l'IT "vert" sans jamais oublier l'idée que la technologie ne doit jamais perdre son but premier : permettre à ses utilisateurs d'accomplir plus facilement leur tâches. Christophe est le développeur principal du Java Operator SDK et de son extension Quarkus. Vous pouvez le trouver sur Twitter (@metacosm) et GitHub [https://github.com/metacosm]

Cet article s'adresse essentiellement aux développeurs Java intéressés par l'écriture d'opérateurs en Java en utilisant Quarkus, un framework Java développé par Red Hat qui permet aux applications Java de s'adapter aux contraintes du Cloud. Il n'est pas nécessaire d'être un expert en opérateurs, ni même en Kubernetes. De même, bien qu'il n'y ait pas besoin d'être un expert de Quarkus, une compréhension des concepts de base est nécessaire : en particulier, le concept d'extension, la configuration des applications, l'utilisation du "Dev Mode" et la compilation native.

L'exemple que nous allons développer utilise Java 11 ainsi que les outils en ligne de commande `operator-sdk` (qui peut être installée en suivant les instructions à [https://sdk.operatorframework.io/docs/installation/]) et `kubectl`. Il devrait fonctionner sur tout cluster Kubernetes suffisamment récent sur lequel le "controller" d'Ingress NGINX [https://kind.sigs.k8s.io/ docs/user/ingress/#ingress-nginx] a été installé. Pour notre part, nous avons testé l'exemple en utilisant un cluster Kubernetes 1.21 local installé avec `kind` via le script disponible à [https://github.com/snowdrop/k8s-infra/tree/master/kind#how-to-installuninstall-the-cluster]. Notre opérateur assume une mise en place similaire (en particulier en ce qui concerne la partie `Ingress`).

Opérateurs : introduction

Kubernetes est devenu la plateforme de choix pour le déploiement d'applications sur le cloud. Cette plateforme n'est néanmoins pas évidente à aborder pour un utilisateur, notamment du fait des relations entre les différentes ressources nécessaires pour configurer une application. Il y a donc une opportunité évidente pour essayer d'en simplifier son utilisation, en automatisant la création des ressources nécessaires pour un type d'application donné.

De manière très simplifiée, un utilisateur interagit avec la plateforme Kubernetes en communicant au cluster l'état dans lequel il désire le placer. Ceci s'accomplit la plupart du temps en spécifiant l'état désiré d'un ensemble de ressources natives de Kubernetes, état matérialisé par un fichier JSON ou YAML envoyé au cluster via `kubectl`. Cet état désiré est ensuite pris en charge par un "controller", un processus qui surveille l'état des ressources du cluster et qui fait en sorte de réconcilier l'état courant de la ressource avec son état désiré.

Les opérateurs (ou "operator") Kubernetes fonctionnent de la même manière avec néanmoins une différence importante: alors que Kubernetes gère automatiquement ses ressources natives (i.e. celles qui font partie de la plateforme), les opérateurs, eux, sont capables de prendre en charge des types de ressources a priori inconnus de la plateforme. Ceci s'effectue via le mécanisme d'extension de Kubernetes sous la forme des "Custom Resources" (CR) auxquelles sont associées des "controllers" chargés de les prendre en charge. Formellement, il y a très peu de différences entre les ressources natives de Kubernetes et les CR : la différence principale étant que, dans le cas des ressources natives, les "controllers" sont fournis par la plateforme tandis que l'utilisateur doit fournir son propre "controller" dans le cas des CR. La combinaison de ces deux parties permet de créer un DSL (Domain-Specific Language, langage spécifique à un domaine) défini par la CR et pris en charge par le controller associé. Ce DSL permet aux utilisateurs de se concentrer sur les aspects métiers tandis que le controller se charge de concrétiser l'état désiré associé sur le cluster, généralement en créant/modifiant un ensemble de ressources Kubernetes natives, mais aussi parfois des ressources externes.

L'intérêt des opérateurs est donc d'étendre la plateforme Kubernetes en lui ajoutant des règles métier qui sont propres à une organisation donnée et ses besoins. Une fois un opérateur installé et configuré sur un cluster, tout utilisateur du cluster peut ainsi avoir accès à une automatisation et son DSL associé sans avoir à se soucier des détails techniques de la plateforme, facilitant ainsi son utilisation.

Pourquoi écrire des opérateurs en Java?

Kubernetes est écrit en Go et, traditionnellement, les opérateurs aussi. Il faut dire que ce langage est particulièrement adapté. D'autre part, il y a plusieurs projets en Go destinés à simplifier l'écriture d'opérateurs: `operator-sdk` et son outil en ligne de commande qui permet de démarrer plus rapidement, `client-go` [https://github.com/kubernetes/client-go/] qui permet d'interagir avec le serveur d'API de Kubernetes de manière programmatique tandis qu`apimachinery` [https://github.com/kubernetes/apimachinery] et `controller-runtime` [https://http://github.com/kubernetes/controller-runtime] fournissent des fonctions et des schémas utiles pour faciliter l'écriture d'opérateurs.

Pourquoi alors utiliser Java? C'est le langage d'applications d'entreprise par excellence et ces applications, souvent complexes, bénéficieraient de mécanismes simplifiés pour les

déployer sur Kubernetes. Par ailleurs, l'approche DevOps veut que les développeurs des applications soient aussi chargés de leur mise (et maintien) en production. Utiliser le même langage pour toutes les étapes du cycle de vie de l'application est donc une proposition attractive.

Un des freins à l'utilisation de Java pour écrire des opérateurs était l'absence de framework similaire à ce qui existe en Go pour simplifier le processus. Heureusement, il existe des clients Kubernetes écrits en Java qui aident la communication avec le serveur mais, aussi utiles soient-ils, les abstractions offertes restent du niveau de ce que `client-go` offre aux développeurs Go. C'est à ce stade qu'intervient Java Operator SDK (JOSDK [https://javaoperatorsdk.io]). Conçu pour simplifier le travail des développeurs, son architecture s'occupe de la gestion bas-niveau des évènements issus de Kubernetes pour permettre aux développeurs Java de se concentrer plutôt sur les aspects métier de leur opérateur.

Par ailleurs, le développement de Quarkus par Red Hat a permis d'améliorer les caractéristiques à l'exécution des applications Java permettant d'améliorer les aspects de performance au démarrage et de consommation mémoire qui ont traditionnellement fait défaut. En déportant à la compilation des traitements qui sont le plus souvent faits au moment de l'exécution par les frameworks Java traditionnels, Quarkus permet des gains de performances appréciables. Par ailleurs, son support pour la compilation native des applications Java permet d'obtenir des applications avec un profil similaire aux applications Go.

Reconnaissant cette opportunité offerte par Quarkus, Red Hat a donc créé une extension `quarkus-operator-sdk`, simplifiant encore plus l'écriture d'opérateurs avec Quarkus, notamment en automatisant des tâches répétitives durant le développement. Red Hat a également développé un plugin pour la ligne de commande d'`operator-sdk` permettant de créer rapidement un projet squelette utilisant JOSDK et son extension Quarkus. Nous allons mettre en œuvre ces trois projets lors de notre exemple.

Cas d'utilisation et architecture

Déployer une application sur Kubernetes nécessite la création de plusieurs ressources associées: il faut a minima créer un `Deployment` et un `Service` associé. Par ailleurs, il faut également créer un `Ingress` (ou une `Route` sur OpenShift) pour exposer l'application en dehors du cluster. Tout ceci n'est certes pas si compliqué mais pour un développeur qui n'a envie de se soucier que d'écrire son application et non pas des détails à mettre en œuvre pour la déployer sur le cluster, c'est une charge de travail supplémentaire. Automatiser le processus est donc intéressant.

Bien évidemment, nous allons grandement simplifier ce cas d'utilisation en ne traitant que le cas particulier d'une application donnée (en l'occurrence, un simple `Hello World` écrit avec Quarkus) mais l'on pourrait imaginer de partir de ce concept pour développer un opérateur plus robuste et général à partir de ce simple scénario. Il faudrait, par exemple, indiquer à notre opérateur quel port doit être exposé pour l'application en question. Dans notre cas, nous exposerons le port 8080 automatiquement.
Tout ceci étant posé, voici à quoi ressemblerait un exemple simple de notre `ExposedApp` CR:

```yaml
apiVersion: "example.com/v1alpha1"
kind: ExposedApp
metadata:
 name: hello-quarkus
spec:
 imageRef: <référence d'une image Docker>
```

Notre but est d'écrire un opérateur capable d'être notifié quand des CR de type `ExposedApp` sont ajoutées, modifiées ou détruites du cluster. En termes d'architecture, en utilisant JOSDK, cela implique de créer une classe représentant notre `CustomResource` puis ensuite un "controller" capable de la prendre en charge en implémentant l'interface `ResourceController`, annotée avec `@Controller`.

Le SDK fournit une classe `Operator` qui gère les différents "controllers" ainsi que l'infrastructure permettant la gestion des évènements bas-niveau envoyés par le serveur d'API de Kubernetes ainsi que de leur envoi sur les méthodes appropriées des "controllers". **Figure 1**

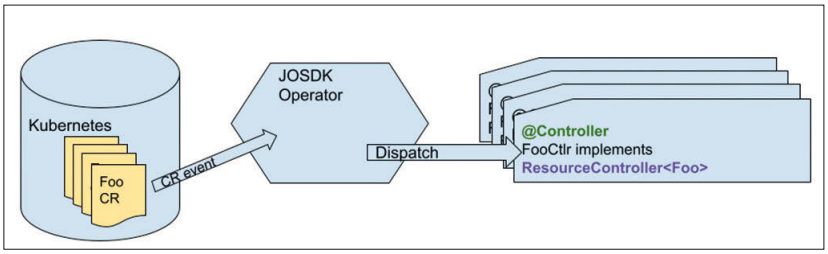
Ainsi, un évènement de création d'une ressource de type `ExposedApp` sera, par exemple, envoyé automatiquement avec la représentation de la nouvelle ressource sur la méthode `createOrUpdateResource` du `ResourceController` associé, qui pourra alors implémenter la logique appropriée pour réconcilier l'état du cluster avec le nouvel état désiré par l'utilisateur et exprimé par cette nouvelle CR. Il n'y a pas besoin de gérer la création d'un `Watcher` ou `Informer`, comme ce serait le cas en utilisant un client directement, pour écouter les évènements, le SDK s'en occupe automatiquement.

Il fournit également une architecture de cache ainsi qu'un mécanisme de gestion des erreurs avec notamment une gestion de nouvelles tentatives graduées quand une exception arrive.

Implémentation

Nous allons voir comment nous pouvons implémenter notre opérateur (ou plus spécifiquement, notre "controller") en utilisant JOSDK et son extension pour Quarkus. L'outil `operator-sdk` nous permet de mettre en place les bases d'un projet très rapidement grâce au plugin `quarkus`.
La commande à utiliser est :

Figure 1



```shell
> operator-sdk init --plugins quarkus --domain halkyon.io --project-name expose
> Next: define a resource with:
> $ operator-sdk create api
```

Nous spécifions que nous voulons initialiser un projet avec le plugin `quarkus` en utilisant le nom de domaine `halkyon.io`, nom utilisé pour le groupe associé à notre CR et aux packages Java de notre projet. `operator-sdk` génère les fichiers suivants :

```shell
.
├── Makefile
├── PROJECT
├── pom.xml
└── src
    └── main
        ├── java
        └── resources
            └── application.properties
```

Elle nous indique aussi, comme nous pouvons le voir sur la structure du projet, que nous avons encore du travail à faire ! Le résultat de cette première étape peut être vu à [https://github.com/halkyonio/exposedapp/tree/step-1].

Démarrons à présent Quarkus en utilisant le Dev mode afin de pouvoir développer notre opérateur alors qu'il tourne :

```shell
mvn quarkus:dev
```

À ce stade, notre projet ne fait pas grand-chose à part mettre en place le code nécessaire pour créer une application Quarkus dans laquelle l'"operator" venant du SDK est injecté et démarré. Tout ceci est fait de manière transparente et nous n'avons pas encore écrit une seule ligne de code. De fait, comme `operator-sdk` nous l'indiquait plus tôt, il n'existe pas encore de "controller" et le SDK nous le fait savoir :

```shell
ERROR [io.qua.run.Application] (Quarkus Main Thread) Failed to start application (with profile dev): io.javaoperatorsdk.operator.OperatorException: No ResourceController exists. Exiting!
    at io.javaoperatorsdk.operator.Operator.start(Operator.java:71)
    at io.quarkiverse.operatorsdk.runtime.AppEventListener.onStartup(AppEventListener.java:31)
...
```

Ajoutons donc une implémentation de `ResourceController` et une classe pour représenter notre CR. JOSDK utilise le client Kubernetes Fabric8 comme couche de communication avec le cluster. L'équipe du client a récemment amélioré le support des CR de manière significative et nous allons pouvoir bénéficier de ces améliorations ici. Pour représenter une CR avec le JOSDK, il nous suffit d'étendre la classe `CustomResource`. De manière générale, il est recommandé, lors de la conception de CR, de n'utiliser que deux champs composés (outre les champs traditionnels des ressources Kubernetes): `spec` et `status`. L'idée est de séparer proprement l'état spécifié par l'utilisateur et qui doit donc être sous son contrôle (la spécification ou `spec`) de l'état actuel de la ressource, communiqué à l'utilisateur, qui ne peut le modifier, par le controller et donc sous le contrôle de ce dernier : le `status`. Cette dichotomie est facilitée par la classe `CustomResource` qui est paramétrée par un type associé à la `spec` et un autre associé au `status`.

Définition itérative de notre CR

Définir une CR revient à définir une API, un contrat avec le cluster. De ce fait, l'outil `operator-sdk` définit une commande `create api` pour ajouter les classes requises à notre projet:

```shell
operator-sdk create api --version v1alpha1 --kind ExposedApp
```

Cette commande crée quatre fichiers dans notre projet: une classe `ExposedApp` représentant notre CR en version `v1alpha1` et à laquelle sont associées une classe pour la `spec` et le `status`, respectivement: `ExposedAppSpec` et `ExposedAppStatus`. Un "controller" configuré pour prendre en charge notre CR est également créé : `ExposedAppController`. Le résultat de cette opération peut être examiné à [https://github.com/halkyonio/exposedapp/tree/step-2].

Observons que Quarkus redémarre automatiquement l'application après ces changements et commence à travailler :

```shell
INFO [io.qua.ope.dep.OperatorSDKProcessor] (build-26) Registered 'io.halkyon.ExposedApp' for reflection
INFO [io.qua.ope.dep.OperatorSDKProcessor] (build-26) Registered 'io.halkyon.ExposedAppSpec' for reflection
INFO [io.qua.ope.dep.OperatorSDKProcessor] (build-26) Registered 'io.halkyon.ExposedAppStatus' for reflection
...
```

Nous pouvons observer que les classes associées avec notre CR ont été enregistrées pour être accédées via la réflection de Java. Ceci est important pour que notre opérateur puisse fonctionner correctement après avoir été compilé nativement grâce au support du mode natif fourni par Quarkus. Sans l'extension, il aurait fallu d'une part savoir que ces classes nécessitent un accès réflectif mais également comment les enregistrer correctement auprès de GraalVM pour assurer un fonctionnement correct.
Examinons à présent la classe qui a été générée pour notre CR `ExposedApp`:

```java
@Version("v1alpha1")
@Group("halkyon.io")
public class ExposedApp extends CustomResource<ExposedAppSpec, ExposedAppStatus> implements Namespaced {}
```

Comme nous allons le voir ensuite, cette classe est au cœur de notre opérateur et de nombreuses informations sont automatiquement inférées à partir du groupe et de la version spécifiés par les annotations `@Group` et `@Version`, respectivement.

En particulier, JOSDK utilise ces informations pour générer une "Custom Resource Definition" (CRD) associée à notre CR. La CRD est l'équivalent d'une classe en Java : elle décrit et valide la structure des CR associées en définissant un schéma de validation qui énumère les noms et types des champs de notre CR. La CRD définit l'API que nous voulons ajouter à notre cluster.

```shell
INFO [io.qua.ope.dep.OperatorSDKProcessor] (build-26) Processed 'io.halkyon.ExposedAppController' controller named 'exposedappcontroller' for 'exposedapps.halkyon.io' CR (version 'halkyon.io/v1alpha1')
INFO [io.fab.crd.gen.CRDGenerator] (build-26) Generating 'exposedapps.halkyon.io' version 'v1alpha1' with io.halkyon.ExposedApp (spec: io.halkyon.ExposedAppSpec / status io.halkyon.ExposedAppStatus)...
INFO [io.qua.ope.dep.OperatorSDKProcessor] (build-26) Generated exposedapps.halkyon.io CRD:
INFO [io.qua.ope.dep.OperatorSDKProcessor] (build-26)  - v1 -> <path absolu de notre app>/target/kubernetes/exposedapps.halkyon.io-v1.yml
```

Le nom de notre CRD a été automatiquement généré à partir des informations de notre CR mais, de manière encore plus intéressante, nous pouvons voir qu'une CRD a également été créée automatiquement à partir de nos classes. Bien évidemment, nos classes étant actuellement vides, notre CRD est très basique mais l'on se rend compte immédiatement, surtout quand on a déjà essayé d'écrire une CRD manuellement, de l'intérêt de pouvoir garder notre CRD synchronisée automatiquement avec les changements faits sur notre code.

Malheureusement, notre opérateur a encore une erreur comme nous pouvons le voir dans la console du Dev Mode de Quarkus (qui tourne toujours, normalement):

```shell
ERROR [io.qua.run.Application] (Quarkus Main Thread) Failed to start application (with profile dev): io.javaoperatorsdk.operator.MissingCRDException: 'exposedapps.halkyon.io' v1 CRD was not found on the cluster, controller 'exposedappcontroller' cannot be registered
```

En effet, par défaut, JOSDK vérifie, avant de démarrer un "controller", que la CRD associée existe bien sur le cluster. Ce comportement peut être désactivé (et c'est d'ailleurs recommandé en production) mais c'est une information intéressante à avoir lors du développement car notre opérateur ne pourra pas fonctionner si la CRD associée n'a pas été déployée sur le cluster cible.

Nous pourrions certes appliquer notre CRD sur notre cluster manuellement en utilisant `kubectl` et en lui passant le fichier qui a été généré :

```shell
kubectl apply -f target/kubernetes/exposedapps.halkyon.io-v1.yml
```

Néanmoins, nous aimerions pouvoir concevoir notre CR de manière interactive en utilisant la fonction de Live Coding du Dev Mode de Quarkus pour ce faire: nous ferions un changement sur notre CR, l'extension régénèrerait la CRD et l'appliquerait automatiquement sur le cluster, nous permettant ainsi de nous concentrer sur notre code!

Là encore, l'extension Quarkus nous vient en aide: il est effectivement possible de demander à l'extension de déployer la CRD automatiquement sur le cluster quand elle change via la propriété `quarkus.operator-sdk.crd.apply` que l'on peut modifier dans `application.properties`:

```properties
# set to true to automatically apply CRDs to the cluster when they get regenerated
quarkus.operator-sdk.crd.apply=false
```

Modifions donc le fichier pour mettre cette propriété à `true` et observons le résultat: Quarkus redémarre l'application et nous voyons:

```shell
INFO  [io.qua.dep.dev.RuntimeUpdatesProcessor] (pool-1-thread-1) Restarting quarkus due to changes in application.properties.
...
INFO  [io.qua.ope.run.OperatorProducer] (Quarkus Main Thread) Applied v1 CRD named 'exposedapps.halkyon.io' from <path absolu de notre app>/target/kubernetes/exposedapps.halkyon.io-v1.yml
...
```

Cette fois notre opérateur démarre correctement, une fois la CRD déployée sur le cluster ! Grâce à ce mode de fonctionnement, nous allons pouvoir progressivement enrichir le modèle de notre CR sans avoir à redémarrer notre "operator" ou même quitter notre IDE.

Ajoutons à présent un champ `imageRef` de type String dans la "spec" de notre CR pour indiquer quelle application nous voulons exposer via notre operator. Nous pouvons voir dans les logs de notre application que la CRD est régénérée et appliquée sur le cluster vu qu'une classe affectant son contenu a été changée.

Configuration du "controller"

En examinant les logs du démarrage nous pouvons voir :

```shell
INFO  [io.jav.ope.Operator] (Quarkus Main Thread) Registered Controller: 'exposedappcontroller' for CRD: 'class io.halkyon.ExposedApp' for namespace(s): [all namespaces]
```

`exposedappcontroller` est le nom de notre "controller" et nous voyons qu'il est enregistré pour tous les "namespaces" du cluster, c'est à dire qu'il recevra tout évènement associé avec notre CR, peu importe le "namespace" dans lequel cet évènement se produit.

Ce comportement n'est pas forcément désirable mais, comme nous allons le voir, nous pouvons contrôler la configuration par différents moyens. Examinons à présent la signature de notre "controller":

```java
@Controller
public class ExposedAppController implements ResourceController
<ExposedApp> { ... }
```

Il implémente l'interface `ResourceController` paramétrée par notre CR `ExposedApp` et est également annoté avec l'annotation `@Controller`. Cette annotation est un des moyens de configurer le comportement du "controller" par rapport au cluster. Nous pouvons, par exemple, spécifier sur quels namespaces le controller va écouter pour des évènements associés à notre CR. Par défaut, i.e. dans la configuration actuelle, le controller va écouter sur tous les namespaces.

Configurons notre "controller" pour n'écouter que les évènements associés au namespace dans lequel il sera déployé sur notre cluster en positionnant le champ `namespaces` de notre annotation `@Controller` à la valeur `Controller.WATCH_CURRENT_NAMESPACE`. Nous allons également renommer notre controller afin de pouvoir utiliser le configurer de manière externe (via le fichier `application.properties`, par exemple) plus simplement en positionnant le champ `name` de l'annotation à la valeur `exposedapp`.

```java
@Controller(namespaces = Controller.WATCH_CURRENT_NAMESPACE,
 name = "exposedapp")
public class ExposedAppController implements ResourceController
<ExposedApp> { ... }
```

Notre extension redémarre notre opérateur et nous voyons que la configuration a bien été prise en compte:

```shell
INFO  [io.jav.ope.Operator] (Quarkus Main Thread) Registered Controller:
'exposedapp' for CRD: 'class io.halkyon.ExposedApp' for namespace(s):
[default]
```

Implémentation de la logique de réconciliation

Il nous faut maintenant ajouter la logique de notre "controller". Lorsqu'une CR `ExposedApp` est créée, nous devons créer un `Deployment`, un `Service` et un `Ingress`. Chacune de ces ressources sera créée avec un "label" `app.kubernetes.io/name` dont la valeur sera le nom de notre CR. Ceci nous permettra de pouvoir récupérer toutes les ressources associées à notre ressource principale.

Cependant, nous voulons pouvoir lier explicitement chacune de ces ressources secondaires à notre ressource principale afin que leur cycle de vie soient liés. Kubernetes implémente ce comportement grâce au mécanisme d'Owner Reference. Cela permet par exemple de détruire toutes les ressources associées automatiquement quand la ressource primaire est détruite : c'est exactement ce que l'on veut dans notre cas car l'on suppose que si l'on détruit notre CR, on ne veut plus exposer notre application et il serait pénible de traquer et manuellement détruire toutes les ressources que notre "controller" crée. Nous ajouterons, par conséquent, une "Owner Reference" au champ `metadata` de nos ressources secondaires.

En examinant la classe de notre "controller", nous voyons qu'elle possède un champ de type `KubernetesClient`. Il s'agit d'une instance du client Kubernetes fourni par le projet Fabric8. L'extension `quarkus-operator-sdk` l'injecte automatiquement dans notre "controller".

Ce client fourni ce que l'on appelle une API "Fluent" pour interagir avec le serveur d'API de Kubernetes. À chaque groupe d'API Kubernetes correspond une interface spécifique permettant aux utilisateurs d'interagir avec l'API de manière guidée.

Par exemple, pour interagir avec les `Deployments` qui sont définis dans le groupe `apps`, nous appelons `client.apps().deployments()`. Pour interagir avec les CRDs en version v1, définies dans le groupe `apiextensions.k8s.io`, nous appelons `client.apiextensions().v1().customResource Definitions()`, etc.

Ce client va nous être utile pour implémenter le cœur de notre "controller" : que doit-il se passer quand une ressource de type `ExposedApp` est créée sur le cluster? Si nous n'utilisions pas JOSDK, nous devrions manuellement créer un `Watcher` ou un `Informer` et s'occuper de gérer les évènements bas niveau. Heureusement, JOSDK fournit des abstractions de plus haut niveau et nous n'avons besoin que d'implémenter la méthode `createOrUpdateResource` dans la plupart des cas simples :

```java
UpdateControl<R> createOrUpdateResource(R resource, Context<R>
 context);
```

Cette méthode est appelée automatiquement à chaque fois qu'une ressource de type `ExposedApp` est créée ou modifiée sur notre cluster, uniquement pour les ressources concernant les "namespaces" que notre "controller" est configuré pour surveiller. Notre "controller" reçoit la représentation de la ressource qui a causé l'évènement sous-jacent sans avoir à se soucier des détails de comment elle a été construite. Ignorons pour l'instant le second paramètre qui n'est utile que pour des cas plus compliqués.

Notre "controller" doit donc implémenter cette méthode. Dans notre cas, il s'agit de créer un `Deployment`, un `Service` et un `Ingress`.

Voici donc le code pour créer notre `Deployment` associé à notre CR `resource` qui nous est automatiquement fourni

par JOSDK lorsqu'il appelle notre méthode `createOr UpdateResource`:
Code complet sur programmez.com & github

Une fois le client spécifique aux `Deployments` récupéré, nous appelons `createOrReplace` en construisant une nouvelle instance via un `DeploymentBuilder` qui nous fournit un DSL facile à utiliser. La méthode `createMetadata` se charge de positionner les étiquettes ainsi que l'"Owner Reference" dont nous avons parlé plus tôt. Nous voyons donc que la référence d'image `imageRef` Docker qui est extraite de notre CR et utilisée pour créer une template de pod avec un container utilisant cette image et le port 8080 exposé.

Nous créons de manière similaire notre `Service` puis notre `Ingress`, ce dernier étant un peu plus compliqué car il dépend du fait qu'un "controller" NGINX soit installé et est configuré spécifiquement pour le "controller" NGINX via les annotations. Nous ne rentrerons pas dans les détails de la configuration ici mais nous réfererons plutôt à la documentation officielle : https://kubernetes.io/fr/docs/concepts/services-networking/ingress/

Intéressons-nous maintenant à la valeur de retour de notre méthode. Nous devons retourner un objet de type `UpdateControl` mais qu'est-ce donc ? Il s'agit ici d'indiquer à JOSDK ce qu'il doit ensuite faire: est-ce que notre CR ou son statut ont été modifiés? Ou, au contraire, n'y a-t-il eu aucune mise à jour de notre ressource principale ? Cela permet à JOSDK de faire les appels nécessaires à l'API Kubernetes pour mettre à jour les ressources sur le cluster si besoin est. Cela lui permet aussi de maintenir l'état interne à jour.

Dans notre cas, notre ressource principale n'a pas été modifiée et nous n'avons pas de statut, donc nous pouvons simplement retourner `UpdateControl.noUpdate()`. Nous ajoutons également le logging d'un minimum d'information afin de pouvoir voir sur notre console ce qu'il se passe et nous devrions avoir fini. Le code complété pour cette étape peut être vu à [https://github.com/halkyonio/exposedapp/tree/step-4].

Créons à présent une ressource de type `ExposedApp` que nous sauvegardons dans un fichier `app.yml`:

```yaml
apiVersion: "halkyon.io/v1alpha1"
kind: ExposedApp
metadata:
  name: hello-quarkus
spec:
  imageRef: localhost:5000/quarkus/hello
```

Appliquons la ensuite sur le cluster :

```shell
kubectl apply -f app.yml
```

Si notre cluster est correctement configuré, nous devrions voir quelque chose de similaire à :

```shell
INFO  [io.hal.ExposedAppController] (EventHandler-exposedapp) Exposing
 hello-quarkus application
from image localhost:5000/quarkus/hello
INFO  [io.hal.ExposedAppController] (EventHandler-exposedapp) Deployment
 hello-quarkus handled
INFO  [io.hal.ExposedAppController] (EventHandler-exposedapp) Service
 hello-quarkus handled
INFO  [io.hal.ExposedAppController] (EventHandler-exposedapp) Ingress
hello-quarkus handled
```

Et, effectivement, nous pouvons voir que plusieurs ressources ont été créées dans notre "namespace" :

```shell
kubectl get all -l app.kubernetes.io/name=hello-quarkus

NAME                         READY STATUS  RESTARTS  AGE
pod/hello-quarkus-66f564dd97-tm4jt  1/1   Running 0       7m51s

NAME              TYPE     CLUSTER-IP    EXTERNAL-IP PORT(S)   AGE
service/hello-quarkus ClusterIP 10.96.233.205 <none>    8080/TCP  7m50s

NAME                    READY UP-TO-DATE  AVAILABLE AGE
deployment.apps/hello-quarkus 1/1  1         1       7m51s

NAME                               DESIRED CURRENT READY AGE
replicaset.apps/hello-quarkus-66f564dd97  1     1     1    7m51s
```

Comme `Ingress` ne fait pas partie des ressources qui sont affichées quand nous faisons un `get all`, il faut faire une requête séparée pour voir notre `Ingress`:

```shell
kubectl get ingresses.networking.k8s.io -l app.kubernetes.io/name=hello
-quarkus
NAME        CLASS   HOSTS ADDRESS   PORTS AGE
hello-quarkus <none> *    localhost 80    9m40s
```

Nous pouvons également vérifier que notre application est bien accessible en dehors du cluster en ouvrant [http://localhost/hello] sur un navigateur.

Ajout d'un statut

Notre application semble fonctionner correctement. Néanmoins, il serait intéressant de pouvoir connaître son état facilement.

Ajoutons, pour ce faire, deux champs `host` et `message` à notre classe `ExposedAppStatus` ainsi que la gestion du statut à la fin de la méthode `createOrUpdateResource` de notre "controller". Notre but est de récupérer le statut de notre `Ingress` et de récupérer le nom de l'hôte associé à l'`Ingress`. Si son statut existe, nous extrayons l'information pour mettre à jour le statut de notre CR avec le stade "exposed", sinon nous indiquons que notre CR est encore au stade "processing" dans son statut. Notons également que nous retournons `UpdateControl.updateStatusSubResource(resource)`

pour indiquer à JOSDK que notre CR a vu son statut modifié et qu'il doit faire le nécessaire vis-à-vis du cluster. Nous ajoutons également du logging pour être informé quand l'application est exposée. Le code de cette étape peut être vu à [https://github.com/halkyonio/exposedapp/tree/step-5].

Détruisons notre CR pour pouvoir ensuite la re-créer et observer le comportement de notre "controller":

```shell
kubectl delete exposedapps.halkyon.io hello-quarkus
```

NOTE: *Il est important que notre controller soit actif quand nous effaçons la CR. En effet, par défaut, JOSDK configure les "controllers" pour qu'ils ajoutent un "finalizer" aux CRs qu'ils contrôlent: tant que le "finalizer" n'est pas enlevé de la CR, celle-ci ne peut être détruite et comme, normalement, un "finalizer" ne peut être enlevé que par le "controller" qui l'a placé, il est nécessaire que le "controller" soit actif au moment où l'on veut effacer notre CR (ou alors, il faut relancer le "controller" pour que celui-ci fasse le nécessaire une fois la CR marquée comme devant être détruite). Voir [https://kubernetes.io/blog/2021/05/14/using-finalizers-to-control-deletion/] pour plus de détails sur les "finalizers".*

Une fois la CR ré-appliquée sur le cluster (avec `kubectl apply`), nous avons beau attendre, le logging ne nous indique jamais que l'application est exposée. De même, si nous examinons notre CR, nous voyons le résultat suivant:

```shell
kubectl describe exposedapps.halkyon.io hello-quarkus

Name:       hello-quarkus
Namespace:  default
...
Status:
 Message:  processing
...
```

Pourtant, si nous attendons suffisamment longtemps (quelques dizaines de secondes en général), notre application est bien disponible mais il ne semble pas que notre "controller" soit appelé pour mettre à jour le statut. Ceci est en fait compréhensible: notre "controller" n'est appelé que pour des évènements concernant les CR `ExposedApp`; or, dans notre cas, nous voudrions que notre "controller" soit appelé quand l'`Ingress` que nous avons créé est mis à jour.

Nous pouvons faire ceci avec JOSDK grâce au concept d'`EventSource` qui représente une source d'évènements associés à un type de CR donné. Dans notre cas, nous voulons écouter les évènements affectant les `Ingress` avec le "label" correspondant à notre application. En créant une telle source, et en l'enregistrant auprès de notre "controller" via la méthode `init`, JOSDK appelera également notre "controller" dans ce cas.

Ajoutons donc une implémentation d'`EventSource` à notre application. Nous allons utiliser un `Watcher` pour écouter les évènements de type `Ingress` et ensuite émettre un nouvel évènement de type `IngressEvent` que nous demanderons au JOSDK de prendre en compte via son `EventHandler`. Pour nous faciliter la tâche, nous étendrons la classe `AbstractEventSource` fournie par le SDK :
Code complet sur programmez.com & github

Nous associons ensuite notre `EventSource` à notre "controller" en implémentant sa méthode `init`:

```java
public void init(EventSourceManager eventSourceManager) {
   eventSourceManager.registerEventSource("exposedapp-ingress-watcher",
 IngressEventSource.create(client));
}
```

Détruisons encore une fois notre CR et ajoutons la à nouveau :
Code complet sur programmez.com & github

Nous voyons donc que notre "controller" est appelé une première fois lorsque notre CR est créé puis, contrairement à précédemment, appelé une seconde fois lorsque l'`Ingress` change de statut et nous avons bien le "logging" que nous espérions.
De même si nous examinons notre CR, nous pouvons constater que son statut a bien été mis à jour :

```shell
kubectl describe exposedapps.halkyon.io hello-quarkus

Name:       hello-quarkus
Namespace:  default
...
Status:
 Host:    https://localhost
 Message:  exposed
...
```

Conclusion

Ainsi se termine notre introduction au monde des opérateurs écrits en Java. Après avoir expliqué l'intérêt pour les développeurs Java de pouvoir interagir avec Kubernetes en utilisant un langage de programmation et des outils familiers, nous avons vu comment il est facilement possible d'étendre Kubernetes en lui ajoutant de nouvelles API grâce à JOSDK qui nous a permis de nous concentrer sur l'aspect métier de notre opérateur. Nous avons également brièvement entrapercu l'intérêt de l'extension Quarkus qui nous a permis de développer notre opérateur alors qu'il tournait, nous permettant ainsi d'avoir une boucle de retour rapide et d'itérer sur le code de l'opérateur efficacement. Il y aurait, bien évidemment, d'autres points à aborder tels que comment tester notre opérateur, sa mise en production ou même la compilation native mais nous avons préféré nous concentrer ici sur la partie développement. Le projet Java Operator SDK est encore jeune et nous travaillons continuellement à son amélioration pour rendre la programmation d'opérateurs en Java toujours plus facile. Vous pouvez retrouver le projet sur https://github.com/java-operator-sdk/java-operator-sdk.
Le code complet de l'opérateur est disponible sur https://github.com/halkyonio/exposedapp.

Les anciens numéros de PROGRAMMEZ!

Le magazine des développeurs

235

236

238

239

240

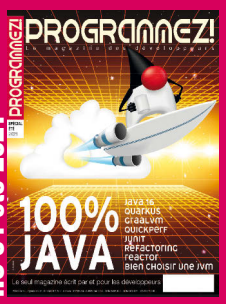
241

HS 01 été 2020

242

246

247



HS 04 été 2021

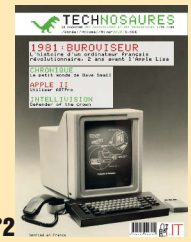
248

Tarif unitaire 6,5 € (frais postaux inclus)

TECHNOSAURES

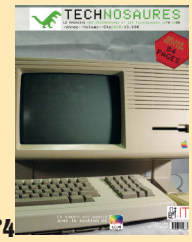
Le magazine à remonter le temps !

N°1



N°2

N°3



N°4

N°5

N°6

N°7 + N°8

Commandez directement sur www.technosaures.fr

☐ 235	: __ ex	☐ 240	: __ ex	☐ 246	: __ ex
☐ 236	: __ ex	☐ 241	: __ ex	☐ 247	: __ ex
☐ 238	: __ ex	☐ HS1 été 2020	: __ ex	☐ HS4 été 2021	: __ ex
☐ 239	: __ ex	☐ 242	: __ ex	☐ 248	: __ ex

Commande à envoyer à :
Programmez!
57 rue de Gisors
95300 Pontoise

soit __ exemplaires x 6,50 € = __ € ..... __ € soit au **TOTAL** = __ €

☐ M. ☐ Mme ☐ Mlle Entreprise : __ Fonction : __

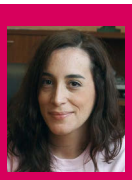
Prénom : __ Nom : __

Adresse : __

Code postal : __ Ville : __

Règlement par chèque à l'ordre de Programmez ! I Disponible sur www.programmez.com

Offres pouvant s'arrêter à tout moment, sans préavis **PROG spécial 5**



Katia Aresti

Katia est Principal Software Engineer chez Red Hat (2017) et Java Champion (2019). Elle est dans l'équipe du produit open-source Infinispan (Red Hat Data Grid), où elle s'occupe principalement de l'intégration avec Quarkus, Spring-Boot, Vert.x mais également des briques telles que la Console Web du Serveur Infinispan, Multimap, Locks Distribués ou des tutoriaux et guides. Membre de la communauté Duchess France depuis 2010 et de Devoxx France depuis 2015.

Statistiques en temps réel dans l'Open Hybrid Cloud avec Quarkus et Red Hat Data Grid (Infinispan)

Les applications et microservices répondent aux besoins des données disponibles à grande échelle, souvent déployées sur plusieurs clouds, où des millions des données circulent. Disponibilité, performance et scalabilité sont critiques dans ces environnements de plus en plus complexes.

Un cache distribué permet de réaliser des opérations très rapides (même transactionnelles) en mémoire sur les données. Il offre la possibilité de distribuer ces données sur différentes instances. L'usage de la mémoire est efficace, scalable et on évite les problèmes d'utilisation des caches locaux (données périmées, ratio hit/miss bas, trop de redondance des données...).

Red Hat Data Grid, dont le projet open-source est Infinispan, est une solution de distribution des données en mémoire (caches distribués), utilisée en production par des nombreuses entreprises pour des cas besoins très variés :

- Off loading d'une base de données
- Backups des données entre différents Data Centers
- Réplication des sessions des utilisateurs
- Statistiques en temps réel
- Catalogue de produits, recherche et shopping cart
- ...

Figure 1

On peut toujours utiliser Red Hat Data Grid comme une librairie embarquée dans une application Java. Aujourd'hui on va avoir tendance à utiliser Data Grid en mode client serveur. Quand on embarque des données dans nos applications, elles ne sont plus sans état (stateless) mais avec état (stateful). Cela ajoute de la difficulté supplémentaire dans le cloud: on ne peut pas faire de scale up and down rapide des microservices si elles ne sont pas stateless. On ne peut pas dissocier les besoins de scalabilité de CPU et mémoire etc.
Quand on travaille en mode client/serveur avec Data Grid, on déploie les clusters des serveurs, et nos applications embarquent simplement un client qui se connectera à la grid des données via le protocole Hot Rod. Hot Rod est le protocole d'Infinispan de communication. Nous pouvons aussi utiliser l'API REST ou d'autres formats pour lire et écrire les données d'un cluster Data Grid.
Les cinq fonctionnalités clés de Data Grid, ayant comme socle toute la technologie des caches distribués, sont :

- Interopérabilité. Nous pouvons utiliser un client Java, .Net ou Node JS.
- Tolérance aux pannes grâce aux différents niveaux de consistances des données et des stratégies de disponibilité.
- Transactions ACID en mémoire sur les données distribuées.
- Exécution en cluster des traitements.
- Recherche Full-Text, avec le support le plus complet sur les caches qui utilisent le format Protobuf pour stocker les données

Statistiques en temps réel et multi cluster

Dans cet article, nous voulons développer un jeu en ligne, et pour la gamification, on veut récolter plusieurs statistiques du jeu en temps réel: combien de joueurs nous avons, combien d'actions ont-t-ils réalisé, le top 10 des joueurs.
Le jeu devra être déployé sur plusieurs Data Centers dans différentes régions de la planète, et avec des Provider Cloud différents (Open Hybrid Cloud). Les participants vont se connecter au cluster proche de sa zone géographique, puisque la vitesse des transactions des données est essentielle. Par contre, les statistiques du jeu doivent être globales. **Figure 1**
Lors de Red Hat Summit 2021, nous avons mis en production un jeu Naval avec succès, où plusieurs milliers d'utilisateurs jouaient depuis toute la planète. La solution que nous avons développée pour répondre à ce cas d'usage, et que

nous allons expliquer dans cet article, nous aide à illustrer le potentiel de Data Grid.
Voici la liste des technologies dont on va vous parler :

- Indexation et recherche full text avec des caches Data Grid en format Protobuf
- Quarkus et les extensions suivantes: Data Grid (Infinispan), Websockets et Scheduler
- La réplication Cross Site de Data Grid
- L'opérateur Data Grid pour OpenShift

Architecture de chaque datacenter

Nous allons déployer sur chaque Data Center (site) un cluster Data Grid et les services impliqués dans le tracking et calcul en temps réel des statistiques. Celui que nous allons regarder est le LeaderBoard, composant en charge de calculer le ranking des joueurs en temps réel. **Figure 2**

- **Data Grid (Infinispan) Server**, le cluster contient un cache appelé "player-scores"..
- **Scoring Service**, qui va s'occuper de traquer des actions et d'incrémenter les points victoire des joueurs.
- **Leaderboard Service**, utilisant les extensions Quarkus Websockets, Scheduler, et Infinispan client, va calculer et mettre à disposition le ranking.

Le format des caches: Protobuf

Nous avons mentionné rapidement le format Protobuf des caches. Nous pouvons utiliser d'autres formats de sérialisation et continuer avec la sérialisation de Java, ou utiliser du text ou json. Protobuf est aujourd'hui le format recommandé puisque c'est le format qui va nous offrir le top des capacités de Data Grid:

- Nous éviterons les problèmes liés à la sérialisation Java
- Protobuf est rapide, compact, léger
- Nous avons la meilleure interopérabilité entre technologies
- Nous pouvons utiliser le potentiel de la recherche full-text au maximum

Configuration du cache

Le cache "playes-scores" stockera les données des joueurs et les points victoire. Afin de permettre la recherche full-text sur les valeurs du cache, nous allons indiquer les entités qui devront être indexées. Le media-type ProtoStream signifie bien que ce cache va utiliser Protobuf.

```
<distributed-cache name="player-scores" statistics="true" owners="2">
   <encoding>
      <key media-type="application/x-protostream" />
      <value media-type="application/x-protostream" />
   </encoding>
   <indexing enabled="true">
         <indexed-entities>
         <indexed-entity>com.redhat.PlayerScore</indexed-entity>
      </indexed-entities>
   </indexing>
</distributed-cache>
```

Jusqu'à la version 8.2 de Data Grid (Infinispan 12), la configuration des caches n'était pas modifiable en runtime. A partir de la version 8.3 (Infinispan 13) en cours de développement lors de l'écriture de cet article, une partie de la configuration pourra être changée sans recréer un nouveau cache.

Les données stockées: PlayerScore

PlayerScore est une classe qui va porter le modèle de données en Java. La propriété la plus importante pour réaliser le ranking des joueurs est le "score".
PlayerScore.java
https://gist.github.com/karesti/9475fc8be5fd238c5b55b4945b3fb941

```
@ProtoDoc("@Indexed")
public class PlayerScore {
  private String gameId;
  private String userId;
  private String username;
  private Integer score;

  @ProtoFactory
  public PlayerScore(String gameId, String userId, String username, Integer score) {
    this.gameId = gameId;
    this.userId = userId;
    this.username = username;
    this.score = score;
  }


  @ProtoField(1)
  @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.NO, store = Store.YES)")
  public String getGameId() {
    return gameId;
  }
  @ProtoField(2)
  public String getUserId() {
    return userId;
  }

  @ProtoField(3)
  public String getUsername() {
    return username;
  }

  @ProtoField(4)
  @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.NO, store = Store.YES)")
```

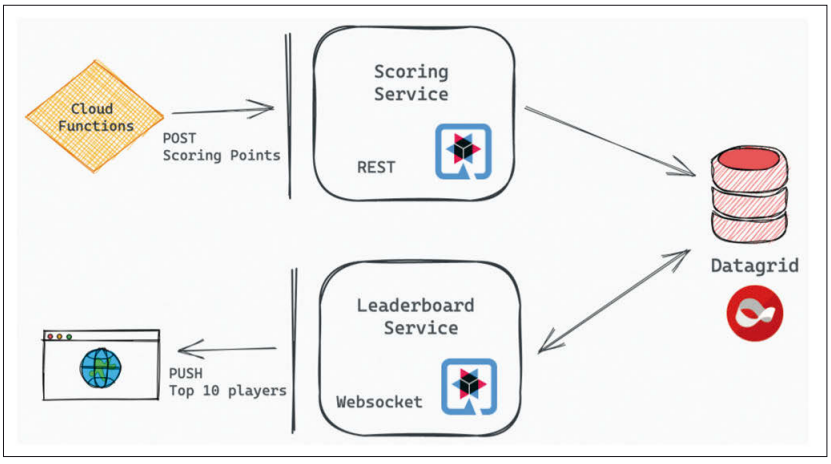


Figure 2
Architecture simplifié des composants

```
@ProtoDoc("@SortableField")
public Integer getScore() {
  return score;
}

...

}
```

L'extension d'Infinispan pour Quarkus embarque ProtoStream pour créer le schéma ".proto" et le marshaller côté client sait comment lire et écrire en Protobuf les données dans la grid.
L'API de ProtoStream utilise des annotations :

- @Protofield détermine quels attributs doivent être inclus dans la sérialisation.
- @ProtoFactory est capable de créer une instance POJO pendant la désérialisation.
- @ProtoDoc permet d'ajouter l'information relative à l'indexation des données qui seront utilisées lors des recherches full-text.

Afin d'indiquer à ProtoStream quelles classes inclure dans les schémas Protobuf, nous allons aussi créer une interface du type GeneratedSchema.

```
import org.infinispan.protostream.GeneratedSchema;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;

@AutoProtoSchemaBuilder(schemaPackageName = "com.redhat",
   includeClasses = { PlayerScore.class})
public interface GameSchema extends GeneratedSchema {
}
```

Notez que le "schemaPackageName" et le package Java du modèle sont dissociés, et ne sont pas les mêmes si nous ne le voulons pas. Ce nom de paquet est celui qu'on utilisera lors de la recherche full-text des entités "PlayerScore" et pour configurer l'indexation comme nous l'avons fait précédemment lors de la configuration du cache.
Ceci est le schéma généré:
https://gist.github.com/karesti/b50cfecac14c2a9a95f712e95af14401

```
syntax = "proto2";

package com.redhat;

/**
 * @Indexed
 */
message PlayerScore {

 /**
  * @Field(index=Index.YES, analyze = Analyze.NO, store = Store.YES)
  */
  optional string gameId = 1;

  optional string userId = 2;

  optional string username = 3;

  /**
   * @Field(index=Index.YES, analyze = Analyze.NO, store = Store.YES)
   * @SortableField
   */
  optional int32 score = 4;
}
```

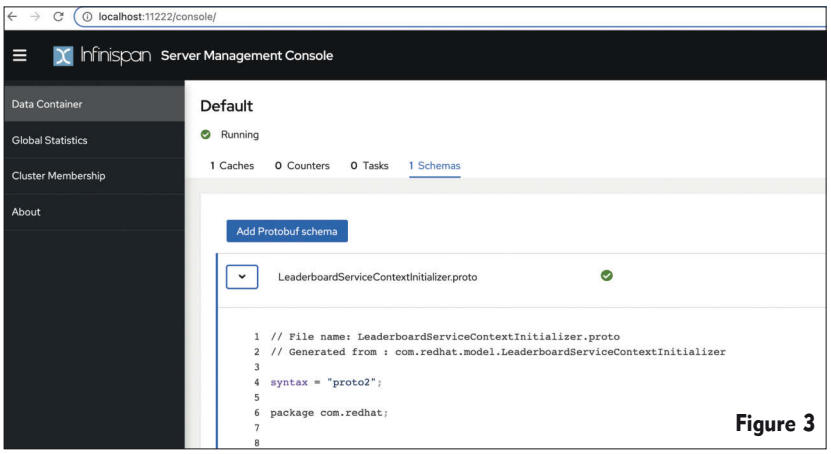


Figure 3

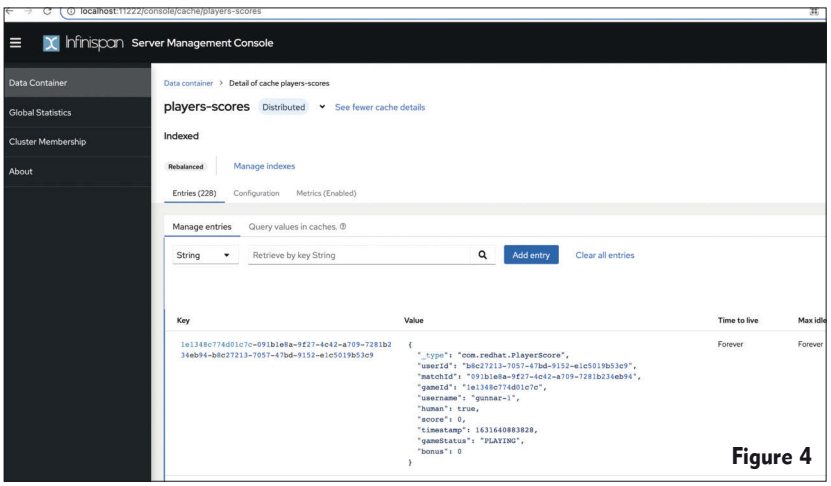


Figure 4

Le processus pour générer le schéma de Protobuf a lieu côté client. Le serveur de Data Grid a aussi besoin de savoir comment écrire, lire et indexer le format Protobuf de PlayerScore. Pour ce faire, le schéma peut être uploadé via la Console Web du serveur, via l'opérateur ou laisser l'extension de Quarkus le faire au démarrage du Micro Service. **Figure 3**
La console web de Data Grid utilise l'API REST. Grâce à l'intégration Protobuf, Data Grid est capable de visualiser au format JSON les PlayerScore. **Figure 4**

Calculer le top 10 local

Pour trouver les top 10 joueurs en temps réel, et les autres statistiques, nous utilisons l'API de recherche de Data Grid.

```
QueryFactory queryFactory =
Search.getQueryFactory(playersScores);
Query topTenQuery = queryFactory.create("from com.redhat.PlayerScore p WHERE
 gameId=:gameId ORDER BY p.score DESC").maxResults(10);
topTenQuery.setParameter("gameId", gameId);
List<PlayerScore> topTen = topTenQuery.execute().list();
```

Data Grid offre une API des query en continu : Continuous Query. Celle-ci n'est pas adaptée à des recherches où le tri(sorting) doit être réalisé. Afin d'exécuter la requête top 10 en continu, on utilise l'extension de scheduler de Quarkus, qui nous permet de mettre en place cette continuité.
Enfin, avec l'extension WebSockets de Quarkus, nous allons envoyer le ranking et les statistiques pour qu'un client reçoive et montre les mises à jour en continu.
Code ici : https://gist.github.com/karesti/912bb6e456f6387d0f19fdff2bcc8510

Top 10 global

Finalement nous avons nos composants déployés dans un Cluster, avec les données locales de chaque joueur. Pour faire en sorte que tous les clusters soient au courant des scores de tous les joueurs, et que les résultats de la query top 10 soit globale, nous allons utiliser la réplication Cross Site en mode "active-active". Chaque cache des scores enverra ses données aux caches qui existent dans les autres clusters de Data Grid. **Figure 5**
La configuration du cache doit contenir une partie pour spécifier les backups de ce cache.
Par exemple, le cluster AZURE définit ses backup de "playerscores" sur AWS et GCP. Data Grid attend qu'un cache appelé "players-scores" existe sur AWS et GCP pour envoyer les données. Le nom du cache doit être le même, la configuration peut être différente. Ainsi on peut adapter les backups en fonction des besoins spécifiques aux clusters.
Dans notre exemple, les 3 caches de AWS, GCP et AZURE ont la même configuration.

```
<distributed-cache name="players-score" ...>
<!-- index and encoding configuration. -->
<backups>
<backup site="AWS" strategy="ASYNC" enabled="true">
 <take-offline min-wait="60000" after-failures="3" />
            </backup>
            <backup site="GCP" strategy="ASYNC" enabled="true">
                  <take-offline min-wait="60000" after-failures="3" />
            </backup>
          </backups>
</distributed-cache>
```

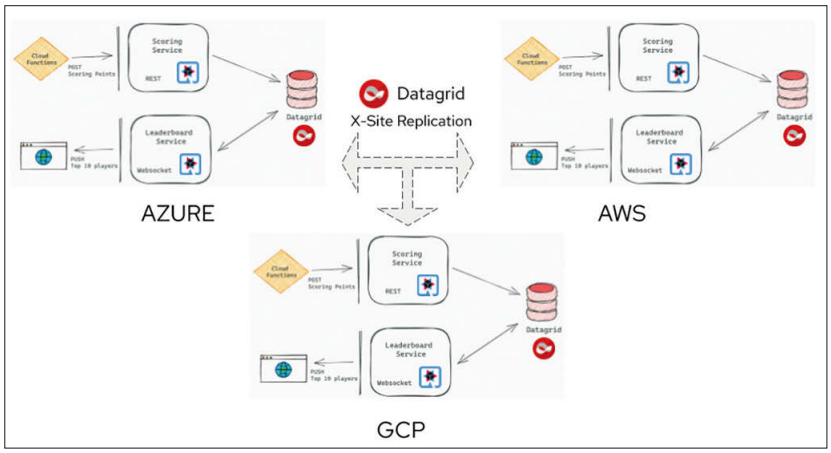


Figure 5
Réplication Cross Site avec 3 trois déploiements cloud

Désormais quand le Leaderboard va exécuter la requête top 10, elle inclura les scores locaux aux joueurs mais également les scores qui ont été transmis par les autres 2 clusters. Le ranking devient ainsi global.

L'operateur Kubernetes de Data Grid

Installer, configurer et maintenir plusieurs clusters Data Grid avec la réplication Cross Site peut s'avérer compliqué. Heureusement, pour simplifier la vie des développeurs, des opérations et nous faciliter énormément la prise en main de Data Grid en Kubernetes/OpenShift, l'opérateur Kubernetes de Data Grid (Infinispan) est là. Il est déjà très stable. Tout comme Infinspan/Data Grid, il est en constante évolution et amélioration.
Le Infinispan Custom Resource nous aide avec le cluster.
https://gist.github.com/karesti/803b396208b11a7c5d4b3ad7631c68b1

Conclusion

Data Grid et les caches distribués offrent des solutions aux différents cas d'usage grâce notamment aux opérations rapides en mémoire, la recherche full-text et la réplication Cross Site. L'intégration avec Quarkus et d'autres frameworks tels que Vert.x, Spring-Boot ou Micronaut sont disponibles. N'oubliez pas que d'autres cas spécifiques à Wildfly, Keycloak ou Camel sont aussi intégrés avec Data Grid (Infinispan). Par exemple, la réplication Cross Site des identités de Keycloak.
Nous espérons que cet article vous a donné envie de découvrir, ou redécouvrir, Data Grid, et Infinispan sur http://infinispan.org et l'utiliser dans vos architectures

Intégration Camel Quarkus

Zineb Bendhiba

Senior Software Engineer chez Red Hat. Elle est contributrice reconnue sur le projet open source Apache Camel. Actuellement, elle contribue principalement au sous-projet Camel Quarkus. Avant de rejoindre Red Hat en 2020, elle a travaillé pendant 12 ans dans différentes entreprises. Elle a participé à la conception, au développement et à la gestion de différents projets, principalement dans la technologie Java. Par ailleurs, elle est membre active de l'association Duchess France.

Apache Camel est un framework open source dont le but principal est de simplifier l'intégration entre systèmes. Avec un large nombre de composants, il permet de connecter vos applications à presque tout type de système externe. Dans ce dossier, nous allons explorer le framework Apache Camel. Par la suite, nous allons découvrir comment réaliser votre première application Camel avec Quarkus, et la déployer sur Openshift.

PARTIE 1: INTRODUCTION À APACHE CAMEL

Le projet Apache Camel a su s'adapter avec le temps. Il se compose depuis sa version 3 de plusieurs sous-projets, dont certains ont le but de porter Camel sur des architectures cloud et serverless. Avec une communauté grandissante, il est aujourd'hui le framework open source d'intégration le plus populaire, et votre allié pour gérer tout type d'évènements de vos architectures EDA (Event Driven Architecture). Par ailleurs, sur les rapports les plus récents de l'ASF (Apache Source Foundation), Camel figure parmi les projets les plus actifs. Avant de s'attaquer à un exemple concret, voici quelques notions de base sur Apache Camel.

Route

Camel se base sur un système de *routing* qui permet d'intégrer les données d'un système A vers un système B. Le principe du framework est de créer des *routes* Camel, qui représentent des use case d'intégration.

La **figure 1** illustre une *route* Camel qui transporte des messages d'un système A vers un système B.

Producer et consumer

Une route se compose d'un *consumer* et d'un ou plusieurs *producer*. Le *consumer* est le point d'entrée de la route. Il est à l'écoute d'évènements. Les évènements peuvent être de plusieurs types : un message JMS, un CDC(Change Data Capture) dans une base de données, un évènement Change Data Capture provenant d'un système externe tel que Salesforce, un fichier, un message HTTP, un email, un message chat… Le *producer* est responsable dans une route Camel de produire le message dans le système externe cible. Les *composants* Camel permettent de créer des producer et consumer dédiés à chaque type de système et protocole.

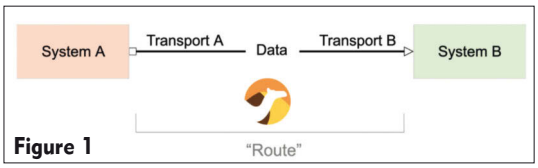
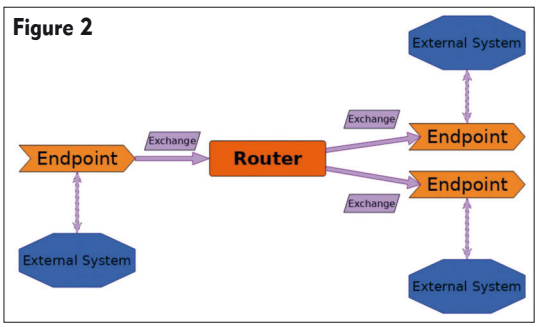


Figure 1

Figure 2



Composants

Afin de gérer plusieurs types de systèmes externes et de protocoles, Camel introduit la notion de composant. Les composants permettent de créer les consumer ou producer dédiés à des systèmes spécifiques. Vous interagissez avec tous ces systèmes externes, en utilisant la même API, quel que soit le protocole ou le type de données que ces systèmes utilisent. Ce sont ces composants qui vont se charger de la complexité technique. Camel regroupe aujourd'hui plus de 360 composants prêts à l'emploi, et vous trouverez presque tout ce dont vous avez besoin. Le cas échéant, il est tout à fait possible de créer ses propres composants Camel.

Voici une liste non exhaustive des composants disponibles :

Filetypes:	Protocols:	Public/private Clouds:
• Plaintext, XML, HTML, CSV, JSON, ZIP, ... • Apache Tika (MS Office, OpenDocument, ...)	• TCP/UDP, DNS, FTP, HTTP, IRC, SSH, ... • REST, GRPC, git, ... • IoT: CoAP, MQTT, PubNub	• AWS: S3, SQS, Kinesis, ... • Azure: Blob, Queue, ... • Google: BigQuery, PubSub • DigitalOcean, Kubernetes, OpenShift, etcd, Docker
Data & messaging	**APIs:**	**Misc:**
• Hazelcast, Infinispan, Redis, ... • Cassandra, MongoDB • JDBC, SQL, JPA • Kafka, Debezium • JMS, AMQP, QPID, STOMP	• Social: Facebook, Twitter, LinkedIn • Box, Dropbox, Google Calendar/Drive/Mail/Docs • Salesforce, SAP, ServiceNow, FHIR • GitHub, Jira, Telegram	• Barcode • Base64, JsonPath, Freemarker, Groovy, Mustache, Ruby • JMX, OpenTelemetry, OpenTracing, OptaPlanner, Syslog

Exchange et Message

L'échange de données entre un système A et un système B est encapsulé dans un objet qu'on appelle *Exchange*. L'*Exchange* contient un *Message*, ainsi que des informations propres au contexte de l'échange de données.

Le *Message* contient plusieurs informations :

- Le *Body* : qui contient le payload
- Les *Headers*: qui contiennent des informations supplémentaires pour le traitement de la donnée.
- Les *Attachements* : qui sont optionnels. Ils sont généralement utilisés dans le cadre de composants qui peuvent recevoir des messages avec pièces jointes, comme des web services ou emails

La **figure 2** illustre plus en détail une *route* Camel. La *route* est composée d'un *endpoint* de type *consumer*. C'est le point d'entrée de la *route*, sur lequel on est à l'écoute de messages depuis le système d'entrée. Elle contient un ou plusieurs endpoint(s) de type *producer*, dont le but est de produire des messages dans les systèmes externes cibles. La donnée transite via l'*Exchange*. Le *Router* Camel va gérer le routage des messages vers les endpoints cibles.

DSL

Camel permet d'interagir avec différents systèmes en utilisant une DSL (Domain Specific Language) unique, quel que soit le protocole ou le type de données que les systèmes utilisent. Ce niveau d'abstraction vous permet de vous focaliser sur vos use case métier. La DSL Camel peut être utilisé depuis plusieurs langages Java, XML....

La **figure 3** illustre une Route Camel qui :

- consomme un fichier depuis Amazon S3 (Amazon Simple Storage Service),
- crée une log du contenu du message,
- transmet le message via un chat bot Telegram.

La Route est présentée dans 2 types de DSL: la DSL Java et la DSL XML. Le keyword *from* représente le *consumer* de type *aws-s3*. Le keyword *to* représente le *producer* type *telegram*. Le keyword *log* permet d'ajouter un message de log. À l'aide d'un langage Camel appelé *Simple language*, on a créé un message dynamique, combinant du texte statique et le *body* de l'*Exchange*.

Langages

Camel offre une liste de langages d'expressions dans sa DSL, permettant de manipuler les messages et headers. Les langages peuvent être simples comme le Simple language, mais beaucoup plus complexes comme les langages MVEL, Groovy et OGNL.

EIP:

Le framework implémente une grande partie des EIP (Entreprise Integration Patterns), qui sont des patterns d'intégration décrits dans l'excellent livre *Entreprise Integration Patterns* de G. Hohpe et B. Woolf. Voici une liste d'exemples d'EIP populaires présents dans Camel :

Content Based Router:

L'EIP Content Based Router permet de router les messages vers la bonne destination, selon le contenu des messages échangés. La **figure 4** illustre l'EIP.

Dans Camel, on retrouve l'EIP sous le nom *choice*. Les cas sont définis avec les keywords *when* et *otherwise*. Le *Simple language* permet de créer des conditions sur le body ou les headers du message. Ceci est un exemple d'utilisation de l'EIP avec la DSL Java. On décide du routage du message selon la valeur du header *foo*, contenu dans le message d'entrée.

```
from("direct:a")
        .choice()
          .when(header("foo").isEqualTo("bar"))
            .to("direct:b")
          .when(header("foo").isEqualTo("cheese"))
            .to("direct:c")
          .otherwise()
            .to("direct:d");
```

Le keyword *direct* désigne le composant Camel de type Direct, permettant d'invoquer directement un consumer ou un producer, qui n'est pas associé à un système externe. On l'utilise généralement pour connecter une route existante avec une autre.

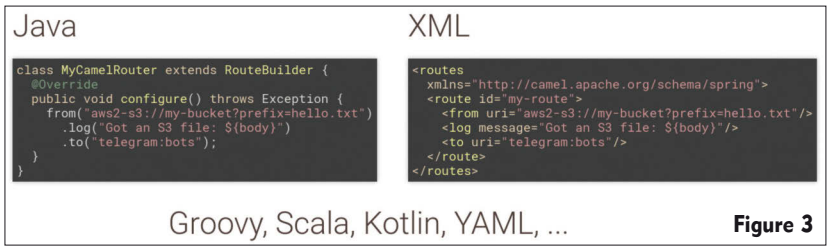


Figure 3

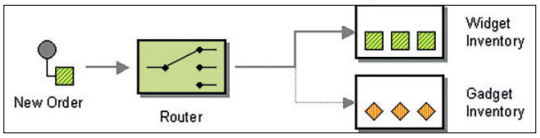


Figure 4

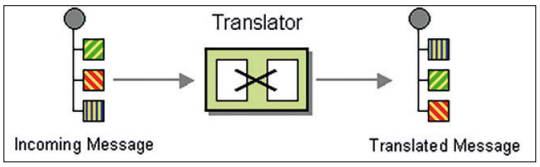


Figure 5

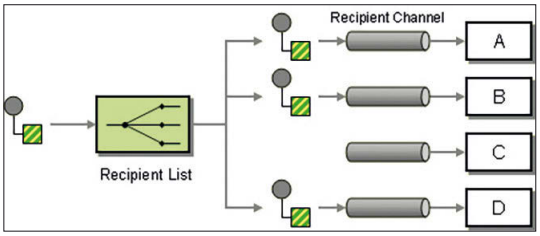


Figure 6

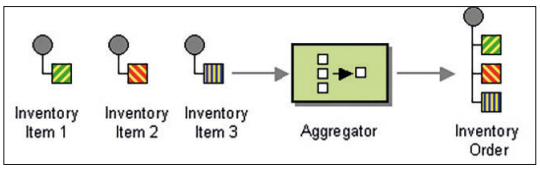


Figure 7

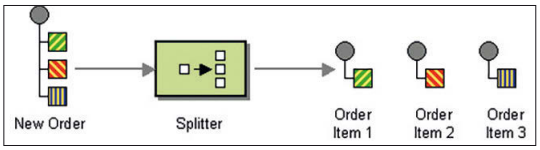


Figure 8

Message Translator EIP:

Cet EIP permet de traduire un message pour le système externe cible. La **figure 5** illustre l'EIP.

Il existe plusieurs implémentations du Message Translator dans Camel. Ceci est une implémentation du Message Translator, en utilisant la méthode *transform* et le *Simple language* de la DSL Java.

```
from("direct:input")
  .transform().simple("Message is : ${body}")
  .to("direct:output");
```

Recipient List

L'EIP Recipient List est utilisé pour transmettre un message d'un système A à plusieurs systèmes. La **figure 6** illustre l'EIP.

Aggregator

L'EIP Aggregator va agréger plusieurs messages entrants vers un seul message sortant. La **figure 7** illustre Aggregator.

Splitter

L'EIP Splitter divise le message entrant en plusieurs messages sortant, à traiter indépendamment. La **figure 8** illustre le Splitter.

Les projets Camel 3

Depuis la sortie de sa version 3, Camel est divisé en plusieurs sous-projets :

- Camel-core : C'est le core de Camel
- Camel-K : Lightweight framework d'intégration utilisant Camel, qui tourne nativement sur Kubernetes et Knative.
  - Camel-K permet d'utiliser plusieurs langages de programmation(Java, XML, YAML, Groovy, Javascript…), pour créer des petites intégrations exécutables dans le cloud
  - Camel-K intègre par défaut le *catalog des Kamelets*, qui sont des templates de routes Camel réutilisables.
- Camel-Kafka-Connector pour utilisation des composants Camel comme des connecteurs Kafka-connect
- Camel-Quarkus pour la plateforme Quarkus: Camel est optimisé pour tourner sur la plateforme Quarkus. Les composants sont disponibles sous forme d'extensions Quarkus.
- Camel-Spring-Boot pour la plateforme Spring Boot: utilisation du contexte Spring pour la détection des routes Camel, et de l'auto-configuration pour la création automatique des routes. Les composants sont disponibles sous formes de starters Spring Boot.
- Camel-Karaf pour la plateforme Apache Karaf, afin de faire tourner Camel dans un conteneur OSGi.

PARTIE 2 : CRÉER UNE APPLICATION CAMEL QUARKUS

Nous allons découvrir comment créer une application Quarkus, en utilisant Camel.

Pré-requis

- Un environnement de développement pour créer des applications Java Quarkus. Selon l'IDE que vous utiliserez, vous pourrez rajouter les plugins Quarkus et Apache Camel, qui pourront vous aider pour l'autocomplétion.
- Docker, si l'on souhaite utiliser le dev services fourni par Quarkus. Durant ce tutoriel, nous utiliserons le dev mode, et Quarkus lancera pour nous automatiquement des containers pour utiliser Kafka et PostgreSQL.
- Un bot Telegram: Pour créer un bot, télécharger l'application Telegram(https://telegram.org/). Créez un compte. Cherchez l'utilisateur @BotFather. Taper */newbot* et suivre les indications de création du bot. Notez le token d'utilisation de l'API Bot. Vous pouvez commencer à dialoguer avec le bot en cherchant le nom de votre Bot parmi les utilisateurs Telegram. Le Bot ne vous répondra pas, nous allons le programmer dans ce tutoriel. **Figure 9**
- Si vous voulez tester le livrable ou déployer votre application, il vous faudra une instance Kafka et une base de données accessibles. Notez que ce n'est pas un prérequis pendant la phase de développement.
- Pour le déploiement sur Openshift, il est nécessaire d'avoir un environnement prêt, et d'installer l'Openshift CLI. Vous pouvez utiliser gratuitement la Developer Sandbox for Red Hat OpenShift pour ce tutoriel.

Notez que lors de la rédaction de cet article, j'ai utilisé les versions suivantes :

- Quarkus 2.2.x
- Camel-Quarkus 2.2.x, qui englobe Camel 3.11.x

L'application

Créons un bot Telegram permettant de recueillir des messages d'utilisateurs qui seront ensuite publiés dans kafka. Par ailleurs, un timer va aussi publier des messages dans le même topic kafka, à intervalle régulier. Enfin, un consommateur va transférer les messages de kafka vers une base de données. Cependant, seuls les messages provenant, à l'origine, de Telegram atteindront la base de données. **Figure 10**

Pour réaliser ce projet, nous allons utiliser les composants Camel suivant:

- Timer: Le composant camel-timer, va permettre de créer des messages automatiquement, sur un intervalle de temps défini.
- Telegram: Telegram est une application de messagerie. Elle permet de créer des bots, et expose la Telegram Bot API: une API destinée aux développeurs pour implémenter le comportement du Bot. Le composant camel-telegram offre un consumer et un producer qui implémentent la Bot API.
- Kafka: Apache Kafka est une plateforme de streaming populaire. Le composant camel-kafka permet d'envoyer et de recevoir des messages à/depuis un broker Kafka.
- JPA: Java Persistance API. Nous utilisons, dans ce blog, JPA pour interagir avec une base de données Postgresql. Notez que l'on aura besoin aussi de l'extension Quarkus *jdbc-postgresql*.
- Platform-http: ce composant va nous permettre d'exposer un endpoint REST, adapté à la plateforme, qui dans ce cas est Quarkus.

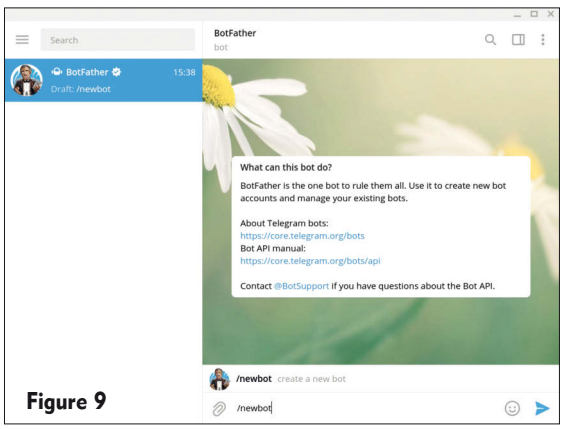


Figure 9

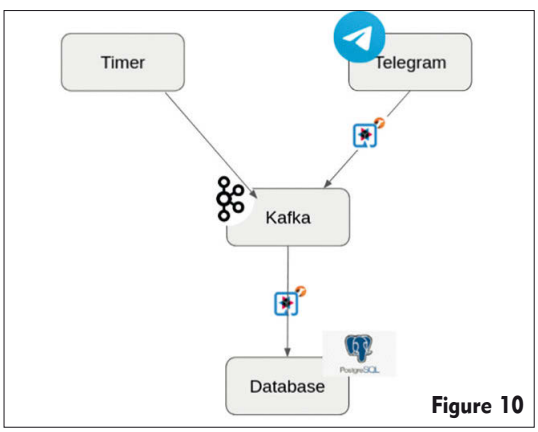


Figure 10

• Camel Jackson: pour transformations JAVA/JSON.

Pour créer le projet, allez sur https://code.quarkus.io/, créer un nouveau projet, en ajoutant les extensions suivantes :

• Camel Timer
• Camel Kafka
• Camel JPA
• JDBC Driver - PostgreSQL
• Camel Platform HTTP
• Camel Jackson
• Camel Telegram

Figure 11

Téléchargez la source du projet, et ouvrez le projet dans votre éditeur préféré. Créer une classe dans laquelle nous allons définir nos routes Camel. Appelons là tout simplement Routes. La classe doit hériter de la classe *org.apache.camel.builder.RouteBuilder* et implémenter la méthode *configure*.

```
public class Routes extends RouteBuilder {
  @Override
  public void configure() throws Exception {

  }
}
```

Maintenant, nous pouvons commencer à créer nos routes Camel à l'intérieur de notre méthode configure.

1ère route : timer to kafka

Nous allons créer une route de type "timer" qui va créer un message automatique et le streamer dans notre topic Kafka, sur un intervalle de temps défini qui est 10000 millisecondes. Le message de type JSON, que l'on veut créer est le suivant :

```
{
 "from": "timer",
 "content": "Message automatique ${numero_sequence}"
}
```

Où ${numero_sequence} représente le numéro de séquence du message généré. Utilisons le Simple Language pour :

• Retrouver dans le Message Camel, la propriété CamelTimerCounter créé par le consumer Timer
• Transformer le contenu du Body du Message

Voici la route Camel:

```
from("timer:foo?period=10000&delay=10000")
        .setBody().simple("{\"from\":\"timer\", \"content\": \"Message automatique #${exchangeProperty.CamelTimerCounter}\"}")
        .to("kafka:test");
```

Le consumer de type *timer* s'appelle *foo*. *Period* détermine l'intervalle en secondes entre chaque message généré. *Delay* détermine le délai en secondes à attendre avant de produire le premier message. Le producer *kafka*, va streamer les messages dans un topic kafka, dont le nom est *test*.

Il faudra configurer l'URL du broker Kafka dans le fichier application.properties. Nous utiliserons pour le moment dev services, et nous allons nous appuyer sur la propriété quarkus kafka.bootstrap.servers. En dev mode, cette variable va être définie automatiquement par Dev services.

Je vous invite donc à créer le fichier src/main/resources/application.properties et d'y renseigner cette valeur:

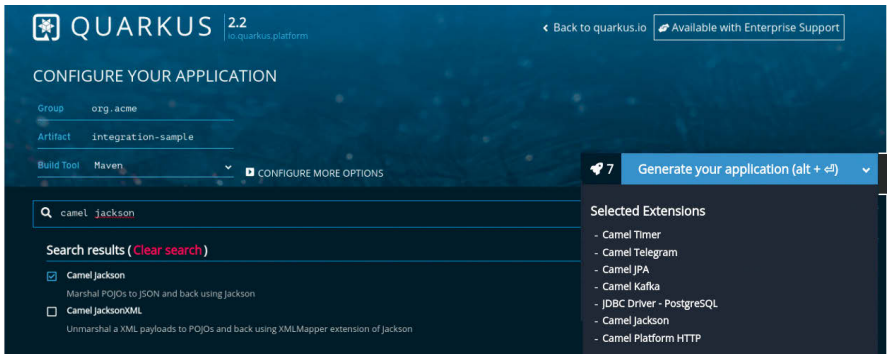


Figure 11

```
%dev.camel.component.kafka.brokers=${brokers:${kafka.bootstrap.servers}}
```

2ème route : Telegram to Kafka

La 2ème route va intercepter les messages que des utilisateurs Telegram vont envoyer à notre chat bot. Ce message va être transformé, en format JSON suivant.

```
{
 "from": "telegram",
 "content": "${body}"
}
```

Où ${body} est le message reçu par le bot. Pour simplifier notre use case, nous supposons que les utilisateurs n'envoient que du texte. À noter que l'API permet de gérer les vidéos, images, fichiers…

La route va ensuite répondre à l'utilisateur, en lui renvoyant le message. Nous utilisons le producer Telegram pour l'envoi. Le message sera routé au bon utilisateur grâce aux informations contenus dans le header. En effet, lorsque le consumer Camel reçoit le message, Camel va injecter les informations du contexte de l'échange, notamment via des headers. Ces informations restent disponibles jusqu'à la fin de l'échange. Lorsque le producer de type Telegram va envoyer des messages, il va chercher dans le contexte l'id Telegram vers lequel il doit envoyer le message. Cela va nous permettre de répondre à l'émetteur du message sur Telegram.

Nous utilisons plusieurs fois des transformations de messages, une fois pour créer le message à produire dans le topic kafka, puis une seconde fois pour créer la réponse à envoyer à l'utilisateur. Afin de transformer le message reçu en un message compatible pour le topic kafka, nous utilisons *transform* comme implémentation du Message Translator EIP. Nous utilisons Simple Language pour manipuler le contenu du message. Voici la route Camel:

```
from("telegram:bots?authorizationToken={{telegram-token-api}}")
    .transform(simple("{\"from\":\"Telegram\", \"content\": \"${body}\"}"))
        .to("kafka:test")
        . transform(simple("Merci pour votre message."))
      .to("telegram:bots?authorizationToken={{telegram-token-api}}");
```

Les URI des consumer et producer Telegram commence par telegram:bots. AuthorizationToken est une propriété de l'endpoint utilisé. Nous pouvons copier directement la valeur de notre token ou utiliser comme décrit ci-dessus une variable, dont la valeur est définie dans le fichier application.properties.

Je vous invite donc à rajouter cette valeur dans le fichier src/main/resources/application.properties :

```
telegram-token-api=VOTRE_AUHTORIZATION_TOKEN
```

3ème route: kafka to postgresql

Créons maintenant le consumer kafka. Il sera à l'écoute de tous les messages du topic test. Nous utilisons la fonction log, afin de visualiser tous les messages consommés. Nous n'insérons, en base de données, que les messages provenant de Telegram. Créons notre entité JPA :

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.NamedQuery;

@Entity
@NamedQuery(name = "findAll", query = "SELECT m FROM TelegramMessage m")
public class TelegramMessage {
  @Id
  @GeneratedValue
  Long id;
  @Column(name = "sender")
  String from;
  String content;

 public  TelegramMessage(){
 }
// getters and setters ici
...
}
```

Les messages provenant de kafka sont au format JSON. Afin de les persister en base de données en utilisant JPA, nous devons impérativement transformer les JSON reçus en objets Java de type TelegramMessage. Pour ce faire, nous utilisons le composant camel-jackson.

```
from("kafka:test")
  .log("Message entrant : ${body}")
  .choice()
        .when(simple("${body} contains 'Telegram'"))
            .unmarshal().json(TelegramMessage.class)
            .to("jpa:" + TelegramMessage.class.getName())
      .end();
```

from("kafka:test") va créer un consumer de type Kafka, qui va consommer des messages depuis le topic "test".
unmarshal().json(TelegramMessage.class) va transformer le message JSON en un objet de type TelegramMessage.
to("jpa:" + TelegramMessage.class.getName()) va produire en base de données l'objet TelegramMessage, en utilisant JPA.
Afin de créer automatiquement notre table en base de données, nous allons rajouter la configuration suivante dans le fichier src/main/resources/application.properties.

```
quarkus.hibernate-orm.database.generation=create
```

Les informations de connexion à la base de données doivent être paramétrées. Pendant la phase de développement, dev services se chargera de démarrer un container Postgresql et d'injecter la configuration. Nous ne nous en soucions pas à ce stade. Lancez l'application en dev mode :

```
$ ./mvnw  compile quarkus:dev
```

Nous remarquons que l'application démarre et que dev services lance automatiquement des containers pour kafka et postgresql.
On commence à voir les logs des messages créés automatiquement. Nous pouvons envoyer un message à notre bot Telegram. On remarque 2 choses :

1. On reçoit une réponse du bot. **Figure 12**
2. On voit dans les logs le message traité par le consumer Kafka

4ème route : lire depuis la base de données

Laissons l'application tourner en dev mode. Créons un nouvel endpoint REST en utilisant le composant Platform-http. Cet endpoint va retourner tous les messages enregistrés en base de données. L'objectif étant de s'assurer du résultat attendu. Remarquez que la classe TelegramMessage, créée précédemment, contient une NamedQuery "findAll". Nous allons l'utiliser pour récupérer tous les messages insérés dans la table.
JPA va nous retourner une liste d'objets Java de type TelegramMessage. Or, nous nous attendons à une réponse de type JSON. Nous utilisons le composant camel-jackson pour cette transformation. Voici la route Camel :

```
from("platform-http:/messages?httpMethodRestrict=GET")
        .to("jpa:" + TelegramMessage.class.getName() + "?namedQuery=findAll")
        .marshal().json();
```

La route ci-dessus démarre avec un endpoint platform-http, dont le chemin est "/messages" et dont la méthode HTTP est "GET". Afin de requêter la base, nous utilisons le producer JPA avec la propriété namedQuery. Grâce à cette propriété, le producer va exécuter la requête findAll sur cette entité.
 .marshal().json() va transformer le résultat en format JSON.
Enregistrez les modifications apportées à la classe JAVA. Quarkus relance automatiquement l'application en dev mode. Renvoyez des messages au chat bot Telegram. Vérifiez le contenu de votre base de données, en invoquant l'endpoint REST. Par défaut, l'application démarre sur le port 8080, et l'endpoint devrait être normalement disponible sur http://localhost:8080/messages. **Figure 13**

Générer et tester les livrables

Pour faire tourner le livrable, il faut disposer d'instances de base de données et kafka. Une des solutions est de lancer des images docker. Modifier le fichier src/main/resources/application.properties, pour configurer les accès aux instances.

```
# kafka
camel.component.kafka.brokers=INSERER_ICI_URI_BROKERS_KAFKA
#postgresql
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=INSERER_ICI_USER_POSTGRESQL
quarkus.datasource.password=INSERER_ICI_PASSWORD_POSTGRESQL}
quarkus.datasource.jdbc.url=INSERER_ICI__JDBC_URL
```

En mode JVM

Créer le livrable :

```
$ ./mvnw clean package
```

Lancer l'application en local :

```
$ java -jar target/quarkus-app/quarkus-run.jar
```

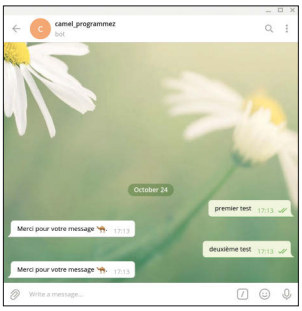


Figure 12

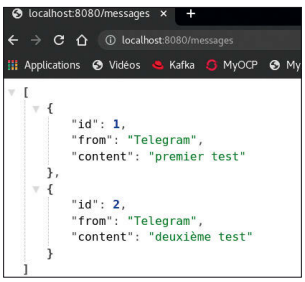


Figure 13

En mode natif:

Créer le livrable :

```
$ ./mvnw clean package -Pnative
```

Lancer l'application en local :

```
$ ./target/*-runner
```

PARTIE 3 : DÉPLOYER SUR OPENSHIFT

Pour déployer l'application sur Openshift, il faut être connecté au préalable à son projet depuis son environnement Openshift.

Il y a plusieurs manières de déployer une application Quarkus sur Openshift. Nous allons créer le livrable et le déployer avec la même commande maven.

Ajoutons les 2 extensions quarkus suivantes dans le fichier pom.xml:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-openshift</artifactId>
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-container-image-openshift</artifactId>
</dependency>
```

Les 2 extensions vont permettre de créer l'image et de déployer directement sur Openshift.

Ajouter l'extension Camel suivante dans le fichier pom.xml. Celle-ci permet de monitorer l'application sur Openshift.

```
<dependency>
    <groupId>org.apache.camel.quarkus</groupId>
    <artifactId>camel-quarkus-microprofile-health</artifactId>
</dependency>
```

Nous allons rajouter certaines configurations au fichier src/main/resources/application.properties, afin de déterminer que l'on souhaite créer le container, et spécifier l'environnement kubernetes de déploiement. Nous spécifions que l'on souhaite exposer nos routes Openshift, afin de relier notre endpoint REST à une URL externe.

```
quarkus.container-image.build=true
quarkus.kubernetes.deployment-target=openshift
quarkus.openshift.route.expose=true
```

Exécuter la ligne de commande suivante, qui va déclencher un build et un déploiement :

```
$ ./mvnw clean package -Dquarkus.kubernetes.deploy=true
```

Allez sur votre console de développement Openshift et attendez que l'application démarre. **Figure 14**

On peut aussi s'assurer que le pod est bien démarré, en exécutant la ligne de commande :

```
$ oc get pods
```

Exemple de réponse, en supposant que notre application est nommée integration-sample:

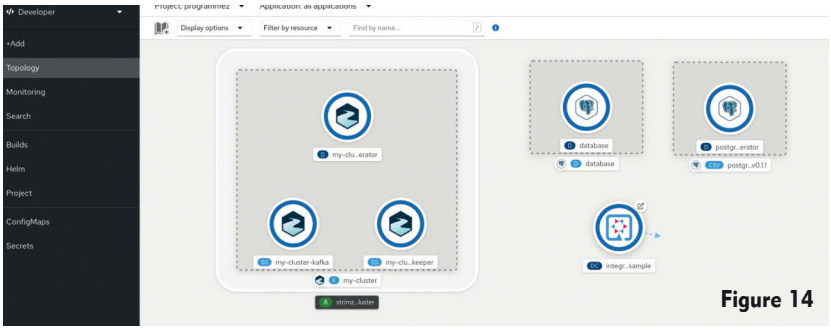


Figure 14

```
$ oc get pods
NAME                                  READY  STATUS   RESTARTS  AGE
integration-sample-dbc56974b-ph29m  1/1    Running      0     2m34s
```

Pour accéder aux logs, on peut lancer la commande suivante:

```
$ oc logs -f integration-sample--dbc56974b-ph29m
```

Pour récupérer l'URL externe du service :

```
$ oc get route integration-sample
```

Pour tester l'endpoint REST

```
$ curl -X GET <URL>/messages
```

Après avoir fini vos tests, vous pouvez supprimer votre application, en exécutant la commande suivante:

```
$ oc delete all -l app.kubernetes.io/name=integration-sample
```

Conclusion

Si vous souhaitez vous lancer dans le développement d'applications qui s'intègrent à divers systèmes externes, Apache Camel est votre allié. Vous gagnerez un temps précieux en utilisant la DSL Camel. Vous vous focaliserez principalement sur vos uses case métier, et non à maîtriser les différents API, librairies ou protocoles des systèmes externes.

Apache Camel s'intègre aussi dans les architectures Cloud, en offrant notamment des projets orientés Kubernetes tels Camel Quarkus et Camel K. Nous avons vu à travers cet article, comment créer et développer son application Camel facilement avec Quarkus. Nous avons aussi découvert comment déployer son application Quarkus sur Openshift avec une seule ligne de commande maven. En effet, Camel Quarkus permet à Apache Camel de bénéficier de toutes les optimisations de Quarkus autour de Java pour des environnements Kubernetes, mais aussi de la superbe expérience développeur propre à Quarkus. Maintenant, il ne vous reste plus qu'à vous lancer et découvrir les différents composants Camel, ainsi que la richesse des fonctionnalités d'intégration qu'offre le framework.

Liens :

https://camel.apache.org/
https://camel.apache.org/camel-kamelets/latest/
https://www.enterpriseintegrationpatterns.com/
https://quarkus.io/
https://telegram.org/
https://camel.apache.org/camel-quarkus/latest/reference/extensions/telegram.html
https://docs.openshift.com/container-platform/4.7/cli_reference/openshift_cli/getting-started-cli.html
https://developers.redhat.com/developer-sandbox
https://quarkus.io/guides/dev-services

Nicolas Massé

Solution Architect chez Red Hat. Il aide ses clients à adopter OpenShift ainsi que les outils de sécurité de Red Hat (API Management, Single Sign On, sécurité des conteneurs, etc.). En dehors de Red Hat, il écrit aussi à propos de l'Open Source sur son blog (itix.fr).

Sécurisez vos applications avec Red Hat SSO !

L'authentification, les habilitations et la traçabilité (aussi connue sous le nom de "AAA" en anglais - pour Authentication, Authorization and Audit) font partie de ce que l'on appelle les "Non-Functional Requirements". Si ces fonctions n'apportent aucune fonctionnalité métier, elles n'en demeurent pas moins nécessaires, voire critiques ! Heureusement, Red Hat SSO aide le développeur en implémentant ces fonctions et comme les autres logiciels de l'écosystème Red Hat, il se déploie très facilement sur OpenShift.

Du SSO dans vos projets !

Le Single Sign-On, « authentification unique » en français, fait partie des technologies incontournables :

- Le SSO améliore l'expérience utilisateur. Vos utilisateurs ne veulent plus avoir à retenir un énième mot de passe pour accéder à votre service. Offrez leur la connexion via leur réseau social préféré (ou pour les applications internes, via l'identité d'entreprise) !
- Le SSO réduit les risques sur le projet. Les contraintes actuelles sur la gestion des identités et accès (RGPD, vol de données, etc.) font qu'il devient de plus en plus difficile de l'imlémenter correctement soi même. Et c'est aussi le risque de voir la liste d'exigences s'agrandir en cours de projet… C'est pourquoi il est préférable de faire appel à un composant logiciel dédié !
- Il ne passe pas une semaine sans qu'une société ne soit victime d'une attaque informatique. Ne pas stocker de mots de passe dans votre application, c'est encore la meilleure façon de ne pas tenter les pirates !

Red Hat SSO est une solution clé en main pour gérer le SSO dans nos application, à moindre frais !

Red Hat SSO est la version packagée par Red Hat de la communauté Keycloak. C'est un logiciel qui dispense au développeur d'avoir à gérer les protocoles de sécurité dans son application : ils sont gérés par une communauté d'experts aguerris et c'est autant de problèmes potentiels en moins !

En particulier, Red Hat SSO prend en charge les protocoles SAMLv2 et OpenID Connect qui sont les deux standards permettant de mettre en œuvre la fédération d'identité. Il supporte aussi les fournisseurs d'identité des principaux réseaux sociaux et peut aussi s'interfacer avec France Connect.

Red Hat SSO prend en charge les protocoles LDAP, Kerberos, les certificats X.509 ainsi que les jetons OTP. C'est un vrai couteau suisse de la gestion des identités et des accès ! La suite de l'article montre comment sécuriser une application Quarkus avec Red Hat SSO. Mais si Java n'est pas votre langage de programmation préféré, pas d'inquiétude ! La plupart des langages de programmation ou framework fournissent un support OpenID Connect qui fonctionnera avec Red Hat SSO.

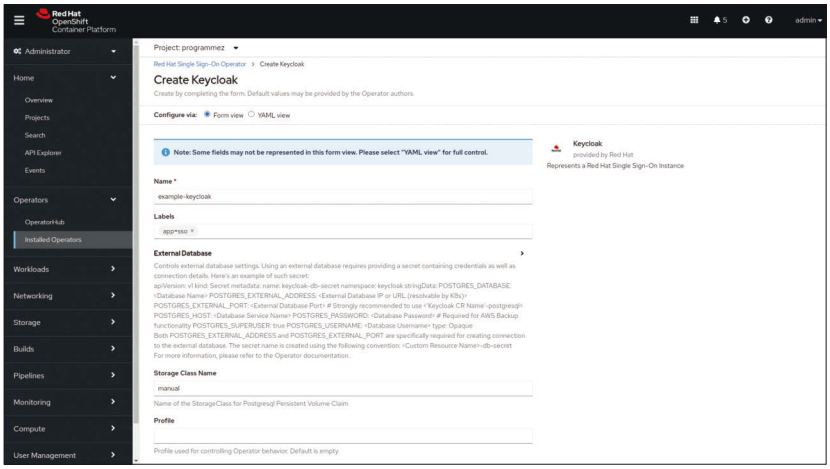


Figure 1 : *Formulaire de création de la Custom Resource Definition "Keycloak".*

Déploiement dans OpenShift

Red Hat SSO se déploie dans OpenShift soit via les templates présents dans le catalogue de services (**Developer** > **+Add** > **All services** et saisir "sso" dans le champ de recherche), soit via un opérateur disponible dans l'Operator Hub.

A la date de l'écriture de cet article, l'installation via les templates OpenShift est la seule méthode documentée et supportée par Red Hat en production. Néanmoins, il est probable qu'à l'avenir l'opérateur devienne la méthode privilégiée. Je m'attacherai donc à documenter ici l'installation via l'opérateur.

L'installation peut s'effectuer, en ligne de commande ou via l'interface graphique. Nous installerons ici Red Hat SSO et son opérateur via cette dernière méthode.

Installation de l'opérateur

Naviguez dans **Administrator** > **Operators** > **Operator Hub**, saisissez "sso" dans le champ de recherche et vous trouverez normalement un opérateur très sobrement intitulé "Red Hat Single Sign-On Operator".

Cliquez sur la tuile et OpenShift vous présentera le détail de l'opérateur ainsi que ses capacités.

Cliquez sur **Install** et laissez les options par défaut. L'opérateur s'installe obligatoirement dans un namespace (ici j'ai choisi le namespace "programmez").

Cliquez à nouveau sur **Install** et laissez OpenShift terminer l'installation de l'opérateur.

Déploiement de Red Hat SSO

Une fois l'opérateur installé, le déploiement s'effectue via la création d'une Custom Resource Definition (CRD) de type "Keycloak".

Naviguez dans **Administrator** > **Operators** > **Installed Operators** et cliquez sur **Red Hat Single Sign-On**. Ouvrez l'onglet **Keycloak**, et cliquez sur **Create Keycloak**. **Figure 1**

Dans ce formulaire, choisissez un nom pour la CRD et saisissez le nom de la classe de stockage à utiliser. La classe de stockage est utilisée pour allouer du stockage pour la base de données PostgreSQL qui stocke la configuration de Red Hat SSO. La classe de stockage dépend de l'infrastructure sous-jacente à votre plateforme OpenShift: par exemple sur amazon, ce sera probablement de l'EBS. Sur mon cluster OpenShift, la classe de stockage s'appelle "manual". Dans le doute, vous pouvez laisser ce champ vide et ainsi utiliser la classe de stockage par défaut si celle-ci a été configurée.
Cliquez sur **Create** et attendez que tous les Pods se stabilisent dans un état **Running** avec les *readiness probes* à 1/1.

Première connexion et création d'un royaume

Maintenant que vous avez une instance fonctionnelle de Red Hat SSO, vous pouvez vous connecter à son interface web d'administration et créer un royaume.
Le royaume est une unité logique qui permet d'isoler les configurations les unes des autres. Cela permet à plusieurs projets d'être hébergés sur la même instance Red Hat SSO et ainsi mutualiser les ressources.

Connexion à l'interface web d'administration

Naviguez dans **Administrator** > **Workload** > **Secrets** et cliquez sur le secret **credential-example-keycloak** (ce sera "credential-", suivi du nom de la CRD Keycloak créé si vous avez changé le nom par défaut).
Une fois le secret ouvert, cliquez sur **Reveal values** pour afficher le login et le mot de passe de l'administrateur initial. Notez soigneusement le mot de passe.
Naviguez dans **Administrator** > **Networking** > **Routes** et ouvrez la route **keycloak**. Juste sous le champ **Location**, cliquez sur l'URL .
Sur la page d'accueil qui s'affiche, cliquez sur **Administration Console**. Une page de connexion s'affiche, saisissez le login de l'administrateur initial (admin) et le mot de passe généré par l'opérateur. Bravo ! Vous êtes maintenant connecté à votre instance Red Hat SSO !
L'interface d'administration (**Figure 2**), affiche par défaut le royaume "master": c'est un royaume particulier et il est plutôt recommandé de ne pas (trop) y toucher. C'est pour cette raison que dans la prochaine section, nous créerons un royaume dédié à nos tests.
Dans chaque royaume, on retrouve la même structure:

- Les **Realm Settings** regroupent les paramètres d'affichage du royaume (nom et thème) et des paramètres de sécurité (durée de validité des jetons, protection contre les attaques et clés cryptographiques utilisées).
- Les **Clients** sont les applications à sécuriser qui viennent s'authentifier sur Red Hat SSO. Les **Client Scopes** sont un moyen de factoriser une partie de la configuration de ces clients.
- Les **Roles** sont un moyen de gérer les habilitations transverses à tout un royaume (par exemple, un auditeur pourrait, via le rôle éponyme, avoir un droit en lecture sur toutes les applications du royaume).
- Les **Identity Providers** permettent aux utilisateurs de se connecter à Red Hat SSO (et à ses clients) en réutilisant leurs identités d'entreprise (SAMLv2 ou OpenID Connect) ou sociales (Facebook, GitHub, etc.)
- La section **User Federation** permet de puiser les identités des utilisateurs (et de les authentifier) depuis des entrepôts d'identités tels que LDAP ou Kerberos.
- La section **Authentication** permet de configurer les cinématiques d'authentification ainsi que les politiques de mot de passe.
- Les interfaces **Users** et **Groups** permettent de consulter et administrer les utilisateurs et les groupes.
- La section **Sessions** permet de consulter les sessions ouvertes par application et de les invalider.
- L'interface **Events** permet de consulter qui s'est connecté sur Red Hat SSO, quelles applications ont été accédées, etc. C'est la partie "audit".
- Les sections **Import** et **Export** offrent à l'administrateur la possibilité, comme son nom l'indique, d'exporter la configuration d'un royaume pour la réimporter plus tard ou sur une autre instance. Et comme c'est un simple fichier JSON dont le format est documenté, on peut aussi s'en servir pour mettre en œuvre de la "Configuration-as-Code" !

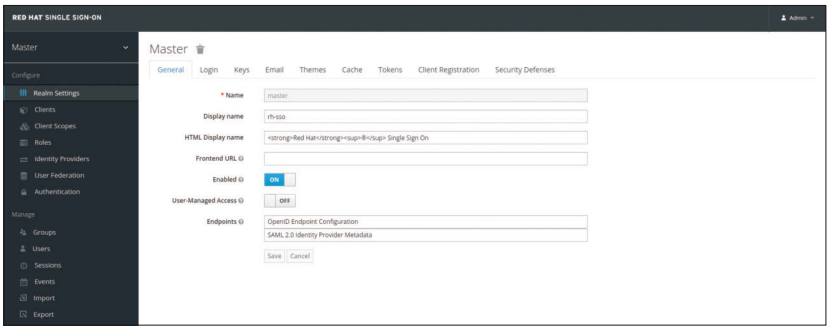


Figure 2 : *L'interface d'administration de Red Hat SSO.*

Création d'un royaume

Dans l'interface d'administration en haut à gauche, juste à droite de **Master**, vous trouverez une flèche vers le bas qui vous permet de sélectionner un royaume. Vu qu'il n'y en a qu'un seul, vous n'aurez qu'un bouton bleu **Add realm**.
Cliquez sur ce bouton, choisissez un nom et cliquez sur **Create**.

Sécurisation d'une application Quarkus

Dans cette section, nous allons sécuriser une application Quarkus en délégant l'authentification, la gestion des habilitations et l'audit à Red Hat SSO.
L'application Quarkus d'exemple est disponible sur le repository Git suivant : github.com/nmasse-itix/programmez-article-sso. Elle se présente sous la forme d'un backend REST permettant de gérer un catalogue d'animaux de compagnie (la fameuse Petstore API). La mise en œuvre est standard pour une application Quarkus : git clone & mvn compile.

```
$ git clone https://github.com/nmasse-itix/programmez-article-sso.git
$ cd programmez-article-sso
$ ./mvnw compile quarkus:dev
```

Dans la suite de cet article, j'utiliserai la commande **http** (projet HTTPie.io) pour illustrer les principales opérations REST disponibles.
Dans sa livrée originelle, cette application d'exemple arrive

nue: ni authentification, ni gestion des habilitations. Pour s'en convaincre, il suffit de requêter l'API.

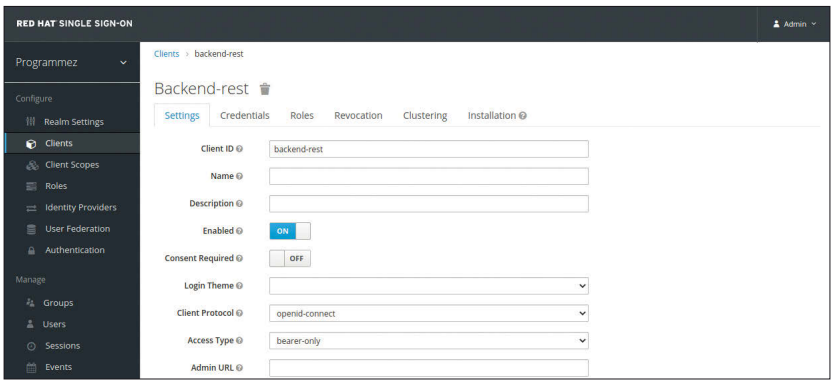
```
$ http http://localhost:8080/pets/
HTTP/1.1 200 OK
Content-Length: 77
Content-Type: application/json

[
  {
    "id": 1,
    "name": "Eclair",
    "tag": "cat"
  },
  {
    "id": 2,
    "name": "Cannelle",
    "tag": "cat"
  }
]
```

L'API REST est effectivement non sécurisée. Afin de la sécuriser, nous devrons :

- créer un client dans Red Hat SSO pour notre backend REST,
- créer un client dans Red Hat SSO pour notre client REST (simulé ici par HTTPie),
- configurer les habilitations par défaut dans Red Hat SSO,
- créer un utilisateur de test,
- et enfin, configurer notre application pour utiliser Red Hat SSO !

Figure 3
le client correspondant à notre backend REST est de type "bearer-only".



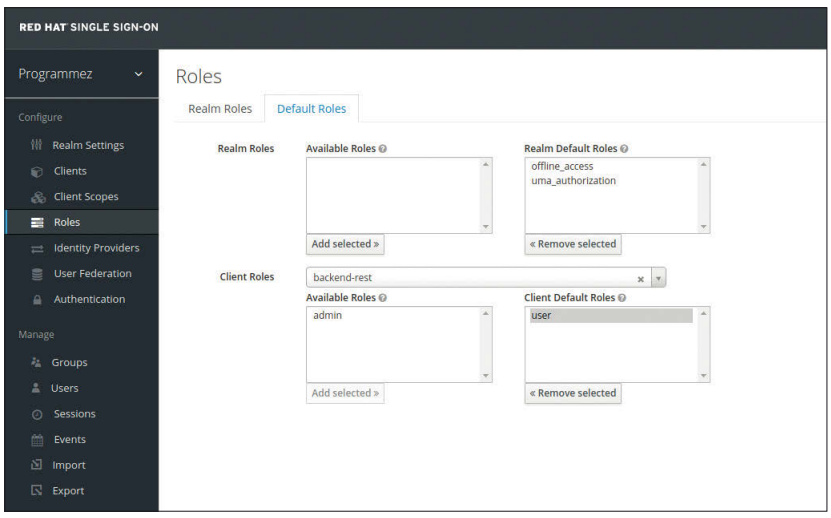


Figure 4 : *tous les utilisateurs du royaume auront maintenant le rôle "user" sur l'application "backend-rest"*

Préparatifs

Création du client "backend-rest"

Dans Red Hat SSO, ouvrez votre royaume et naviguez dans **Configure** > **Clients**. Cliquez sur **Create**, choisissez un **Client ID** ("backend-rest" par exemple) et cliquez sur **Save**.
Dans le formulaire qui s'affiche (**Figure 3**), passez le Access Type à **bearer-only**. Notre backend REST ne va pas demander de jetons, il va juste les valider : nous n'avons pas besoin de plus.
Cliquez sur **Save**.
Passez sur l'onglet **Roles** et cliquez sur **Add Role**. Dans le champ **Role Name**, saisissez la valeur **user**. Cliquez sur **Save**.
En suivant la même procédure, créez aussi un rôle **admin** qui nous servira pour démontrer la gestion des habilitations.

Création du client "client-rest"

Créez maintenant le client REST en naviguant dans **Configure** > **Clients**. Cliquez sur **Create**, choisissez un **Client ID** ("client-rest" par exemple) et cliquez sur **Save**.
Passez le champ **Access Type** à **Confidential**, décochez l'option **Standard Flow Enabled** pour ne laisser que **Direct Access Grant Enabled**. Cette configuration n'est en général pas utilisée en production mais elle a le mérite de nous permettre de recetter notre backend REST sans avoir à faire appel à un Selenium pour la partie authentification de l'utilisateur.

> A ce stade, quelques explications s'imposent sur les différents types de clients et de *flows* !
> Les clients de type **Public** sont utilisés pour la partie "front" des applications (Single Page App en React, Angular, etc.). Ces clients sont souvent utilisés avec un flow de type **Standard** ou **Implicit**.
> Les clients de type **Confidential** sont utilisés pour la partie "back" des applications (le backend Java, NodeJS, PHP, etc). Comme le backend est considéré comme sécurisé, on lui affecte un **Client Secret**. Ça renforce la sécurité. Ces clients sont souvent utilisés avec un flow de type **Standard** (*Authorization Code* dans la terminologie OpenID Connect) lorsqu'il faut authentifier l'utilisateur ou **Service Account** (*Client Credentials* dans la terminologie OAuth 2.0) lorsque authentifier l'application appelante est suffisant.
> Le flow **Direct Access Grants** (*Resource Owner Password Credentials* dans la terminologie OAuth 2.0) est rarement utilisé car il impose de faire confiance de manière démesurée à l'application cliente. Mais il est utile pour les tests !

Cliquez sur **Save** pour enregistrer vos modifications. Notez l'apparition d'un onglet **Credentials** contenant un **Secret**. Copiez-collez le quelque part, nous en aurons besoin plus tard !

Configuration des habilitations par défaut

Afin que nos utilisateurs puissent accéder à notre backend REST, nous allons leur affecter d'office le rôle **user**. Pour cela, naviguez dans **Configure** > **Roles** et sélectionnez l'onglet **Default Roles**. Dans le champ **Client Roles**, saisissez "backend-rest". Dans la liste de rôles qui apparaissent, sélection-

nez **user** et cliquez sur **Add selected >>** pour le passer dans la liste de droite (**Figure 4**).

Création de l'utilisateur de test

Dernière étape préparatoire avant de plonger dans le code de l'application Quarkus: créer un utilisateur de test dans Red Hat SSO. Celui-ci nous permettra de faire nos tests de bout en bout.

Naviguez dans **Manage** > **Users** et cliquez sur **Add user**. Dans le formulaire qui s'affiche choisissez un login, un nom et un prénom pour cet utilisateur et cliquez sur **Save**. Dans l'onglet **Credentials**, saisissez un mot de passe pour cet utilisateur, confirmez, décochez la case **Temporary** et cliquez sur **Set Password**. Confirmez en cliquant à nouveau sur **Set Password**.

Mise en place de l'authentification

À partir de maintenant, nous pouvons nous concentrer sur le code de notre application et mettre en place l'authentification. Pour cela, nous il nous faudra :

- éditer le fichier application.properties pour indiquer à la bibliothèque OpenID Connect où se trouve le serveur Red Hat SSO,
- ajouter la bibliothèque OpenID Connect au projet Quarkus,
- ajouter les annotations sur nos méthodes REST pour indiquer celles qui doivent être sécurisées,
- récupérer un jeton auprès de Red Hat SSO,
- et appeler notre API REST en fournissant le jeton en paramètre.

Ouvrez le fichier src/main/resources/application.properties et définissez la clé quarkus.oidc.auth-server-url. Cette clé doit contenir l'URL de votre royaume Red Hat SSO, au format https://<SSO_HOSTNAME>/auth/realms/<REALM>.

Définissez ensuite la clé quarkus.oidc.client-id. Cette clé doit contenir le nom du client que vous avez choisi pour le backend REST ("backend-rest" dans les captures d'écran ci-dessus).

Si comme moi, votre cluster OpenShift n'a pas de certificat TLS valide, il vous faudra également définir la clé quarkus.oidc.tls.verification.

Dans mon environnement, le fichier application.properties ressemble à ça :

```
quarkus.oidc.auth-server-url=https://keycloak-programmez.apps.itix-dev.ocp.itix/auth/realms/programmez
quarkus.oidc.client-id=backend-rest
quarkus.oidc.tls.verification=none
```

Ajoutez à votre projet Quarkus la bibliothèque io.quarkus.quarkus-oidc. Vous pouvez le faire directement en éditant le pom.xml ou bien via maven.

```
./mvnw quarkus:add-extension -Dextensions="oidc"
```

Ouvrez maintenant le fichier src/main/java/com/redhat/petstore/PetstoreResource.java et ajoutez l'annotation javax.annotation.security.RolesAllowed à la classe PetstoreResource.

Cette annotation prend en paramètre le ou les rôles requis pour appeler la méthode (si l'annotation est positionnée sur une méthode) ou toutes les méthodes (si l'annotation est est positionnée sur une classe).

```
import javax.annotation.security.RolesAllowed;

@RolesAllowed({"user"})
public class PetstoreResource {
 [...]
}
```

Requêtez à nouveau l'API REST et vous devriez être gratifié d'un code 401 Unauthorized, signe que l'authentification a bien été mise en place.

```
$ http http://localhost:8080/pets/
HTTP/1.1 401 Unauthorized
Content-Length: 0
www-authenticate: Bearer
```

Pour utiliser l'API REST de notre backend, nous devons maintenant récupérer un jeton auprès de Red Hat SSO et le passer en paramètre à notre appel REST. Seulement, par défaut, les jetons expirent au bout d'une minute dans Red Hat SSO. Autant dire que l'on peut passer un temps non négligeable durant la mise au point à récupérer des jetons !

Pour éviter ce problème, je vous propose l'astuce suivante. Définissez la fonction Bash get_token. Elle est composée d'une commande curl qui exécute le flow "Resource Owner Password Credentials" du protocole OAuth 2.0 et d'une commande jq qui extrait l'Access Token de la réponse si celui-ci est présent ou affiche la réponse entière dans le cas contraire (quand Red Hat SSO retourne une erreur notamment).

```
function get_token () {
 curl -sk -XPOST -d grant_type=password -d scope=openid \
 https://$SSO_HOSTNAME/auth/realms/$SSO_REALM/protocol/openid-connect/token \
 -d client_id=$CLIENT_ID -d client_secret=$CLIENT_SECRET \
 --data-urlencode username=$LOGIN \
 --data-urlencode password=$PASSWORD \
 | jq -r 'if .access_token then .access_token else . end'
}
```

Définissez les variables SSO_HOSTNAME, SSO_REALM, CLIENT_ID, CLIENT_SECRET, LOGIN et PASSWORD. Dans mon environnement, ça donne :

```
SSO_HOSTNAME=keycloak-programmez.apps.itix-dev.ocp.itix
SSO_REALM=programmez
CLIENT_ID=client-rest
CLIENT_SECRET=1fb49759-ebf5-4891-868c-bb0df7530f31
LOGIN=john
PASSWORD=s3cr3t
```

Vérifiez que vous obtenez bien un jeton :

```
$ get_token
eyJhbGci [...]
```

Vous pouvez maintenant requêter l'API REST en passant un jeton dans l'en-tête HTTP "Authorization".

```
$ http http://localhost:8080/pets/ "Authorization:Bearer $(get_token)"
HTTP/1.1 200 OK
Content-Length: 77
Content-Type: application/json

[
  {
```

```
    "id": 1,
    "name": "Eclair",
    "tag": "cat"
  },
  {
    "id": 2,
    "name": "Cannelle",
    "tag": "cat"
  }
]
```

Si tout est bien configuré, vous devriez obtenir un code HTTP 200.

Gestion des habilitations

Dans l'implémentation que nous avons faite jusqu'à présent, n'importe quel utilisateur ayant le rôle "user" (qui rappelons-le, est donné par défaut à tout utilisateur du royaume) peut accéder à la totalité des méthodes REST de notre API.

Pour ajouter un peu de vraisemblance à ce scénario, nous allons différencier les rôles requis pour chaque méthode de l'API REST : les méthodes en lecture nécessiteront le rôle "user" et les méthodes en écriture nécessiteront le rôle "admin".

Ouvrez à nouveau le fichier src/main/java/com/redhat/petstore/PetstoreResource.java et laissez l'annotation RolesAllowed sur la classe PetstoreResource: ce sera la valeur par défaut pour les méthodes non-annotées. Ajouter cette même annotation aux méthodes add, update et remove en prenant soin de passer en paramètre le rôle "admin".

Code complet sur programmez.com & github

Tentez de supprimer "Éclair" du catalogue et vérifiez que vous obtenez bien un code HTTP 403.

```
$ http DELETE http://localhost:8080/pets/1 "Authorization:Bearer $(get_token)"
HTTP/1.1 403 Forbidden
Content-Length: 0
```

Maintenant, retournez dans l'interface de Red Hat SSO. Naviguez dans **Manage** > **Users**. Cliquez sur **View all users** et cliquez sur le bouton **Edit** en regard de "john".

Ouvrez l'onglet **Role Mappings**, saisissez dans le champ **Client Roles** "backend-rest". Sélectionnez le rôle "admin" dans la colonne **Available Roles** et cliquez sur **Add Selected >>** pour le passer dans la colonne **Assigned Roles**.

Re-tentez de supprimer "Éclair" du catalogue et vérifiez que vous obtenez cette fois-ci un code HTTP 200.

```
$ http DELETE http://localhost:8080/pets/1 "Authorization:Bearer $(get_token)"
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json

{
  "id": 1,
  "name": "Eclair",
  "tag": "cat"
}
```

Traçabilité

Après avoir implémenté l'authentification et la gestion des habilitations, mettons en place la traçabilité.

Naviguez dans **Manage** > **Events**. Ouvrez l'onglet **Config** et sous la section **Login Events Settings**, cochez la case **Save Events** et cliquez sur **Save**. Faites une nouvelle requête à l'API REST (incluant un nouveau jeton), par exemple pour vérifier qu'Éclair ne fait plus partie du catalogue.

```
$ http http://localhost:8080/pets/ "Authorization: Bearer $(get_token)"
HTTP/1.1 200 OK
Content-Length: 40
Content-Type: application/json

[
  {
    "id": 2,
    "name": "Cannelle",
    "tag": "cat"
  }
]
```

Maintenant, naviguez dans **Manage** > **Events**. Dans l'onglet **Login Events**, cliquez sur **Update** et observez le résultat.

Pour faire le lien entre l'authentification de John sur Red Hat SSO et ses actions effectuées dans l'API REST, nous allons devoir instrumenter un peu notre code.

Ouvrez à nouveau le fichier src/main/java/com/redhat/petstore/PetstoreResource.java et ajoutez les lignes suivantes.

```
import org.eclipse.microprofile.jwt.JsonWebToken;
import org.jboss.logging.Logger;

public class PetstoreResource {
  @Inject
  JsonWebToken jwt;

  private static final Logger LOG = Logger.getLogger(PetstoreResource.class);

  public Set<Pet> list() {
    LOG.info("Access from " + jwt.getTokenID() + " to method list");
    [...]
  }

  // Les méthodes add, get, update et delete sont à modifier sur le même principe
}
```

En relançant une nouvelle requête à l'API REST, vous verrez apparaître le log suivant:

```
Access from a35a5ae1-0509-4edb-ad52-44d933625ff8 to method list
```

Et vous pourrez constater que l'UUID ci-dessus correspond au champ token_id de l'événement **LOGIN** dans Red Hat SSO. Voilà qui nous permet de faire le lien entre les logs de Red Hat SSO et les logs applicatifs !

Conclusion

Nous avons présenté brièvement Red Hat SSO et l'avons déployé dans OpenShift. Nous l'avons configuré pour prendre en charge l'authentification, la gestion des habilitations et la traçabilité. Notre application Quarkus a été modifiée pour s'intégrer à Red Hat SSO.

Cybersécurité : RESTEZ INFORMÉ

L'actualité quotidienne
News, avis d'experts, témoignages, livres blancs, etc.
https://www.solutions-numeriques.com/securite/

La newsletter
Chaque lundi, comme 40 000 professionnels et décideurs, recevez la synthèse des informations.
C'est gratuit, inscrivez vous :
https://www.solutions-numeriques.com/securite/inscription/

L'annuaire : Guide papier et en ligne
Trouvez l'éditeur de solution, le prestataire de services qu'il vous faut.
https://www.solutions-numeriques.com/securite/annuaire-cybersecurite/

softfluent

Rejoignez **l'ESN** créée par des **développeurs** pour les Développeurs

Vous possédez une ou plusieurs de ces compétences ?

- Azure DevOps
- Cloud
- SQL Server
- Angular
- React
- C#
- CI/CD
- .NET Core
- ASP.NET MVC
- Microservices

Contactez nous !

Retrouvez-nous sur notre page carrière : **softfluent.fr/nous-rejoindre**

Ou contactez Marine, en charge du recrutement chez SoftFluent :

06 69 26 27 87

marine.genetay@softfluent.com

- Conseil
- Expertise
- Partage

softfluent.fr