PROGRAMMEZ!

Le magazine des développeurs

PROGRAMMEZ!

SPÉCIAL
ÉTÉ
2021

100% JAVA

Java 16
Quarkus
GraalVM
QuickPerf
JUnit
Refactoring
reactor
Bien choisir une JVM

© Swillkditch

Le seul magazine écrit par et pour les développeurs

M 01642 - 4H - 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

SPÉCIAL ÉTÉ 2021

Disponible dès maintenant

Kiosque / Abonnement - Version papier / Version PDF

Contenus

6 Agenda
Les événements développeurs
La rédaction

8 Ecosystème, évolutions, enjeux…
Comment le marché Java à évoluer ? Quelles compétences ?
Quelles attitudes pour le développeur ?
Océane Roudier & Nourdine Bouaghaz

12 Il était une fois Java
Aux origines de Java
François Tonic

14 Développer en Java en 2021
Tout ce qui a changé pour le développeur Java depuis la version 11
Lilian Benoit

22 Java 16 : quoi de neuf ?
Focus sur la version 16 de Java et les 17 nouvelles JEP
Loïc Mathieu

25 Introduction au Machine Learning avec Tribuo
Comment faire du machine learning en Java avec Tribuo ?
Elvadas Nono

Architecture microservice…
Faisons un peu de DDD dans une architecture microservice tout en utilisant des concepts de l'architecture hexagonale.
Badr Nass Lahsen

31 Compiler et exécuter du CoBOL avec GraalVM !
Java et CoBOL incompatibles ? Avec GraalVM et GnuCOBOL, on peut faire des merveilles et exécuter du vénérable code CoBOL dans la JVM.
Christophe Brun

33 Mieux maîtriser la performance applicative avec QuickPerf
La performance a toujours été une question sensible dans le monde Java. QuickPerf peut vous aider.
Jean Bisutti

36 Reactor
Et si on codait réellement des applications non bloquantes ? C'est l'ambition de Reactor.
Antoine Michaud

44 Le refactoring de code legacy
Le refactoring est un classique de la programmation. Petit rappel que tout développeur devrait connaître.
Bruno Boucard

51 Dossier cloud native avec Quarkus !
Java s'adapte à tout et aujourd'hui avec Quarkus, on peut aisément s'intégrer au monde du cloud computing.
Daniel Petisme, Fabien Pomerol, Julien Millau

64 Quarkus et GCP
Regardons maintenant du côté de Google Cloud Platform et comment on peut utiliser et déployer Quarkus
Loïc Mathieu

66 JUnit : il est temps de passer la 5e
Redécouvrons JUnit et particulièrement la version 5.
Juliette de Rancourt & Julien Topçu

71 Mutation testing
La qualité du code est un enjeu mais comment faire ? Prenez en main Pitest, un outil de test de mutation.
Samuel Marques Antunes

74 Adoptez un JDK !
Passons en revue les JDK du marché et comment choisir la meilleure pour mon développement ?
Fayssal Merimi

82 Le strip du mois
CommitStrip

Divers

4 Edito
Du chêne au café, il n'y a qu'une ligne de code

42 43 Abonnement & boutique

Programmez! est une publication bimestrielle de Nefer-IT.
Adresse : 57, rue de Gisors 95300 Pontoise – France. Pour nous contacter : redaction@programmez.com

#HS4JAVA

Du chêne au café, il n'y a qu'une ligne de code

© Excelsior, L'Heure du dev

Java est une technologie importante de notre paysage informatique depuis 25 ans ! Qui aurait pu prédire un destin aussi incroyable pour un « truc » développé par une toute petite équipe dans un recoin des locaux de Sun Microsystems ? Au départ, il s'agit d'un projet de R&D.

Et pourtant, c'est bien ce fabuleux destin que Java va acquérir en quelques années ! Dès les années 2000, il est incontournable en entreprise et une des briques essentielles du Web de l'époque. Si, si, souvenez-vous des applets ! Les versions se succèdent, parfois, dans la douleur. James Gosling a été la tête pensante de l'univers Java durant 15 ans, et même plus. Car Gosling fut à l'origine des fondations techniques qui allèrent donner Java en 1995-96.

Le rachat de Sun par Oracle, le procès Oracle vs Google sur les codes Java utilisés dans Android, avaient jeté un froid sur les communautés : quel futur pour la plateforme ? Oui, Oracle a donné plusieurs grands projets à des fondations, mais le cœur du langage reste à la maison. Et il faut reconnaître ce mérite à l'éditeur : avoir remis de l'ordre dans l'évolution de Java avec une cadence tous les 6 mois pour lisser les versions. Et ce rythme a été repris par d'autres langages.

Il faut dire qu'il fallait parfois attendre 3-4 ans entre deux versions majeures et le développeur stressait devant tous les changements, les casses de compatibilités, etc. Ce changement de cadence, qui peut paraître infernal et beaucoup trop rapide a le mérite de rompre cette logique de version big bang. On fluidifie les évolutions même si chaque version apparaît peu innovante.

Le risque est de devoir être obligé de changer de versions plus rapidement que prévu. Cette situation est toujours un stress pour les entreprises, les équipes techniques. Mais exécuter du code basé sur une JDK vieille de 4-5 versions n'est pas neutre surtout si la JDK n'est pas LTS, support long terme avec l'assurance d'avoir des patchs de sécurité et des mises à jour sur 5 à 7 ans. Les versions LTS apportent cette stabilité attendue. Et ce n'est pas un hasard si Red Hat, Amazon et Microsoft proposent leurs propres distributions Java LTS.

Dans ce numéro spécial Java, nous allons vous démontrer que Java est au cœur des architectures modernes, de l'informatique actuelle et de demain. Le projet Quarkus montre toute la vivacité du langage et sa capacité à se dépasser. On pouvait crainte une perte de vitesse avec l'échec de Java sur mobile et sur les IoT. Le retard pris dans le monde cloud computing avait aussi inquiété.

Incontestablement, qu'on aime ou non le langage, Java est un des rares langages à être durablement présent. On peut dire la même chose de C#, C++ et du C. Cobol est hors compétition. On sait qu'il sera toujours présent dans 10 ans. Le colossal patrimoine de codes Java lui assure cette pérennité. Car impossible de migrer, réécrire des milliards de lignes de codes ni de changer le paradigme technique des centaines de millions d'applications Java.

Nous allons vous montrer toute la vivacité du langage avec JUnit 5, Quarkus, QuickPerf, GraalVM, Java 16.

Merci à toutes les communautés Java et aux contributrices et contributeurs techniques de ce numéro.

François Tonic
Rédacteur en chef plus vieux que Java

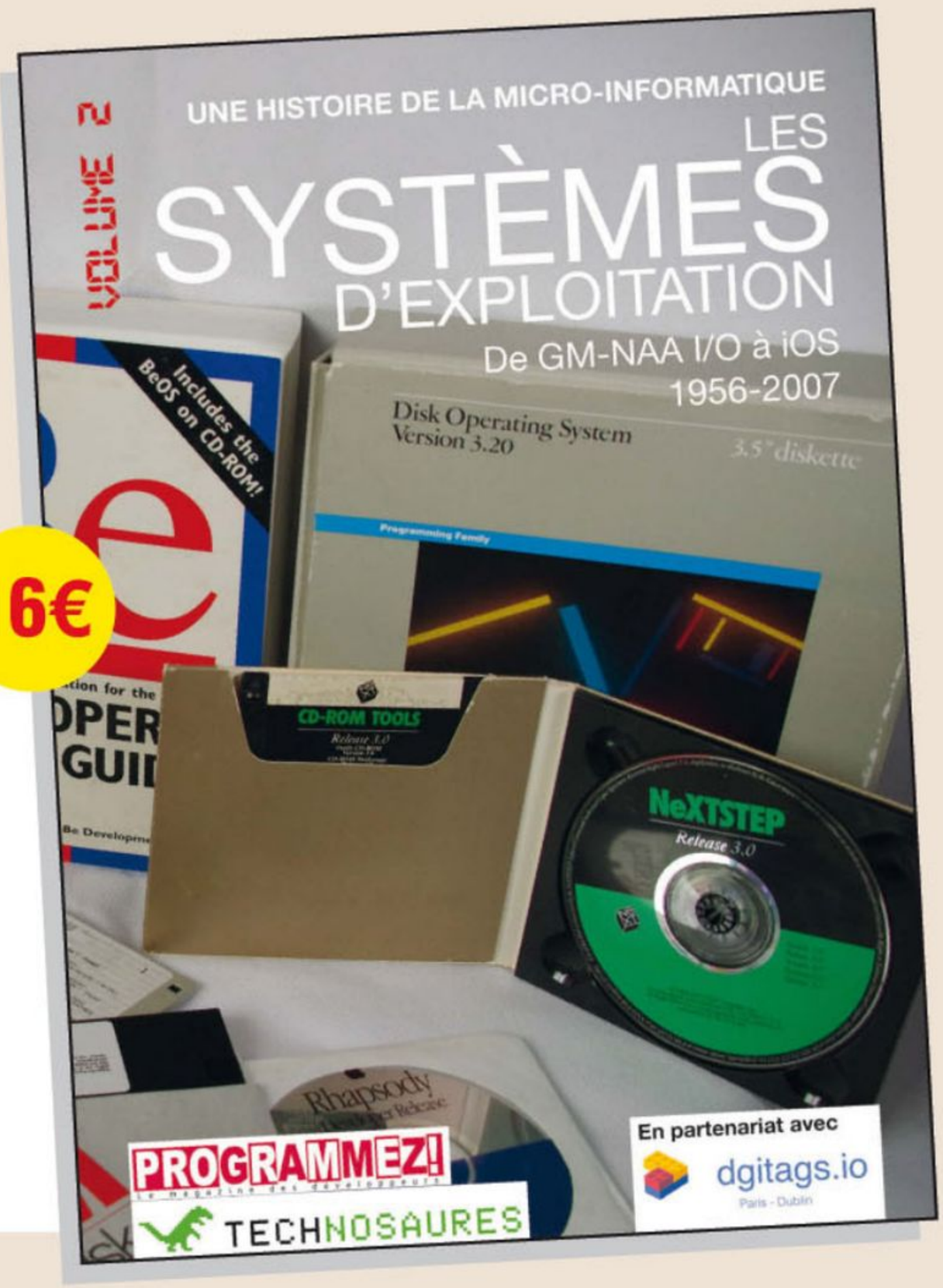
Le CADEAU idéal pour toutes/tous les geeks !

UNE HISTOIRE DE LA MICRO-INFORMATIQUE

Volume 3 : 90 nouvelles machines, + d'ordinateurs français !

116 pages. Format mook.

16€



16€

UNE HISTOIRE DE LA MICRO-INFORMATIQUE

Volume 2 : les systèmes d'exploitation de 1956 à 2007

100 pages. Format mook.

Commandez directement sur
www.programmez.com/catalogue/livres

Les événements Programmez!

Meetups Programmez!
Nous débuterons la session 2021-2022, le 7 septembre !
19 octobre - 2 novembre - 7 décembre

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

DevCon by Programmez !
23 septembre : Spécial .Net 2e édition

Où : Epitech
1 journée - 12 sessions

Informations & inscription : programmez.com

septembre

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
		1	2	3	4	5
6	7	8	9	10	11	12
	Meetup Programmez!			API Platform Conference (Lille + virtuel)	Angers GeekFest / Angers	
				JUG Summer Camp / La Rochelle		
13	14	15	16	17	18	19
	JFTL (Montrouge)					
	WAX, 100 % / Marseille					
20	21	22	23	24	25	26
			DevCon .Net 6 / Azure / Windows (Epitech Paris)			
27	28	29	30			
	Big Data Paris					
	Serverless Days Paris	Devoxx France (Paris)				

octobre

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
				1	2	3
				Devoxx France		
4	5	6	7	8	9	10
			Paris Web (Paris)			
			Cloud Bord			
11	12	13	14	15	16	17
18	19	20	21	22	23	24
	Meetup Programmez !		DevFest Nantes			
25	26	27	28	29	30	31

novembre

1	2	3	4	5	6	7
	Meetup Programmez!					
8	9	10	11	12	13	14
	Open Source Experience (Paris)					
	DevFest Strasbourg					
15	16	17	18	19	20	21
	Hack in Paris (virtuel)		Codeurs en Seine (virtuel)	DevFest Lille		
22	23	24	25	26	27	28
	Paris Test Conference 2021					
29	30	31				

RENDEZ-VOUS EN 2022

- DevOps Rex
- Sunny Tech
- NCrafts Paris
- BreizhCamp
- DevFest du bout du monde
- Best of Web
- Flutter Con Paris
- DevFest Toulouse

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

Les partenaires 2021 de

PROGRAMMEZ!

Le magazine des développeurs

Niveau maître Jedi

softfluent

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

Écosystème Java : évolutions du marché, des postures et enjeux à venir pour tout développeur

Cet article a été rédigé entièrement au masculin par souci de fluidité de lecture. CodeWorks est une entreprise inclusive ouverte à toutes les personnalités éveillé.e.s, curieu.x.ses et animé.e.s par leur passion !

Forts de notre expérience dans l'IT, d'une dizaine d'années passées à accompagner nos clients et recruter des talents, nous souhaitons vous partager ici nos observations et retours d'expériences sur l'évolution du marché Java, les fortes adhérences qui se poursuivent avec la communauté Spring, ainsi que les mutations qui s'opèrent dans le rôle de développeur.

Nous aborderons les évolutions de compétences et de posture nécessaires à nos yeux pour répondre aux enjeux d'architectures modernes de plus en plus complexes (microservices, services managés, décentralisation, performance, déploiement, etc.).

Nous partagerons également un regard critique sur l'utilisation des frameworks qui peuvent éloigner les développeurs des concepts fondamentaux.

Enfin, nous tenterons d'apporter une réponse concrète aux compétences sur lesquelles nous préconisons d'investir pour affronter sereinement les défis techniques.

Retrouvez plus de détails sur notre modèle alternatif en consultant notre manifeste ici :
https://medium.com/codeworksparis

Un rapide tour d'horizon

Développé par Sun Microsystems depuis 1995, Java devient un produit d'Oracle suite au rachat de Sun par ce dernier en 2009, jusqu'à se hisser au premier rang des langages orientés objet (quelques chiffres sur https://www.java.com/fr/about/).
La sortie régulière de versions majeures avec chacune son lot de features structurantes consolide son adoption par le marché, notamment dans le cadre de grands projets.
De plus, challengé par de nouveaux langages dans le core de la JVM (Clojure, Scala et Kotlin principalement) et intégré comme premier langage sous Spring, Java ne cesse de s'adapter et évoluer. Le succès de Spring MVC en 2008 puis de Spring Boot l'a aidé à se maintenir au rang des langages préférés.
Pour en savoir plus :
https://blog.jetbrains.com/idea/2020/09/a-picture-of-java-in-2020/
https://www.jetbrains.com/lp/devecosystem-2020/java/

Un langage qui convient plutôt aux projets importants, pourquoi ?

L'idée n'est pas de développer une liste exhaustive de pour et contre, mais de balayer rapidement les principales caractéristiques qui confortent notre idée qu'il semble adapté aux projets de grands groupes.

Démocratisation : le langage dispose du support d'une communauté très importante et particulièrement active de plus de 9 millions de développeurs qui participent à sa forte adoption dans le monde entier (1) ainsi qu'à un nombre important de ressources (documentation, tutoriels, vidéo, etc.) qui rendent son apprentissage plus accessible.

Maturité : des millions, voire milliards d'applications pour différents secteurs et entreprises ont permis de tirer des enseignements, consolider les connaissances, résoudre les problèmes et documenter largement.

Rétrocompatibilité ascendante : il offre la possibilité d'exécuter du code conçu il y a plusieurs années sur une version plus récente du JDK. Ce point rassure les organisations qui sont assurées que le code, modulo certains ajustements, fonctionnera encore pour les années à venir.

POO (2) : Ce paradigme offre différents principes (encapsulation, polymorphisme, héritage, etc.) qui assurent une expression du métier plus aisée et des facilités dans sa conception. L'arrivée, depuis la version 8, du paradigme fonctionnel ouvre également de nouveaux horizons.

Indépendance : Il s'exécute sur n'importe quelle plateforme et n'a besoin que d'une Java Virtual Machine qui s'adapte à tout type de système d'exploitation et assure une portabilité du Java Byte Code.

Langage de haut-niveau : à la différence de C ou C++, les problématiques d'allocation de la mémoire et des erreurs inévitables sont totalement prises en charge par le système. Le développeur peut alors se concentrer sur l'expression du métier.

Multithreading : il dispose de la capacité à exécuter plusieurs threads (tâches) en même temps et en parallèle.

Distribution : à l'ère des services managés (cloud computing), il fournit les fonctionnalités nécessaires pour relier des ressources distantes et optimiser les performances des applications développées.

Océane Roudier

Co-CEO chez CodeWorks et Co-organisatrice de NewCrafts Conference. Recruteuse puis Responsable RH & Communication durant 5 ans dans une ESN parisienne, Océane s'est ensuite lancée comme consultante RH freelance avant de rencontrer Nourdine et CodeWorks en 2020.

Nourdine Bouaghaz

Fondateur et Co-CEO chez CodeWorks. Ancien développeur passé par la société Linagora, puis consultant au sein du groupe Pierre Fabre et AXA France Services, Nourdine a contribué au développement commercial d'ESN durant 5 ans avant de lancer sa propre initiative CodeWorks en mars 2018.

CodeWorks *est une organisation horizontale fondée sur une promesse audacieuse : proposer un modèle alternatif, équitable et solidaire par la redistribution des richesses, l'équilibre vie personnelle / vie professionnelle, le développement des compétences et l'accompagnement empathique orchestrés par un cercle de mentors seniors. Retrouvez plus de détails sur notre modèle alternatif en consultant notre manifeste ici :*
https://medium.com/codeworksparis

Quelques limitations connues qui peuvent freiner son adoption

Performances : il reste gourmand en mémoire et plus lent que les langages compilés nativement tels que C ou C ++. C'est d'ailleurs une de ses limites lorsque des contraintes de temps réel se présentent, même si des alternatives, type GraalVM, voient aujourd'hui le jour.

Gestion de la mémoire : elle est facilitée via le garbage collection qui affecte fortement les performances de l'application puisque tous les autres threads doivent être arrêtés pour permettre au thread "ramasse-miettes" de fonctionner.

Bouleversement du modèle économique : une nouvelle cadence de version est mise en place depuis 2018, où de nouvelles « versions » de Java sont disponibles tous les 6 mois. Cependant, cette nouvelle cadence n'inclut pas systématiquement le support à long terme.
Seules les versions 11 (sortie en septembre 2018) et 17 (prévue pour septembre 2021) sont une LTS.
(https://www.oracle.com/java/technologies/java-se-support-roadmap.html)

Un fort ancrage sur le JDK8

Depuis sa sortie en mars 2014, la version 8 a apporté d'importantes fonctionnalités dans la mise en place d'architectures modernes et "Cloud Ready".
L'apparition des streams, des lambdas, la refonte de l'API de la gestion des dates, les implémentations d'interface, etc. améliorent grandement le quotidien des développeurs.
Aujourd'hui encore, nous constatons que la majorité des projets de nos clients restent figés sur la version 8 du JDK.
Cependant, ceci pose la question de la maintenance et surtout de la fin du support officiel de cette version.
L'étude publiée par JetBrains (https://www.jetbrains.com/lp/devecosystem-2020/java/) conforte notre vision avec une version 8 qui reste la plus répandue. D'autres études, comme celle publiée par Snyk (https://snyk.io/blog/jvm-ecosystem-report-2020), précisent également que le JDK 8 reste le plus utilisé en production.

De fortes adhérences avec la communauté Spring

Spring Boot toujours autant plébiscité par la communauté Java

Spring MVC et plus récemment Spring Boot ont grandement contribué à la démocratisation de Java.
Les applications REST basées sur Spring MVC prenant en charge le format JSON dans Tomcat demandaient la mise en place d'une dizaine de dépendances avant de compléter la configuration.
Le choix de Spring Boot s'est donc imposé en facilitant le quotidien des développeurs en supprimant certaines de ces difficultés grâce à :

- Starters
- L'autoconfiguration
- BOM (Bill Of Materials) de dépendances
- Applications "Production-Ready" via Actuator

Spring Boot, bonjour les architectures microservices !

Une accélération du démantèlement des monolithes
Spring Boot coïncide également avec le lancement à grande échelle de projets de démantèlement de monolithes. Le duo Java 8 / Spring Boot devient le choix plébiscité pour initier ces changements de paradigmes d'architecture avec la promesse d'obtenir des systèmes plus ouverts (facilitant l'accès par des API), modulaires et évolutifs.

L'avènement de Spring Cloud et la facilitation des déploiements "Cloud Ready"

Avec Spring Cloud, Spring contribue à la démocratisation du microservice grâce à la gestion de la configuration, la découverte de services, passerelles de service, au pattern circuit breaker, fallback, etc.
Nous constatons aussi que ces facilitations s'accompagnent, à notre sens, de compétences Ops qui deviennent de plus en plus importantes dans le quotidien d'un développeur pour minimiser les problèmes de déploiements, notamment en production.
D'ailleurs, il est judicieux de se demander à quoi bon disposer d'une profusion de microservices si la gestion de la séparation des responsabilités, des déploiements, de l'interconnexion de services, etc., est douloureuse ?

Les frameworks, des abstractions nécessaires pas toujours bénéfiques

Spring MVC, Spring Boot, Spring Cloud, etc. sont des frameworks qui apportent une réelle valeur indéniable.
Cependant, la couche d'abstraction apportée peut éloigner nos développeurs des concepts fondamentaux et les amener à appliquer le "FDD"*, sans prise de recul, se contentant ainsi de laisser faire la "magie" de Spring.
C'est pourquoi nous préconisons d'investir judicieusement son temps d'apprentissage et de pratique dans des concepts fondamentaux agnostiques aux langages, aux frameworks ou aux librairies : injection de dépendances, inversion de contrôle, immutabilité, effets de bord, idempotence, etc.
Ces concepts fondamentaux restent un investissement sûr, transposable et durable quelles que soient les évolutions de frameworks à venir.

**FDD, Frameworks Driven Development, concept humoristique que nous utilisons chez CodeWorks pour décrire le comportement d'un développeur ou d'une équipe qui sait utiliser, voire devient spécialiste d'un framework, mais n'a pas ou peu de connaissances ni prise de recul dans la conception système ou l'architecture logicielle sans ledit framework.*

Pourquoi vous ne devez pas investir TOUT votre temps dans l'apprentissage des frameworks ?

Les technologies évoluent très vite et cette fuite en avant est déjà perdue d'avance pour quiconque souhaiterait suivre cette cadence infernale. La technologie évoluera toujours plus vite que votre capacité d'apprentissage. Il est alors plus judicieux de prendre un peu de recul, capitaliser et investir votre temps sur des compétences "transférables".
Ces compétences vous permettront d'appréhender plus facilement les évolutions technologiques à venir en vous

appuyant sur des connaissances qui perdurent depuis plusieurs dizaines d'années maintenant. Pour cela, concentrez-vous sur l'apprentissage et la mise en pratique des principes fondamentaux. Un mentor ou une communauté de pratique, voire une participation au développement d'un projet open source peuvent également vous être d'une grande aide. Le contexte Covid a aussi favorisé la mise en place d'événements en visioconférence en France, en Europe, voire dans le monde entier.

Voici quelques conseils que nous pouvons vous apporter pour accompagner votre parcours de développement de compétences :

- Lancez-vous dans l'apprentissage des bonnes pratiques de design de code avant de vous lancer dans un langage
- Lancez-vous dans l'apprentissage des architectures évolutives avant de vous lancer dans l'apprentissage des microservices, par exemple
- Lancez-vous dans l'apprentissage des concepts de livraison continue avant de vous intéresser à une technologie de conteneurisation, type Docker, par exemple
- Lancez-vous dans l'apprentissage du fonctionnement du Web, HTTP et REST avant de vous intéresser à un framework ou bibliothèque Front, type Angular, React ou Vue, par exemple

Et si vous avez fait le chemin inverse, rien n'est perdu pour commencer à appliquer ces conseils dès maintenant.

De même, voici quelques conseils sur l'apprentissage des technologies :

- Identifiez une technologie qui peut vous apporter de nouvelles compétences ou un regard critique et remettre en question vos compétences déjà acquises
- Le temps est votre meilleur conseiller, apprenez à être patients avec vous-même
- Les frameworks, bibliothèques et outils vont et viennent, les principes et concepts fondamentaux restent
- Modulez votre investissement entre 70% sur les fondamentaux et 30% sur des frameworks, bibliothèques et outils.

Comment contribuer à des évolutions de culture technique ?

Les offres de poste de développeurs s'apparentent de plus en plus aux compétences recherchées pour assurer le fonctionnement optimal d'un service IT complet.

L'écosystème Java confirme cette vision puisqu'il imbrique, à lui seul, une quantité parfois déconcertante de technologies. Cela conforte, notre idée qu'une évolution des logiciels de pensée est nécessaire.

Commencer par écrire un code simple et compréhensible par tout le monde

Qu'il soit écrit en Java ou dans un autre langage, un code compréhensible par la machine est facile à écrire, mais un code lisible et facilement compréhensible par d'autres développeurs l'est beaucoup moins. La base de code se doit d'être facilement compréhensible par d'autres afin qu'elle ne se transforme pas en dette technique comportant des bugs douloureux à corriger, de la complexité non justifiée qui aboutirait à un sentiment d'incertitude et un manque de confiance. C'est la connaissance et la mise en pratique des principes d'architecture, de design, des méthodologies de tests, de découverte et l'appétence pour les connaissances métier qui permettent l'élaboration d'un code lisible et compréhensible.

Définir un haut niveau d'exigence qualité

Au sein de nombreux projets informatiques, les rapports de force entre acteurs métier et équipes de développement existent, parfois au détriment de la qualité du produit final. Ils ne permettent pas d'instaurer un climat serein et apaisé nécessaire à la production d'un code de qualité, fiable, qui répondra aux exigences et aux enjeux business partagés par toute l'équipe. En effet, il est parfois difficile de faire comprendre aux parties prenantes métier d'un projet IT que la satisfaction du client et la qualité du produit final dépendent en grande partie des pratiques mises en œuvre pour concevoir ce code. Il est donc de notre devoir d'accompagner ces évolutions de culture technique et méthodologique.

S'inscrire dans un dépassement de fonction

Nous sommes également convaincus que le développeur doit s'inscrire dans un dépassement de fonction. Savoir s'extirper de la posture de "doer" pour accompagner, comme pourrait le faire un coach, la mise en œuvre de bonnes pratiques pour concevoir un code :

- Facile à lire
- Facile à comprendre
- Facile à tester
- Facile à débugger

Développer une culture de la responsabilité et de l'opérabilité

Dans le prolongement de la culture DevOps qui consiste à rapprocher les équipes de développement et systèmes, les compétences Ops deviennent de plus en plus précieuses pour développer une vision plus responsable : "You build it, You run it".

La montée en puissance d'applications "Cloud Ready" et l'émergence de nouveaux paradigmes, type Serverless, promus par les Big 3 que sont AWS, Azure et GCP, ne font que confirmer cette tendance.

S'adapter à l'évolution de culture et pratiques agiles

Scrum, par exemple, repose sur trois piliers : transparence, inspection et adaptation.

La promesse de l'agilité consiste à offrir de la visibilité sur le travail d'une équipe développement pour :

- éviter les effets tunnels
- permettre des retours itératifs et constructifs
- définir des mises en production plus régulières
- fluidifier les interactions

Malgré toutes ces belles perspectives, certains projets s'inscrivent aujourd'hui dans une approche "Scrum Zombie". À ce titre, Christiaan Verwijs, Johannes Schartau et Barry Overeem apportent des retours d'expérience intéressants : https://medium.com/the-liberators/zombie-scrum/home

Le "Scrum Zombie" est quelque chose qui ressemble à Scrum et où l'on peut voir implémenter les rôles, les événe-

ments artefacts du framework, mais sans aucune collaboration entre les parties prenantes et où personne ne semble se soucier d'améliorer quoi que ce soit.
Il se produit souvent dans des organisations qui se concentrent sur l'optimisation d'autre chose que l'agilité réelle et ont rarement une réponse claire sur ce qui fait la valeur de leur produit. Comme les zombies titubant sans savoir où ils vont, ces équipes travaillent très dur, mais sans arriver à un résultat précis ou efficace. Les équipes se cachent alors derrière la complexité du produit, les limites technologiques ou le manque de connaissances métier. Enfin, dans ce type de contexte, les organisations ne créent pas d'espace protégé pour assumer sereinement les échecs. A contrario, les équipes s'attachent à développer des stratégies défensives pour éviter cette incertitude.

Accueillir et assumer une posture d'apprenant

Tout développeur doit être conscient qu'il n'écrit pas du code que pour lui, mais un code universel, simple, performant, accessible et compréhensible par tous. Pour cela, il doit être en capacité d'éliminer toute complexité inutile.
Pour ce faire, nous recommandons quelques ouvrages qui peuvent accompagner votre cheminement vers ce noble objectif et vous permettre d'investir judicieusement votre temps d'apprentissage :

- The Pragmatic Programmer | Auteurs : David Thomas and Andrew Hunt
- Clean Code | Auteur : Robert C. Martin
- Domain-Driven Design | Auteur : Eric Evans
- Continuous Delivery | Auteur : Jez Humble
- The Software Craftsman | Auteur : Sandro Mancuso
- Soft Skills : The Software Developer's Life Manual | Auteur : John Sonmez

Attention, nous attirons votre attention sur le fait que ces ouvrages vous permettront d'explorer de nouveaux horizons ou d'approfondir vos connaissances déjà acquises, mais nous vous recommandons vivement de pratiquer par du code, du code et encore du code !

De l'importance des compétences comportementales

Nous constatons depuis plusieurs années un attrait grandissant des équipes projets tant pour les compétences comportementales que techniques.
Nous ne pouvons que nous en réjouir, car nous sommes convaincus que le prisme technique seul ne peut suffire pour contribuer efficacement, comprendre les enjeux métier et interagir convenablement au sein d'un projet.

Savoir dire "je ne sais pas"

Cela en dit beaucoup sur votre honnêteté intellectuelle et facilitera l'apprentissage et l'évolution. Nous le valorisons comme un terreau fertile pour accepter le lâcher-prise notamment dans les phases d'accompagnement (mentoring, pair ou code review, par exemple) et la prise de conscience de ses propres limites.

Faire preuve de curiosité

Sans elle, pas d'évolution possible. Assurez-vous donc de ne jamais en manquer ou faites en sorte de la stimuler sans arrêt pour maintenir un intérêt pour votre projet.

Du pragmatisme pour combattre l'over-engineering

Toute complexité inutile a un coût important pour la viabilité d'un produit et c'est cette sensibilité qui permettra de produire une base de code fiable qui ne virera pas rapidement à la dette technique impossible à maintenir.

Le mentor apprenant

Le "mentor/mentoré.e" sait qu'il ne peut pas tout savoir. De ce fait, sur un sujet X ou Y tantôt il aura les connaissances qui permettront d'aider les autres, tantôt il bénéficiera et apprendra de celles des autres.

Des choix de carrière qui évoluent : la portée du produit prend peu à peu le pas sur l'intérêt pour les technologies

Depuis quelques d'années, nous constatons également une réelle évolution des critères qui motivent l'engagement des développeurs pour tel ou tel projet. Les critères technologiques avaient une place importante qui encourageait les offres de poste type inventaire à la Prévert des technologies présentes sur les projets, même les plus mineures.
Ces critères ont évolué pour laisser plus de place à :

- des projets qui ont un impact positif sur l'utilisateur final
- un équilibre vie professionnelle / vie personnelle en disposant de temps libre, par exemple
- un développement de compétences financé et assumé par l'entreprise
- un collectif riche, divers et compétent pour poursuivre son évolution en qualité de "mentor/mentoré"

François Tonic

Il était une fois Java

La première version de Java sort le 23 mai 1995. Pourquoi Java ? Comment naît le projet chez Sun Microsystems ? La genèse remonte à 1991 avec le projet oak. Retour sur 26 ans d'évolutions.

En 1991, Naughton, Sheridan et Gosling forment une petite équipe interne chez Sun. Elle regroupe 13 personnes sous le nom de Green Team. Ces ingénieurs et développeurs doivent penser et concevoir le futur de l'informatique. Une des technologies qu'ils voient arriver est la convergence des terminaux et des ordinateurs et l'interaction entre tous les équipements : le fameux hub numérique que Steve Jobs réalisera dix ans plus tard. Le premier prototype est une set-top TV : une sorte d'écran autonome, tactile, avec un système media. Il faut 18 mois pour développer l'OS et les interfaces. Dès ce prototype, l'équipe introduit une petite mascotte : Duke. Eh oui, Duke apparaît avant même la création de Java. Le prototype est le *7 et tout naturellement, l'étape suivante est la téléphonie. Pour créer les applications, les interfaces, un langage dédié est nécessaire. Gosling va alors jeter les bases du langage Oak. Ce langage s'inspire directement du C, reposant sur une machine virtuelle.

Le nom de Oak fut choisi par Gasling, car il aimait un chêne (oak en anglais) près de son bureau… Le nom Java arrivera plus tard alors que le nom Oak était acté, mais un problème de marques apparut. Les équipes hésitaient et plusieurs noms furent proposés : Java, Silk, Ruby. Finalement, Java fut choisi, en référence au café. Initialement, le projet fut appelé en interne Greentalk.

Les bases de Java, et donc de Oak étaient, selon Gosling :

- Un langage le plus simple possible
- Ensemble d'API pour faciliter le développement
- Une abstraction du langage : indépendance matérielle et système, 1 code pour être exécuté partout
- Intégration dans les navigateurs web (qui viennent d'arriver sur le marché)
- Exécution de plusieurs threads / tâches simultanément

L'équipe fondatrice de oak – java. 1994-95.

Des évolutions sur le nom, les sorties, etc.

Java n'a jamais été un long fleuve tranquille : création chez Sun, rachat de Sun par Oracle, réorganisation des équipes et des priorités, Java Oracle et projet Open Source, etc. Il n'y a que l'embarras du choix !

La cadence de mise à jour a souvent été critiquée. Il faut dire que les nouvelles versions sortent de manière aléatoire et souvent dans la douleur. Entre Java 6 et Java 7, 5 ans s'étaient écoulés. Une éternité ! Entre Java 8 et 9, nous avions eu 3 ans. Les dérives de développement de ces ver-

Plusieurs JDK officielles

Java

OpenJDK	Oracle JDK
Version Open Source, gratuite, sans support officiel Oracle. Licence GPL	Version officielle d'Oracle, support de l'éditeur (LTS), les mises à jour sont gratuites. Licence Oracle

Les deux branches partagent le même code et les mêmes fonctionnalités. Comme nous l'avions dit en 2018, les deux JDK sont identiques à 99,9 %. Il s'agit d'avoir une synchronisation entre les deux branches dans les rajouts, les modifications, les retraits de fonctions / modules. Oracle JDK est donc une JDK payante sur le desktop et le serveur. En dehors d'OpenJDK et d'Oracle JDK, il existe plusieurs autres JDK. Ces Java se basent sur la version open source donc OpenJDK. Ce sont des distributions repackagées et optimisées. Ces distributions doivent passer avec succès le TCK. Ce sont des batteries de tests permettant de vérifier la compatibilité de la JDK.

Les différentes JDK suivent les sorties officielles d'OpenJDK.

Azul	Amazon web Services	Red Hat
Azul Zulu	Corretto	OpenJDK
Distributions commerciales avec support moyen et long terme.	OpenJDK version AWS. Support LTS.	Distribution de Red Hat pour Windows et Red Hat Linux.

En 2020, Microsoft avait annoncé son soutien pour porter et développer une OpenJDK sur les architectures Aarch64 (ARM 64) pour Windows ARM et Apple Silicon. Les développeurs Microsoft travaillent avec Red Hat. Ce travail et le rachat de jClarity avaient aussi pour objectif d'optimiser la JDK sur Azure.

sions ont tourné au cauchemar : retard sur retard, report de projets. Java 9 est sorti dans la douleur. Cette version est une rupture et Oracle va imposer une nouvelle roadmap : une version tous les 6 mois, en mars et en septembre. L'objectif est d'arrêter les mises à jour incluant x nouveautés et améliorations. L'évolution sera lissée : Oracle souhaite éviter des changements trop brutaux et une trop longue attente. Ce modèle itératif s'est imposé à d'autres langages.

Oracle veut canaliser l'effort :

- Recentrer le développement sur quelques éléments clés
- Raccourcir les délais de déploiement des versions, ne plus faire de big bang à chaque version
- Définir clairement les licences commerciales et open source
- Modèle de tarification simplifiée

Oracle a annoncé deux types de versions : un Java LTS et un Java non LTS. LTS signifie support long terme. Cela signifie que cette version sera supportée 8 ans par Oracle. La version LTS sort tous les 3 ans. Il s'agit d'une version spécifique à Oracle.

Depuis 1995, plusieurs noms ont été utilisés pour désigner la même chose : Java édition standard ou plus simplement la JDK, OpenJDK. Sun puis Oracle a utilisé JDK, J2SE, Java SE, Java SE et OpenJDK.

Java Professional Edition = Java Enterprise = J2EE = Jakarta EE

À côté de Java SE, Sun développe une plateforme dédiée aux entreprises et aux serveurs : Java Enterprise Edition. La première version sort officiellement en décembre 1999. J2EE évolue tous les 3-4 ans. La plateforme commence à apparaître au printemps 1998 sous le nom de Java Professional Edition.

Java EE a une histoire chaotique surtout depuis le rachat de Sun par Oracle. Le projet a été confié à la fondation Eclipse qui prend en charge le support et le développement de la plateforme. La première version est apparue en septembre 2019 : Jakarta EE 8. Elle reprend le périmètre de Java EE 8. Le changement de nom vient aussi du fait que Java reste une marque d'Oracle. Les changements vont jusqu'aux classes internes : java.servlet devient jakarta.servlet.

Jakarta EE doit avoir une cadence de mise à jour annuelle pour éviter les dérives de développements de la JDK. La v8 était surtout une version de transition après le retrait d'Oracle. La v9 était la première véritable version de la fondation. Les équipes veulent un alignement plus serré avec les évolutions d'Open JDK. / Java SE. Cela doit éviter de développer des API et des modules trop différents vis-à-vis du cœur de Java. L'objectif est aussi de mieux intégrer les standards du marché. Pourquoi réinventer la roue ?

Jakarta 10 devrait apporter plusieurs évolutions : Jakarta NoSQL, Eclipse JNoSQL, Jakarta MVC. La v10 sera la 1ere véritable version de la fondation.

Janvier – mai 1995 : 5 événements cruciaux

Event 1

Depuis plusieurs mois, l'équipe Oak – Java travaille à stabiliser le langage et l'environnement. Gasling lance le développement d'un navigateur web avec le nouveau langage : WebRunner. Il s'agit d'un clone de Mosaic. La première démo publique se fait dans les premières semaines de 1995 durant la conférence Hollywood meets Silicon Valley. Gosling et Cage (un des responsables techniques de Sun) dévoilent le langage. L'audience s'intéresse peu à la présentation, mais quand des animations et des manipulations à la souris se font en live directement sur WebRunner, l'audience commence à s'y intéresser. Gasling avait passé des dizaines d'heures, avant la session, à corriger et à patcher WebRunner et le code.

Event 2

En février, WebRunner est prêt à être distribué officiellement.

Event 3

En mars 1995, le langage est packagé pour être lui aussi distribué en dehors de Sun. Cette version est encore très loin d'être la 1.0 de 1996, il s'agit de la 1.0a2, une version alpha. Le code source fut ouvert sur le web pour attirer les développeurs comme l'avait raconté Lisa Friendly, une des développeuses de l'équipe Java. En quelques heures, ce sont des dizaines de téléchargement. En quelques mois, des milliers de développeurs téléchargent les sources. Les questions techniques, par mail, se multiplient : 10, 20 puis des milliers par jour. Mais l'équipe Java est réduite et elle doit essayer de répondre aux questions tout en continuant à stabiliser le langage. Une équipe support fut créée à plein temps : avec une personne…

Event 4

Le site officiel java.sun.com est ouvert (mars – avril 1995).

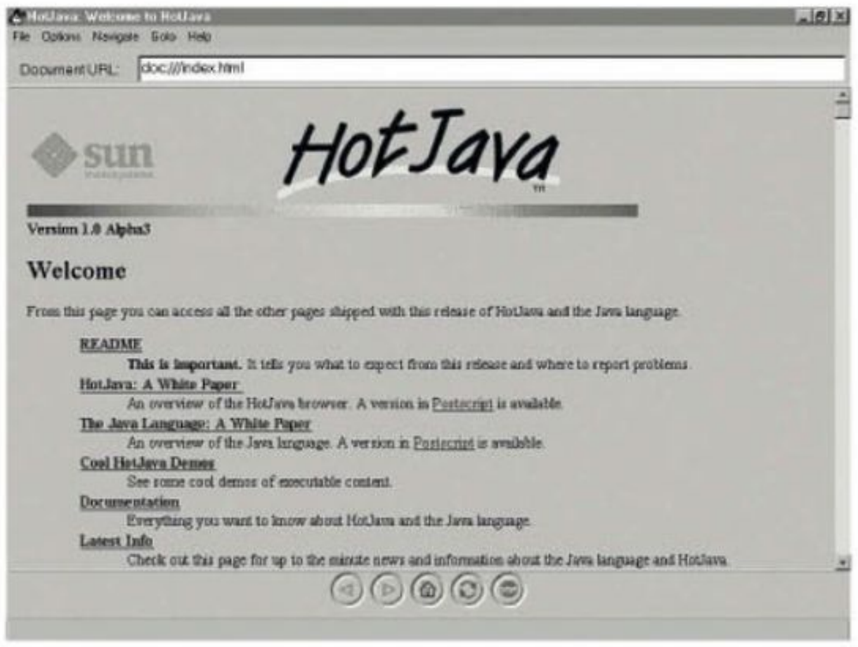
Event 5

Si Sun est avant tout un constructeur de machines, de processeurs et d'OS, Java commence à agiter l'ensemble de la société et la SunWorld de mai 95 est l'officialisation de Java. En effet, le 23 mai, le langage Java est officiellement intronisé sur la scène de la SunWorld. Et la diffusion de la version alpha ne va pas cesser de croître.

(Event 6)

Rajoutons un dernier élément important. Si WebRunner / HotJava ne connaît pas un succès fou, il permet d'exécuter Java sur le web via les applets. Netscape comprend vite l'intérêt de Java notamment suite à la présentation de mai 95. L'éditeur va prendre une licence du langage pour l'intégrer dans son navigateur. Les applets sont disponibles dans Netscape Navigator avant fin 95.

*Le projet *7 ne sera jamais produit mais sans lui, Java n'aurait jamais été développé.*



Le navigateur WebRunner / HotJava

Développer en Java en 2021

En mars 2021, le JDK 16 est sorti. Nous allons voir quelles sont les nouveautés que nous allons pouvoir profiter en tant que développeurs. Il est à noter que dans le dernier rapport 2021 de la société Snyk sur écosystème JVM [1], nous pouvons voir que deux développeurs sur trois utilisent la version 8 du JDK en production et seulement un développeur sur quatre utilise un JDK 11. Nous allons donc en profiter pour voir ou revoir rapidement les nouvelles fonctionnalités depuis le JDK 11.

Lilian BENOIT

J'ai plus de 21 ans d'expérience. JugLeader du BordeauxJUG, Tech Leader à IMC, une ESN Bordelaise. Je suis passionné par l'informatique depuis bien plus longtemps. J'aime travailler sur la plateforme Java : JVM et JakartaEE, notamment sur ma distribution préférée : Debian. Vous pouvez me contacter sur @Lilian_Benoit ou venir à une soirée du JUG.

Mais avant, un point de friction ou d'inquiétude est les cadences de sortie des JDK. En effet, par rapport au constat précédent sur l'utilisation des JDK 8 et 11, nous pouvons nous dire que nous sommes déjà à la version 16.

Nouvelle cadence et LTS

La sortie du JDK 9 a été particulièrement difficile, car elle a été repoussée à maintes reprises, notamment à cause de l'ajout et le support de la modularité au sein de la JVM et pour les applications Java. Cela a apporté aussi un frein sur l'adoption de cette version avec la croyance que les modules étaient obligatoires. Cela n'est pas le cas. Le mode « classpath » fonctionne et fonctionne toujours même avec le JDK 16. De plus, Oracle avait aussi annoncé un changement de cadence avec une sortie du JDK : deux fois par an (en mars et en septembre). De plus, ils ont introduit la notion de version LTS et non-LTS.

Cadence

L'objectif est de rendre prévisibles les sorties du JDK. Le corollaire est que si une fonctionnalité n'est pas prête, elle est sortie du périmètre de la version. Un exemple a été fait avec la fonctionnalité « Raw String Literals » prévue initialement pour le JDK 12. Les retours de la communauté n'étaient pas bons, il a été décidé de la retirer purement et simplement. De plus, la fonctionnalité a été entièrement revue dans le JDK 13, puis affinée dans le JDK 14 à travers la fonctionnalité des « Text Blocks ». Nous aurons l'occasion de revenir dessus plus loin dans cet article. Nous pouvons voir la cadence de sortie du JDK 8 au JDK 16 avec la **Figure 1.**

LTS

Les versions LTS sont des versions avec un support étendu. C'est-à-dire qu'il existe des versions intermédiaires contenant des correctifs. Par exemple, le JDK 11 a eu droit le 19 janvier 2021 à une nouvelle version intermédiaire 11.0.10 incluant 118 correctifs.

Les versions non-LTS sont maintenus tant qu'il n'y a pas de nouvelle version. Par exemple, avec la sortie du JDK 16, le JDK 15 n'est plus maintenu. Durant ces six mois, il a bénéficié de deux versions intermédiaires.

L'avantage de ces versions non-LTS est d'avoir la possibilité de voir arriver et même d'utiliser ces nouvelles fonctionnalités plus tôt qu'avec un cycle LTS. En comparaison, le JDK 9 avait plus de 80 nouvelles fonctionnalités. C'est difficile d'aborder toutes les nouveautés. Avec une version tous les 6 mois, c'est l'occasion d'avoir techniquement moins de fonctionnalités, mais c'est surtout plus facile à appréhender les nouveautés comme nous l'avons le voir plus bas avec le JDK 16.

Le choix de son rythme

À travers cette cadence et ces versions LTS, nous avons en réalité le choix. Pour cela, Mark Reinhold (Architecte en Chef, Plateforme Java chez Oracle) l'illustrait très bien en 2019 au FOSDEM [2]. Nous avons le choix entre deux chemins (ou deux pilules, petit clic d'œil à Matrix) :

- Choisir un chemin plus conservateur avec les versions LTS (pilule bleue)
- Choisir un chemin plus agressif sur les nouvelles fonctionnalités (pilule rouge)

En prenant la pilule bleue, c'est à dire en choisissant uniquement les versions LTS, nous avons une version tous les 3 ans, comme avant (cf. Figure 1). Cela est adapté aux entreprises qui possèdent des socles techniques assez lourds et où les évolutions sont délicates et prennent du temps au sein de leur infrastructure.

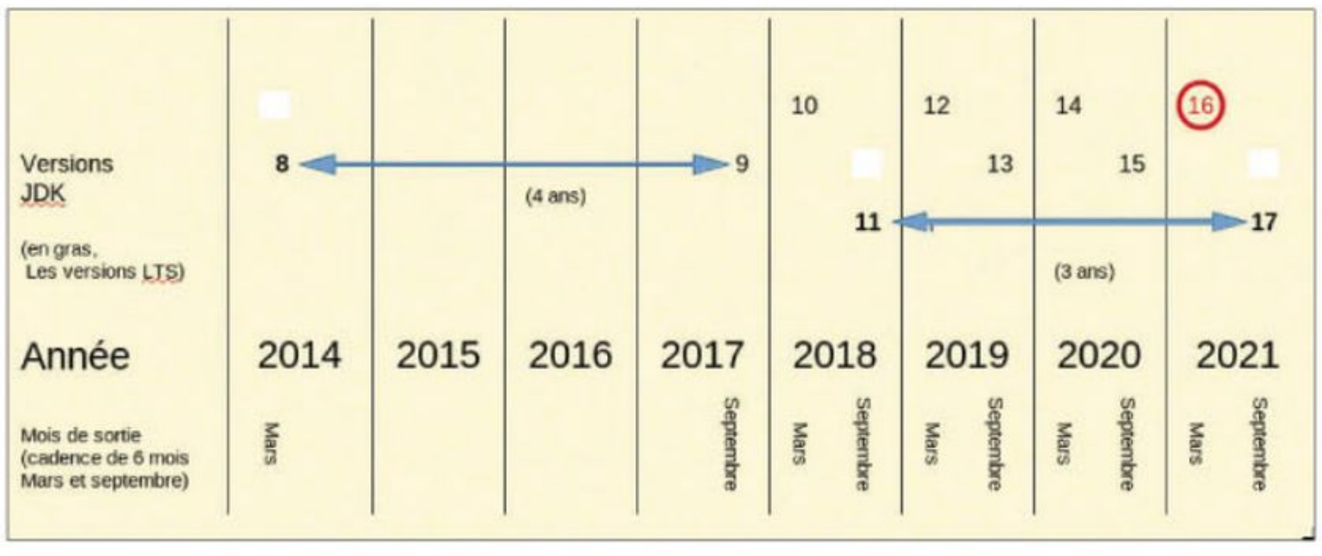
D'un autre côté, la pilule rouge est une voie où l'infrastructure est assez souple pour suivre rapidement les évolutions. En ce sens, les technologies avec les conteneurs (Docker, Kubernetes) permettent de profiter plus rapidement des dernières versions, car il suffit de modifier l'image de base.

Aperçu (« Preview »)

Le principe de l'aperçu ou de préversion est d'avoir une nouvelle fonctionnalité au niveau langage, JVM ou API pleinement spécifiée, pleinement développée, mais pas encore permanente.

L'objectif est ainsi de mettre en place complètement cette fonctionnalité (cela n'est pas une phase d'incubation et de bêta). La communauté Java peut ainsi réaliser des retours dans des cas d'usages réels (projets, librairies, cadriciel). Le

Figure 1 – *Cadence de sortie du JDK 8 au JDK 16*



but est ensuite d'intégrer cette fonctionnalité de manière permanente dans une future version.
Parmi les fonctionnalités que nous allons étudier, un certain nombre a bénéficié de ce mode. Cela a permis d'enrichir la fonctionnalité et de prendre en compte les retours des développeurs. Par exemple, les classes scellées ont été introduites dans le JDK 15 en aperçu. Suite aux retours, la fonctionnalité est présente en deuxième aperçu en apportant un certain nombre de raffinements.
Cela démontre le souhait d'Oracle et du projet OpenJDK de manière générale de collaborer avec la communauté Java. Inversement, la communauté Java est impliquée sur la réalisation des nouvelles fonctionnalités au sein de la plateforme.
Les fonctionnalités en aperçu sont accessibles uniquement si l'option –enable-preview est utilisée lors de la compilation et lors de l'exécution.

```
javac Foo.java                                  // Pas d'aperçu activé
javac --release 16 --enable-preview Foo.java    // Tous aperçu du JDK 16 activés
javac --release 15 --enable-preview Foo.java    // PAS AUTORISE
```

La contrainte est effectivement que les aperçus sont spécifiques à une version du JDK donnée. Le JDK 16 ne peut pas faire fonctionner des aperçus d'une autre version.

```
java Foo                            // Pas d'aperçu activé
java --enable-preview Foo           // tout aperçu du JDK 16 activé
java --enable-preview -jar App.jar  // tout aperçu du JDK 16 activé
java --enable-preview -m App        // tout aperçu du JDK 16 activé
```

Rappel sur les principales nouveautés de JDK 11 à 15

JDK 14 Switch Expressions

L'objectif est d'étendre l'instruction switch par deux actions. La première est que l'instruction switch permet de retourner une valeur dont le prolongement est une expression. La seconde est de supporter une nouvelle notation flèche (->) ainsi que la notation existant avec le deux-points (:).
La variable jour étant une énumération, un exemple classique est le code suivant :

```
int nbLettres;
switch (jour) {
   case LUNDI:
   case MARDI:
   case JEUDI:
      nbLettres = 5;
      break;
   case SAMEDI:
      nbLettres = 6;
      break;
   case MERCREDI:
   case VENDREDI:
   case DIMANCHE:
      nbLettres = 8;
      break;
   default:
      throw new IllegalStateException(" ?! : " + jour);
}
```

Pour rappel, avec la notation classique, il ne faut pas oublier les instructions break pour sortir au bon endroit.
Avec la nouvelle notation, c'est à dire la notation flèche (->), cela devient tout simplement :

```
int nbLettres = switch (jour) {
   case LUNDI, MARDI, JEUDI        -> 5;
   case SAMEDI                     -> 6;
   case MERCREDI, VENDREDI, DIMANCHE -> 8;
};
```

Ce qui est intéressant, c'est l'exhaustivité. En effet, la variable jour étant une énumération. Le compilateur peut savoir s'il faut ou non la clause default.
De ce fait, s'il manque un cas, le compilateur le signalera. De même, si l'énumération évolue avec une nouvelle valeur possible. Le compilateur viendra nous aider en indiquant toutes les instructions switch à regarder pour savoir comment il faut traiter la nouvelle valeur.
Avec la notation flèche, il reste possible de définir un bloc de code. Voici un exemple de code d'illustration :

```
int j = switch (jour) {
   case LUNDI -> 0;
   case MARDI -> 1;
   default    -> {
      int k = jour.toString().length();
      int result = f(k);
      yield result;
   }
};
```

Nous pouvons remarquer l'utilisation du mot clé yield pour indiquer la valeur de l'expression qui sera affectée à la variable j.

JDK 14 Helpful NullPointerExceptions

Cette fonctionnalité n'introduit pas de nouveau mot clé ou de nouvelle API. En revanche, cela va sûrement vous aider en tant que le développeur dans le diagnostic des problèmes.
Qui n'a jamais eu un problème d'exécution avec ce message lapidaire

```
java.lang.NullPointerException: null
```

Alors, effectivement, nous avons le numéro de ligne, mais cela n'est pas forcément si simple. Prenons un exemple concret où, dans notre programme, nous avons la méthode suivante :

```
public boolean isVide(StringHolder holder) {
    return (holder.getChaine().length() > 0);
}
```

Une erreur NullPointerException sur la ligne de l'instruction return peut nous laisser perplexe. Cela n'est pas lié à la complexité, mais sur le fait de déterminer quelle est la variable nulle. Est-ce la variable holder ? Holder.getChaine() ?
Et c'est là qu'intervient la nouvelle fonctionnalité. Avec le JDK 16, nous obtenons le message suivant :

```
java.lang.NullPointerException: Cannot invoke "String.length()" because
the return value of "fr.lbenoit.....StringHolder.getChaine()" is null
```

Et là, c'est gagné ! Nous savons la variable qui est nulle (ici c'est StringHolder.getChaine()) et l'action qui ne peut pas

être réalisée : invoquer la méthode String.length(). Cerise sur la gâteau, c'est le comportement par défaut à partir du JDK 15, donc vous n'avez rien à faire pour en bénéficier.

JDK 15 Text Blocks

L'objectif est de simplifier l'écriture d'une chaîne de caractères qui tient sur plusieurs lignes. Cela se produit notamment pour les langages non-java. Ci-dessous, un exemple de requête SQL que nous affectons à la variable query.

```
String query = "SELECT \"EMP_ID\", \"LAST_NAME\" FROM \"EMPLOYEE
_TB\"\n"+
        "WHERE \"CITY\" = 'INDIANAPOLIS'\n" +
        "ORDER BY \"EMP_ID\", \"LAST_NAME\";\n";
```

Les bases

Les délimiteurs choisis sont trois guillemets : """
Prenons un cas simple :

```
"""
ligne 1
ligne 2
ligne 3
"""
```

Cela correspond au code suivant :

```
"ligne 1\nligne 2\nligne 3\n"
```

Avant, nous aurions vu fréquemment les lignes suivantes :

```
"ligne 1\n" +
"ligne 2\n" +
"ligne 3\n"
```

Si on revient sur l'exemple de la requête SQL, le code devient aussi beaucoup plus clair sans caractère et échappement parasite :

```
String query = """
        SELECT "EMP_ID", "LAST_NAME" FROM "EMPLOYEE_TB"
        WHERE "CITY" = 'INDIANAPOLIS'
        ORDER BY "EMP_ID", "LAST_NAME";
        """;
```

Il est à noter que l'emplacement de la fermeture de la chaîne n'est pas anodin. Il permet de différencier les espaces significatifs et les espaces accidentels.
Nous pouvons avoir le code suivant. Nous utilisons le caractère point (.) pour représenter les espaces accidentels :

```
String html = """
..............<html>...
..............  <body>
..............    <p>Hello, world</p>....
..............  </body>.
..............</html>...
..............""";
```

Maintenant, en prenant, le caractère pipe (|) pour mieux représenter la valeur finale. Nous avons la valeur suivante :

```
|<html>|
|  <body>|
|    <p>Hello, world</p>|
|  </body>|
|</html>|
```

En avançant les guillemets, nous obtenons le code suivant :

```
String html = """
.......  <html>
.......    <body>
.......      <p>Hello, world</p>
.......    </body>
.......  </html>
.......""";
```

Cela donne le résultat suivant :

```
|  <html>
|    <body>
|      <p>Hello, world</p>
|    </body>
|  </html>
```

Nouvelles fonctionnalités JDK 16

JDK 16 Pattern Matching for instanceof

Avant de regarder la fonctionnalité, nous allons nous pencher sur la notion de Filtrage par motif (Pattern Matching). Nous pouvons retrouver la définition sur Wikipédia [3]: « Le filtrage par motif permet de vérifier si l'objet du filtrage possède une structure donnée, s'il s'y trouve telle ou telle sous-structure spécifique et laquelle, pour y retrouver des parties par l'organisation de leur contenu, et/ou éventuellement pour substituer quelque chose d'autre aux motifs reconnus »
La définition n'est pas forcément très limpide. Heureusement, le billet de Brian Goetz et Gavin Bierman de septembre 2018 [4], est très instructif, car cela permet de voir les idées fondamentales de ce sujet, des exemples d'illustration en Java (pour rappel en 2018, c'est de la projection, car ces exemples ne compilent pas à l'époque). En revanche, cela permet de voir l'intérêt d'avoir ce mécanisme au sein du langage Java. Prenons le code suivant que tout développeur Java aura déjà écrit une multitude de fois :

```
if (obj instanceof String) {
   String s = (String) obj;
   // utilisation de la variable s
   int lg = s.lenght() ;
   // utilisation de la variable lg
}
```

Sur ce code, nous avons en réalité trois étapes :

- Test pour s'assurer que la variable obj est une instance de String
- Une conversion de la variable obj en String
- Extraction d'information (« déconstruction ») de la classe String (Récupération de la longueur de la chaîne)

Dans notre exemple, le motif est instanceof. L'objectif est de définir une variable associée au motif. Pour cela, nous plaçons le nom de la variable après le type.

```
if (obj instanceof String s) {
   // utilisation directe de la variable s
   int taille = s.lenght() ;
} else {
```

```
    // la variable s n'est pas connue dans ce bloc
}
```

Dans l'exemple ci-dessus, la variable s est visible et disponible dans le premier bloc (if), mais elle n'existe pas dans le second bloc (else). Si nous essayons de l'utiliser quand même dans le bloc (else), le compilateur nous dira que la variable n'est pas connue. La portée de variable est aussi vrai au niveau de la condition. Un exemple va permettre d'éclaircir tout de suite. Nous avons le code suivant :

```
if (obj instanceof String) {
   String s = (String)obj ;
   if (s.lenght() > 5) {
      System.out.println(« Valeur : »+s.toUppercase()) ;
   }
}
```

Cela peut être remplacé tout simplement par le code suivant :

```
if (obj instanceof String s && s.lenght() > 5) {
      System.out.println(« Valeur : »+s.toUppercase()) ;
}
```

Un dernier exemple pour illustrer et montrer que cela n'est pas qu'une fonctionnalité gadget. Voici le code de la méthode de equals() classique pour une classe Point ayant deux attributs : x et y.

```
public boolean equals(Object o) {
   if (!(o instanceof Point))
      return false;
   Point param = (Point) o;
   return x == param.x
      && y == param.y;
}
```

Maintenant, nous pouvons écrire tout simplement le code suivant :

```
public boolean equals(Object o) {
   return (o instanceof Point param)
      && x == param.x
      && y == param.y;
}
```

JDK 16 Records

Le principe est d'ajouter un nouveau mot clé record au langage Java. Cela correspond à des classes immutables dont la structure est connue. Nous pouvons les considérer comme des tuples nommés. Prenons un exemple pour illustrer le propos. Si en java, nous souhaitons représenter une position GPS, nous irons rapidement à définir la classe suivante :

```
class Position {
   private float longitude
   private float latitude
}
```

Cependant, pour bien faire, nous devrons ajouter des accesseurs, un constructeur, la méthode hashcode(), equals() et toString(). Nous sommes aidés par nos IDE pour réaliser les opérations, mais cela devient rapide plus long à écrire (et à lire) :

```
public class Position {
   private final float longitude;
   private final float latitude;

   public Position(float longitude, float latitude) {
      super();
      this.longitude = longitude;
      this.latitude = latitude;
   }

   public float getLongitude() {
      return longitude;
   }

   public float getLatitude() {
      return latitude;
   }

   @Override
   public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + Float.floatToIntBits(latitude);
      result = prime * result + Float.floatToIntBits(longitude);
      return result;
   }

   @Override
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj == null)
         return false;
      if (getClass() != obj.getClass())
         return false;
      Position other = (Position) obj;
      if (Float.floatToIntBits(latitude) != Float.floatToIntBits(other.latitude))
         return false;
      if (Float.floatToIntBits(longitude) != Float.floatToIntBits(other.longitude))
         return false;
      return true;
   }

   @Override
   public String toString() {
      return "Position [longitude=" + longitude + ", latitude=" + latitude + "]";
   }
}
```

Nous sommes loin de la simple définition que nous avons pensé au départ. C'est là que les enregistrements (« records ») entrent en jeu. Nous allons pouvoir écrire simplement :

```
record Position(float longitude, float latitude) {}
```

Par cette définition, nous avons automatiquement :

- un attribut final privé pour chaque composant de l'enregistrement,

- un accesseur public pour chaque composant de l'enregistrement,
- un constructeur contenant tous les composants de l'enregistrement,
- une méthode equals(),
- une méthode hashcode(),
- une méthode toString() pour visualiser les valeurs de chaque composant,

Vous aurez remarqué que la définition des attributs est réalisée au niveau de la définition de l'enregistrement. D'ailleurs, nous ne pouvons pas ajouter d'autre attribut dans le scope de la définition (hormis des attributs statiques).
Les données d'un enregistrement sont immutables. Nous avons seulement des accesseurs. De plus, la protection est accrue. En effet, dans la solution initiale avec la classe Position, il était encore possible de modifier la valeur des attributs finaux avec la réflexion. Avec les enregistrements, vous aurez droit à une exception : IllegalAccessException.

NOTE : *dans les sources fournies, vous avez les tests correspondant à la classe Position et au record Position.*

Un constructeur est automatiquement défini. Cependant, il est possible de définir son propre constructeur pour effectuer ces propres contrôles. Par exemple, nous pouvons avoir la définition suivante :

```
record Periode(LocalDate debut, LocalDate fin) {
  Periode {
    if ((debut == null || fin == null) || debut.isAfter(fin)) {
      throw new IllegalArgumentException(
       String.format("la date de début doit être antérieure à la date de fin
( debut : %s, fin :%s", debut, fin));
    }
  }
}
```

Restrictions :

- Nous ne pouvons pas lever des exceptions vérifiées (checked exception). C'est-à-dire que les exceptions que nous devons lever doivent être des RuntimeExceptions, comme dans l'exemple avec l'exception IllegalArgumentException.
- Au niveau des restrictions, nous ne pouvons pas étendre d'une autre classe ou d'un enregistrement. En effet, chaque enregistrement hérite automatiquement de la classe java.lang.Record. En revanche, il est possible de réaliser une ou plusieurs interfaces.

Réflexion

L'API de Réflexion a été mise à jour afin de manipuler et récupérer des informations sur les enregistrements. Pour illustrer les nouvelles méthodes, nous utiliserons jshell dans les exemples.

```
jshell> var bordeaux = new Position(44.841225, -0.5800364);
bordeaux ==> Position[longitude=44.841225, latitude=-0.5800364]
```

Nous pouvons savoir qu'une instance est un enregistrement avec la méthode isRecord()

```
jshell> bordeaux.getClass().isRecord()
$4 ==> true
```

Nous pouvons récupérer les composants de l'enregistrement :

```
jshell> bordeaux.getClass().getRecordComponents();
$5 ==> RecordComponent[2] { double longitude, double latitude }
```

Nous retrouvons nos deux composants : longitude et latitude et non des attributs. Au niveau de la JVM, cela est bien différent, car notre enregistrement ne possède pas d'attributs. En effet, si nous demandons la liste des attributs, nous n'avons rien.

```
jshell> bordeaux.getClass().getFields();
$6 ==> Field[0] { }
```

Nous retrouvons bien nos méthodes equals(), toString(), hashCode(), latitude() et longitude(), ainsi que notre constructeur Position()

```
jshell> bordeaux.getClass().getDeclaredMethods();
$7 ==> Method[5] { public final boolean Position.equals(java.lang.Object),
 public final java.lang.String Position.toString(), public final int Position.
hashCode(), public double Position.latitude(), public double Position.
longitude() }

jshell> bordeaux.getClass().getDeclaredConstructors();
$8 ==> Constructor[1] { public Position(double,double) }
```

Enregistrement local

Un cas d'usage est d'utiliser un enregistrement local au niveau d'une méthode. Cela permet d'enregistrer des valeurs intermédiaires. Cela est utile par exemple lors des traitements via l'API Stream. Passons directement à l'exemple :

```
List<Client> findTopClient(List<Client> clients, int annee) {
  // enregistrement local
  record ClientAchat(Client client, double montant) { };

  return clients.stream()
    .map(c -> new ClientAchat(c, calculMontantAchats(c, annee)))
    .sorted((r1, r2) -> Double.compare(r1.montant(), r2.montant()))
    .map(ClientAchat::client)
    .collect(Collectors.toList());
}
```

Dans l'exemple ci-dessous, nous utilisons l'API Stream en passant par les étapes suivantes :

- map() : Nous stockons temporairement le client et le montant calculé de ces achats dans un enregistrement local,
- sorted() : Nous effectuons un tri pour classer par montant,
- map() : Maintenant que c'est classé, nous récupérons seulement le client,
- collect() : Nous collectons les clients dans une liste qui sera retournée par la méthode.

L'enregistrement ClientAchat est utilisé uniquement au sein de la méthode. Il n'a pas d'autres visibilités. Le développeur n'a plus besoin de créer une classe ou une classe interne pour cela.

Enregistrements et Sérialisation.

Pour rappel, la sérialisation est un processus qui permet de convertir une instance d'objet en un format qui permet le transfert réseau ou la sauvegarde sur disque. Naturellement, l'opération inverse existe qui permet de transférer cette représentation en une instance d'objet. En Java, la sérialisation d'une instance d'une classe est possible. Pour cela, il suffit de

réaliser l'interface java.io.Serializable. Cela est suffisant, car la JVM est capable de réaliser les opérations en prenant tous les attributs non transitoires (« transient ») et en les écrivant d'un flux d'octet. Si le développeur le souhaite, il peut personnaliser les opérations de sérialisations avec les méthodes writeObject() et readObject() pour implémenter son propre format. Pour les enregistrements, c'est la même démarche. Il suffit de réaliser l'interface Serializable.

```
record Position(float longitude, float latitude) implements java.io.Serializable
```

Cela est très pratique pour le développeur, car la démarche est la même qu'une classe. En revanche, sous le capot, la prise en charge par la JVM est différente. Le principe est de rester simple avec les deux règles suivantes :

- La sérialisation d'un enregistrement est basée seulement sur l'état de l'instance.
- La desérialisation d'un enregistrement utilise seulement le constructeur canonique

De ce fait, il n'est pas possible de personnaliser le processus de sérialisation. En effet, l'état consiste à sauvegarder la valeur de chaque composant. Ayant une valeur pour chaque composant, il suffit d'appeler le constructeur canonique pour l'opération inverse.

Passons à la pratique. Nous allons réaliser l'interface Serializable au niveau de notre enregistrement Position

```
public record Position(double latitude, double longitude) implements
Serializable {
}
```

Suite à cela, nous allons pouvoir à présent sérialiser notre enregistrement

```
try ( FileOutputStream fos = new FileOutputStream("target/position.
serial");
  ObjectOutputStream oos = new ObjectOutputStream(fos)) {
  oos.writeObject(new Position(48.856614, 2.352221)); // Paris
  oos.writeObject(new Position(44.841225, -0.5800364)); // Bordeaux
}
```

puis, nous réalisons l'opération inverse

```
try ( FileInputStream fis = new FileInputStream("target/position.serial");
  ObjectInputStream ois = new ObjectInputStream(fis)) {
  System.out.println(ois.readObject());
  System.out.println(ois.readObject());
}
```

Ainsi, nous obtenons le résultat suivant :

```
Position[latitude=48.856614, longitude=2.352221]
Position[latitude=44.841225, longitude=-0.5800364]
```

La sérialisation appelle les accesseurs de l'enregistrement pour récupérer l'état : latitude() et longitude() pour notre exemple. Nous allons voir qu'il y a trois points intéressants sur la sérialisation des enregistrements.

Performance

La sérialisation des enregistrements est plus performante que celle utilisée pour les classes Java. En effet, la sérialisation des instances Java utilise l'API Réflexion pour récupérer la liste des attributs et leurs états. Ce qui est coûteux de performance. Pour les enregistrements, c'est l'API « Handle » de méthode de java.lang.invoke qui est utilisé (Pour rappel, l'API a été introduite en Java 7). Elle permet notamment de rechercher et d'invoquer des méthodes. Cela est plus performant que l'API Réflexion de Java 1.1.

Sécurité

Un autre problème de la sérialisation classique des objets Java. C'est aussi lors de la desérialisation. Le processus passe par la réflexion afin de positionner les valeurs de chaque attribut. Cela implique que des règles mis en place dans la phase de construction peuvent être ignorées et nous nous retrouvons avec des instances dans un état invalide. En revanche, au niveau des enregistrements, comme mentionnée plus haut, c'est le constructeur canonique qui est appelé et les règles que vous aurez codées seront par conséquent appelées. Cela vous semble un peu obscur. Illustrons cette problématique avec un cas pratique. Nous allons écrire la classe Java pour représenter une position.

```
public class PeriodeClass implements Serializable {

  private static final long serialVersionUID = 1L;

  private double latitude;
  private double longitude;

  public PositionClass(double latitude, double longitude) {
    super();
    this.longitude = longitude;
    this.latitude = latitude;
  }

  // Tous le reste est présent dans les sources fournies
}
```

Donc, nous pouvons sérialiser notre classe et notre l'enregistrement

```
try ( FileOutputStream fos = new FileOutputStream("target/position.serial");
  ObjectOutputStream oos = new ObjectOutputStream(fos)) {
  oos.writeObject(new PositionClass(250, 2.352221));  // Notre classe
  oos.writeObject(new Position(205, -0.5800364));     // Notre enregistrement
}
```

Cela fonctionne et sans surprise, nous obtenons le résultat suivant :

```
PositionClass [latitude=250.0, longitude=2.352221]
Position[latitude=205.0, longitude=-0.5800364]
----
```

Cependant, les valeurs ne sont pas cohérentes, car les latitudes et les longitudes sont normalement des valeurs comprises entre -90 et +90. Dans l'exemple, nous avons des latitudes supérieures à 200. Pour éviter cela, nous allons rajouter des contrôles au niveau du constructeur. Du côté de la classe PeriodeClass, nous modifions notre constructeur

```
public PositionClass(double latitude, double longitude) {
  super();
  if (latitude > 90 || latitude < -90 || longitude > 90 || longitude < -90) {
```

```
        throw new IllegalArgumentException(String.format("latitude et
longitude doivent être comprises entre -90 et 90. (latitude : %d, longitude :
 %d)", latitude, longitude));
    }
    this.longitude = longitude;
    this.latitude = latitude;
  }
```

Du côté de l'enregistrement, nous ajoutons le constructeur canonique

```
public Position {
    if (latitude > 90 || latitude < -90 || longitude > 90 || longitude < -90) {
        throw new IllegalArgumentException(String.format("latitude et
longitude doivent être comprises entre -90 et 90. (latitude : %f, longitude :
 %f)", latitude, longitude));
    }
  }
```

NOTE : *pour rappel, pas d'argument pour le constructeur de l'enregistrement.*

Maintenant, nous exécutons de nouveau le code pour la désérialisation. Et nous obtenons le résultat suivant :

```
PositionClass [latitude=250.0, longitude=2.352221]
java.io.InvalidObjectException: latitude et longitude doivent être comprises
 entre -90 et 90. (latitude : 245,000000, longitude : -0,580036)
    at java.base/java.io.ObjectInputStream.readRecord(ObjectInputStream.
java:2342)
    at java.base/java.io.ObjectInputStream.readOrdinaryObject(ObjectInput
Stream.java:2229)
    at java.base/java.io.ObjectInputStream.readObject0(ObjectInputStream.
java:1712)
    at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.
java:519)
    at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.
java:477)
    at fr.lbenoit.demo.jdk16.records.serialisation.PositionSerialisationTest.
Deserialiser(PositionSerialisationTest.java:32)
...
...
Caused by: java.lang.IllegalArgumentException: latitude et longitude
doivent être comprises entre -90 et 90. (latitude : 245,000000, longitude
: -0,580036)
    at fr.lbenoit.demo.jdk16.records.Position.<init>(Position.java:8)
    at java.base/java.io.ObjectInputStream.readRecord(ObjectInputStream.
java:2340)
```

Nous pouvons bien remarquer que l'instance de PeriodeClass a pu être créée malgré des valeurs incorrectes. Ce qui est malheureusement normal, car la sérialisation des classes Java ne passe pas par notre constructeur. La création de l' instance est faite avec un constructeur sans argument, puis les attributs sont initialisés par réflexion. De ce fait, les contrôles de notre constructeur ne sont pas pris en compte.
En revanche, côté enregistrement, nous avons bien une exception de type InvalidObjectException lié à notre exception IllegalArgumentException.
Ce mécanisme permet de renforcer la sécurité des données et d'assurer que les contrôles soient toujours bien exécutés.

Évolution

Un autre point intéressant lors de la desérialisation d'un enregistrement est l'utilisation des valeurs par défaut. C'est-à-dire que si un composant est manquant dans le flux d'octet, c'est la valeur par défaut qui est injectée lors de l'appel au constructeur canonique.
Cela permet de gérer les évolutions de nos structures en toute sécurité. Par exemple, si nous souhaitons ajouter l'altitude pour avoir la position en trois dimensions. Il nous suffit de rajouter le composant au niveau de la déclaration.

```
public record Position(double latitude, double longitude, double altitude)
 implements Serializable {
}
```

De ce fait, la lecture du flux d'octet généré avec l'ancienne structure reste possible sans changement de code. Concrètement, en exécutant le code précédent de désérialisation, nous obtenons le résultat suivant :

```
---
Position[latitude=48.856614, longitude=2.352221, altitude=0.0]
Position[latitude=44.841225, longitude=-0.5800364, altitude=0.0]
```

Nous pouvons remarquer que la propriété altitude prend la valeur 0, qui est bien la valeur par défaut pour un double.
Cela permet des évolutions aisées de nos structures de type enregistrement par rapport à l'utilisation des classes.

JDK 16 Sealed Classes (Preview)

En Java, jusqu'à présent, le seul contrôle sur l'héritage est de définir la classe finale avec le mot clé final. Par ce biais, il n'est plus possible de réaliser des classes dérivées de cette dernière. Les classes scellées sont un moyen pour le développeur d'avoir plus de contrôle sur les héritages en précisant ce qui est autorisé. Pour cela, nous avons deux nouveaux mots clés :

- sealed : permet d'avoir des classes dérivées à condition qu'elles fassent partir des classes autorisées. Pour cela, il y a aussi le mot clé permits.
- non-sealed : permet de ne pas limiter les classes dérivées.

Voici un exemple d'utilisation :

```
public sealed class Vehicule permits Voiture, Moto {

}
```

Nous avons une classe Vehicule dont nous devons hériter seulement avec les classes dérivées : Voiture et Moto. Cela est vérifié dès la compilation. À partir du moment où nous avons utilisé le mot sealed, il faut donner des précisions sur les sous-classes avec les mots suivants: final, sealed, non-sealed. Par exemple, nous pouvons avoir la définition suivante pour la classe Moto

```
public final class Moto extends Vehicule {

}
```

Naturellement, nous pouvons de nouveau utiliser le mot clé sealed, dans ce cas, il faudra indiquer les classes dérivées autorisées. En revanche, il est possible d'indiquer qu'une classe dérivée ne soit pas scellée, dans ce cas, nous pourrons avoir autant de classes dérivées que nous voulons.

Par exemple, nous pouvons avoir la classe Voiture non scellée.

```
public non-sealed class Voiture extends Vehicule {

}
```

Ainsi, nous pouvons voir la classe VoitureThermique, VoitureElectrique.

```
public class VoitureThermique extends Voiture {

}

public class VoitureElectrique extends Voiture {

}
```

Types scellés et les enregistrements

La combinaison des enregistrements et des types scellés permet de se référer à des types produits et des types sommes. Combinée à la récursivité, les données structurées peuvent exprimer des listes et des arbres.

NOTE : *pour en savoir plus, vous pouvez consulter la page Wikipédia sur les types algébriques de données [5].*

De notre côté, nous allons prendre un exemple sur la représentation d'un arbre. Nous avons une interface Nœud qui autorise uniquement nos enregistrements Feuille, Addition, Multiplication. Nous utilisons une interface scellée, car nous souhaitons maîtriser complètement notre hiérarchie. La caractéristique d'un nœud est de pouvoir évaluer sa valeur, un réel dans notre cas.

Nous avons ensuite trois enregistrements, chacun réalise l'interface Nœud et implémente la méthode eval() :

- Feuille qui contient un réel et qui retourne la valeur lors de l'évaluation.
- Addition qui contient 2 instances de Nœud, l'évaluation retourne l'addition de la valeur des deux nœuds.
- Multiplication qui contient 2 instances de Nœud, l'évaluation retourne la multiplication de la valeur des deux nœuds.

```
public sealed interface Noeud
    permits Noeud.Feuille, Noeud.MultiNoeud, Noeud.AdditionNoeud {

    public double eval();

    public record Feuille(double val) implements Noeud {
        public double eval() {
            return val;
        }
    }

    public record Multiplication(Nœud gauche, Nœud droite) implements Noeud {
        public double eval() {
            return gauche.eval() * droite.eval();
        }
    }

    public record Addition(Nœud gauche, Nœud droite) implements Noeud {
        public double eval() {
            return gauche.eval() + droite.eval();
        }
    }
}
```

Avec le code mentionné, le fait d'appeler la méthode eval() sur la racine de l'arbre permet d'avoir le résultat des calculs en fonction des types de nœuds (sans avoir besoin de parcourir l'arbre). Prenons un exemple d'arbre avec les nœuds M, A, F, qui représentent respectivement des instances de Multiplication, Addition et Feuille. Posons l'arbre suivant :

```
//          M(r)
//        /     \
//     M(n1)     F(n2)
//    /   \
// F(n3)    À(n4)
//         /    \
//      F(n5)    F(n6)
```

Si les Feuilles n2, n3, n5 et n6 ont respectivement les valeurs 2, 4, 3 et 1. L'évaluation de la racine donne la valeur 32.0. Le code de tests associé est le suivant :

```
Nœud n6 = new Noeud.Feuille(1.0);
Noeud n5 = new Noeud.Feuille(3.0);
Nœud n4 = new Noeud.Addition(n5, n6);
Noeud n3 = new Noeud.Feuille(4.0);
Noeud n2 = new Noeud.Feuille(2.0);
Noeud n1 = new Noeud.Multiplication(n3, n4);
Noeud r  = new Noeud.Multiplication(n1, n2);
```

Assert.assertEquals("L'évaluation de l'arbre a échoué", 32.0, r.eval(), 0); L'évaluation du nœud racine (r dans notre exemple) donne bien la valeur 32.0.

Conclusion

L'introduction des enregistrements est très intéressante au sein du langage Java. Certains les comparent à l'arrivée des lambdas dans le JDK 8. Il est indéniable que leurs utilisations vont se populariser. Des librairies comme jackson les prennent d'ores et déjà en charge.

Nous avons aussi vu le filtrage par motif avec le mot clé instanceof. Nous avons déjà une série de fonctionnalités dans ce sens en préparation. Par exemple, le « Pattern Matching for switch (Preview) » est une fonctionnalité candidate pour le JDK 17.

À vous de profiter de toutes ces nouvelles fonctionnalités !

Référence

[1] https://snyk.io/blog/java-ecosystem-survey-2021/ - Rapport Snyk sur l'écosystème Java en 2021

[2] https://twitter.com/steveonjava/status/1091634904025178112 – Tweet de Stephen Chin présentant Mark Reinhold au FOSDEM 2019

[3] https://fr.wikipedia.org/wiki/Filtrage_par_motif – Article Wikipedia sur le filtrage par motif

[4] https://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html – Article de Brian Goëtz et Gavin Bierman sur le filtrage par motif pour Java

[5] https://fr.wikipedia.org/wiki/Type_alg%C3%A9brique_de_donn%C3%A9es – Article Wikipedia sur le type algébrique des données

Java 16 : quoi de neuf ?

Loïc Mathieu

Consultant, formateur et speaker.
Zenika Lille.

zenika
<animés par la passion>

Java 16 est sorti le 16 mars. Cette nouvelle version compte 17 JEP (Java Enhancement Proposals). Records et le pattern matching pour instanceof sortent du « preview » et viennent en version finale. C'est une bonne nouvelle. Dans cet article, nous allons vous faire un panorama des nouveautés.

JEP 380: Unix-Domain Socket Channels

Tout d'abord une petite description de ce qu'est un socket de type Unix-Domain : *les sockets Unix-Domain sont utilisés pour la communication inter-processus (IPC) sur le même hôte. Ils sont similaires aux sockets TCP/IP à bien des égards, sauf qu'ils sont adressés par path de filesystem plutôt que par des adresses IP et des numéros de port.*

Les classes SocketChannel et ServerSocketChannel peuvent dorénavant être créées depuis un socket de type Unix-Domain.

Exemple (non testé):

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.socket().bind(UnixDomainSocketAddress.of("path/to/socket/file");
while(true){
  SocketChannel socketChannel = serverSocketChannel.accept();
  //do something with socketChannel...
}
```

Plus d'informations dans la JEP-380 : https://openjdk.java.net/jeps/380

JEP 338: Vector API (Incubator)

Vector API est une nouvelle API qui permet d'exprimer des calculs de vecteur (calcul matriciel par exemple), qui seront exécutés via des instructions machines optimales en fonction de la plateforme d'exécution.

Ces optimisations incluent des changements au sein du Just In Time compiler de la JVM, des intrinsics (un intrinsic est une méthode que la JVM peut remplacer par une implémentation manuellement optimisée pour une architecture CPU), et utilisent les instructions AVX/SSE des CPU qui permettent une vectorisation des calculs (instructions de type SIMD - Single Instruction Multiple Data).

La JEP contient l'exemple suivant, implémenté **avant** la Vector API par :

```
void scalarComputation(float[] a, float[] b, float[] c) {
  for (int i = 0; i < a.length; i++) {
    c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
  }
}
```

Et **avec** la Vector API par :

```
static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_256;

void vectorComputation(float[] a, float[] b, float[] c) {

  for (int i = 0; i < a.length; i += SPECIES.length()) {
    var m = SPECIES.indexInRange(i, a.length);
    // FloatVector va, vb, vc;
    var va = FloatVector.fromArray(SPECIES, a, i, m);
    var vb = FloatVector.fromArray(SPECIES, b, i, m);
    var vc = va.mul(va).
          add(vb.mul(vb)).
          neg();
    vc.intoArray(c, i, m);
  }
}
```

On utilise ici un FloatVector.SPECIES_256 qui permet de gérer des vecteurs de float sur 256 bits. Les opérations mul, add et neg seront donc faites via des instructions SIMD sur 256 bits au lieu d'être faites unitairement sur chaque float.

Plus d'informations dans la JEP-338 : https://openjdk.java.net/jeps/338

JEP 389: Foreign Linker API (Incubator)

Avec la JEP-393 Foreign-Memory API, qui permet de gérer des segments mémoire (on heap ou off heap), cette nouvelle fonctionnalité pose les bases du projet Panama en permettant l'interconnexion de la JVM avec du code natif.

La Foreign Linker API permet d'appeler du code natif (en C par exemple) de façon facile et performant.

Voici un exemple d'appel de la fonction strlen de la librairie standard C :

```
MethodHandle strlen = CLinker.getInstance().downcallHandle(
    LibraryLookup.ofDefault().lookup("strlen").get(),
    MethodType.methodType(long.class, MemoryAddress.class),
    FunctionDescriptor.of(C_LONG, C_POINTER)
);

long len = strlen.invokeExact(CLinker.toCString("Hello").address())
```

Plus d'informations sur les appels de fonction native dans cet article de Maurizio Cimadamore : State of foreign function support - https://cr.openjdk.java.net/~mcimadamore/panama/ffi.html

Ce code peut sembler un peu complexe, c'est pour cela qu'a été créé **jextrac**, qui permet d'extraire le code nécessaire à l'appel d'une librairie C automatiquement depuis un fichier header C.

Exemple pour l'appel de la méthode getpid de la librairie standard C.

Utilisation de **jextract** pour générer le code d'invocation de la méthode getpid :

```
echo "int getpid();" > getpid.h
jextract -t com.unix getpid.h
```

Utilisation du code Java généré :

```
import static com.unix.getpid_h.*;

class Main2 {
  public static void main(String[] args) {
```

```
    System.out.println(getpid());
  }
}
```

Plus d'informations sur **jextract** dans cet article, de Sundar Athijegannathan : Project Panama and jextract - https://inside.java/2020/10/06/jextract/

Deux nouveaux ports de la JVM

OpenJDK 16 ajoute le support Alpine Linux, et donc Musl comme implémentation de la librairie standard C, pour les architectures x64 et AArch64. Plus d'informations dans la JEP-386 : https://openjdk.java.net/jeps/386

OpenJDK 16 ajoute aussi le support de l'architecture AArch64 sous Windows (précédemment uniquement supporté sous Linux) via la JEP 388 - https://openjdk.java.net/jeps/388.

Le support de macOS sur cette architecture (Apple Silicon) est en cours et sera certainement livré dans une prochaine version de Java, voir la JEP 391 : https://openjdk.java.net/jeps/391.

Les fonctionnalités qui passent de preview à standard

Les fonctionnalités suivantes, qui étaient en preview (ou en incubator module), sont maintenant en standard.

- JEP 392 - https://openjdk.java.net/jeps/392 : Packaging Tool.
- JEP 394 - https://openjdk.java.net/jeps/394 : Pattern Matching for instanceof.
- JEP 395 - https://openjdk.java.net/jeps/395 : Records.

Pattern Matching for instanceof.

Chaque développeur a déjà écrit du code qui ressemble à ça avec l'opérateur instanceof :

```
if (obj instanceof Integer) {
  int intValue = ((Integer) obj).intValue();
  // use intValue
}
```

Le cast après l'**instanceof** semble superflu, car on vient de tester le type de l'objet.

Et c'est là qu'entre en scène le pattern matching, il va permettre de vérifier qu'un objet est d'un type précis (comme **instanceof** le fait) et "extraire" la "forme" de l'objet dans une nouvelle variable.

On va donc pouvoir remplacer le code précédent par celui-ci :

```
if (obj instanceof Integer intValue) {
  // use intValue
}
```

Plus besoin de cast, et on assigne à une variable locale au bloc qui va permettre d'utiliser directement l'objet via son type vérifié par l'opérateur **instanceof**.

Records

Un **Record** est un nouveau type Java (comme class et enum), son but est d'être un conteneur de données. Les Records sont implicitement final et ne peuvent être abstraits.

Les Records fournissent une implémentation par défaut pour du code boilerplate que vous auriez sinon généré via votre IDE.

```
record Point(int x, int y) { } // definition d'un record

Point p = new Point(1, 1); //utilisation du constructeur canonique

p.x(); //utilisation de l'accesseur pour l'attribut x
```

Tous les Records ont des accesseurs publics (mais les champs sont privés) et une méthode **toString**. Ils ont aussi les méthodes equals et hashCode dont les implémentations sont basées sur le type du Record et son état (ses champs donc).

Les fonctionnalités qui restent en preview

Les fonctionnalités suivantes restent en preview (ou en incubator module).

- JEP 393 - https://openjdk.java.net/jeps/393 : Foreign-Memory Access API (Third Incubator).
- JEP 397 - https://openjdk.java.net/jeps/397 : Sealed Classes (Second Preview)

Les premiers impacts du projet Valhalla

La JEP 390 - https://openjdk.java.net/jeps/39 : Warnings for Value-Based Classes apporte un changement en préparation des ***Primitive Objects*** (auparavant inline classes) du projet Valhalla.

Certaines classes du JDK sont appelées **Value-Based Classes**, ce sont des classes qui ne sont que des véhicules pour de la donnée (data carrier), et sont donc susceptibles d'être transformées en *Primitive Objects* quand le projet Valhalla sortira. C'est le cas par exemple de la classe Optional ou des wrappers sur les primitives.

À partir de Java 16, l'utilisation de ces classes d'une manière incompatible avec les *Primitive Objects* va générer des warnings : utilisation des constructeurs (qui sont dépréciés et seront supprimés dans une release future), synchronisation, utilisation incorrecte du == ou du !=.

Divers

Divers ajouts :

- JDK-8238286 : **Stream.mapMulti()** : permet de mapper un élément T en n éléments R via un Consumer<R>?
- JDK-8180352 : **Stream.toList()** : accumule les éléments de la Stream dans une liste immuable.
- JDK-8255150 : Ajout de méthodes utilitaires pour vérifier le range d'un index de type long : Objects.checkIndex(long index, long length, Objects.checkFromToIndex(long fromIndex, long toIndex, long length, Objects.checkFromIndexSize(long fromIndex, long size, long length. Ces nouvelles méthodes sont optimisées via l'ajout d'intrinsics.

Lors de la modularisation du JDK via le projet Jigsaw, certaines API internes du JDK qui ne devraient pas être utilisables en dehors de celui-ci, ont quand même été rendues utilisable (par manque d'alternatives, ou pour donner aux applications les utilisant un temps de migration). C'est ce qui a été appelé *Relaxed Strong Encapsulation* - https://openjdk.java.net/jeps/261#Relaxed-strong-encapsulation. Un simple WARNING était alors affiché dans les logs de la JVM lors de la première utilisation de ces API (comportement modifiable via --illegal-access).

Avec la JEP 396 - https://openjdk.java.net/jeps/396 : *Strongly Encapsulate JDK Internals by Default*, l'accès à ces API (qui

ont quasiment toutes un remplacement officiel dans le JDK), est maintenant interdit par défaut.
La valeur par défaut du flag --illegal-access passe donc de permit à deny. Le flag étant toujours présent on peut le modifier pour retrouver le comportement précédent si nécessaire.

Performance

Beaucoup d'intrinsics ont été ajoutés à la JVM :

- JDK-8250902 : MD5 intrinsic.
- JDK-8248188 : Base64 intrinsic.
- JDK-8173585 : Intrinsic pour StringLatin1.indexOf(char).

Plusieurs nouveaux intrinsics dédiés pour AArch64 (déjà présent sur d'autres plateformes).

Temps de démarrage

Java 16 a vu son lot d'optimisations du temps de démarrage de la JVM. Claes Redestad, l'un des ingénieurs de chez Oracle a écrit un article assez intéressant qui reprend les différentes optimisations du temps de démarrage de la JVM depuis Java 8 et se pose la question des optimisations encore à faire pour Java 17 : Towards OpenJDK 17 - https://cl4es.github.io/2020/12/06/Towards-OpenJDK-17.html.
Release après release, le temps de démarrage de la JVM à quasiment été divisé par 2 depuis Java 8 (après un accroissement non négligeable en Java 9 suite à l'introduction des modules). Sachant qu'une JVM démarre en moins de 40ms, Claes se pose la question de savoir si c'est encore nécessaire de travailler sur le sujet ?
Pour ce qui concerne Java 16, on peut noter, entre autres, de nombreuses améliorations de la fonctionnalité CDS :

- JDK-8244778 : Archive full module graph in CDS.
- JDK-8247536 : Support for pre-generated java.lang.invoke classes in CDS static archive.
- JDK-8247666 : Support Lambda proxy classes in static CDS archive.
- CDS - Class Data Sharing, permet d'enregistrer dans une archive les données des metadatas des classes lors du lancement de la JVM, pour les réutiliser lors de lancements successifs, optimisant alors le temps de démarrage de cette dernière.

Vous pourrez trouver plus d'informations dans mon article : https://www.loicmathieu.fr/wordpress/informatique/ quarkus-jlink-et-application-class-data-sharing-appcds.
La JVM contient une archive par défaut avec les metadatas de certaines classes du JDK, des changements sur la fonctionnalité CDS profitent donc automatiquement à tout le monde.

Conclusion

Java 16 est une version riche en nouvelles fonctionnalités, et qui en plus voit le passage en final de fonctionnalités très importantes du langage telle que les **Records** et le pattern matching pour **instanceof**. Via la nouvelle Vector API, elle va ouvrir la voie à de nouvelles classes d'algorithme dans la JVM, et on voit déjà de nombreuses expérimentations positives dans la communauté.
Même si de nombreuses personnes vont préférer attendre Java 17 qui sera une LTS (Long Term Support), je pense que Java 16 est une version intéressante à utiliser pour ceux qui le peuvent.

DÉPRÉCIATIONS & RETRAITS

Peu de choses à noter dans Java 16 :
Suppression :

- le constructeur de javax.tools.ToolProvider.

Côté dépréciation :

- les constructeurs des Wrapper sur les primitives

Introduction au Machine Learning avec Tribuo

Tribuo est une librairie Java open source dédiée au Machine Learning. Elle fournit tous les outils pour réaliser les opérations traditionnelles de l'apprentissage automatique telles que : la résolution des problèmes de classification, régression, Clustering. etc. Découvrons ensemble Tribuo.

Elvadas NONO

Elvadas est Principal Solution Architect chez Oracle, spécialiste de la transformation digitale et des technologies Java, Middleware et Cloud. Elvadas accompagne les entreprises françaises et internationales dans la modernisation de leur parc applicatif et la transition vers le multi-cloud. Il a travaillé de nombreuses années chez Red Hat et Docker Inc en tant que consultant spécialiste des Middlewares et des Plateforme de Containers Docker, Kubernetes. Passionné des nouvelles technologies, toujours curieux d'apprendre et porté par les causes humanitaires.

Tribuo : une librairie fortement typée

Distribué sous licence Apache 2.0 et développé principalement par l'Oracle Labs Machine Learning Research Group, Tribuo permet de construire et de déployer des modèles de machine learning, à travers d'API Java fortement typées.

Les **modèles** Tribuo s'appliquent à des objets de type **Examples**, produisent des résultats encapsulés dans des objets de type **Prédictions** et pas des tableaux de flottants / double comme le font la plupart des librairies Python dédiées au Machine Learning.

Une librairie extensible et favorisant la réutilisation des modèles existant

Tribuo fournit une interface unifiée d'accès à différentes librairies éprouvées dans le domaine du machine learning telles que Xgboost et liblinear. XGBoost (comme eXtreme Gradient Boosting) est une implémentation open source optimisée de l'algorithme d'arbres de boosting de gradient massivement utilisé dans le domaine du Machine learning : au lieu d'utiliser un seul modèle, l'algorithme XGBoost va en utiliser plusieurs qui seront ensuite combinés pour obtenir un seul résultat. Tribuo permet également l'export import des modèles, le déploiement de modèles produits à partir des librairies python tels que scikit-learn et pytorch dans un programme Java

Example

Dans cet exemple, nous allons exploiter un échantillon de données issue de la plateforme Kaggle pour mettre sur pied un modèle simple de prédiction du prix d'assurance santé d'une population.

Les données collectées sont représentées sous forme de fichier CSV contenant plusieurs colonnes dans l'ordre suivant :

- l'âge de la personne interrogée
- le sexe (0 : Femme, 1: Homme)
- l'indice de masse corporelle IMC,
- le nombre d'enfants,
- l'indication si la personne fume (0) ou non (1) ,
- un identifiant du département de résidence de la personne
- en dernier lieu la prime d'assurance qui est la variable à prédire

Ci-dessous un extrait du fichier en question.

$ head insurance.csv

```
19,0,27.9,0,1.0,0,16884.924
18,1.0,33.77,1.0,0,1.0,1725.5523
28,1.0,33,3,0,1.0,4449.462
33,1.0,22.705,0,0,3,21984.47061
32,1.0,28.88,0,0,3,3866.8552
31,0,25.74,0,0,1.0,3756.6216
46,0,33.44,1.0,0,1.0,8240.5896
37,0,27.74,3,0,3,7281.5056
37,1.0,29.83,2,0,2,6406.4107
60,0,25.84,0,0,3,28923.13692
```

Afin de résoudre ce problème, il est possible d'utiliser un notebook Jupiter. Le support du langage Java par Jupiter est assurée par la librairie IJava qui requiert une version 10 minimum.

Le problème à résoudre vise à prédire une valeur numérique (prix de l'assurance maladie en USD). C'est donc un problème de régression pour lequel nous allons utiliser la librairie XGBoot pour créer et entraîner un modèle simpliste.

```
var regressionFactory = new RegressionFactory();
var csvLoader = new CSVLoader<>(regressionFactory);


//Load data
var insuranceHeaders = new String[]{"age","sexe","imc","enfants","fumeur","departement",
"prime"};
var datasource = csvLoader.loadDataSource(Paths.get("src/main/resources/insurance
.csv"),"prime",insuranceHeaders);
```

La première étape consiste à charger le fichier de données à traiter, via la classe csvLoader. Pour les problèmes de classification(Régression), un Label Factory(resp RegressionFactory) doit être utilisé.

```
//split train and test data
var splitter = new TrainTestSplitter<>(datasource,0.95, 0L);
var trainingDataset = new MutableDataset<>(splitter.getTrain());
var  testingDataset = new MutableDataset<>(splitter.getTest());

//display data overwiew
System.out.println(String.format("Training data size = %d, number of features = %
d",trainingDataset.size(),trainingDataset.getFeatureMap().size()));
System.out.println(String.format("Testing data size = %d, number of features = %
d",testingDataset.size(),testingDataset.getFeatureMap().size()));
```

Le fichier de données chargé est séparé en deux catégories via la classe **TrainTestSplitter** : 95% des lignes seront utilisées dans l'échantillon d'entraînement contre seulement 5% pour des données de tests du modèle construit.

```
// create and Train Model on Train data set
var xgb = new XGBoostRegressionTrainer(XGBoostRegressionTrainer.RegressionType.
LINEAR,50);
```

Par la suite un modèle XGBoost de régression linéaire est créé avec 100 arbres pour affiner la précision du modèle.
Une fois un algorithme choisi, la phase d'entraînement peut être lancée afin de produire un modèle cible :

```
var xgbModel = xgb.train(trainingDataset);
```

Puis procéder à des prédictions sur la base du modèle créé :

```
// Make prediction
Example<Regressor> xPredict = testingDataset.getExample(0);
System.out.println("xPredict="+xPredict);
List<Prediction<Regressor>> yPredict = Collections.singletonList(xgbModel.predict(xPredict));
System.out.println("yPredict="+yPredict);
```

Sortie console :

```
Training data size = 1271, number of features = 6
Testing data size = 67, number of features = 6
xPredict=ArrayExample(numFeatures=6,output=(DIM-0,8162.71625),weight=1.0,
features=[(age, 40.0)(departement, 3.0), (enfants, 4.0), (fumeur, 0.0), (imc, 30.875),
(sexe, 1.0), ])
yPredict=[Prediction(maxLabel=(DIM-0,13242.43359375),outputScores})]
```

Dans ce cas d'usage par exemple, la valeur prédite **(13242.43359375)** est assez éloignée de la valeur cible **(8162.71625)**. Il faut procéder à des améliorations et réaliser un tuning du modèle par des évaluations et configurations des hyperparamètres.
Plusieurs autres paramètres peuvent être définis à la création du modèle XGBoost, pour des besoins de simplicité nous ne les aborderons pas tous: Voir les constructeurs de la classe XGBoostRegressionTrainer.

```
Create an XGBoost trainer.
Params:
rType – The type of regression to build.
numTrees – Number of trees to boost.
eta – Step size shrinkage parameter (default 0.3, range [0,1]).
gamma – Minimum loss reduction to make a split (default 0, range [0,inf]).
maxDepth – Maximum tree depth (default 6, range [1,inf]).
minChildWeight – Minimum sum of instance weights needed in a leaf (default 1,
range [0, inf]).
subsample – Subsample size for each tree (default 1, range (0,1]).
featureSubsample – Subsample features for each tree (default 1, range (0,1]).
lambda – L2 regularization term on weights (default 1).
alpha – L1 regularization term on weights (default 0).
nThread – Number of threads to use (default 4).
silent – Silence the training output text.
seed – RNG seed.
```

Le modèle ML construit peut être évalué à la fois sur les données d'entraînement ou les données de test :

```
// Evaluate Model towards TrainingDataSet
RegressionEvaluator evaluator = new RegressionEvaluator();
var evaluationTrain = evaluator.evaluate(xgbModel,trainingDataset);
System.out.println("Train data eval:"+evaluationTrain.toString());

// Evaluate Model towards TestingDataSet
var evaluationTest = evaluator.evaluate(xgbModel,testingDataset);
System.out.println("Test data Eval:"+ evaluationTest.toString());
```

Chaque évaluation fournit un certain nombre d'indicateurs tels que l'écart type RMSE (Root Mean Square Error) ou la variance permettant d'apprécier la qualité du modèle.

```
Train data eval:Multi-dimensional Regression Evaluation
RMSE = {DIM-0=1121.8739=247.9502241994838}
Mean Absolute Error = {DIM-0=1121.8739=9.84057398481758}
R^2 = {DIM-0=1121.8739=0.9995727916141816}
explained variance = {DIM-0=1121.8739=0.9995727916141817}

Test data Eval:Multi-dimensional Regression Evaluation
RMSE = {DIM-0=1121.8739=6018.547409168213}
Mean Absolute Error = {DIM-0=1121.8739=3171.0083601463393}
R^2 = {DIM-0=1121.8739=0.8085782492452122}
explained variance = {DIM-0=1121.8739=0.8156063825533252}
```

Un modèle est intéressant si l'écart type est proche de la valeur 0. Le coefficient de détermination R^2 quant à lui mesure l'éloignement des données par rapport à la ligne de régression et de fait l'adéquation du modèle par rapport à ces données.

Source du projet : https://github.com/nelvadas/tribuo-ml-demos

Bon machine learning !

Architecture microservice : DDD l'arme absolue mais pas que...

Cet article abordera le Domain Driven Design (DDD) appliqué à la conception des microservices en se basant sur l'architecture hexagonale. Tout d'abord nous allons commencer par présenter le découpage en couches d'un microservice afin de respecter les principes du DDD. Ensuite, nous allons zoomer sur l'implémentation de ces principes en nous basant sur les standards Java et le Framework de développement Spring Boot. Enfin, nous allons conclure avec quelques bonnes pratiques, et un exemple de code sera fourni pour illustrer l'ensemble.

BADR NASS LAHSEN

Principal Cloud Architect chez Oracle, 13 ans d'expérience IT, il a contribué à la transformation digitale de plusieurs banques en France et à l'international. Il est auteur de la librairie springdoc-openapi.

Qu'est-ce qu'un microservice ?

Un microservice selon **Gartner** est un composant logiciel, responsable d'un certains nombres d'opérations inhérente à un domaine métier. Il est le seul et unique composant responsable de les exposer via des interfaces explicites de communication (API). De plus, il est scalable et déployable indépendamment des autres composants du SI. Parmi les avantages d'une architecture microservices, on peut citer :

- Une meilleure résilience : en cas de panne, on sait isoler le problème et démarrer une nouvelle instance.
- Une meilleure maintenance du code, grâce à la modularité
- Une accélération du Time To Market
- Une meilleure interopérabilité entre des applications hétérogènes technologiquement.
- Une meilleure disponibilité
- La réutilisation et l'évolutivité
- Une intégration **native** dans une architecture **cloud** (Voir The Twelve-Factor App - https://12factor.net/fr/)
- Une performance accrue, grâce à une montée en charge à la demande tout en tirant profit des bénéfices du cloud

Figure 1

Le constat dans les entreprises

Une des lois à laquelle beaucoup d'organisations n'échappent pas, c'est la **loi de Conway** : « Très souvent, les applications sont conçues de façon à refléter l'organisation de l'entreprise ». Au vu de l'évolution des organisations d'une approche orientée coûts vers une approche orientée création de valeur pour les métiers grâce notamment à l'organisation en mode produit, l'architecture microservices trouve tout son sens, pour permettre cet alignement. **Figure 2**

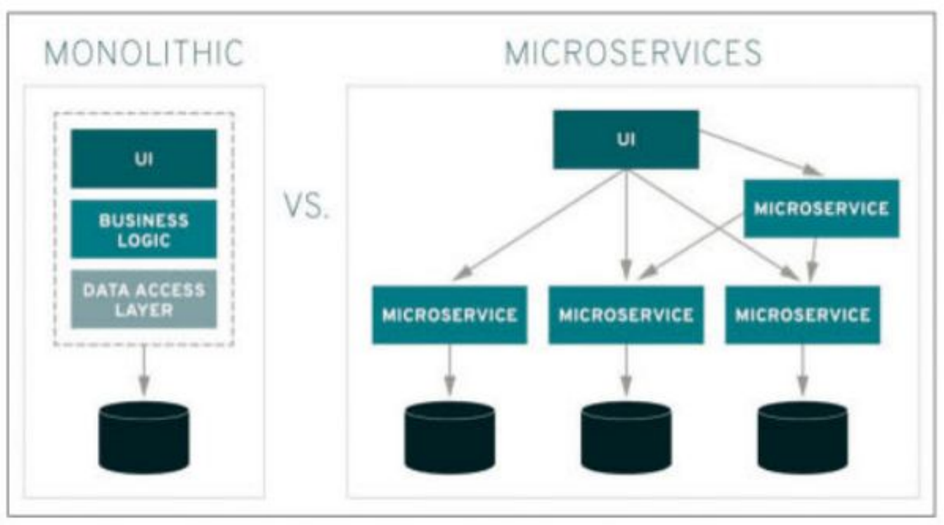


Figure 1 : *Illustration d'une architecture microservices*

Le constat est que plusieurs entreprises disposent d'un Legacy applicatif et une énorme dette technique. Ce Legacy, qui était vu par le passé, il y a 40 ans, comme le succès de ces entreprises, devient désormais très difficile à faire évoluer. Par conséquent, toute évolution d'un périmètre métier devient douloureuse et complexe pour tout le système d'information, ce qui dénote d'un manque d'agilité.

Une des approches les plus communément utilisées pour moderniser ce type systèmes d'informations est le **Strangler Pattern** : Il permet de découper une application monolithique en plusieurs domaines et de migrer progressivement un périmètre métier, en dehors de l'application monolithique vers un service plus fin, et qui peut être un microservice.

Par ailleurs, vous pouvez trouver plusieurs patterns, dans l'ouvrage de Sam Newman (2019) « Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith ».

Comment déterminer le périmètre d'un microservice ?

Au vu de la complexité pour réunir tous les paramètres pour concevoir un microservice, respectant ces principes, une méthodologie se démarque pour concevoir efficacement un microservice. Il s'agit du Domain Driven Design (DDD).

DDD est une approche de conception qui permet de définir une vision et un langage partagés par toutes les personnes impliquées dans la construction d'une application (experts métier, concepteur, développeur, testeurs…). Cette approche est indépendante de tout langage, elle est centrée sur le

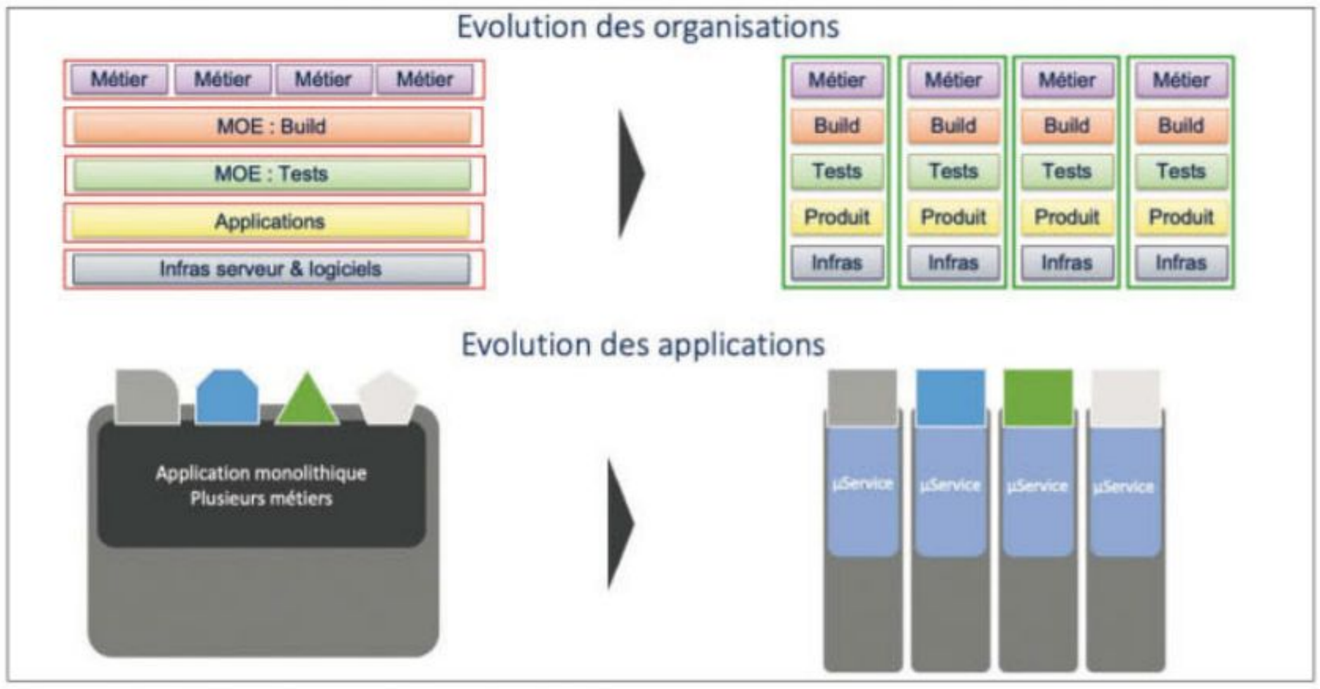


Figure 2 : *Évolution des organisations et applications*

métier et permet de mieux appréhender la complexité.
Eric Evans dans son ouvrage « Domain-Driven Design: Tackling Complexity in the Heart of Software - 2004» reste une bonne référence pour aller dans le détail.
Voici quelques concepts clés du DDD :

- **Bounded Context** : un contexte borné fournit le cadre logique à l'intérieur duquel le modèle évolue. La division d'une application monolithique en contextes limités rend l'application encore plus maintenable, faiblement couplée et réutilisable.
- **Ubiquitous Language** : langage omniprésent, il est très important de comprendre tous les termes d'un domaine métier spécifique, car il y a beaucoup de mots qui ont des significations différentes selon le contexte dans lequel ils sont utilisés (homonymes). Cette langue doit être basée sur le domaine métier. Le langage omniprésent est le lien entre les développeurs) et toutes les autres parties prenantes du domaine. Il aide à éliminer les contradictions au cours du cycle de vie du projet.
- **Le domaine** : la sphère de connaissance d'un métier, on y trouve des notions permettant de représenter la réalité du métier, comme les entités, les Value Objects, agrégats et événement.

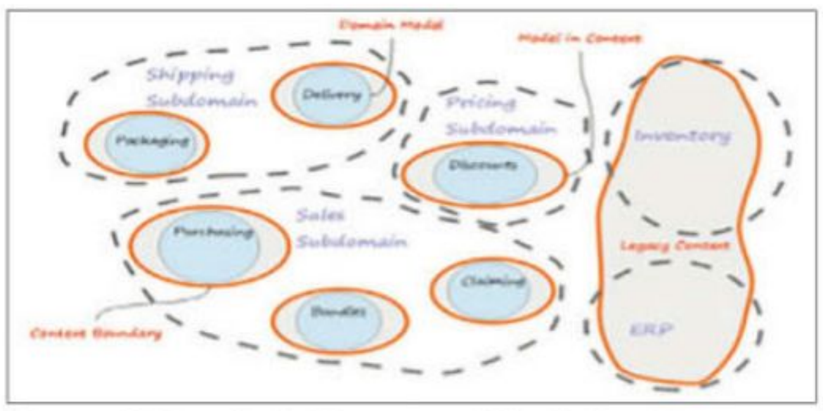


Figure 3 : *Exemple de découpage en différents domaines*

Principes de développement à adopter ?

Il existe plusieurs principes complémentaires de conception visant à rendre le développement logiciel compréhensible, flexible et maintenable.
Certains sont proposés par Robert C. Martin (2000) dans son livre « Design Principles and Design Patterns » et doivent être utilisés dans le cadre d'une démarche DDD : on y trouve notamment les principes **SOLID**, qui permettent à un composant logiciel, de couvrir une seule responsabilité.
D'autres, peuvent compléter ces principes comme **DRY**, pour « **D**on't **R**epeat **Y**ourself», afin d'éviter les duplications de code, ou bien **KISS**, pour « **K**eep **I**t **S**imple **S**tupid », éviter de compliquer les choses quand on peut faire simple et **YAGNI** « **Y**ou **A**in't **G**onna **N**eed **I**t», afin d'ajouter des fonctionnalités, que quand cela est nécessaire.

L'architecture hexagonale

L'architecture hexagonale concentre la logique métier dans une couche domaine (le modèle) ne dépendant d'aucune autre couche, ne contenant aucune référence technique (annotation ORM, I/O, middleware...).
La préoccupation première du développeur est de rendre l'application agnostique à son usage (utilisateur, autre service, batch), développée et testée hors conditions d'exécution (Mobile, Navigateur, base de données, middleware...).
Ce pattern permet de :

- Changer de technologies sans impacter le code du domaine (modèle)
- Construire la couche domaine :
  - En faisant abstraction de comment le modèle sera consommé
  - En différant les choix technologiques infrastructurels
- Favoriser le bouchonnement permettant de paralléliser le développement de services
- Favoriser les approches **BDD** et **TDD**
- Préserver la logique métier pure et cohésive

Quels frameworks utiliser pour développer des microservices ?

Un des grands avantages de l'architecture microservices, est qu'on n'est pas contraint à utiliser une technologie particulière.
En fonction de la maîtrise de l'équipe et des orientations de votre organisation IT en termes de langage, on peut parfaitement avoir plusieurs équipes utilisant des langages différents (java / python / GO/ Kotlin...).
Certes, cela permet d'avoir une meilleure autonomie, mais attention à la multiplication des langages, qui peut devenir en soit une énorme dette technique à absorber.
Il y a un grand besoin de rationaliser, cadrer et d'utiliser des Frameworks matures notamment avec l'émergence de nouveaux langages.
Multiplier les technologies requiert plusieurs modèles opérationnels avec une complexité d'appliquer les patchs pour chaque technologie, une difficulté de mise à l'échelle et une difficulté à garantir un service de niveau entreprise pour les problématiques de haute disponibilité et plan de reprise de l'activité, car il faut gérer les multiples plateformes où ces différents langages sont déployés.
Il est aussi important de rationaliser la gestion de la donnée, il est important de s'appuyer sur une architecture multitenant de la BD. Par exemple, la base de données Oracle dispose d'une capacité à gérer plusieurs BD de plusieurs types (Relationnel, Spatial, JSON, Column, Blockchain,...), en instanciant plusieurs PDB (Portable Databases) au sein d'un même CDB (Container Database), en fonction des usages.
Ce modèle permet de garantir un bon niveau d'isolation entre les différentes bases de données et permet surtout de développer plus rapidement des microservices sans avoir à ajouter une complexité technologique liée aux multiples technologies sous-jacentes pour la gestion de la donnée.
Figure 4
Par ailleurs, côté framework de développements, **Spring Boot** reste le framework le plus populaire dans le monde Java pour le développement des microservices.
Spring Boot propose notamment le mode réactif offrir une meilleure expérience à l'utilisateur et garantir une utilisation intelligente des ressources, qui est un facteur déterminant du coût et surtout dans le cloud.
Si vous souhaitez en savoir davantage, je vous recommande le livre de Josh Long (2020) « Reactive Spring ».

Quel découpage en couche choisir au sein d'un microservice ?

Nous pouvons combiner l'architecture hexagonale et les principes du DDD, pour disposer d'un découpage optimal.
Chacune de ces couches à une responsabilité :

- La couche présentation : Elle est responsable de la présentation de l'information à l'utilisateur et de l'interprétation de ces commandes.
  Elle peut exposer des API (REST, gRPC, JMS ou tout par tout autre protocole)
  Il s'agit d'une couche technique, on y trouve les **DTO** (Data Transfer Objects).
  On peut y trouver des fonctions techniques, d'audit et de monitoring (respect des SLA), ou de health-check (up, down, busy, ...)
- La couche application : Elle s'occupe de la coordination. Elle effectue les contrôles de surfaces, mais n'effectue aucun contrôle métier, pour ensuite préparer les données pour la couche domaine. Elle ne contient pas l'état d'un objet, mais peut contenir l'état de progression d'une tâche. Elle est responsable de la gestion des transactions métiers.
- La couche domaine : C'est le cœur métier de l'application. L'état des objets métier y est géré. La persistance des objets métier est déléguée à la couche infrastructure.
- La couche infrastructure : La vraie vie, la couche de geek. Cette couche agit comme une bibliothèque de soutien pour toutes les autres couches. Elle facilite la communication entre les couches, implémente la persistance des objets métier et contient les bibliothèques auxiliaires de la couche présentation.

Dans ce découpage, on utilise le principe d'inversion de dépendances (Un des principes SOLID - **DIP**), qui permet d'avoir un couplage faible entre les composants :

- Le domaine ne dépend d'aucune couche
- L'infrastructure dépend du domaine.
- L'application dépend du domaine
- La présentation dépend de l'application et de l'infrastructure.
- On peut se baser sur l'injection des dépendances - fournie à travers la JSR 330 (dont spring est à l'origine) – afin d'injecter la bonne dépendance à l'exécution.

Le **schéma 5** illustre ce découpage en couche avec Spring Boot.

Comment déployer ces microservices dans le cloud ?

Une application **Cloud Native** est une application qui exploite la puissance du Cloud. Elle tire profit de la conteneurisation, de l'infrastructure as code et des outils **DevOps (CI/CD)**.
Un des facteurs qui ont fait le succès des architectures microservices est qu'elles sont nées grâce au cloud. À titre d'exemple, Netflix a profité des capacités du cloud pour son architecture microservices et se base sur le chaos engineering pour tester la résilience de leur architecture. Bien évidemment ce type de tests n'est pas valable pour tout type de business.
Le paradigme **cloud natif** a permis l'émergence de l'architecture microservices, mais également de nouveaux métiers comme le **SRE** (**S**ite **R**eliability **E**ngineering), qui nécessite une très bonne maîtrise des principes de l'infrastructure programmable (Infrastructure As Code) ou bien le **FinOps** pour optimiser la gestion des coûts du cloud.
Par contre ce modèle pose des challenges aux entreprises qui doivent faire cohabiter des applications on premise et dans le cloud. D'où la nécessité de disposer d'une stratégie cloud

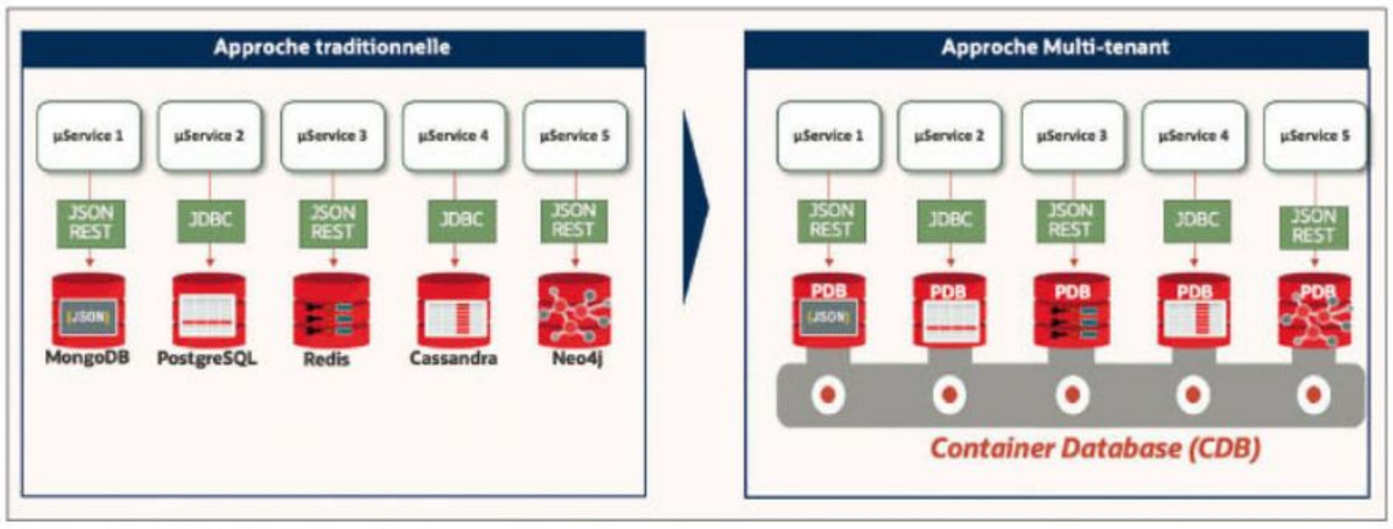


Figure 4 : *Illustration de l'architecture multitenant*

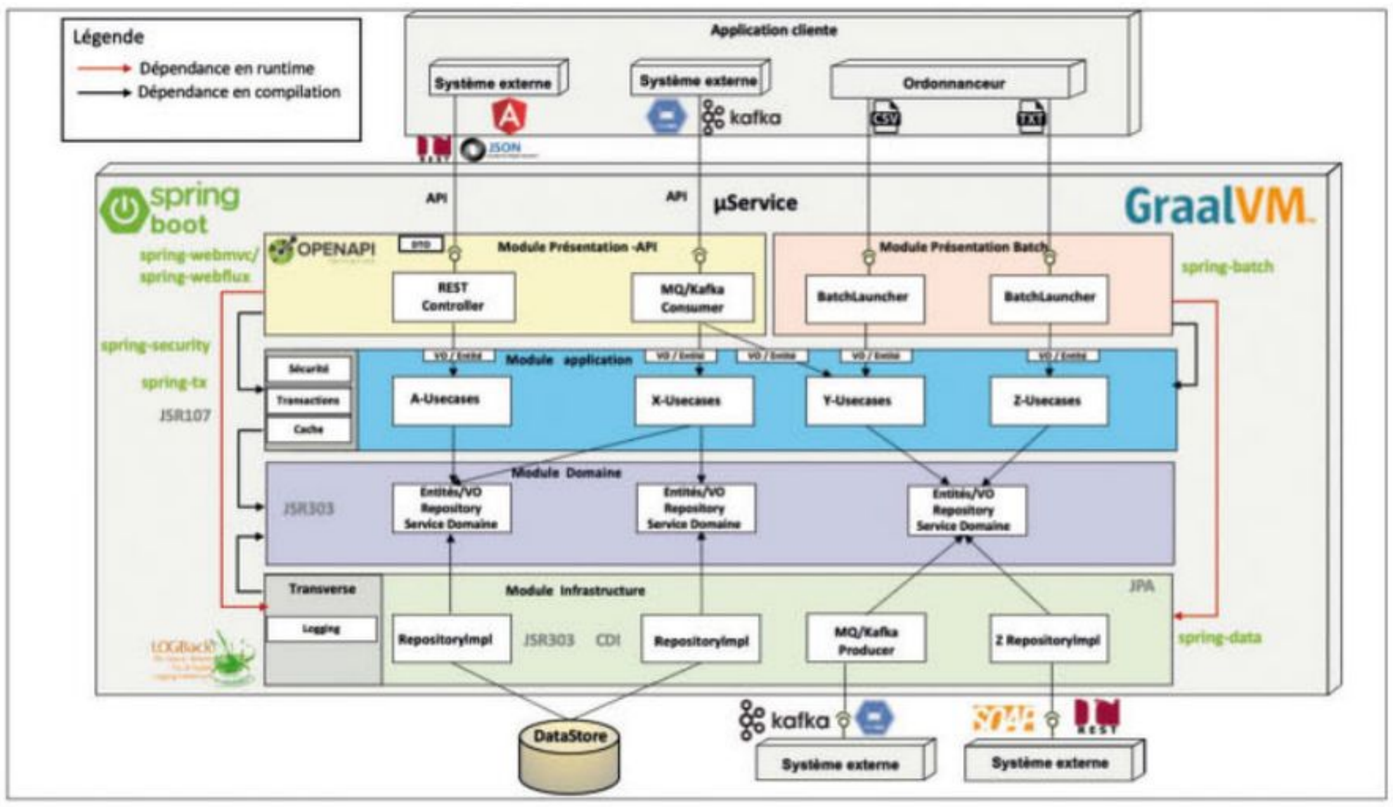


Schéma 5 : *Découpage en couches avec DDD et Spring Boot*

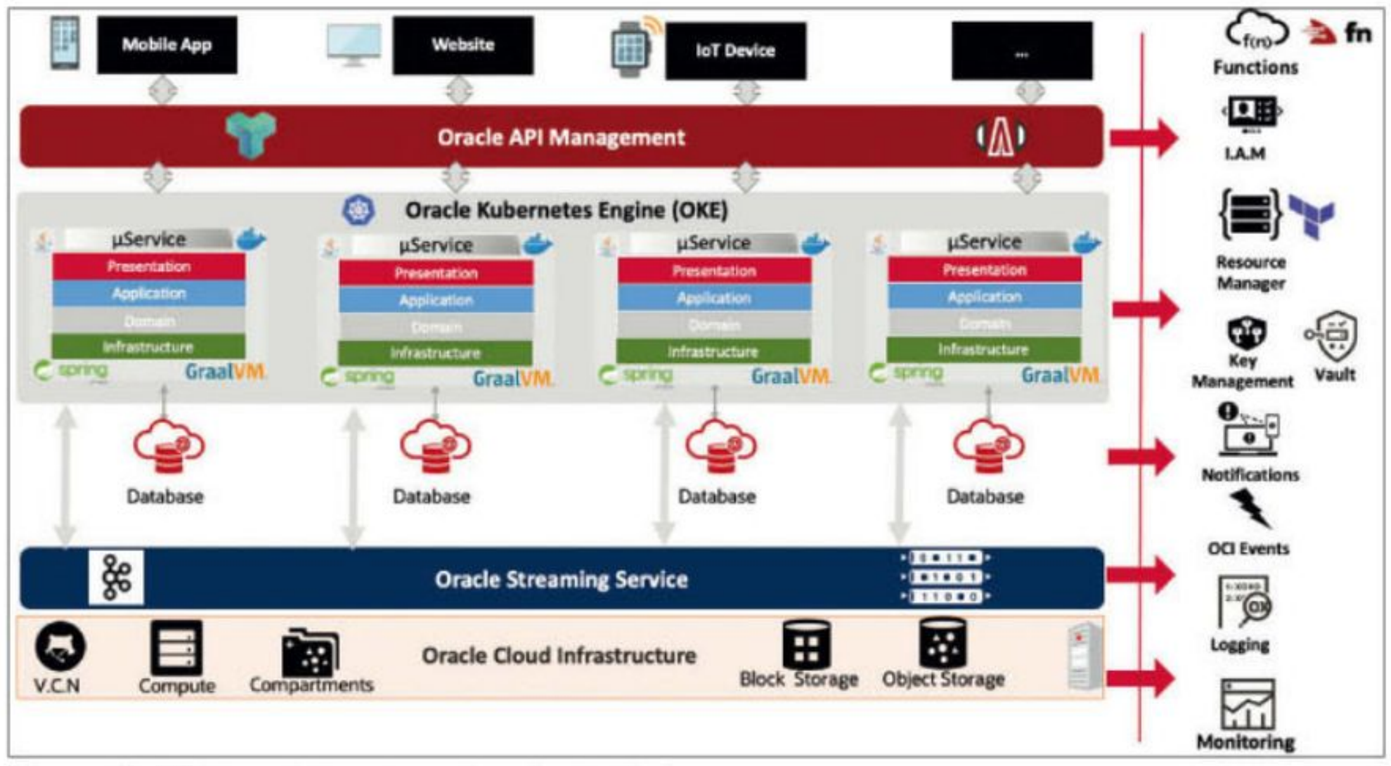


Figure 6 : *Déploiement des microservices dans le cloud*

claire, afin d'éviter d'aggraver la dette technique et empiler une couche technologique supplémentaire.
Cette stratégie doit être drivée par la création de valeur pour le métier et par le gain en termes de T.C.O (Total Cost Ownership) pour l'IT.
Un exemple de déploiement d'une architecture microservices avec Oracle cloud : **Figure 6**

Bonnes pratiques

Le DDD a fait ses preuves sur des projets avec un métier complexe, mais il ne faut pas en abuser (un peu comme tout). Le DDD n'est pas la seule façon de faire. Il est impor-

tant de rester DDD pragmatique par rapport aux contextes d'une application.

Voici un ensemble de bonnes pratiques que je souhaite partager avec vous :

- Définir des SLA pour vos API : Définir un temps de réponse moyen pour chaque API.
- Veiller à garantir le niveau de disponibilité optimale du service en fonction des contraintes de RTO et RPO.
- Veiller sur le niveau de la consistance des données (temps réel, J+1,...)
- Mettre en place une gouvernance des services et des données, en Build et en Run
- Accompagner les différentes parties prenantes : DDD nécessite de disposer de bons développeurs et d'un métier qui adhère aux principes de l'agile.
- Utiliser la loi de Conway: le paramètre organisationnel est très important dans la conception d'un microservice.
- Privilégier des équipes en mode produit : « TWO PIZZA TEAM », semble en moyenne un bon compromis.
- Implémenter un **circuit breaker** pour éviter les erreurs en cascade. Des Frameworks existent, comme **resilience4j.**
- Utiliser le **service mesh** quand c'est opportun : par exemple pour avoir du **mTLS** entre microservices.
- Privilégier des images natives : le projet qui se démarque est **GraalVM.** Il permet d'avoir une JVM polyglotte, de déployer des images natives et de garantir des améliorations des temps de démarrage et une réduction de l'empreinte mémoire des programmes.
- Privilégier la chorégraphie à l'orchestration pour la gestion des opérations transactionnelles.
- Centraliser les logs demeure indispensable pour le monitoring dans une architecture distribuée.
- Automatiser les tests et les contrôles de qualité de code.
- Privilégier les standards pour les échanges : CNCF Cloud Events / OpenAPI

L'architecture microservice n'est pas une solution qui va résoudre tous les problèmes. Parfois, une simple montée en version de l'application monolithique, peut s'avérer une meilleure solution. Fondamentalement, utilisez un cloud public autant que possible et privilégiez un PaaS réversible.

Compiler et exécuter du CoBOL avec GraalVM Community Édition, la machine virtuelle polyglotte par Oracle, et GnuCOBOL

Cet article a pour but d'expliquer comment GraalVM peut compiler et lancer un programme CoBOL sur presque toutes les plateformes sans package manager, uniquement avec GraalVM et son package LLVM. En tant qu'ancien développeur mainframe/CoBOL, j'aimerais voir ce type de VM moderniser le CoBOL existant dans des environnements ouverts.

Christophe Brun

Ancien développeur mainframe reconverti au Python, Java, JavaScript. Il a depuis fondé PapIT en 2017, https://papit.fr, société de développement logiciel et intégration logiciel présente dans les domaines de la logistique, de la domotique, de l'énergie et de la construction.

GraalVM supporte de nombreux langages, Java, JavaScript, Ruby, Python, R, WebAssembly, C/C++, selon le site web d'Oracle (https://www.graalvm.org/docs/why-graal/). La documentation Oracle (https://www.graalvm.org/uploads/graalvm-language-level- virtualization-oracle-tech-papers.pdf) explique que la VM peut interpréter du code natif avec le compilateur Low-Level Virtual Machine (LLVM). Ces codes natifs sont C, C++, FORTRAN, Rust, COBOL, et Go. Cependant, concernant CoBOL, il n'y a pas plus d'explication.

Modernisation du CoBOL

Les services informatiques, particulièrement dans la finance, les administrations cherchent désespérément des experts CoBOL et mainframe, mais le manque de formations et l'UX inspirée du minitel de Z/OS TSO n'encourage pas les vocations. Un autre point noir de ces technologies est le coût des licences. Je pense, j'espère, que ces problèmes pourraient se résoudre en portant le CoBOL existant dans des environnements plus modernes. Si vous pensez que le code existant peut être réécrit pour éteindre les mainframes, lisez https://thenewstack.io/cobol-everywhere-will-maintain/ qui explique pourquoi c'est quasi impossible.

Exécution de code natif GraalVM

L'installation est décrite sur le site https://www.graalvm.org/ docs/getting-started/. L'exécution de code natif utilise un package appelé llvm-toolchain voir : https://www.graalvm.org/docs/reference-manual/languages/llvm/#llvm-toolchain. LLVM étant déjà sur mon système, les commandes clang et lli existent déjà. J'ai donc créé les symlinks g-clang, g-llc, g-llvm-as et g-lli pour accéder aux deux LLVM. L'installation de ce package installe la version 10.0.0-4 (Optimized build).

GnuCOBOL

Grâce à Flex pour l'analyse lexicale et bison le compilateur de compilateur, GnuCOBOL transpile le CoBOL en C. L'actuelle version 3 implémente entre autres une partie des standards COBOL 85, 2002, 2014 et 202X. GNUCoBOL supporte de nombreuses extensions propriétaires apparues tout au long de l'histoire de CoBOL comme MF for Micro Focus IBM, MVS, BS2000, ACU, RM, REALIA. La bêta de la version 4 promet une continuité et une évolution du support des extensions de CoBOL.

Il peut directement compiler un exécutable grâce à la toolchain de votre plateforme, mais ce n'est pas notre but ici, car nous souhaitons l'exécuter dans GraalVM comme indiqué dans la documentation Oracle. Ce compilateur et sa librairie libcob peuvent être compilés avec Clang de GraalVM que l'on vient d'installer. La dernière version est sur le site officiel (https://sourceforge.net/p/gnucobol/code/HEAD/tree/trunk/). L'autoconfiguration du build se fait grâce au script shell:

```
sh ./autogen.sh
```

Configuration avec Clang de GraalVM (sans Berkeley DB support dans notre exemple) :

```
./configure --with-cc=g-clang --without-db

Build et installation avec make:

make install
```

Ceci installe la version 3.1.2.0 de GNU CoBOL.

Compilation du CoBOL en C, puis représentation intermédiaire LLVM et son exécution

La représentation Intermédiaire LLVM offre un ensemble d'instructions indépendant de tout langage et de tout système. Prenons l'exemple de l'ensemble de Mandelbrot, voire mandelbrotset.cbl dans le dépôt de code.

Génération de l'intermédiaire en C

Avec l'exécutable GNU CoBOL, le code C peut être produit avec la commande suivante :

```
cobc -C -x mandelbrotset.cbl
```

Compilation du C en représentation intermédiaire (IR) LLVM

Un point qui n'est pas détaillé dans la documentation Oracle est le bénéfice de LLVM et comment exécuter le code. Un

binaire peut-être généré directement avec Clang ou GCC. Petite précision, pour compiler, il ne faut pas oublier d'inclure la dépendance libcob avec l'argument -lcob.
Mais le vrai bénéfice de LLVM vient de la représentation intermédiaire, un code interprétable et donc portable qui peut être exécuté directement ou compilé sur toute plateforme ou dans notre cas la machine virtuelle Graal. Il peut être utilisé dans un environnement de développement ou de tests et compilé sur la machine de production. Pour compiler en IR :

```
g-clang mandelbrotset.c -S -emit-llvm -o "bin/mandelbrotset.ll"
```

Exécution de l'IR dans l'interpréteur LLVM

L'interpréteur LLVM peut exécuter ce code en chargeant la dépendance libcob de GNUCobol précédemment compilé :

```
g-lli -load /usr/local/lib/libcob.so ./bin/mandelbrotset.ll
```

Performance de l'IR GraalVM

Pour faire la comparaison avec la LLVM classique, la version 10.0.0 a été récupérée du dépôt officiel https://github.com/llvm/ llvm-project/tree/llvmorg-10.0.0. La compilation est configurée avec le moteur de production Ninja et les paramètres suivants pour obtenir un *optimized build* de la même version :

```
cmake -GNinja ../llvm -DCMAKE_BUILD_TYPE=Release\
-DLLVM_ENABLE_ASSERTIONS=off
```

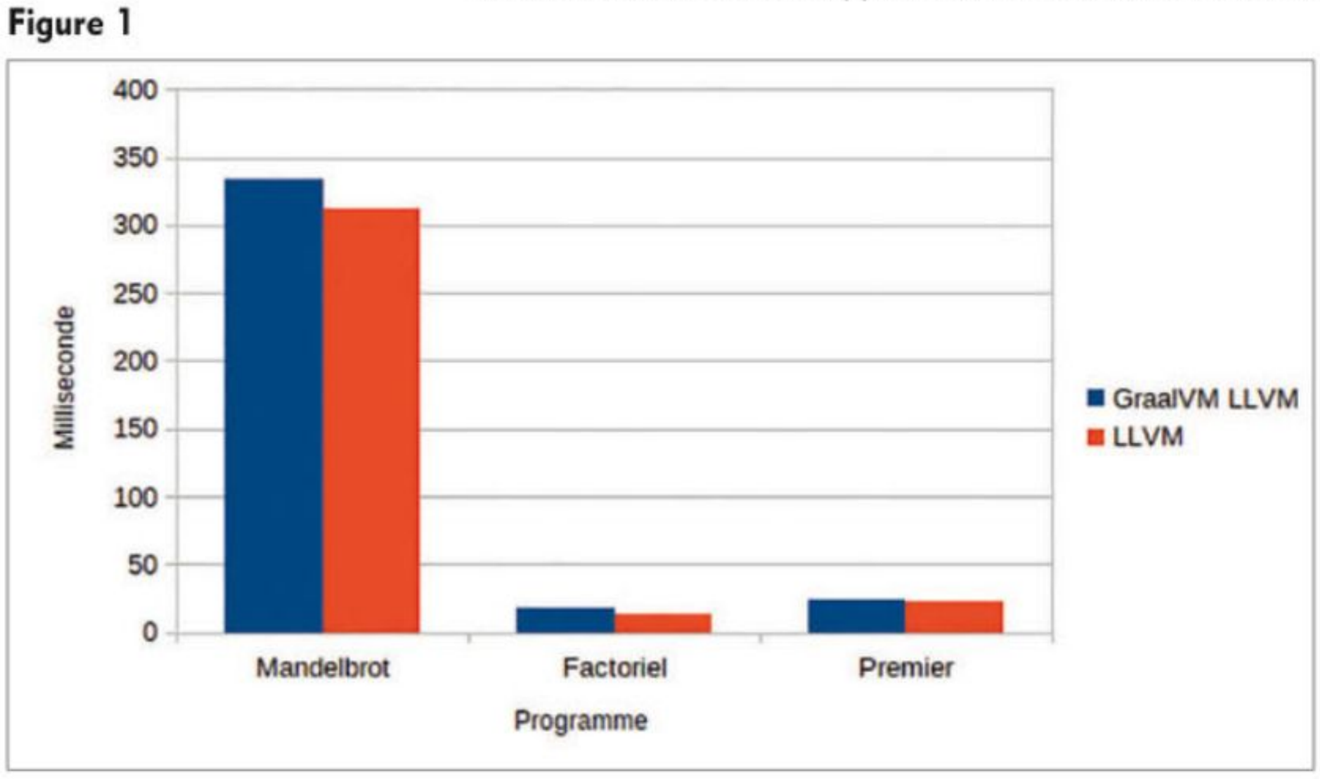
LLVM est ensuite compilé et installé avec la commande ninja install.
Tous les IR générés ont pu être lancés avec succès par l'interpréteur LLVM classique. Les performances de la LLVM de GraalVM se sont avérées identiques.
Le dépôt https://github.com/phe-sto/CoBOL-GraalVM contient actuellement les 4 programmes de benchmarks suivants:

- L'ensemble de Mandelbrot
- Les 1899 premiers nombres premiers par le crible d'Eratosthène
- Les 15 premiers nombres de la suite de Fibonnacci (ne compile plus avec GNU CoBOL 3.x)
- Les factoriels jusqu'à 16. **Figure 1**

Séparation des environnements de développement et production

La plupart des processus de développement logiciels actuels impliquent des environnements bien distincts. L'environnement du développeur tourne sous Windows ou un Linux convivial avec des outils comme des IDE modernes et ergonomiques, celui des serveurs de production est le plus souvent un Linux minimaliste.
A l'heure actuelle, beaucoup de développements CoBOL se font encore sur le mainframe, dans le TSO, un terminal des années 60 qui rappelle le minitel.
Avec GraalVM et son package LLVM le pool de code CoBOL à maintenir pourrait être développé et testé sur la machine du développeur.

Figure 1



Portabilité de l'IR

Sa portabilité assure que ce code peut être exécuté sur toute autre machine possédant GraalVM et son package LLVM, dans un environnement de production par exemple. De plus, la représentation intermédiaire reste un code lisible, pas des plus conviviales mais cela reste un code source avec tous ses avantages.
Jugez par vous même le *hello world* de l'article https://www.informit.com/articles/article.aspx?p=1215438&seqNum=2.
Il peut donc être versionné comme du code classique, partagé entre collègues et mis en production sur une autre machine sans modification, ni même compilation.

Compilation de l'IR

La compilation de l'IR en bytecode se fait avec l'assembleur LLVM et sa commande :

```
g-llvm-as bin/mandelbrotset.ll -o bin/mandelbrotset.bc
```

Le bytecode lui aussi peut être lancé par l'interpréteur g-lli avec la dépendance libcob :

```
g-lli -load /usr/local/lib/libcob.so ./bin/mandelbrotset.ll.
```

Mais l'IR peut tourner directement dans un environnement de production, le plus souvent sans perte de performance significative. Aucune différence significative en termes de temps d'exécution n'a été relevée dans nos exemples par rapport à une compilation directe avec Clang (de GraalVM) ou l'interprétation du bytecode de l'IR. De nombreux exemples sur le web rapportent même une IR plus rapide que le code natif grâce au compilateur JIT de la LLVM. Pour être certain d'obtenir des performances optimales, il est nécessaire de comparer ces 3 solutions.

Conclusion

Avec le package LLVM, GraalVM permet à la VM d'exécuter tous les langages transpilables en C/C++ en plus de Java et des langages apportés par les packages Python, R et Rubby, JavaScript/Node et WebAssembly. Ces technologies open sources, GraalVM et GNU CoBOL, permettent un développement hors de l'environnement mainframe avec des outils classiques. La production peut elle aussi se libérer du mainframe pour être remplacée par un cloud public, environnement serverless, Faas, etc. C'est là que réside probablement la motivation principale d'Oracle à développer cette technologie : récupérer un maximum de code, y compris le legacy, dans son Cloud. Ce type de cloud public permet un scale up et scale down, c'est à dire concrètement que c'est la consommation des ressources et non une infrastructure qui est facturée. Cela permet minimiser les coûts de production.

Mieux maîtriser la performance applicative avec QuickPerf

Jean Bisutti
Consultant à Zenika Paris,
Java Champion,
Créateur de QuickPerf

zenika
<animés par la passion>

QuickPerf est une librairie de test Java permettant d'évaluer simplement et rapidement des propriétés liées à la performance à l'aide d'annotations, comme le nombre de requêtes envoyé à la base de données ou l'allocation mémoire. Cette librairie va vous permettre de confirmer ou d'infirmer certaines de vos intuitions sur des propriétés de performance. Elle va également vous aider à améliorer ces propriétés. La librairie de test QuickPerf permet aussi d'assurer une non-régression automatisée sur ces propriétés afin de détecter au plus tôt de potentiels goulets d'étranglement de performance.

Un exemple !

La méthode suivante exécute une requête JPA :

```
public List<Joueur> findAll() {
  Query query = entityManager.createQuery("FROM Joueur");
  return query.getResultList();
}
```

À première vue, combien de requêtes SQL sont envoyées à la base de données ? Un SELECT ? Si nous voulions vérifier cette hypothèse, nous pourrions activer les logs Hibernate ou les statistiques Hibernate. Ce travail de vérification pour chaque requête JPA est long et fastidieux. Sur les projets, j'ai remarqué qu'il n'était souvent pas effectué.
Notre application possède un test automatisé permettant de vérifier que la méthode récupère l'ensemble des joueurs de la base de données :

```
@ExpectSelect(1)
@Test
public void should_find_all_players() {

  JoueurRepository joueurRepository = new JoueurRepository(entityManager);

  List<Joueur> players = joueurRepository.findAll();

  assertThat(players).hasSize(2);

}
```

L'annotation QuickPerf ExpectSelect(1) est apposée sur la méthode de test. Elle va permettre de vérifier ou non notre hypothèse.
Exécutons ce test automatisé. Il échoue avec le rapport d'erreur suivant : **Figure 1**

Ce rapport comporte différentes informations :

1. Une propriété liée à la performance a échoué.
2. 3 SELECT sont envoyés à la base de données. Nous avions émis l'hypothèse qu'il y en avait un seul. QuickPerf a analysé le SQL produit par l'application et a vérifié le nombre de SELECT générés.
3. Il est possible qu'un plus grand nombre de requêtes SELECT soit envoyé à la base de données avec la volumétrie de production. Nous sommes alertés que les aller-retour JDBC ont un impact négatif sur les performances. Un lien vers un article de blog détaillant ce point est fourni.
4. QuickPerf a détecté que l'application utilise Hibernate et suggère que nous pourrions faire face à un N+1 SELECT. La librairie propose des solutions pour corriger le N+1 SELECT avec Hibernate. Si l'application utilisait Spring Data JPA, QuickPerf proposerait une solution de correction avec ce framework.
5. Le rapport d'erreur contient les requêtes envoyées à la base de données. Nous pouvons remarquer la présence d'une requête de sélection sur la table Joueur et deux autres sur la table Équipe.

Une fois le N+1 select supprimé, l'annotation ExpectSelect(1) va garantir une non-régression automatisée sur le nombre de requêtes SELECT envoyé à la base de données. Cette annotation peut aussi servir de documentation. En lisant le code d'une méthode de test annotée ExpectSelect(1), nous savons qu'un et un seul SELECT doit être généré.

Annotations SQL

ExpectSelect est une annotation de type SQL. Il en existe d'autres, détaillées dans la documentation de QuickPerf (https://github.com/quick-perf/doc/wiki/SQL-annotations).
Le tableau ci-dessous donne un aperçu de certaines annotations de type SQL :

ExpectJdbcQueryExecution	Vérifie le nombre d'exécutions JDBC
ExpectSelectedColumn	Vérifie le nombre de colonnes sélectionnées
ExpectMaxQueryExecutionTime	Vérifie la durée des requêtes SQL
ExpectJdbcBatching	Vérifie que les INSERT, UPDATE et DELETE sont envoyés par lots à la base (batching JDBC)

L'annotation ExpectMaxQueryExecutionTime doit être utilisée avec une base de données représentative de celle utilisée en production (même volumétrie de données avec les statistiques à jour), et non avec une base mémoire (H2, HSQLDB). Si vous souhaitez utiliser une base embarquée dans un conte-

```
1 java.lang.AssertionError: a performance-related property is not respected

2 [PERF] You may think that <1> select statement was sent to the database
       But there are in fact <3>...

3 💣 You may have even more select statements with production data.
Be careful with the cost of JDBC roundtrips: https://blog.jooq.org/2017/12/18/the-cost-of-jdbc-server-roundtrips/

4 💡 Perhaps you are facing an N+1 select issue
   * With Hibernate, you may fix it by using JOIN FETCH
                                            or LEFT JOIN FETCH
                                            or FetchType.LAZY
                                            or ...
     Some examples: https://stackoverflow.com/questions/32453989/what-is-the-solution-for-the-n1-issue-in-jpa-
and-hibernate";
                    https://stackoverflow.com/questions/52850442/how-to-get-rid-of-n1-with-jpa-criteria-api-in-
hibernate/52945771?stw=2#52945771
                     https://thoughts-on-java.org/jpa-21-entity-graph-part-1-named-entity/

5 [JDBC QUERY EXECUTION (executeQuery, executeBatch, ...)]
  Time:0, Success:True, Type:Prepared, Batch:False, QuerySize:1, BatchSize:0, Query:["
    select
      joueur0_.id as id1_2_,
      joueur0_.equipe_id as equipe_i4_2_,
      joueur0_.NOM as NOM2_2_,
      joueur0_.PRENOM as PRENOM3_2_
    from
      Joueur joueur0_"], Params:[()]

  Time:0, Success:True, Type:Prepared, Batch:False, QuerySize:1, BatchSize:0, Query:["
    select
      equipe0_.id as id1_1_0_,
      equipe0_.championnat_id as champion3_1_0_,
      equipe0_.nom as nom2_1_0_
    from
      Equipe equipe0_
    where
      equipe0_.id=?"], Params:[(1)]

  Time:0, Success:True, Type:Prepared, Batch:False, QuerySize:1, BatchSize:0, Query:["
    select
      equipe0_.id as id1_1_0_,
      equipe0_.championnat_id as champion3_1_0_,
      equipe0_.nom as nom2_1_0_
    from
      Equipe equipe0_
    where
      equipe0_.id=?"], Params:[(2)]
```

Figure 1

neur Docker, des exemples d'utilisation de QuickPerf avec la librairie Testcontainers (https://www.testcontainers.org/) sont disponibles dans ce repository Github : https://github.com/quick-perf/quickperf-examples. L'utilisation d'une base mémoire pour les autres annotations SQL permet d'avoir des tests plus rapides.

Annotations JVM

Le tableau ci-dessous donne un aperçu de certaines annotations de type JVM :

MeasureHeapAllocation	Mesure l'allocation dans le tas (heap) de la JVM
MeasureRSS	Mesure le Resident Set Size
ProfileJvm	Profile le fonctionnement de la JVM

L'annotation MeasureHeapAllocation va par exemple permettre de mesurer l'allocation mémoire d'une structure de données ou celle d'un batch.
Cette annotation a permis de mesurer l'allocation de distributions de Maven (https://github.com/quick-perf/maven-test-bench).
L'annotation ProfileJvm affiche dans la console des éléments relatifs au fonctionnement de la JVM :

ALLOCATION (estimations)	GARBAGE COLLECTION	THROWABLE
Total : 3,68 GiB	Total pause : 1,264 s	Exception: 0
Inside TLAB : 3,67 GiB	Longest GC pause: 206,519 ms	Error : 36
Outside TLAB: 12,7 MiB	Young: 13	Throwable: 36
Allocation rate: 108.1 MiB/s	Old : 3	

COMPILATION	CODE CACHE
Number : 157	The number of full code cache events: 0
Longest: 1,615 s	

Le résultat du profilage peut être analysé plus en détails à l'aide de l'outil graphique *JDK Mission Control* :

```
[QUICK PERF] JVM was profiled with JDK Flight Recorder (JFR).
The recording file is available here:
 C:\Users\user~1\AppData\Local\Temp\QuickPerf-9292511997956298899\jvm-profiling.jfr
You can open it with JDK Mission Control (JMC).
Where to find JDK Mission Control?  https://tinyurl.com/find-jmc
```

Nous pouvons ainsi par exemple comprendre quelles méthodes allouent le plus.
D'autres annotations JVM sont décrites dans la documentation de QuickPerf :
https://github.com/quick-perf/doc/wiki/JVM-annotations

Portée des annotations

Une annotation peut être ajoutée à une méthode de test. Il est aussi possible de la placer sur une classe de test. L'annotation va alors s'appliquer à l'ensemble des méthodes de test de cette classe.
Supposons que nous souhaitions appliquer des vérifications, comme la détection de N+1 select, à une application possédant un grand nombre de tests automatisés. Nous pouvons configurer des annotations afin qu'elles s'appliquent à l'ensemble des tests QuickPerf de l'application. Pour cela, il est nécessaire de définir une classe implémentant l'interface *SpecifiableGlobalAnnotations* et avec le package *org.quickperf*. Nous parlons alors de portée (scope) globale.
Nous donnons ci-dessous un exemple de configuration d'annotations avec une portée globale :

```
package org.quickperf;

import org.quickperf.config.SpecifiableGlobalAnnotations;

import java.lang.annotation.Annotation;
import java.util.Arrays;
import java.util.Collection;

import static org.quickperf.sql.annotation.SqlAnnotationBuilder.*;


public class QuickPerfConfiguration implements SpecifiableGlobalAnnotations {

  public Collection<Annotation> specifyAnnotationsAppliedOnEachTest() {

    return Arrays.asList(
          // La présence d'un N+1 select peut entrainer la création de plusieurs SELECT du même type avec une valeur de paramètre différent.
          // Dans l'exemple donné au début de cet article, les deux requêtes de sélection sur la table Équipe ne se différencient que par les valeurs
          // l'identifiant de l'équipe (1 et 2).
```

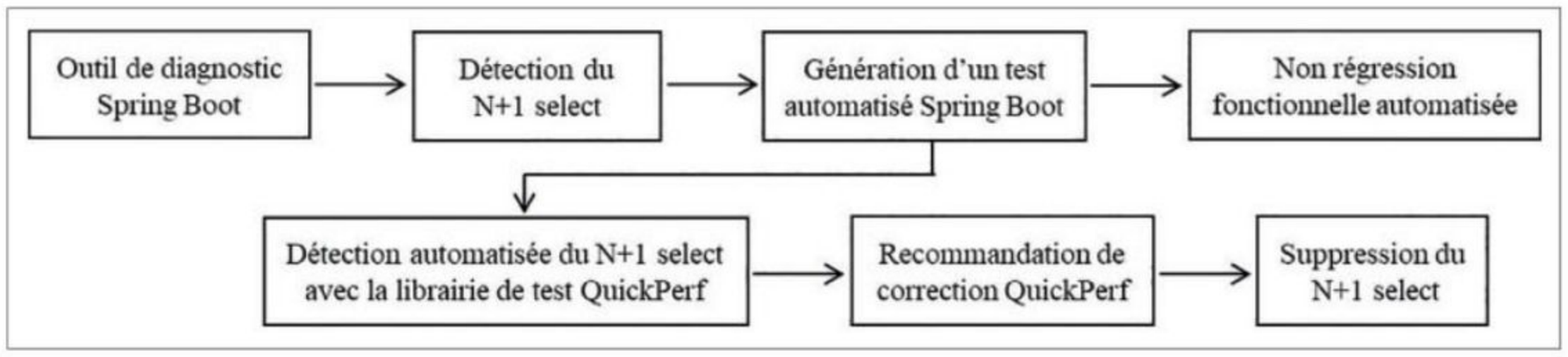


Figure 2

```
        // Par conséquent, l'annotation DisableSameSelectTypesWithDifferent
ParameterValues configurée avec une portée globale permet de détecter des
        // N+1 SELECT sur l'ensemble des tests d'une application.
        disableSameSelectTypesWithDifferentParamValues()

        , expectJdbcBatching()

        , expectMaxQueryExecutionTime(30)

        , disableStatements()

        , disableQueriesWithoutBindParameters()

    );

}

}
```

Prochainement avec QuickPerf

Nous avons vu que QuickPerf permet d'évaluer simplement et rapidement des propriétés impactant la performance applicative par l'ajout d'annotations aux tests automatisés. Nous allons continuer à développer de nouvelles annotations QuickPerf, pour par exemple détecter la non-fermeture d'une connexion à la base après l'exécution de la logique métier (*connection leak*), ou pour repérer des fuites mémoire. Prochainement, nous allons aussi rendre public un outil de diagnostic pour Spring Boot. Lors de l'utilisation d'applications web, il alertera de caractéristiques empêchant l'application d'être performante, comme la présence de N+1 select, des requêtes SQL lentes ou une forte allocation mémoire. Cet outil permettra aussi de générer des tests automatisés QuickPerf pour reproduire et corriger les N+1 select. L'outil de diagnostic sera capable de générer automatiquement un script SQL pour reconstituer un jeu de données en base utilisé par le test automatisé. Améliorer la performance applicative peut-être une bonne chose pour les utilisateurs, encore faut-il ne pas introduire de régression sur les règles fonctionnelles. L'outil de diagnostic permettra aussi que les tests générés assurent une non-régression fonctionnelle. Le risque de régression fonctionnelle est sans doute faible lors de la correction d'un N+1 select, mais il pourrait être important si le code devait être restructuré de manière significative pour, par exemple, diminuer l'empreinte mémoire.

Le diagramme de la **Figure 2** illustre la complémentarité de l'outil de diagnostic Spring Boot et de la librairie de test QuickPerf dans le cas d'un N+1 select.

Si vous souhaitez vous tenir informé de l'actualité de QuickPerf, n'hésitez pas à suivre le compte Twitter @QuickPerf ou à consulter le compte Github de QuickPerf (https://github.com/quick-perf) !

N'ayez plus peur de la programmation non bloquante avec Reactor !

Avec l'amélioration continue des techniques et des besoins, la programmation réactive est devenue la référence pour les programmes demandant disponibilité, résilience, souplesse et répondant à des événements asynchrones. Elle est d'autant plus une référence lorsque l'on souhaite répartir la charge et utiliser les ressources tout en permettant de mieux découpler le code d'une application. L'écosystème Java, toujours soucieux de permettre aux développeurs de créer des applications performantes, se devait de suivre le mouvement, et ça n'a pas manqué !

Antoine MICHAUD

Consultant Senior Java

www.invivoo.com

Jeune, dynamique, et passionné depuis son enfance, Antoine veille aux tendances et nouveautés de Java. Après une carrière dans l'armée, il a rejoint l'expertise Java d'Invivoo Bordeaux en 2019 où il a effectué une mission chez Dolby. Il est en ce moment chez ManoMano.

INVIVOO
BEYOND TECH

Après la publication de la première version du *Manifeste Réactif* en 2013 par Jonas Bonér, plusieurs librairies sont apparues pour profiter de ce nouveau modèle de programmation, dont certaines sont particulièrement devenues populaires :

- RxJava (**R**eactive e**X**tension for **Java**): principalement connu sur Android, elle se distingue par sa compatibilité officielle avec les différentes versions du SDK et utilise le design pattern Observable
- Project Reactor: venant de la communauté Spring, elle en est le cœur de WebFlux (qui permet le développement d'API réactives et non bloquantes), implémente la spécification Reactive Streams et profite de l'avantage d'être basée sur les APIJava (CompletableFuture, Stream, Duration, ExecutorService, Flow...) et utilise aussi le design pattern Observable
- Eclipse Vert.X: utilisée par beaucoup de grandes enseignes, et à l'époque faisant partie des plus performantes. Cette librairie se distingue par sa modularité et sa simplicité, et intègre un bus d'événements permettant de centraliser et multiplexer les messages

Ces différentes librairies évoluent, et la différence de performance est de moins en moins présente, voire quasi inexistante. De plus, chacune d'entre elles s'importe très facilement par un simple ajout d'une dépendance.

Nous allons utiliser le projet Reactor qui m'a permis d'aborder très sereinement ces nouvelles contraintes lorsque j'y ai été confronté chez ManoMano.

Reactor, kézako ?

La programmation réactive a pour base de concept chez Reactor *un flux d'éléments, fini ou infini*, représenté par deux classes principales :

- *Mono* : flux d'éléments pouvant en émettre un seul au maximum, ayant donc une cardinalité de 0 à 1
- *Flux* : flux d'éléments pouvant en émettre à l'infini, ayant une cardinalité de 0 à N

Chacun de ces deux types de flux représente un **Publisher** et Reactor apporte tout un tas de méthodes nommées *"opérateurs"*, permettant la manipulation des éléments émis par ces flux afin de les regrouper, les diviser, les différer, les filtrer, les convertir, et bien plus encore... Il vous permet aussi de gérer les erreurs (je sais, personne n'est parfait) qui peuvent survenir pendant le traitement de votre flux en vous donnant la possibilité de les ignorer, de les convertir en un autre élément, etc. Ces outils permettent donc une manipulation simple et très lisible (syntaxe encourageant la programmation fonctionnelle) des flux « en mode asynchrone » et Reactor, de par sa structure, vous fait gagner en disponibilité, résilience et souplesse de manière transparente !

Dans un premier temps, je vais vous exposer le contexte dans lequel Reactor sera mis en œuvre avec le code final (pour attiser votre curiosité) et ensuite je vais le décrire pas-à-pas. Au travers de cette description, je souhaite vous apporter une meilleure compréhension des problématiques soulevées par la programmation réactive et vous montrer comment Reactor vous permet de les maîtriser facilement pour que vous n'ayez plus aucune excuse pour vous y mettre ! Et si cette librairie vous intrigue, je vous indiquerai quelques concepts avancés pour d'aller plus loin !

Le cas d'usage

Prenons le côté positif d'un sujet d'actualité : le nombre de guéris du Covid-19. L'objectif va être d'appeler une API pour récupérer le nombre de guéris d'hier et d'avant-hier en parallèle pour en afficher la différence le plus rapidement possible, tout en faisant attention à une possible erreur de format de données ou de réseau. Pour se faire, nous devons l'appeler deux fois : une pour chaque date (hier et avant-hier).

L'API en question est coronavirusapi-france.now.sh qui expose plusieurs routes permettant d'obtenir les statistiques du covid-19 de manière globale et mondiale, par pays, par département... Et par date, ce qui nous intéresse donc ici.

En se basant sur le 15 novembre 2020, la route à appeler serait : GET coronavirusapi-france.now.sh/FranceGlobalDataByDate?date=2020-11-15, et le résultat en JSON serait par exemple :

```
{
 "FranceGlobalDataByDate" : [
  {
   "date" : "2020-11-15",
   "sourceType" : "ministere-sante",
```

```
  "gueris" : 1111
  [...]
 },
 {
  "date" : "2020-11-15",
  "gueris" : 2222,
  "sourceType" : "opencovid19-fr"
  [...]
 }
]
}
```

Le JSON obtenu nous permet de savoir le nombre absolu de guéris en fonction des sources, comme les données open data de opencovid19-fr, ou du Ministère de la santé. Ce sera cette dernière source que nous allons traiter, en ignorant les autres. Voici le code final permettant de faire notre calcul :

```
@SpringBootApplication
public class LesGuerisDuCovid19 implements CommandLineRunner {
  private static final Logger log = LoggerFactory.getLogger(LesGuerisDuCovid19.class);
  private static final DateTimeFormatter DATE_PARAM_FORMATTER = DateTime
Formatter.ISO_DATE;
  private static final String SOURCE_INTERESSANTE = "ministere-sante";

  private final WebClient httpClient = WebClient.create("https://coronavirusapi-france
.now.sh");

  public static void main(final String[] args) {
    var app = new SpringApplication(LesGuerisDuCovid19.class);
    app.setWebApplicationType(WebApplicationType.NONE);// Pour ne pas lancer le
serveur web
    app.run(args);
  }

  @Override
  public void run(final String... args) {
    var hier = LocalDate.now().minusDays(1);
    var avantHier = hier.minusDays(1);

    Integer nouveauxGueris = Mono.zip(getNombreGueris(hier, SOURCE_INTERESSANTE),
getNombreGueris(avantHier, SOURCE_INTERESSANTE)) // 8
        .doFirst(() -> log.info("Récupération des données entre hier et avant-hier...")) // 9
        .map(tuple -> tuple.getT1() - tuple.getT2()) // 10
        .onErrorResume(error -> Mono.fromRunnable(() -> log.error("Problème lors
de la récupération des données: {}", error.getMessage()))) // 11
        .block(); // 12
    // .... À vous d'écrire la suite ....
  }

  private Mono<Integer> getNombreGueris(final LocalDate date, final String source) {
    return httpClient.get()
          .uri(uri -> uri.path("/FranceGlobalDataByDate").queryParam("date", DATE
_PARAM_FORMATTER.format(date)).build())
          .retrieve()
          .bodyToMono(JsonNode.class) // 1
          .doFirst(() -> log.info("Récupération des données pour le {} sur la source
'{}'...", date, source)) // 2.a
          .doOnNext(jsonData -> log.info("Données reçues: {}", jsonData.toPretty
String())) // 2.b
          .map(jsonData -> jsonData.get("FranceGlobalDataByDate")) // 3
        .flatMapMany(dataByDate -> Flux.range(0, dataByDate.size()).map(dataBy
Date::get)) // 4
          .filter(dataByDate -> source.equals(dataByDate.get("sourceType").text
Value())).next() // 5
          .flatMap(data -> Mono.justOrEmpty(data.get("gueris"))).map(JsonNode
::intValue) // 6
          .switchIfEmpty(Mono.error(new RuntimeException(String.format("Pas de
données pour le %s sur la source '%s' !", date, source)))); // 7
  }
}
```

En résumé, getNombreGueris(...) va récupérer la valeur du nombre de guéris dans le résultat (en JSON) de la réponse de l'API, et émettre une erreur si jamais ce champ n'est pas trouvé. La méthode run()est le point d'entrée de notre programme, qui va appeler getNombreGueris pour la date d'hier et d'avant-hier en parallèle, afin de pouvoir calculer la différence de guéris et l'afficher dans la console.

Dans un premier temps, je vais vous expliquer comment initialiser votre projet en ajoutant les dépendances nécessaires pour utiliser Reactor et Spring WebFlux. Ensuite nous verrons comment récupérer le nombre de guéris pour une date donnée. Pour finir, j'expliquerai comment combiner le résultat des deux appels.

Dépendance

La première chose à faire est d'ajouter les dépendances nécessaires pour utiliser Reactor, ainsi que WebClient, une fonctionnalité de Spring nous permettant d'appeler une API de manière asynchrone, basé sur... Reactor ! Quelle coïncidence ;) Il vous faudra donc ajouter qu'une seule dépendance à votre projet :

- Avec gradle

```
implementation group: 'org.springframework.boot', name: 'spring-boot-starter-
webflux', version: '2.4.0'
```

- Avec maven

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-webflux</artifactId>
 <version>2.4.0</version>
</dependency>
```

Explications pas-à-pas

Nous allons partir d'une base très simple, en laissant Spring boot configurer le nécessaire. Ici, l'auto-configuration va surtout nous servir à avoir un format de logs sympa, avec le niveau par défaut à INFO.

```
@SpringBootApplication
public class LesGuerisDuCovid19 implements CommandLineRunner {
 private static final Logger log = LoggerFactory.getLogger(LesGuerisDuCovid19.class);
 public static void main(final String[] args) {
   var app = new SpringApplication(LesGuerisDuCovid19.class);
   app.setWebApplicationType(WebApplicationType.NONE);// Pour ne pas lancer le
serveur web
   app.run(args);
 }
```

```
  @Override
  public void run(final String... args) {
   // Notre code à exécuter ici
  }
}
```

1

Pour faire notre requête HTTP vers l'API, il va falloir initialiser un WebClient en donnant l'url de base, ici https://coronavirusapi-france.now.sh:

```
WebClient clientHttp = WebClient.create("https://coronavirusapi-france.now.sh")
```

WebClient nous propose une abstraction de tous les verbes HTTP habituels, à savoir GET, POST, PUT, DELETE, etc... Ici, nous allons utiliser GET pour le chemin /FranceGlobalDataByDate qui attend le paramètre de requête date formatée en date ISO (aaaa-MM-jj), et récupérer le JSON sorti tout droit de l'API:

```
Mono<JsonNode> resultatJson = httpClient.get()
      .uri(uri -> uri.path("/FranceGlobalDataByDate").queryParam("date", DateTime
Formatter.ISO_DATE.format(date)).build())
      .retrieve()
      .bodyToMono(JsonNode.class);
```

Si nous voulons passer plus de paramètres de requête, il suffit de chaîner les appels avec .queryParam("param", value)
JsonNode.class pourrait être remplacé par FranceGlobalData.class où FranceGlobalData serait un simple POJO, représentant le schéma du JSON à extraire, plutôt que d'utiliser JsonNode qui est générique et pourra représenter n'importe quel schéma JSON.
Si la réponse est totalement vide (pas de JSON retournée), le Mono retourné sera lui aussi vide, mais chut ! Un peu de suspens avant d'en parler.

2.a

Il est possible d'effectuer une action lorsqu'un flux se déclenche (autrement dit lors d'un *subscribe*), avant tout le reste. Ici, on veut écrire un log juste avant d'exécuter la requête HTTP :

```
.doFirst(() -> log.info("Récupération des données pour le {} sur la source '{}'...", date,
source))
```

2.b

Mais on peut aussi effectuer une action lorsqu'un élément est émis dans un flux. Ici, on veut écrire un log juste après avoir exécuté la requête HTTP :

```
.doOnNext(jsonData -> log.info("Données reçues: {}", jsonData.toPrettyString()))
```

3

Nous avons dans le Mono l'objet JSON suivant :

```
{
 "FranceGlobalDataByDate" : [ ... ]
}
```

Nous allons mapper la donnée reçue pour récupérer le contenu du champ FranceGlobalDataByDate:

```
.map(jsonData -> jsonData.get("FranceGlobalDataByDate"))
```

4

Nous avons maintenant dans le Mono l'élément JSON suivant (un tableau d'objets) :

```
[
 {
 "date" : "aaaa-MM-jj",
 "sourceType" : "xxx",
  "gueris" : 1111
  [...]
 },
 {
  ...
 }
]
```

Nous allons convertir le Mono émettant le tableau d'objets vers un Flux d'objets, rendant bien plus simple les futures opérations pour chaque objet. La particularité du JsonNode est qu'il ne fournit pas directement une liste ou un stream de tous les objets. Pour avoir accès à chacun d'eux, on aurait naturellement fait :

```
for (int i = 0; i < dataByDate.size(); i++){
 var object = dataByDate.get(i);
}
```

Reactor nous permet de générer un flux d'index où nous allons par la suite mapper chaque index vers une case de notre tableau, et c'est plus concis :

```
Flux.range(0, dataByDate.size()).map(dataByDate::get)
```

Enfin, pour convertir le Mono vers un Flux, il faut utiliser .flatMapMany. Voici à quoi ressemble le code assemblé :

```
.flatMapMany(dataByDate -> Flux.range(0, dataByDate.size()).map(dataByDate::get))
```

Si le tableau est vide, le Flux retourné sera lui aussi vide. Le suspense est encore de la partie... L'explication arrive!

5

Nous avons maintenant dans le Flux aucun à plusieurs objets JSON de type :

```
{
 "date" : "aaaa-MM-jj",
 "sourceType" : "xxx",
 "gueris" : 1111
 [...]
}
```

Ce que nous voulons, c'est seulement récupérer les données du *Ministère de la santé*. Il va donc falloir filtrer avec .filter le flux en ne récupérant que les objets dont le champ sourceType contient la valeur ministere-sante.
Comme chaque tableau n'a qu'un seul objet par type de source, nous prendrons uniquement le premier qui satisfait cette condition avec .next, ce qui aura pour effet de convertir le Flux en Mono . Voici le code :

```
.filter(dataByDate -> source.equals(dataByDate.get("sourceType").textValue())).next()
```

Si aucun élément ne correspond à notre source, le Mono retourné sera vide, il pourra donc être géré par la suite avec des opérateurs spécifiques ! Mais... encore du suspens #roulementDeTambours

6

Nous avons maintenant dans le Mono un objet JSON suivant :

```
{
 "date" : "aaaa-MM-jj",
 "sourceType" : "ministere-sante",
 "gueris" : 1111
 [...]
}
```

Nous allons mapper l'objet pour récupérer le contenu du champ gueris :

```
.map(data -> data.get("gueris").intValue())
```

Le problème du .map avec Reactor, c'est qu'il n'accepte pas que la valeur obtenue dans le mapper soit null.
Pour des raisons d'apprentissage, on va faire *comme si* le champ gueris pouvait être null. Pour gérer ce cas, le seul moyen est d'utiliser le .flatMap qui permet de mapper un élément vers 0 à 1 élément (et 0 à N éléments pour un Flux). Pour représenter 0 élément, il faut utiliser Mono.empty() et pour 1 élément, nous avons Mono.just(valeur):

```
.flatMap(jsonData -> jsonData.get("gueris") == null ? Mono.empty() : Mono.just(json
Data.get("gueris")))
```

Mais Reactor ne s'arrête pas là. On peut simplifier notre ternaire en un seul morceau avec Mono.justOrEmpty(valeur). Si la valeur est nulle, elle est mappée toute seule vers un Mono.empty() !

```
.flatMap(jsonData -> Mono.justOrEmpty(jsonData.get("gueris")))
```

Ce code aurait pu être écrit avec un Optional de cette manière :

```
.map(jsonData -> Optional.ofNullable(jsonData.get("gueris")))
.filter(Optional::isPresent)
.map(Optional::get)
```

Avec la gestion du champ null, le code sera donc le suivant :

```
.flatMap(data -> Mono.justOrEmpty(data.get("gueris"))).map(JsonNode::intValue)
```

Oui, encore un Mono pouvant être vide si le champ gueris est vide ou n'existe pas.

7

À plusieurs reprises, je vous ai dit que nous pouvions obtenir un Mono ou un Flux vide. Et nous allons gérer cette particularité en générant une erreur, non pas avec throw, mais avec Mono.error.
En déclenchant cette erreur, n'importe quel opérateur qui interviendrait après serait totalement ignoré (excepté certains qu'on va voir juste après), permettant de court-circuiter un Mono/Flux :

```
.switchIfEmpty(Mono.error(new RuntimeException()))
.doOnNext(v -> log.info("Voici un log qui ne sera pas écrit si aucun élément est émis"))
```

Cependant, on pourrait très bien imaginer retourner une valeur par défaut -1, faisable de cette manière (et dans ce cas, les opérateurs qui suivront pouront intervenir sur notre valeur par défaut) :

```
.defaultIfEmpty(-1)
```

Avec un message d'erreur plus parlant, voici le code :

```
.switchIfEmpty(Mono.error(new RuntimeException(String.format("Pas de données
pour le %s sur la source '%s' !", date, source))))
```

8

Maintenant que nous avons extrait la logique de récupération d'un nombre de guéris en fonction d'une date et d'une source dans un Mono<Integer>, nous pouvons récupérer ce nombre pour la date d'hier et d'avant-hier. Le mieux serait de déclencher en parallèle les deux requêtes et joindre les résultats ensemble, même si elles ne répondent pas à la même vitesse. Et c'est faisable facilement à l'aide de Mono.zip(requete1, requete2), qui va prendre nos deux appels, et joindre les résultats dans un Tuple2<Resultat1, Resultat2>, où tuple.getT1() correspondra au résultat de l'appel en 1re position dans le .zip, et tuple.getT2() correspondra à celui de l'appel en 2e position. Si un appel déclenche une erreur, l'autre appel sera annulé et l'erreur sera retransmise dans la suite des opérateurs. Si un appel ne retourne rien, l'autre appel sera annulé, mais aucune erreur ne sera déclenchée (le Mono retourné par le .zip sera annulé)
Il est possible de combiner les résultats autrement qu'avec un tuple, mais ce cas, bien qu'il soit intéressant, ne sera pas abordé. Voici le code, aussi simple qu'efficace, permettant de récupérer le nombre de guéris d'hier et d'avant hier pour la source ministere-sante, et qui nous retournera un Mono<Tuple2<Integer, Integer>>:

```
var hier = LocalDate.now().minusDays(1);
var avantHier = hier.minusDays(1);
Mono.zip(getNombreGueris(hier, "ministere-sante"), getNombreGueris(avantHier,
"ministere-sante"))
```

9

Mettons une trace écrite, qui sera affichée **avant** le .doFirst de chaque appel (voir le pas #2.a) :

```
.doFirst(() -> log.info("Récupération des données entre hier et avant-hier..."))
```

10

Lorsque les deux requêtes sont terminées, nous obtenons un tuple contenant le nombre de guéris d'hier (T1) et avant-hier (T2). Pour récupérer le nombre de nouveaux guéris, il faut simplement soustraire les guéris d'hier à ceux d'avant-hier:

```
.map(tuple -> tuple.getT1() - tuple.getT2())
```

11

Pour la gestion de notre erreur précédente, nous pouvons convertir une erreur en un autre Mono (ou un autre Flux lorsque l'erreur est attrapée dans un flux) en utilisant .onErrorResume.
Dans notre cas, nous allons nous contenter d'écrire un log

avec le message d'erreur sans émettre d'élément supplémentaire, et donc le Mono retourné sera vide :

```
.onErrorResume(error -> Mono.fromRunnable(() -> log.error("Problème lors de la récupération des données: {}", error.getMessage())))
```

12

Voici la méthode **la plus importante** et souvent la plus oubliée :

```
.subscribe();
```

Ou celle-ci :

```
var value = mono.block();
```

Imaginez-vous dans votre vieille Clio par temps pluvieux, vos essuie-glaces ne font pas leur travail, car vous ne les avez pas enclenchés (oui, il fut un temps où ils n'étaient pas automatiques). Ici, c'est le même problème : si nous n'appelons pas cette méthode, le flux ne sera pas lancé. Cependant, il existe une réelle différence entre ces deux méthodes :

- .subscribe() va déclencher le flux de manière asynchrone et non bloquante, et le retour de cette méthode sera donc immédiat. Voici un exemple qui va d'abord afficher Hello ! puis ensuite Valeur reçut :

```
void run() {
 mono.doOnNext(value -> log.info("Valeur reçue")).subscribe();
 log.info("Hello !");
}
```

- .block() va déclencher le flux, mais le retour de cette méthode sera synchrone, c'est-à-dire **lorsque le flux sera terminé**. Comme son nom l'indique, elle va bloquer la méthode run() jusqu'à ce que le Mono se complète. Voici un exemple qui va d'abord afficher Valeur reçue puis ensuite Hello ! :

```
void run() {
 var value = mono.doOnNext(value -> log.info("Valeur reçue")).block();
 log.info("Hello !");
}
```

Dans notre cas, nous voulons récupérer le nombre de nouveaux guéris dans une variable pour vous laisser la possibilité d'imaginer une suite non-réactive. Mais attention, si une erreur est déclenchée, nous avons choisi d'avoir un log, et de renvoyer un Mono vide. Et si le Mono est vide, la valeur retournée par .block() sera null :

```
Integer nouveauxGueris = mono.block();
```

Si jamais vous appelez 10 fois la méthode block ou subscribe, le publisher va s'exécuter 10 fois ! Et c'est là tout l'intérêt des **Cold publishers**

Comment tester ?

Reactor met à disposition l'objet StepVerifier qui permet de tester les différentes étapes de manipulation de votre flux. Voici quelques use cases avec le code pour tester :

- un Flux ou un Mono ne devant retourner aucune valeur :

```
StepVerifier.create(mono).verifyComplete();
```

- un Flux ou un Mono devant retourner exactement une valeur :

```
Object expectedValue = ...;
StepVerifier.create(mono).expectNext(expectedValue).verifyComplete();
```

- un Flux devant retourner plusieurs valeurs, dans l'ordre donné :

```
StepVerifier.create(flux).expectNext(value1).expectNext(value2).expectNext(value3)
.verifyComplete();
```

- un Flux ou un Mono devant déclencher une erreur :

```
StepVerifier.create(Mono.empty()).expectError().verify();
```

Comment faire un debug ?

Le debug de Reactor peut être compliqué lorsqu'il s'agit d'une erreur remontée en plein milieu de la chaîne d'opérateurs. En effet, comme l'exception est déclenchée dans une suite d'appels au sein d'un Mono ou d'un Flux (car ce n'est pas votre code qui appelle directement votre lambda présent dans un map par exemple, mais celui de Reactor), on aura une stacktrace pas très claire !

Prenons le code suivant, où l'on retourne une valeur nulle dans un map, ce qui est formellement interdit :

```
public class Lanceur {
    public static void main(String[] args) {
     Service.disBonjour().block();
    }

}

class Service {
    static Mono<String> disBonjour() {
        return Mono.just("hello")
                .map(t -> null);
    }
}
```

Pour obtenir de plus amples informations non pas sur l'exécution, mais l'emplacement des opérateurs où ils ont été assemblés, il suffit juste d'avoir la dépendance :
io.projectreactor:reactor-tools:3.4.0
Et d'avoir cette ligne avant le code à debug (en première ligne de notre main()) :

```
ReactorDebugAgent.init();
```

Cela va préfixer notre stacktrace avec l'emplacement de l'assemblage (ou déclaration) de nos opérateurs :

```
Exception in thread "main" java.lang.NullPointerException: The mapper returned a null value.
    at java.base/java.util.Objects.requireNonNull(Objects.java:246)
    Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Assembly trace from producer [reactor.core.publisher.MonoMap] :
    reactor.core.publisher.Mono.map
    Service.execution(Lanceur.java:16)
Error has been observed at the following site(s):
    |_ Mono.map ⇢ at Service.execution(Lanceur.java:16)
Stack trace:
        at java.base/java.util.Objects.requireNonNull(Objects.java:246)
        at reactor.core.publisher.FluxMap$MapSubscriber.onNext(FluxMap.java:106)
```

```
        at reactor.core.publisher.Operators$ScalarSubscription.request(Operators.java:2346)
        at reactor.core.publisher.FluxMap$MapSubscriber.request(FluxMap.java:162)
        at reactor.core.publisher.BlockingSingleSubscriber.onSubscribe(BlockingSingle
Subscriber.java:50)
        at reactor.core.publisher.FluxMap$MapSubscriber.onSubscribe(FluxMap.java:92)
        at reactor.core.publisher.MonoJust.subscribe(MonoJust.java:54)
        at reactor.core.publisher.Mono.subscribe(Mono.java:3987)
        at reactor.core.publisher.Mono.block(Mono.java:1678)
        at Lanceur.main(Lanceur.java:8)
    Suppressed: java.lang.Exception: #block terminated with an error
        at reactor.core.publisher.BlockingSingleSubscriber.blockingGet(BlockingSingleSubs
criber.java:99)
        at reactor.core.publisher.Mono.block(Mono.java:1679)
        at Lanceur.main(Lanceur.java:8)
```

Conclusion

Grâce à ce tutoriel, vous allez pouvoir appeler des micro-services JSON en parallèle, et manipuler les données de manière réactive en un temps record, tout en ayant la main sur le debug. Comme nous avons pu le voir, on peut convertir un élément vers d'autre(s) élément(s). Mais ces éléments pourraient très bien provenir d'une autre API réactive, d'une requête SQL, une entrée utilisateur, et bien encore... C'est la puissance de Reactor : tout est flux de données, et quand nous utilisons une méthode permettant d'obtenir un Mono ou un Flux, ce n'est pas la source de la donnée qui nous intéresse, mais la ou les valeurs obtenues. Reactor va naturellement imposer cette abstraction adaptable dans tous les cas possibles. Bien évidemment, nous avons à peine frôlé la puissance et la flexibilité de Reactor, mais vous avez les éléments de base pour vous lancer.

Aller plus loin

Nous avons vu toutes ces méthodes : map, flatMap, flatMapMany, doFirst, doOnNext, switchIfEmpty, onErrorResume, filter, Mono.just, Mono.justOrEmpty, Mono.zip, Flux.range, subscribe et block... Mais il existe près de 400 méthodes rien que pour Flux et 200 pour Mono !
Je vous laisse avec quelques concepts à découvrir :

- Les Sinks : une extension de Mono et Flux, n'émettant qu'une seule fois chaque élément, qu'il y ait des Subscribers ou non
- Les Schedulers : permet de gérer le multithreading de votre appli
- Retry : relance un traitement lors d'une erreur (réseau coupé par exemple)
- Mono.using() et Flux.using() : Try-with-resource réactif
- … pour le reste, à vous de jouer !

TL;DR;

Reactor en quelques lignes :

- À quoi ça sert : faciliter la mise en place d'applications réactives et non bloquantes sous forme de flux d'événements
- Facilité d'intégration : une dépendance principale pour le cœur, et une par module (client web, serveur web, client Kafka ...), voire une seule lors de l'utilisation avec Spring
- Facilité d'implémentation : écriture fonctionnelle, seulement 2 types centraux (Mono et Flux) qui partagent les mêmes opérateurs (map, flatMap, zip...)
- Large communauté : projet porté par Spring, donc beaucoup de développeurs actifs
- Design pattern Obvservable
- Utilise les API officielles de la JDK (reactive streams, Duration, ...)

Je vous laisse, mais pour ceux qui veulent jouer un peu, je vous soumets un petit quizz, pour voir si vous avez bien suivi :

- Que renvoie le code suivant ?

```
Mono.just(42).map(v -> v * 2);
```

R: rien, car ni .block(), ni .subscribe() a été appelé donc la chaîne n'a pas été déclenchée

- Que va contenir la variable value ?

```
var mono = Mono.just(42)
mono.map(v -> v * 2);
var value = mono.block();
```

R: 42, il aurait fallu faire mono = mono.map(v -> v * 2); (ou chaîner les appels) pour obtenir 84

- Prenons la méthode returnMono(int value): Mono<Integer>. Quelle est la bonne manière de mapper une valeur d'un Mono<Integer> avec cette méthode ?
  - 1: mono.map(this::returnMono)
  - 2: mono.flatMap(this::returnMono)

R: 2. Avec 1. nous obtiendrions un Mono<Mono<Integer>>

- Que peut-on faire quand une erreur est déclenchée dans un Mono ?
  - 1: rien
  - 2: émettre un élément
  - 3: appeler un microservice HTTP
  - 4: appeler une BDD
  - 5: écrire un log

R: Toutes les réponses sont justes, car une erreur peut être convertie en Mono, et un Mono peut être vide, ou avec une valeur provenant d'une BDD ou d'un MS, et bien encore...

Abonnez-vous à Programmez! Abonnez-vous à Programmez! Abonne

PROGRAMMEZ!
Le magazine des développeurs

nos classiques

1 an → 10 numéros (6 numéros + 4 hors séries) — 49€*

2 ans → 20 numéros (12 numéros + 8 hors séries) — 79€*

Etudiant
1 an → 10 numéros (6 numéros + 4 hors séries) — 39€*

Option : accès aux archives — 19€

* *Tarifs France métropolitaine*

abonnement numérique

PDF ..... 39€
1 an → 10 numéros
(6 numéros + 4 hors séries)

Souscription uniquement sur
www.programmez.com

OFFRES 2021

Profitez dès aujourd'hui de nos offres d'abonnements.

1 an soit 18 numéros en tout

Programmez! + Technosaures + Pharaon Magazine
+ carte PybStick + accès aux archives :

89€*
au lieu de 137 €

1 an soit 14 numéros

Programmez! + Technosaures + carte PybStick :

75€*
au lieu de 93 €

1 an soit 10 numéros

Programmez! + carte PybStick :

55€*
au lieu de 63 €

(*) Tarifs France. Dans la limite des stocks disponibles de la PybStick.
Ces offres peuvent s'arrêter à tout moment. Sans préavis.

Toutes nos offres sur www.programmez.com

Oui, je m'abonne

ABONNEMENT à retourner avec votre règlement à :
PROGRAMMEZ, Service Abonnements
57 Rue de Gisors, 95300 Pontoise

- ☐ **Abonnement 1 an :** 49 €
- ☐ **Abonnement 2 ans :** 79 €
- ☐ **Abonnement 1 an Etudiant :** 39 €
  Photocopie de la carte d'étudiant à joindre
- ☐ **Option :** accès aux archives 19 €

- ☐ **Abonnement 1 an :** 89 €
  Programmez! + Technosaures + Pharaon Magazine + carte PybStick + accès aux archives
- ☐ **Abonnement 1 an :** 75 €
  Programmez! + Technosaures + carte PybStick
- ☐ **Abonnement 1 an :** 55 €
  Programmez! + carte PybStick

☐ Mme ☐ M. Entreprise : ..... Fonction : .....

Prénom : ..... Nom : .....

Adresse : .....

Code postal : ..... Ville : .....

Adresse email indispensable pour la gestion de votre abonnement

E-mail : ..... @ .....

☐ Je joins mon règlement par chèque à l'ordre de Programmez !

☐ Je souhaite régler à réception de facture

*** Tarifs France métropolitaine**

Offres pouvant s'arrêter à tout moment, sans préavis **PROG Spécial 4**

Les anciens numéros de PROGRAMMEZ!

Le magazine des développeurs

Complétez votre collection....

Tarif unitaire 6,5 € (frais postaux inclus)

TECHNOSAURES

Le magazine à remonter le temps !

Prix unitaire : **7,66 €** (frais postaux inclus)

N°4 Standard **10 €**

N°4 Deluxe **15 €**

☐ 234	: __ ex	☐ 240	: __ ex
☐ 235	: __ ex	☐ 241	: __ ex
☐ 236	: __ ex	☐ HS1 été 2020	: __ ex
☐ 238	: __ ex	☐ 242	: __ ex
☐ 239	: __ ex	☐ 246	: __ ex

soit __ exemplaires x 6,50 € = ____ €

Technosaures ☐ N°1 ☐ N°2 ☐ N°3 ☐ N°5 ☐ N°6

soit __ exemplaires x 7,66 € = ____ €

☐ N°4 Deluxe ..... 15 €

☐ N°4 Standard ..... 10 €

soit au **TOTAL** = ____ €

Commande à envoyer à :
Programmez!
57 rue de Gisors
95300 Pontoise

☐ 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 4**

LE REFACTORING DE CODE LEGACY

Bruno Boucard

@brunoboucard

Co-fondateur de la société 42skillz (42skillz.com), c'est aussi un développeur depuis plus de 30 ans. Il possède une longue expérience à travers différents langages de programmation et systèmes d'exploitation. Aujourd'hui, il est à la fois coach agile et coach technique, formateur sur les pratiques XP, speaker international et organisateur du meetup BDD Paris.

Le refactoring de code legacy est une activité peu pratiquée chez les développeurs, car malheureusement peu enseigné. Cependant si vous souhaitez vous initier, sachez que son apprentissage réclame discipline et prudence. Dans cet article je présenterai, quelques éléments qui vous permettront de mieux appréhender la démarche pour remanier un code en profondeur, en toute sécurité. J'utiliserai indifféremment les mots, « remaniement de code » et « refactoring de code » pour désigner la modification de code afin de l'améliorer sans jamais altérer son comportement.

Attention : Tous mes propos dans cet article ne sont pas propres à un langage de programmation. Les exemples de code sont écrits en Java, mais vous pouvez facilement les retranscrire dans votre langage de programmation préféré. Professionnellement, je pratique les langages C++, C#, Java, JavaScript et dernièrement Python et j'applique les principes et les pratiques de l'eXtreme Programing (XP) avec chacun sans difficulté.

Les enjeux associés au remaniement de code

Avant de nous plonger dans l'art du remaniement de code, intéressons-nous aux motivations qui devraient nous conduire à un refactoring de code. La raison essentielle doit être à la suite d'une demande métier, par exemple une évolution fonctionnelle ou une nouvelle fonctionnalité. Interrogez-vous, si votre base de code permet d'envisager sans risque, l'implémentation de cette demande métier. Étudier l'impact de cette demande vis-à-vis de la qualité des codes présents. Si les codes sources souffrent d'un manque de design ou d'un nommage désastreux, ou si les comportements métier ne sont pas rattachés aux bonnes entités, alors il est peut-être risqué de faire cette évolution. Enfin, si le nommage des variables n'est pas suffisamment explicite, alors la capacité à ajouter cette nouvelle fonctionnalité sera compliquée, car le code est difficile à comprendre. Cette revue de code devrait potentiellement déclencher des idées de remaniement de code afin de mitiger les risques de briser un comportement existant.

Si vous pratiquez le Test-Driven Development (TDD), vous pratiquez une forme de refactoring. Cette phase arrive lorsque le code produit devient difficile à comprendre et ne permet plus de développer naïvement. C'est le moment où nous décidons de remanier le code afin de produire les prochains tests au même rythme que les tout premiers. C'est donc bien dans le souci de produire une nouvelle fonctionnalité que nous pratiquons le remaniement de code.

Si votre application contient du code dont le design est catastrophique, mais dont le métier ne réclame aucun changement, alors ne vous lancez pas dans le refactoring de ce code. Il fonctionne et donne satisfaction à vos utilisateurs. Dans ce cas, garder votre énergie dans l'apport de plus de valeur métier dans les fonctionnalités réclamées.

Comment pratiquer le refactoring de code

Si les enjeux d'un refactoring sont avérés, alors vous devez vous assurer d'un remaniement de code à la fois efficace et sans risque. Voici les six étapes qui vous permettront de faire du refactoring en toute sécurité :

- Comprendre les comportements relatifs du code à remanier.
- Comprendre l'architecture du code à remanier.
- Utiliser les meilleurs outils de refactoring.
- Construire un harnais de test (harnais de sécurité) avant de modifier le code.
- Repérer les codes smells en visitant les fichiers à remanier.
- Appliquer le remaniement du code.

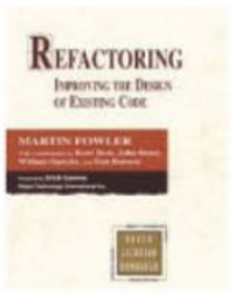
Comprendre le métier relatif du code à remanier

Avant de se lancer dans un refactoring de code, le ou les développeurs devraient rafraîchir leur mémoire auprès des experts métier. L'objectif est de porter des actions de remaniement de code en toute sécurité en ayant compris les enjeux métier. *Si vous comprenez parfaitement votre contexte métier, il devient plus simple de prendre des décisions de design*.

Pour illustrer notre propos, je vous propose un code kata tiré de la première édition de l'ouvrage Refactoring : Improving the Design of Existing Code – Martin Fowler. Pour rappel, celui-ci a pour objet d'imprimer un ticket affichant les films loués.

Le programme imprime un ticket affichant les films loués par un client et pour combien de temps, il calcule ensuite les frais, qui dépendent de la durée de location du film pour chacun des types de films loués. Il existe trois types de films : ***Regular***, ***Children*** et ***NewRelease***. En plus de calculer les frais, le programme évalue également les points de fidélité de

location, qui varient selon que le type de film loué est de type ***NewRelease***.



L'objectif du *Product Owner* est de préparer l'implémentation de l'impression du ticket de location au format HTML dans un futur proche. Le but n'est pas de réaliser l'implémentation du support HTML, mais bien de préparer son arrivée. À terme le format texte devra coexister avec le nouveau format HTML.

Comprendre l'architecture du code à remanier

Le programme est composé de trois classes :

- La classe *Customer* représente le client louant des films et réclamant un ticket de location incarné par la méthode *statement()*. Elle maintient une liste de location de type *Rental*.
- La classe *Rental* représente une location et gère le nombre de jours de location via le champ *daysRented* de type entier. Elle contient, une instance de film de type *Movie*.
- La classe Movie représente le film loué, elle contient un champ nommé *priceCode*, représentant le type de film : *Regular*, *Children* et *NewRelease*.

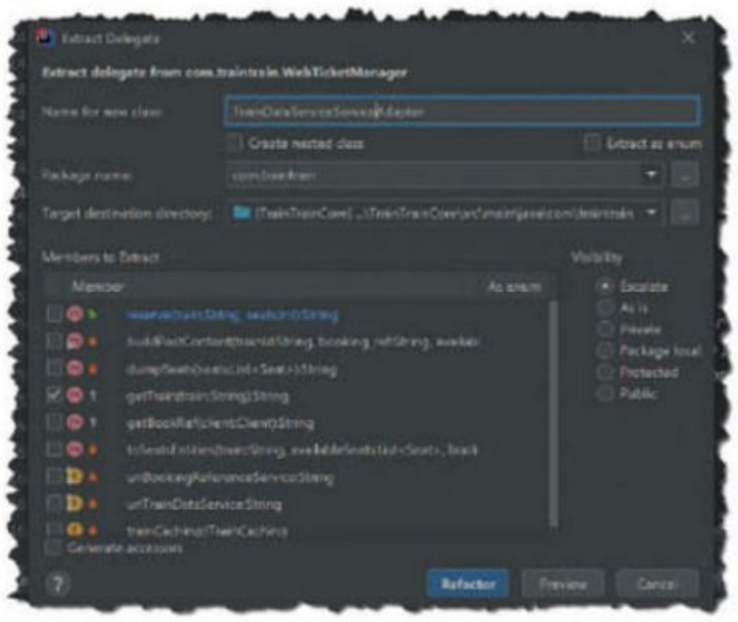
Voici le diagramme de classes UML synthétisant les relations entre *Customer*, *Rental* et *Movie*. **Figure 1**

La décomposition des entités au regard du contexte métier semble légitime. Cependant, on note que seule que la classe Customer contient une méthode nommée statement, alors que les classes Rental et Movie exposent des champs et aucune méthode. Il n'y a donc pas de comportement pour ces dernières, ce qui est peut-être le signe d'un mauvais design. Nous reviendrons sur ce point dans un instant.

Utiliser les meilleurs outils de refactoring

En fonction du langage de programmation, il existe des outils spécifiques au refactoring de code. Pour les cas de Java et .NET voici les outils conseillés. :

En Java : **IntelliJ IDEA** est pour moi le meilleur outil pour le remaniement de code.



Intellij IDEA est sans doute le premier outil à avoir implémenté toutes les actions de refactoring. Si vous souhaitez urbaniser votre code Java, je vous invite à l'utiliser, car il offre à ce jour le meilleur support des actions de refactoring définies dans l'ouvrage : Improving the Design of Existing Code – Martin Fowler.

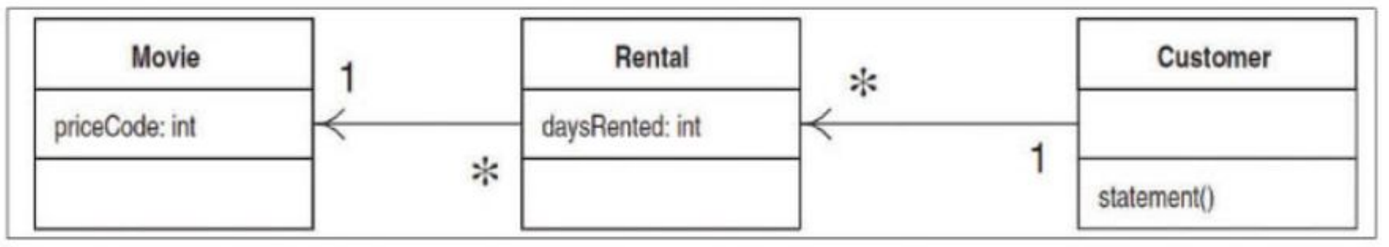


Figure 1

Construire un harnais de test avant de modifier le code

La construction d'un *harnais de test* est obligatoire. Pour rappel : *Faire du refactoring, c'est modifier du code sans jamais modifier les comportements associés*. Le *harnais de test* est notre seul recours pour constater un dysfonctionnement, suite à une action de refactoring. De plus, il est important de considérer les tests comme faisant partie intégrante des codes sources. Les noms des tests doivent explicitement révéler les comportements métiers. Enfin, les noms doivent contenir les conséquences d'une action. La lecture du test doit permettre de comprendre, sans ambiguïté, sa motivation. Code source du *harnais test* :

```
public class CustomerShould {
  @Test
  public final void print_statement_when_rent_a_regular_movie_for_3_days() {
    Customer vincent = new Customer("Vincent");
    vincent.addRental(new Rental(new Movie("Star wars", Movie.REGULAR), 3));
    assertThat(vincent.statement())
        .isEqualTo("Rental Record for Vincent\n" +
            "\tStar wars\t3.5\n" +
            "Amount owed is 3.5\n" +
            "You earned 1 frequent renter points");
  }

  @Test
  public final void print_statement_when_rent_a_new_release_movie_for_4_days() {
    Customer vincent = new Customer("Vincent");
    vincent.addRental(new Rental(new Movie("No Time to Die", Movie.NEW_
RELEASE), 4));
    assertThat(vincent.statement())
        .isEqualTo("Rental Record for Vincent\n" +
            "\tNo Time to Die\t12.0\n" +
            "Amount owed is 12.0\n" +
            "You earned 2 frequent renter points");
  }

  @Test
  public final void print_statement_when_rent_a_children_movie_for_5_days() {
    Customer vincent = new Customer("Vincent");
    vincent.addRental(new Rental(new Movie("Frozen 2", Movie.CHILDREN), 5));
    assertThat(vincent.statement())
        .isEqualTo("Rental Record for Vincent\n" +
            "\tFrozen 2\t4.5\n" +
            "Amount owed is 4.5\n" +
            "You earned 1 frequent renter points");
  }
```

```
@Test
public final void print_statement_when_rent_several_movies() {
    Customer vincent = new Customer("Vincent");

    Movie star_wars = new Movie("Star wars", Movie.NEW_RELEASE);
    star_wars.setPriceCode(Movie.REGULAR);
    vincent.addRental(new Rental(star_wars, 1));
    vincent.addRental(new Rental(new Movie("No Time to Die", Movie.NEW_
RELEASE), 2));
    vincent.addRental(new Rental(new Movie("Frozen 2", Movie.CHILDREN), 5));

    assertThat(vincent.statement())
        .isEqualTo("Rental Record for Vincent\n" +
            "\tStar wars\t2.0\n" +
            "\tNo Time to Die\t6.0\n" +
            "\tFrozen 2\t4.5\n" +
            "Amount owed is 12.5\n" +
            "You earned 4 frequent renter points");
  }
}
```

Une fois le harnais de test terminé, vous devez le lancer après chaque action de refactoring. L'objectif du harnais de test est de vous fournir un feed-back permanent. C'est votre principale source de feed-back et elle doit être la plus courte possible. L'objectif est de corriger au plus tôt une potentielle erreur de remaniement. Ainsi vous gagnerez en sécurité et vous obtiendrez une plus grande confiance dans votre développement. Ci-dessous, une illustration de la fenêtre des tests dans IntelliJ IDEA. **Figure 2**

Il est très important de relancer le harnais de test après chaque modification afin de constater que rien n'est cassé. Si vous oubliez de lancer votre harnais de test régulièrement, vous risquez de réaliser un peu tard que vous avez cassé un ou plusieurs comportements, et vous perdrez du temps à comprendre quelle modification de code qui a occasionné le problème.

Ne vivez pas dangereusement

Si vous n'êtes pas convaincu par l'importance d'un harnais de test, que pensez-vous de casser un comportement métier au sein de votre application et sans même le savoir ? Gagner la confiance de vos utilisateurs en produisant des applications propres et soignées est sans doute la meilleure récompense dans notre métier. Alors, ne vivez pas dangereusement et construisez un harnais de test dès que vous en avez besoin.

Les vertus du harnais de test

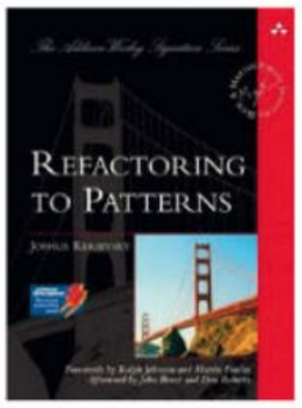
Vous protéger contre une erreur de remaniement.
Mieux comprendre le fonctionnement attendu lorsque vous le construisez.
Gagner en confiance sur votre capacité à ajouter une nouvelle fonctionnalité. Un harnais de test bien écrit est aussi une forme de documentation vivante comme décrit dans l'approche Behaviour-Driven Development (BDD).

Addiction à la rapidité exécution

Dans un remaniement de code, toute la valeur de votre harnais de test passe dans son utilisation systématique. C'est votre principale source de feed-back et elle doit être la plus courte possible. Avoir une attitude paranoïaque vis-à-vis du code en cours de remédiation est légitime, rassurez-vous en validant chacune de vos actions de remédiation en lançant votre harnais de test. Il y a pourtant un cas où créer un harnais de test peut s'avérer être compliqué à construire : les dépendances externes. Et il n'est pas rare que le code contienne de nombreuses dépendances externes qui vous empêchent de tester facilement. Quelques exemples de dépendances externes : Base de données, Connexion réseau, Système de fichiers, Singleton, Méthode statique, API externe…
Sachez qu'avec du temps, de la technique et un peu de courage, il est parfaitement possible de venir à bout des dépendances externes. Je reviendrai dans le cadre d'un autre article, sur l'art de casser des dépendances externes.

Repérer les codes smells en visitant les fichiers à remanier

On appelle mouvement de refactoring, une action permettant de corriger une erreur de design (code smell). Il existe un catalogue des mouvements de refactoring tiré du dernier ouvrage « Refactoring » de Martin Fowler : https://refactoring.com/catalog/



Avant de débuter tous les mouvements de refactoring nécessaires, il est essentiel de faire le tour du propriétaire afin de repérer les codes smells présents. Les codes smells (décris en annexe) constituent les défauts de design qui ne permettent pas d'appréhender sans risque une

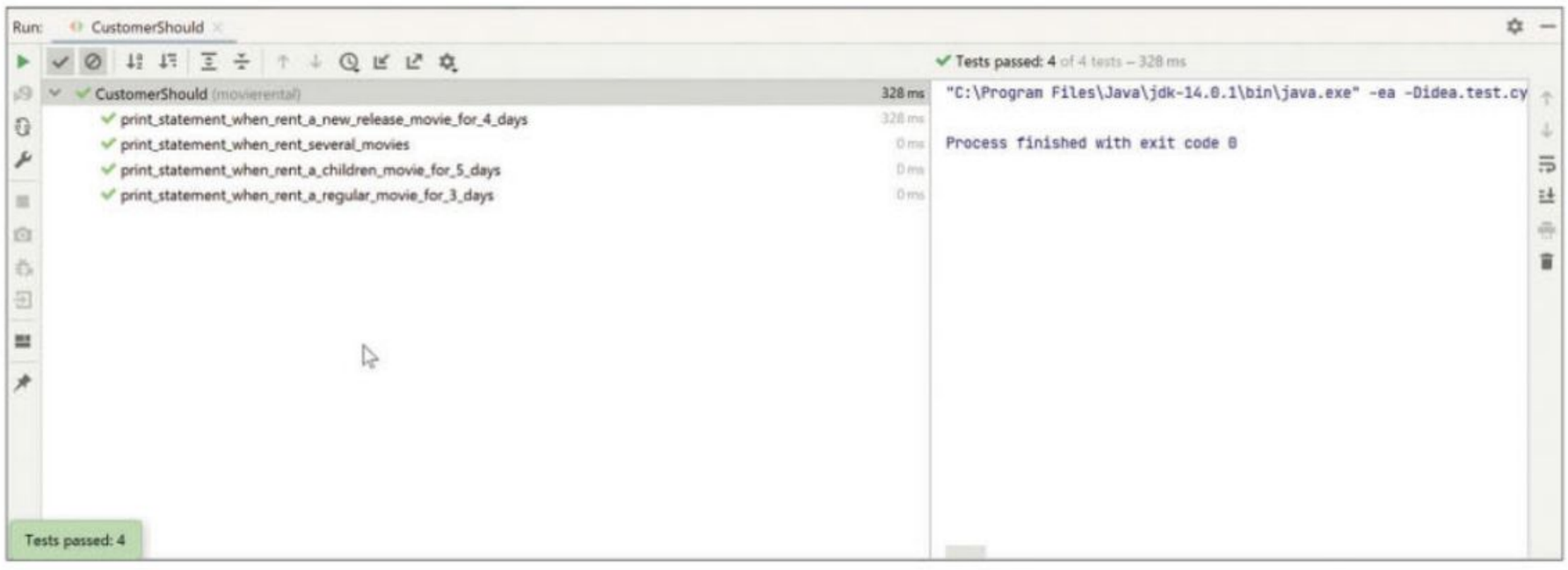


Figure 2

évolution ou une nouvelle fonctionnalité. Bien les connaître permet de remanier le code efficacement.
Tous les codes smells sont des réactions à des anti-patterns de code. Il y a donc une correspondance entre les codes smells et les modèles de design. Les modèles de design sont décomposables en mouvements unitaires de refactoring. L'ouvrage de *Joshua Kerievsky, Refactoring To Patterns* est la référence sur ce sujet.

Visite de la classe Rental

Lorsqu'une classe ne contient pas de méthode, mais que des accesseurs, alors nous pouvons qualifier cette classe d'anémique.

```
public class Rental {

  private Movie _movie;
  private int _daysRented;

  public Rental(Movie movie, int daysRented) {
    _movie = movie;
    _daysRented = daysRented;
  }

  public int getDaysRented() {
    return _daysRented;
  }
  public Movie getMovie() {
    return _movie;
  }
}
```

Visite de la classe Movie

À l'instar de la classe *Rental*, la classe *Movie* contient plusieurs défauts de design, dont le plus important est le manque de comportement.

```
public class Movie {

  public static final int CHILDREN = 2;
  public static final int NEW_RELEASE = 1;
  public static final int REGULAR = 0;

  private String _title;
  private int _priceCode;

  public Movie(String title, int priceCode) {
    _title = title;
    _priceCode = priceCode;
  }

  public int getPriceCode() {
    return _priceCode;
  }

  public String getTitle() {
    return _title;
  }
}
```

Les deux classes, Rental et Movie souffrent du même code smell : ***Data Class***

Visite de la classe Customer

```
public class Customer {

  private String _name;
  private List<Rental> _rentals = new ArrayList<>();

  public Customer(String name) {
    _name = name;
  }

  public void addRental(Rental arg) {
    _rentals.add(arg);
  }

  public String getName() {
    return _name;
  }

  public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    String result = "Rental Record for " + getName() + "\n";

    for (Rental each: _rentals) {
      double thisAmount = 0;
      //determine amounts for each line
      switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
          thisAmount += 2;
          if (each.getDaysRented() > 2)
            thisAmount += (each.getDaysRented() - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          thisAmount += each.getDaysRented() * 3;
          break;
        case Movie.CHILDREN:
          thisAmount += 1.5;
          if (each.getDaysRented() > 3)
            thisAmount += (each.getDaysRented() - 3) * 1.5;
          break;
      }
      // add frequent renter points
      frequentRenterPoints++;
      // add bonus for a two day new release each
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.get
DaysRented() > 1)
        frequentRenterPoints++;
      // show figures for this each
      result += "\t" + each.getMovie().getTitle() + "\t" + thisAmount + "\n";
      totalAmount += thisAmount;
    }
    // add footer lines
    result += "Amount owed is " + totalAmount + "\n";
    result += "You earned " + frequentRenterPoints + " frequent renter points";
    return result;
```

```
  }
}
```

Cette classe détonne vis-à-vis des deux précédentes, car à elle seule, elle contient l'essentiel des comportements métier du programme. La méthode « *statement* » contient trop de code et surtout trop de responsabilités :

- Le formatage du ticket de location
- Le calcul du montant total des films
- Le nombre de points de fidélités accumulés

Il y a donc trois responsabilités alors que les règles de design nous disent : une seule responsabilité par classe. La responsabilité la plus légitime de la méthode « statement » est le formatage du ticket de location, les deux autres doivent être repoussées dans d'autres classes.

Liste des codes smells repérés dans la méthode « statement » :

- ***Long Method***
- ***Feature Envy***
- ***Switch Statements***

Avec cette dernière classe, on note qu'il serait très difficile d'envisager le support d'un ticket HML sans prévoir un refactoring.

Appliquer le remaniement de code

Voici la correspondance entre des codes smells repérés durant la visite des classes et leurs mouvements respectifs à réaliser pour les corriger :

- Long Method: Extract Method
- Feature Envy: Extract Method, Move Instance
- Switch Statements : appliquer le pattern stratégie

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  String result = "Rental Record for " + getName() + "\n";

  for (Rental each: _rentals) {
// Start Feature Envy -> Rental -> rental.computeAmount()
    double thisAmount = 0;
// Start switch statements -> strategy pattern
    switch (each.getMovie().getPriceCode()) {
      case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
          thisAmount += (each.getDaysRented() - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE:
        thisAmount += each.getDaysRented() * 3;
        break;
      case Movie.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) * 1.5;
        break;
    }
// End switch statements
// End Feature Envy class Rental
// Start Feature Envy -> Rental -> rental.computeFrequentRenterPoints()
    // add frequent renter points
    frequentRenterPoints++;
    // add bonus for a two day new release each
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDays
Rented() > 1)
      frequentRenterPoints++;
// End Feature Envy class Rental
    // show figures for this each
    result += "\t" + each.getMovie().getTitle() + "\t" + thisAmount + "\n";
    totalAmount += thisAmount;
  }
  // add footer lines
  result += "Amount owed is " + totalAmount + "\n";
  result += "You earned " + frequentRenterPoints + " frequent renter points";
  return result;
}
```

Urbanisation du code

La mise en application des cinq principes *SOLID*, permet d'obtenir une application correctement urbanisée offrant un coût évolutif faible. L'urbanisation du code est une forme de répartition des responsabilités par classe. Si le design est correct, toutes les classes devraient avoir une seule responsabilité.

La métaphore des cartons de déménagement

Si plusieurs classes sont anémiques (sans comportement) tandis qu'une autre contient plusieurs responsabilités, alors nous sommes sans doute en présence du code smell *Feature Envy*. Pour comprendre le problème, je vous propose d'utiliser une métaphore pédagogique : Le **carton de déménagement**.

Vous venez juste de déménager et tous vos cartons sont déposés dans le salon. Par exemple : un carton est libellé « cuisine », un autre carton libellé « chambre des enfants », un autre carton libellé « salle de bain ». À l'instant présent, tous ces cartons ne sont pas dans la bonne pièce, ils devraient être déplacés dans la pièce relative à leurs contenus. Vous déplacerez sans doute tous ces cartons dans leur pièce respective progressivement, mais au final, tous les objets contenus dans les cartons seront alignés avec leurs fonctions relatives à leur pièce : les ustensiles de cuisine contenus dans le carton libellé "cuisine" trouveront pleinement leurs fonctions une fois installées dans la cuisine.

Prenons un exemple avec le traitement du calcul des points de fidélité contenu dans la méthode *statement()*. Nous avons sélectionné la partie du code relatif au calcul afin de matérialiser la métaphore du carton. **Figure 3**

Puis nous utilisons notre IDE pour réaliser le mouvement ***Extract Method***. **Figure 4**

Nous avons désélectionné le paramètre ***frequentRenter Points*** car nous préférons ne pas avoir de dépendance sur le cumule des points à l'intérieur de la méthode.

Après avoir initialisé la variable interne, notre carton est prêt à être déplacé dans la bonne pièce :

La classe *Rental*.

```
public int computeFrequentRenterPoints(Rental rental) {
  int frequentRenterPoints = 0;
  // add frequent renter points
  frequentRenterPoints++;
  // add bonus for a two day new release rental
  if ((rental.getMovie().getPriceCode() == Movie.NEW_RELEASE) && rental.getDays
Rented() > 1)
     frequentRenterPoints++;
  return frequentRenterPoints;
}
```

Vous noterez que la méthode prend en paramètre un type ***Rental***, ce qui nous indique que ce carton est déplaçable dans la classe ***Rental***.

Il est temps de réaliser une action de refactoring ***Move Instance Méthode***. **Figure 5**

Et appuyer sur le bouton ***Refactor***. Dans la méthode ***statement()***, le code devient :

```
frequentRenterPoints += rental.computeFrequentRenterPoints();
// show figures for this rental
result += "\t" + rental.getMovie().getTitle() + "\t" + thisAmount + "\n";
totalAmount += thisAmount;
```

La classe *Rental* contient maintenant son calcul des points de fidélité.

```
public class Rental {

private Movie _movie;
private int _daysRented;

public Rental(Movie movie, int daysRented) {
  _movie = movie;
  _daysRented = daysRented;
}

public int getDaysRented() {
  return _daysRented;
}
public Movie getMovie() {
  return _movie;
}

public int computeFrequentRenterPoints() {
  int frequentRenterPoints = 0;
  // add frequent renter points
  frequentRenterPoints++;
  // add bonus for a two day new release rental
  if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) && getDaysRented() > 1)
     frequentRenterPoints++;
  return frequentRenterPoints;
 }
}
```

Après quelques mouvements de refactoring supplémentaires

Dans la situation précédente, nous n'avons pas encore ouvert le code afin de supporter un autre média d'impression. Les

```
    // add frequent renter points
    frequentRenterPoints++;
    // add bonus for a two day new release rental
    if ((rental.getMovie().getPriceCode() == Movie.NEW_RELEASE) && rental.getDaysRented() > 1)
        frequentRenterPoints++;
    // show figures for this rental
    result += "\t" + rental.getMovie().getTitle() + "\t" + thisAmount + "\n";
    totalAmount += thisAmount;
}
```

Figure 3

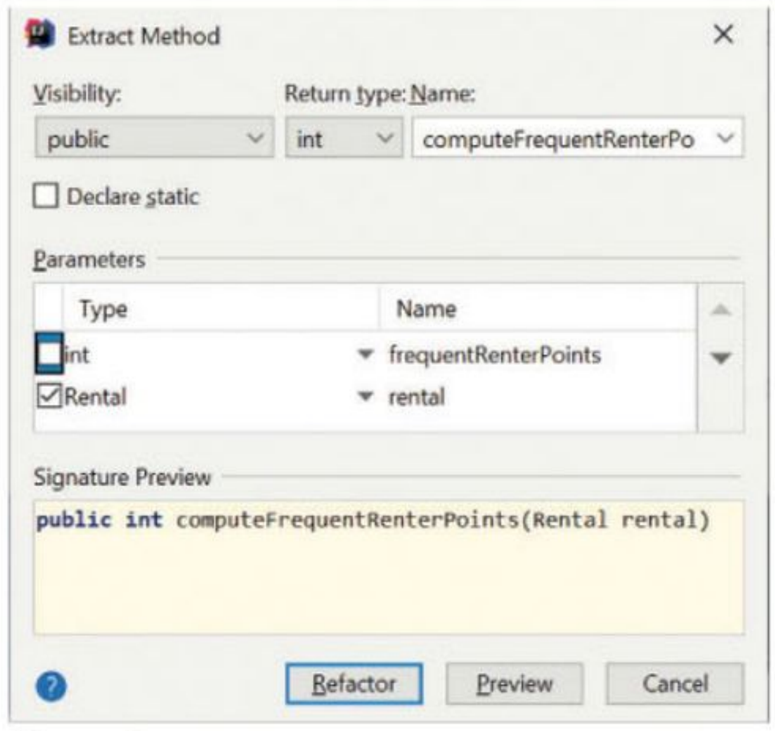


Figure 4

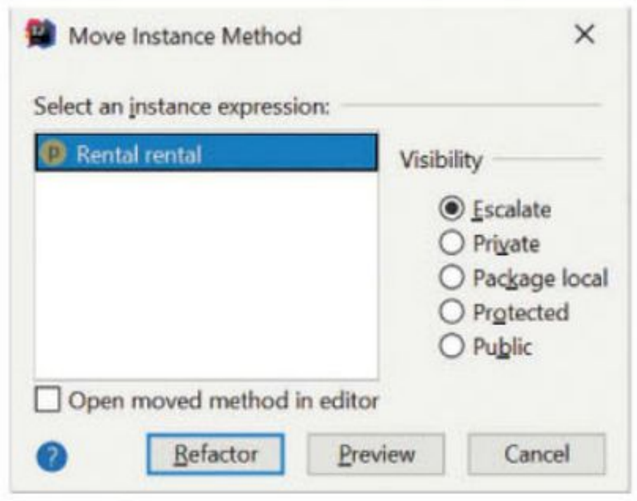


Figure 5

codes sources ci-dessus illustrent une manière de traiter cette demande.

La nouvelle classe *Customer* est simple à appréhender et offre une lecture beaucoup plus agréable.

```
public class Customer {

  private final PrintStatement printStatement;
  private String _name;
  private List<Rental> _rentals = new ArrayList<>();

  public Customer(String name) {
    this(name, new PrintTextStatement());
  }

  public Customer(String name, PrintStatement printStatement) {
    _name = name;
    this.printStatement = printStatement;
  }

  public void addRental(Rental arg) {
    _rentals.add(arg);
  }

  public String getName() {
    return _name;
  }

  public String statement() {

    return printStatement.header(getName())
          .body(_rentals)
            .footer(computeTotalAmount(), computeFrequentRenterPoint())
              .getStatement();
  }
```

```
  public Integer computeFrequentRenterPoint() {
    return _rentals.stream().map(Rental::computeFrequentRenterPoints).reduce(0,
Integer::sum);
  }

  public Double computeTotalAmount() {
    return _rentals.stream().map(Rental::computeAmount).reduce(0.0, Double::sum);
  }
}
```

La méthode ***statement()*** est réduite à l'expressivité de son comportement : imprimer un ticket de location. Techniquement, nous nous sommes inspirés du Pattern Builder afin de chaîner les actions d'impression : ***header***, ***body*** et ***footer***. On note la mise en place du pattern ***Parameterized Constructor*** dans le but de rester compatible avec l'API original, tout en ouvrant une nouvelle possibilité : un nouveau constructeur acceptant un paramètre supplémentaire, une interface ***PrintStatement***.

```
public Customer(String name) {
  this(name, new PrintTextStatement());
}

public Customer(String name, PrintStatement printStatement) {
  _name = name;
  this.printStatement = printStatement;
}
```

Interface PrintStatement

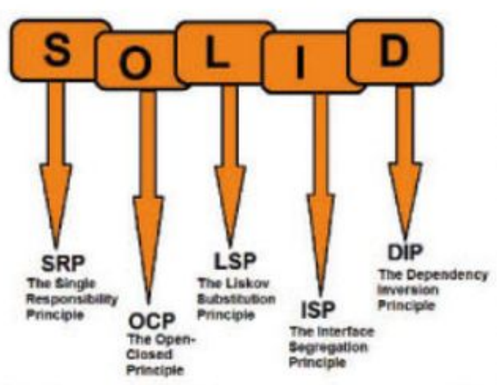
```
public interface PrintStatement {

  String getStatement();

  PrintStatement header(String name);

  PrintStatement footer(double totalAmount, int frequentRenterPoints);

  PrintStatement body(List<Rental> rentals);
}
```

Cette interface nous offre la possibilité d'implémenter facilement un autre support d'impression, comme l'impression du ticket en HTML. Nous avons implémenté l'interface ***PrintStatement*** afin d'assurer l'impression du ticket en mode texte, qui est actuellement le mode défaut.

PrintTextStatement

```
public class PrintTextStatement implements PrintStatement {

  @Override
  public String getStatement() {
    return _statement;
  }

  private String _statement;

  @Override
  public PrintStatement header(String name) {
    _statement = "Rental Record for " + name + "\n";
    return this;
  }

  @Override
  public PrintStatement footer(double totalAmount, int frequentRenterPoints) {
    String statement = "";
    statement += "Amount owed is " + totalAmount + "\n";
    statement += "You earned " + frequentRenterPoints + " frequent renter points";
    _statement += statement;
    return this;
  }

  @Override
  public PrintStatement body(List<Rental> rentals) {
    String statement = "";
    for (Rental rental : rentals) {
      statement += "\t" + rental.getMovie().getTitle() + "\t" + rental.compute
Amount() + "\n";
    }
    _statement += statement;
    return this;
  }
}
```

Nous avons respecté la demande métier, tout en éliminant l'essentiel des *codes smells*



Techniquement, nous avons appliqué plusieurs des ***principes SOLID*** : SRP, OCP, DIP. Leurs motivations sont de vous permettre de modifier votre code régulièrement sans pour autant tout casser. Typiquement, lorsque vous travaillez en mode itératif, vous ajoutez au fil des itérations (Sprint) de nouvelles fonctionnalités qui auront sûrement un impact sur le code existant. Les principes **SOLID** permettent de limiter l'impact afin de garder la même productivité de développement. Ces principes sont donc parfaitement légitimes dans des projets agiles.

En conclusion

Le refactoring de code legacy doit être appliqué à la fois avec discipline et prudence. La pratique des mouvements de refactoring ainsi que l'apprentissage des **codes smells** est un atout pour pratiquer des remaniements de code efficaces.

Casser les dépendances ?

Dans un prochain article, nous aborderons un sujet plus avancé : le cassage des dépendances externes. En effet, les applications contiennent souvent des dépendances externes qui pénalisent la mise en place d'un harnais de test. Le refactoring de design et le cassage de dépendances externes sont deux activités complémentaires lorsqu'on pratique le remaniement de code legacy.

Cloud Native : les limites de Java et la genèse de Quarkus

Mars 2019 : le projet Quarkus devient public avec une accroche prometteuse “Supersonic Subatomic Java”. Dans ce dossier, nous allons revenir sur les raisons qui ont poussé Red Hat à proposer un nouveau framework Cloud Native.

Daniel Petisme
Customer Success Architect Confluent
(ex. System Designer Michelin)
@danielpetisme

En se basant sur l'index TIOBE (https://www.tiobe.com/ tiobe-index/), Java apparaît comme l'un des langages les plus utilisés au monde (et souvent 1er). Cependant, pour les raisons évoquées précédemment, à l'heure de choisir une technologie pour développer des applications micro-services ou serverless, Java est souvent écarté.
L'objectif derrière Quarkus est de repenser la manière dont on construit une application Java pour permettre à la plateforme de rester compétitive dans le contexte Cloud.

Le “Cloud Native”, c'est quoi ?

Pour comprendre pourquoi Quarkus est né, il faut regarder côté “cloud”. Le paradigme du cloud a complètement changé la manière dont on consomme l'infrastructure et par conséquent, la manière dont on construit et délivre les applications. D'un côté, les concepts de « à la demande » (As A Service) et de vélocité nous permettent d'obtenir les ressources nécessaires pour une application « Juste à temps » (Just In Time) fournissant une magnifique capacité de réaction aux aléas. De l'autre, le concept d'élasticité nous donne le sentiment de « ressource infinie jetable » qui nous permet d'optimiser la consommation d'infrastructure au plus juste dans un contexte donné. Une application Cloud Native va être l'application qui tire profit des grandes propriétés du cloud.
Quand on parle « Cloud Native », on retrouve les termes « Microservices » ou « Serverless » par opposition aux « Monolithes ». Historiquement, les applications dites « Monolithes » sont les applications « classiques » développées dans un monde « non-cloud », elles rassemblaient l'intégralité du périmètre d'une application dans un seul artefact. En termes de vélocité, lorsqu'une fonctionnalité avait été mise à jour, il fallait attendre le déploiement de la solution complète pour pouvoir en profiter. Même constat pour l'utilisation de ressources, la fonctionnalité n'était pas utilisée ; tant pis, elle occupait de la place en mémoire au détriment des besoins d'une autre fonctionnalité. L'architecture Micro-service vise justement à isoler des modules autonomes du monolithe pour des besoins d'agilité ou d'optimisation de ressources.
Enfin l'architecture « serverless » pousse le concept un cran plus loin en définissant la *fonction* comme unité de déploiement et optimisation. **Figure 1**
En croisant les paradigmes des applications Cloud Native et les architectures microservices/serverless, on obtient une capacité d'adaptation incroyable. On peut, par exemple, utiliser ces concepts pour améliorer l'agilité d'un produit/service (en accélérant le nombre de déploiements sans nuire à la disponibilité), on peut également travailler sur un axe économique en isolant et optimisant les consommations d'infrastructure au juste minimum.
Là où ces enjeux de réactivité au besoin business n'étaient réservés qu'aux géants du web il y a quelques années, ils deviennent désormais des leviers pour beaucoup d'entreprises traditionnelles, pour des entreprises qui placent l'informatique comme un moteur de croissance. À titre d'exemple, Michelin a pour stratégie de doubler son chiffre d'affaires sur les services autour du pneu. Ce qui se traduit, entre autres, par la construction de solutions IoT pour transformer de la donnée physique autour du pneu en conseils pour les conducteurs, les gestionnaires de flottes poids lourds ou les constructeurs de routes.

Les limites de Java dans le Cloud

Une des forces de Java est sa capacité d'introspection et réflexion qui apporte une forte composante dynamique aux frameworks et applications. Cet aspect dynamique permet au développeur d'attendre la phase d'exécution du code (voir parfois l'invocation de partie spécifique du code) pour activer un comportement. Un exemple concret est la configuration de l'application. Durant le démarrage de l'application, une des premières étapes des frameworks est de scanner les classpaths de l'application pour charger les configurations (via des fichiers, des annotations, différentes classes fournissant de manière programmatique de la configuration). Ce scan est long et demande le chargement en mémoire de l'ensemble du code. Le chargement de configuration est un exemple, mais le même principe s'applique pour la configuration de votre ORM, moteur d'injection de dépendances, couche web, etc.
Souvent les développeurs dimensionnent leur besoin de mémoire en fonction de la consommation de l'application « une fois chargée » (par ex : 1Go) or il n'est pas anodin de voir localement un besoin plus élevé de ressources pour le chargement d'application (par ex : 2Go). Avec moins de ressources disponibles, l'impact direct est un temps de démarrage plus long.
Ce modèle d'application est difficilement compatible avec le

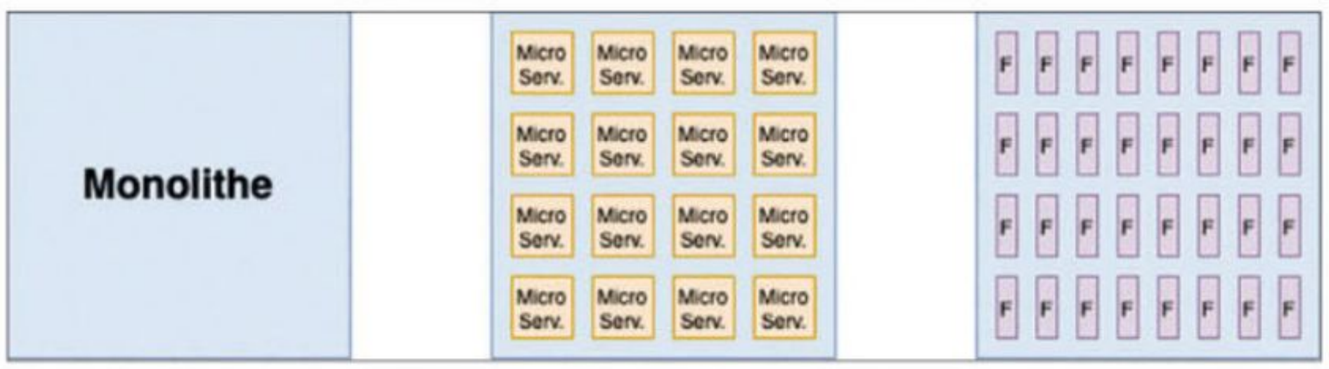


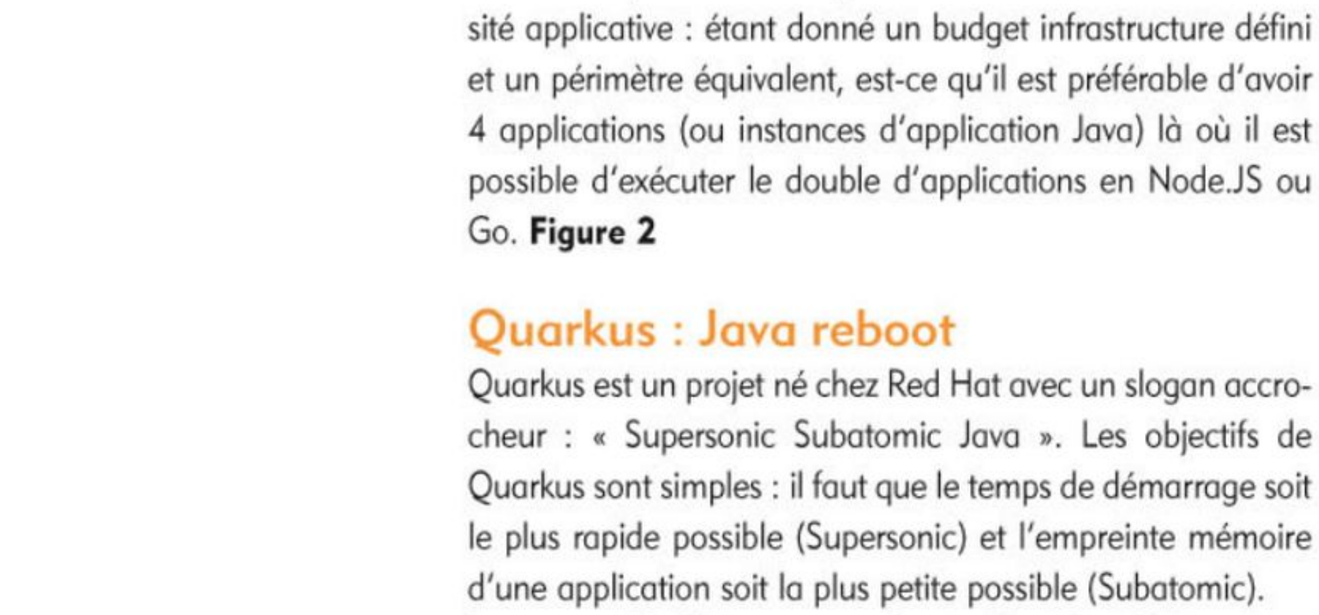
Figure 1 : *Comparaison des architectures monolithiques, micro-services et serverless*

cloud où la consommation de ressources est directement facturée. Soit l'infrastructure est surdimensionnée pour absorber le démarrage (impact financier) soit dimensionné au plus juste au détriment du temps de démarrage (impact vitesse).
Ce constat est encore plus flagrant lorsque l'application s'exécute sur une plateforme d'orchestration de containers (par exemple : Kubernetes). Pour garantir la mutualisation des ressources d'infrastructure, les plateformes comme Kubernetes recommandent de dimensionner au plus juste les quantités de processeurs et de mémoire allouée. Une erreur bien connue est la « OutOfMemory Crash Loop ». Le développeur demande 1Go de RAM à Kubernetes et passe en paramètres les fameux -Xms -Xmx pour limiter la taille de la Heap de machine virtuelle. La réalité est que la JVM est composée d'autres espaces mémoire (comme le off-HEAP, le metaspace) qui s'ajoute à la mémoire du container et de son swap. Résultat, sur le Go de RAM demandé, l'application ne dispose finalement que d'une partie de cette mémoire. Dans ce contexte et étant donné la quantité d'informations à charger au démarrage (scan de classpath), l'application crashe par manque de mémoire (OutOfMemoryException) avant d'être relancée par la plateforme d'orchestration de conteneurs, mais sans grand espoir (Note : dans ce contexte, il est préférable d'utiliser la valeur de la "Resisdent state Size" qui représente l'empreinte mémoire totale du process java et qui est la valeur utilisée dans les quotas RAM de Kubernetes).
Une fois encore, est-il intelligent de surdimensionner les ressources pour régler ce problème ?
Le dilemme d'une application Java dans le cloud est son empreinte mémoire et son temps de démarrage. Ces contraintes posent un vrai problème de rentabilité de la densité applicative : étant donné un budget infrastructure défini et un périmètre équivalent, est-ce qu'il est préférable d'avoir 4 applications (ou instances d'application Java) là où il est possible d'exécuter le double d'applications en Node.JS ou Go. **Figure 2**

Quarkus : Java reboot

Quarkus est un projet né chez Red Hat avec un slogan accrocheur : « Supersonic Subatomic Java ». Les objectifs de Quarkus sont simples : il faut que le temps de démarrage soit le plus rapide possible (Supersonic) et l'empreinte mémoire d'une application soit la plus petite possible (Subatomic).
Le constat de départ est que la manière dont l'écosystème Java consomme des frameworks n'a pas beaucoup changé depuis des décennies alors que l'infrastructure sous-jacente avait subi plusieurs révolutions (Serveur d'application, puis approche fat-jar, cloud, serverless, Kubernetes). De manière générale, les développeurs de solutions avaient abusé de l'aspect dynamique de la plateforme en restant dans un état d'esprit où la mémoire était peu chère.
Il fallait donc retirer de la phase d'exécution de l'application tous les éléments techniques qui n'étaient pas strictement nécessaires ou qui pouvaient s'anticiper.
Cela ne vous rappelle rien ? Essayez d'anticiper le comportement d'une application dans un but d'optimisation ? Rendre la dynamique statique ? Si on voulait être provocateur, on dirait que Quarkus remet au goût du jour le concept de « pré-compilation » (et pour les mêmes raisons qu'à l'époque, pour de l'économie de ressources).
Le premier changement de paradigme est la capacité à faire initialiser les frameworks durant la construction de votre application. Prenons l'exemple du framework Hibernate qui fait le lien avec la base de données. Sans Quarkus, votre application embarque l'API JPA et son implémentation (ici Hibernate) ainsi que le driver et tous les dialectes de votre base de données, par exemple PostgreSQL. Votre application démarre et Hibernate est appelé. Le framework va alors scanner les classpaths à la recherche du fichier `persistence.xml` ou de vos entités (classes annotées avec @Entity). Ensuite, ces classes vont être décorées pour faciliter leur gestion. Dans une autre séquence, basée sur le driver découvert dans le classpath, les implémentations spécifiques de la base de données ciblée vont se charger et se configurer en fonction des configurations données à l'application. Il est important que ces actions soient faites au démarrage de votre application, elles se répètent donc à chaque redémarrage (et dans un mode cloud/container, des redémarrages, il y en a beaucoup...).
En prenant un peu de recul, l'intégralité des informations citées dans le paragraphe précédent est connue au moment de la construction de votre application. Quarkus introduit une nouvelle phase durant la construction des applications Java, qui est la phase « d'augmentation ». L'objectif de cette phase est de forcer les frameworks à pré-calculer un maximum d'informations dans le but de générer le byte code Java résultat. En reprenant l'exemple de l'ORM, Quarkus va permettre à Hibernate de faire le classpaths scanning, la détection des entités, le chargement et la configuration du code spécifique au driver de base de données pendant la construction de l'application et de générer le byte code résultat dans l'archive finale. En appliquant ce paradigme aux différents composants de votre application, la quantité de travail nécessaire au démarrage est drastiquement réduite ce qui a un impact direct sur le temps de démarrage. De plus, comme l'application démarre avec uniquement la partie des frameworks déjà précablée, il n'est plus nécessaire de faire un chargement massif de classes en mémoire, réduisant ainsi l'empreinte mémoire de l'application. Pour finir, en décalant le démarrage à la construction, ce sont tous les démarrages de l'application qui sont optimisés par opposition à l'approche traditionnelle qui se répète à chaque fois. **Figure 3**
La contrepartie du paradigme Quarkus est qu'il implique une évolution des frameworks et de l'écosystème. Le rôle du projet Quarkus est de donner les outils au fournisseur de frameworks pour développer des « extensions » Quarkus. Une

Figure 2 : *: Comparaison des densités applicatives JVM, Node.JS et Go.*



extension correspond à une fonctionnalité qui va implémenter la fameuse phase d'augmentation et ainsi proposer des gains sur le temps de démarrage et de consommation mémoire. À date, le projet Quarkus fournit un ensemble d'extensions reprenant les grands classiques du développement d'application Java (il est intéressant de noter que Quarkus a fait le choix de suivre les traces de JakartaEE/JavaEE en supportant MicroProfile). **Figure 4**

Produire un code natif à partir de Java

Souvent, quand on parle de Quarkus, on aborde le sujet de l'exécutable natif ou encore GraalVM. Il est important de comprendre que le sujet de la compilation native est assez indépendant de Quarkus. Depuis 3-4 ans quasi tous les frameworks majeurs Java ont apporté leur implémentation qui s'appuie sur GraalVM.

GraalVM n'est pas fourni par défaut dans les distributions de la plateforme Java et qu'il faudra installer un nouveau JDK (https://www.graalvm.org/docs/getting-started/). Dans le contexte Quarkus, les éléments importants sont la machine virtuelle Substrate VM et le Graal Compiler. Substrate VM va s'appuyer sur le Graal Compiler pour traduire du byte code Java en code natif compilé. **Figure 5**

La particularité de Substrate VM est son agressivité sur la compilation en avance de phase (Ahead of Time). Lors de la compilation, Substrate VM va chercher à scanner le code de votre application afin d'identifier quels sont les passages vraiment utilisés. En partant de cette analyse, Substrate VM va faire une hypothèse forte qui est que toutes les portions de code identifiées constituent l'ensemble du code nécessaire au fonctionnement de votre application. Si une portion n'a pas été identifiée alors elle ne sera pas présente dans l'artefact final de votre application. Cette hypothèse s'appelle « Closed-world Assumption » et encourage à supprimer (ou du moins réduire) le côté dynamique de votre application. L'effet bénéfique est que, comme avec Quarkus, le travail d'optimisation et de pré-calcul de Substrate VM est fait pendant la compilation ce qui a un impact direct sur l'empreinte mémoire et le temps de démarrage. **Figure 6**

En revenant à Quarkus, on peut voir l'utilisation de GraalVM comme une seconde couche d'optimisation. Dans un premier temps, Quarkus va réduire le coût lié au démarrage en demandant aux extensions de générer, durant la phase de construction, le byte code correspondant au démarrage des frameworks de votre application ; et dans un second temps, GraalVM va optimiser le bytecode pour le réduire au juste minimum avant de le compiler en exécutable natif. Grâce à ces deux mécaniques, on obtient des performances inimaginables pour une application écrite en Java (REST+CRUD= 28MB/0.042s). **Figure 7**

Dans le contexte de la compilation native, Quarkus va simplifier l'accès à GraalVM pour le développeur. Nous avons évoqué la « Closed-World Assumption » qui décourage l'usage de code dynamique; cependant, il se peut que l'application ou les frameworks aient besoin d'utiliser de la réflexion/introspection. Le compilateur Graal permet d'exprimer ce genre de contrainte à travers un paramétrage qui peut vite être complexe. Le rôle de Quarkus est de masquer ces détails aux développeurs pour proposer une expérience la plus simple possible.

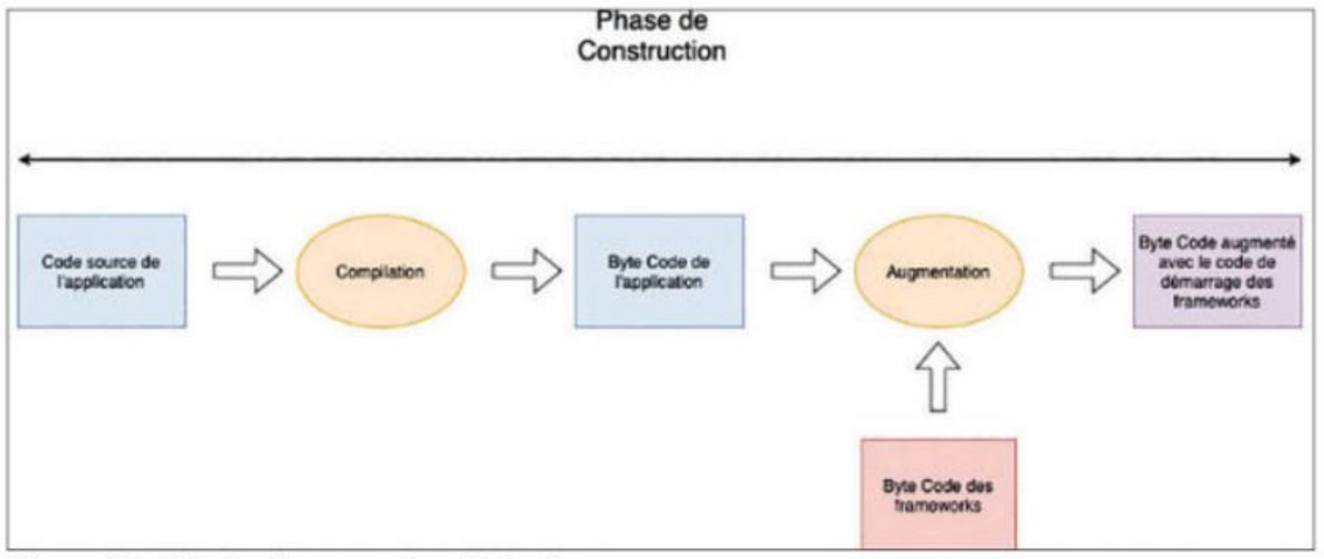


Figure 3 : *Principe d'augmentation de Quarkus*

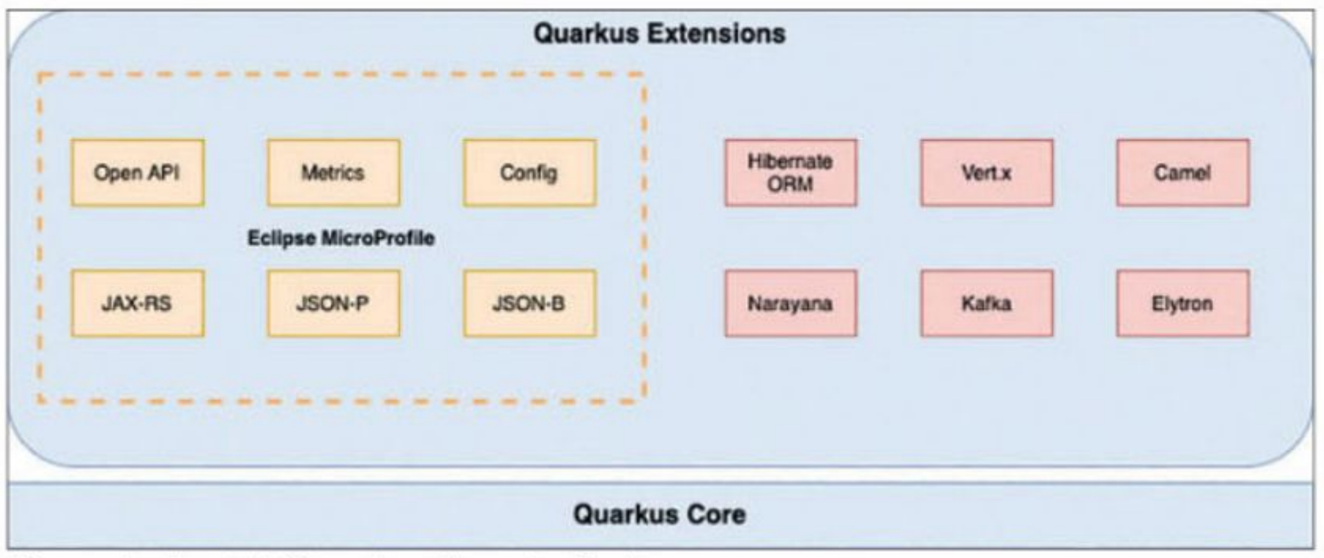


Figure 4 : *Extrait de l'écosystème d'extension Quarkus*

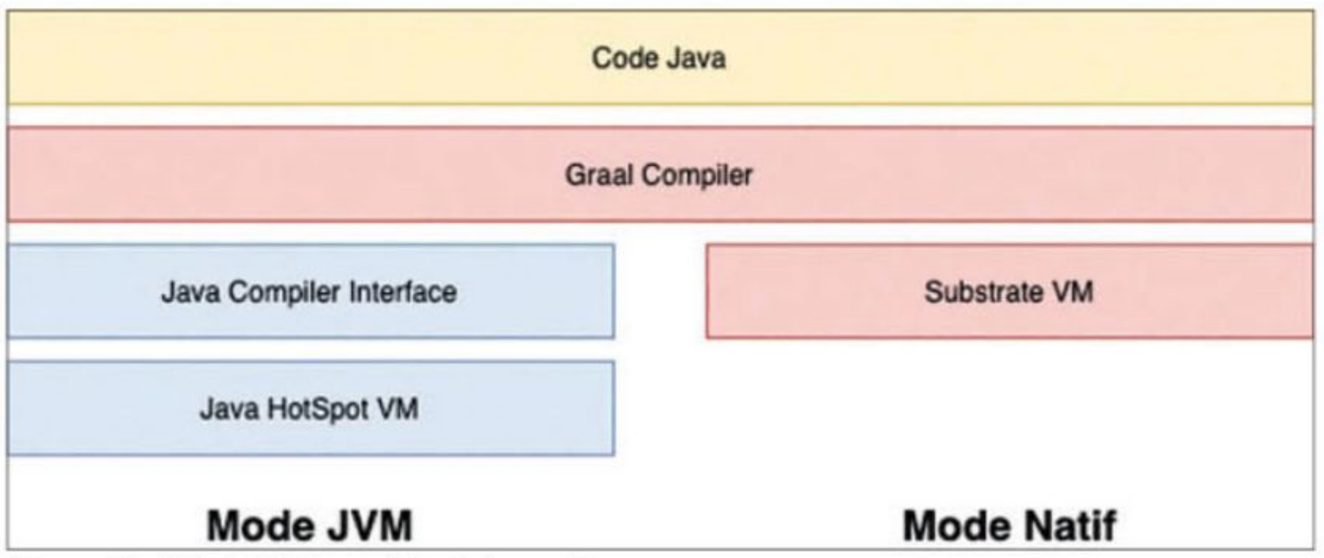


Figure 5 : *Substrate VM pour faire du Java natif*

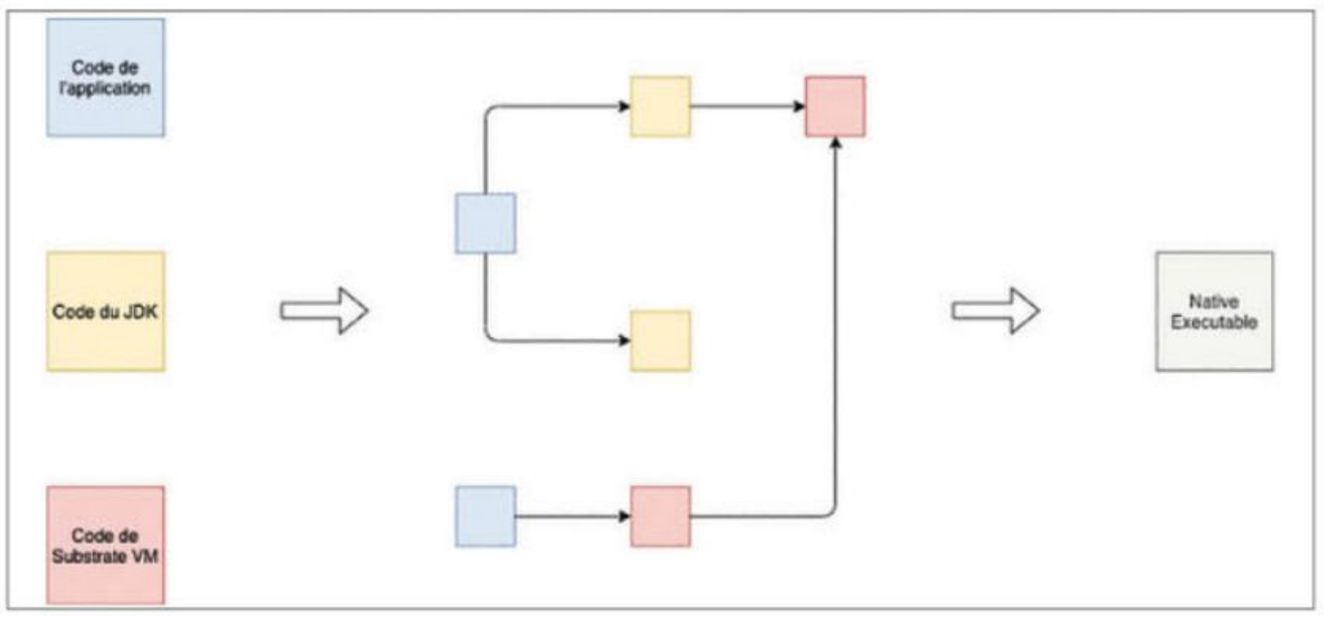


Figure 6 : *Analyse et élimination de code mort par Substrate VM*

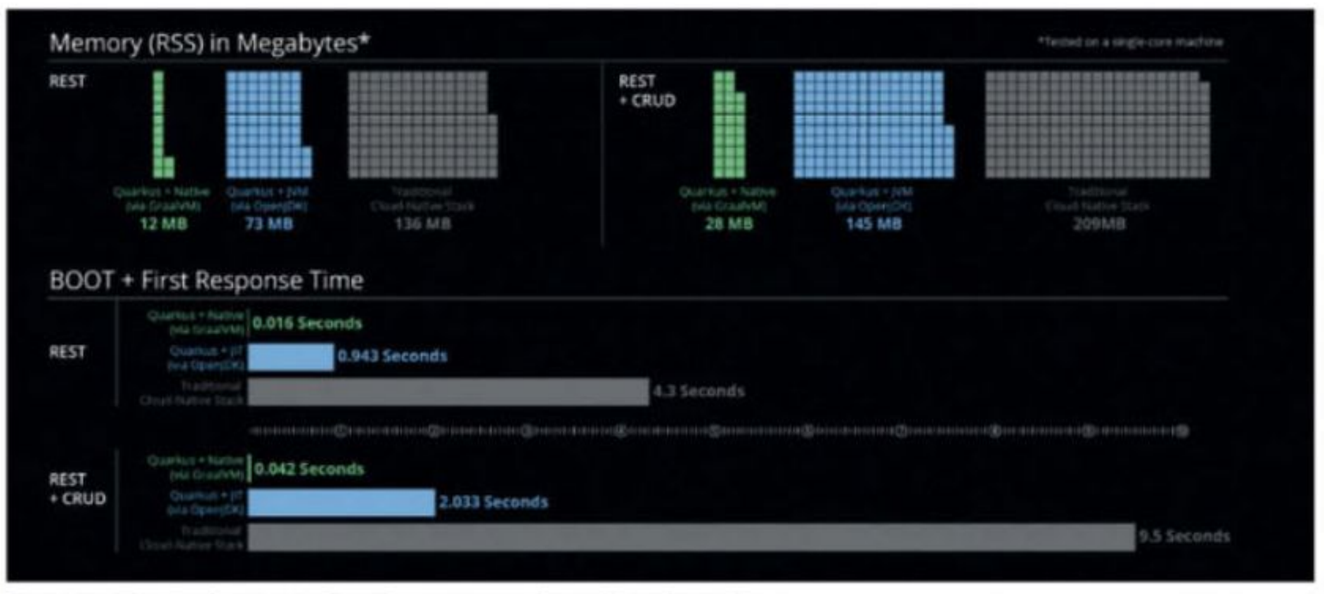


Figure 7 : *Performance Quarkus (source : https://quarkus.io/)*

Conclusion

Dans cette introduction à Quarkus, nous sommes repartis du constat que le mode de fonctionnement des frameworks classiques de l'écosystème Java avait un impact direct sur la consommation mémoire et le temps de démarrage. De ce fait, il n'était pas économiquement intéressant d'utiliser la technologie Java dans des contextes Cloud ou orchestration de containers dans lesquels la ressource et le temps coûtent cher.

Dans ce contexte, Quarkus apporte un changement de paradigme dans la manière dont les frameworks fonctionnent. L'enjeu est de supprimer du démarrage toutes les tâches qui peuvent être décalées à la phase de construction (comme le chargement des configurations, le scan de classpaths, etc.). Ce changement d'approche apporte des gains significatifs sur les performances de nos applications ; cependant, il implique une adaptation des frameworks existants. Pour faciliter cette adoption, Quarkus propose une logique d'extension pour proposer aux fournisseurs de bibliothèques les outils pour optimiser les performances au démarrage de l'application.

Pour finir, nous avons vu comment Quarkus s'intègre avec GraalVM. En s'appuyant sur la « Closed-World Assumption » et la compilation native, Substrate VM nous permet de générer un exécutable compact et doté de performances exceptionnelles pour une application Java.

Mais Quarkus, c'est bien plus que juste de l'optimisation de performances ! C'est aussi un sentiment de fraîcheur sur la manière de développer des applications et c'est ce que vous allez découvrir dans le prochain dossier.

PARTIE 2

Ma première application Quarkus

Venez découvrir comment réaliser votre première application Quarkus.

Todo App

La « Todo App » est une application fictive permettant la gestion d'une liste de tâches à faire. Durant le tutoriel, nous allons réaliser une API REST qui permettra d'interroger une base de données PosgreSQL grâce à un ORM.

Le tutoriel se veut volontairement simpliste pour faciliter la compréhension des fonctionnalités Quarkus. Nous ne saurions être responsables d'une montée en production de cette application.

Merci à Clement Escoffier (https://twitter.com/clementplop) de nous avoir permis de réutiliser le code disponible à https://github.com/cescoffier/quarkus-todo-app.

Par où commencer ?

Avant de se jeter la tête la première dans du code, il est intéressant de faire un tour d'horizon des différentes ressources disponibles pour nous aider dans la découverte de Quarkus.

https://quarkus.io/

Sans être très original, le site officiel Quarkus est le meilleur point d'entrée pour retrouver toutes les informations. Une des grandes forces de Quarkus est d'avoir fait le choix d'avoir des documentations « pratiques ». La section « Guides » (https://quarkus.io/guides/) reprend les grandes fonctionnalités de Quarkus démontrées au travers d'exemples pratiques et le code est disponible sur GitHub (https://github.com/quarkusio/quarkus-quickstarts). Une autre source d'informations intéressantes est les différentes vidéos disponibles sur YouTube. Les vidéos (presque) hebdomadaires « Quarkus Insights » (https://www.youtube.com/quarkusio) sont présentées par des membres de l'équipe « core » Quarkus. Elles mettent en avant l'écosystème de manière générale (retour d'expérience, présentation d'outils construits avec Quarkus, revue en détail de fonctionnalités). Côté français, Sébastien Blanc (https://twitter.com/sebi2706) anime « Quarkus Facile » qui a pour but d'expliquer avec des exemples certaines fonctionnalités du framework.

Coté support, on retrouve les grands classiques. Si vous rencontrez une difficulté, Stack Overflow sera votre meilleur ami (https://stackoverflow.com/questions/tagged/quarkus) et dans le cas où vous souhaitez remonter un bug ou une demande de fonctionnalité, il faudra vous tourner vers le projet GitHub officiel (https://github.com/quarkusio/quarkus).

Une des particularités du projet Quarkus est d'utiliser la plateforme Zulip (https://quarkusio.zulipchat.com/). Zulip est l'outil de discussion de l'équipe de développement de Quarkus, mais

aussi celui de la communauté. C'est l'endroit parfait pour échanger de manière conviviale sur des éléments structurants du framework (besoin de clarification sur une implémentation, discussion sur un bug, etc.).

Comment commencer à coder ?

La manière la plus simple de commencer votre projet est d'aller sur https://code.quarkus.io/. Il s'agit d'une application qui va vous permettre de facilement initialiser un projet Quarkus.

Figure 1

Comme beaucoup d'applications web de génération de projet, code.quarkus.io nous permet de définir les détails de notre projet (groupId, artifactId, version) ainsi que l'outil de construction souhaité (Maven ou Gradle). Nous pouvons également choisir des extensions que nous souhaitons embarquer dans notre squelette de projet. Dans notre contexte, nous allons utiliser l'implémentation RESTEasy de JAX-RS pour définir nos ressources REST, JSON-B pour l'encodage/décodage JSON, Hibernate comme ORM (nous détaillerons plus tard le concept de « panache ») et outil de validation. Enfin, nous utiliserons le driver JDBC correspondant à notre base PostgreSQL.

Après avoir cliqué sur « Generate your application », vous extrayez le projet de l'archive ZIP, vous serez prêt à démarrer le développement.

Pour les adeptes de la ligne de commande :

```
mvn io.quarkus:quarkus-maven-plugin:create \
  -DprojectGroupId=io.quarkus \
  -DprojectArtifactId=sample \
  -DprojectVersion=1.0.0-SNAPSHOT \
  -DclassName="io.quarkus.sample.TodoResource" \
  -Dextensions="resteasy,resteasy-jsonb,hibernate-validator,hibernate
-orm-panache,jdbc-postgresql"
```

Analyse du projet

Nous allons regarder en détail comment se compose le projet que nous venons de générer. Pour commencer, regardons l'aspect construction du projet. Le projet embarque Maven Wrapper qui va permettre d'utiliser la commande `./mvnw`. L'intérêt de Maven Wrapper est de gérer automatiquement le téléchargement et l'exécution de Maven sur votre ordinateur, c'est extrêmement pratique pour rendre la construction répétable sur différents postes de développement (il existe `gradlew` qui est le pendant Gradle).

Puisque nous avons utilisé Maven, nous disposons d'un fichier `pom.xml` qui décrit notre projet. Ce fichier définit une dépendance à la « Bill Of Materials » Quarkus.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>${quarkus.platform.group-id}</groupId>
      <artifactId>${quarkus.platform.artifact-id}</artifactId>
      <version>${quarkus.platform.version}</version>
      <type>pom</type>
      <scope>import</scope>
```

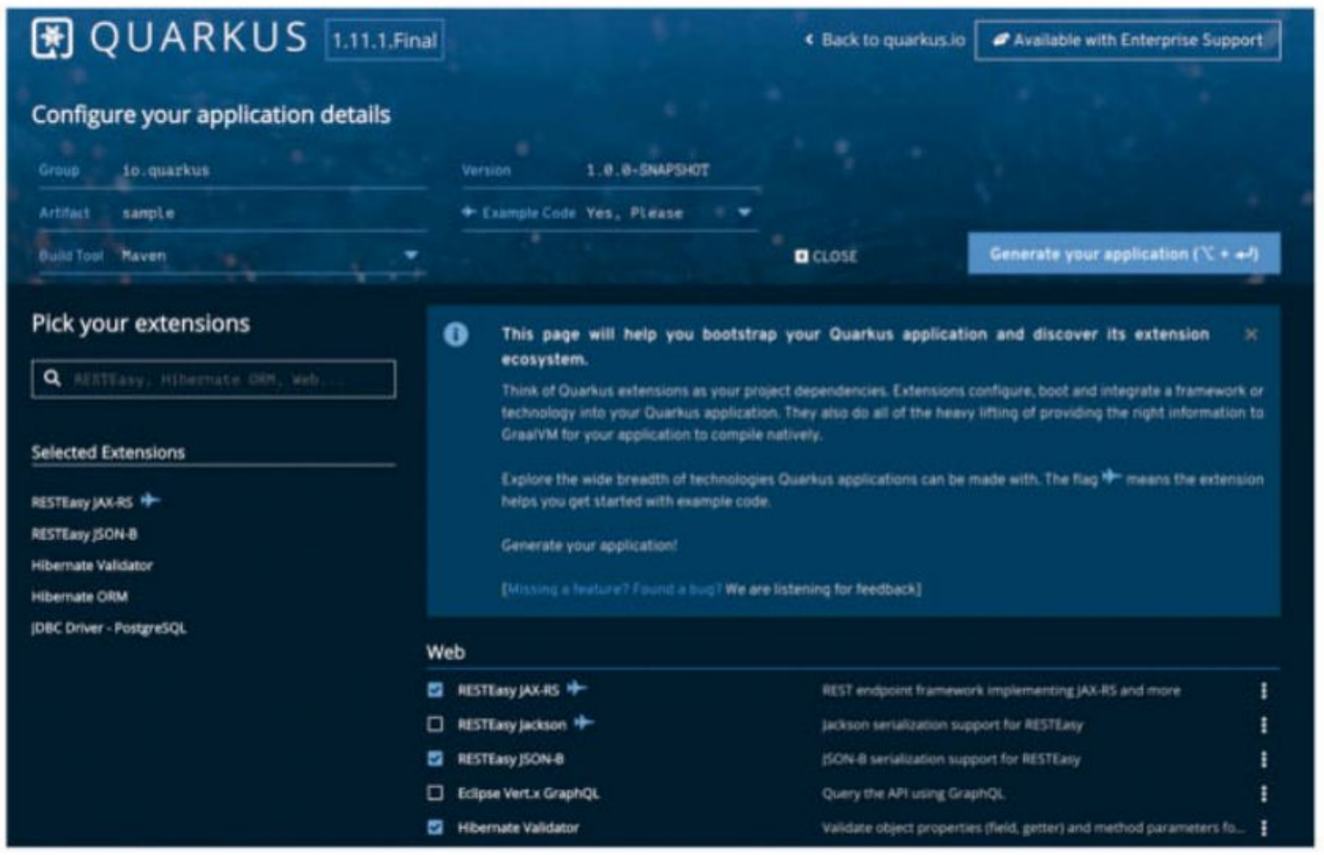


Figure 1 : https://code.quarkus.io

```
  </dependency>
 </dependencies>
</dependencyManagement>
```

Une BOM est une dépendance un peu particulière, il s'agit d'un fichier pom.xml que la plateforme Quarkus met à disposition et qui reprend la définition des principaux packages nécessaires au framework avec leur version à jour. Grâce à cette définition, les développeurs n'ont pas à se soucier des versions des dépendances liées à Quarkus.

Ensuite, dans la section `dependencies`, nous allons retrouver nos dépendances comme `resteasy` ou `hibernate-orm-panache`. On peut constater que la génération du projet s'est occupée de rajouter les dépendances de test `junit5` ainsi que `rest-assured` qu'on abordera plus tard.

Comme expliqué dans notre dossier sur la Genèse de Quarkus, le framework intervient lors de la chaîne de construction du projet pour « augmenter » le byte code de l'application pour notamment optimiser le temps de démarrage et l'empreinte mémoire de l'application. Cette étape est gérée par le `quarkus-maven-plugin` que l'on définit comme suit :

```
<plugin>
<groupId>io.quarkus</groupId>
<artifactId>quarkus-maven-plugin</artifactId>
<version>${quarkus-plugin.version}</version>
<executions>
  <execution>
  <goals>
    <goal>build</goal>
  </goals>
  </execution>
</executions>
</plugin>
```

Pour finir, nous disposons également d'un profil `native` qui s'activera de manière explicite en lançant la commande

Maven `./mvnw package -Pnative`.
Code complet sur programmez.com et GitHub

Ce profil indique à Quarkus de construire un exécutable natif grâce à GraalVM, mais aussi d'exécuter des tests d'intégration pour valider le comportement de l'artefact.
Pour finir, sur la partie construction, le projet généré dispose de fichiers `Dockerfile` génériques. Le premier fichier `Dockerfile.jvm` permet de copier l'archive JAR obtenu lors de la construction de l'application dans une image Docker qui définit l'ensemble du contexte Java nécessaire. Le second fichier `Dockerfile.native` sera utilisé pour créer une image Docker qui s'appuie sur le résultat du paquetage natif.
Le reste du projet concerne les ressources Java. Nous allons retrouver une classe Java exemple, les squelettes de tests (mode JVM et mode natif) et un fichier de configuration `application.properties`. Sans plus de suspens, rentrons dans le vif du sujet en codant une API REST.

Exposer une Resource REST avec Quarkus

Quarkus a fait le choix de s'appuyer sur un certain nombre de standards du monde Java Entreprise. Au sujet de REST, la spécification s'appelle JAX-RS (https://projects.eclipse.org/projects/ee4j.jaxrs) et l'implémentation disponible pour Quarkus est RESTEasy (https://resteasy.github.io/ - sponsorisé par Red Hat).
Dans le projet que nous avons généré, nous disposons par défaut d'une Resource exemple. Dans le cas où le projet a été généré par code.quarkus.io, la ressource est présente dans le fichier `src/main/java/io/quarkus/sample/GreetingResource.java` alors que si vous avez utilisé la ligne de commande fournie plus haut, le fichier se nomme ` src/main/java/io/quarkus/sample /TodoResource.java`. Afin de simplifier la suite du tutoriel, nous vous invitons à renommer le fichier en `TodoResource.java`.

```
package io.quarkus.sample;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class TodoResource {

  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String hello() {
    return "hello";
  }
}
```

On peut constater dans cette classe que la ressource sera accessible en utilisant le chemin `/hello`. Cette configuration est définie par l'utilisation de l'annotation `@Path`. Le protocole REST s'appuie sur les verbes http (GET, POST, PUT, DELETE, PATH, etc.) pour définir les actions à réaliser sur la ressource. Dans l'exemple de code, on peut voir que l'on peut obtenir une représentation de l'état de la ressource en utilisant l'annotation `@GET` qui sera activitée sur l'opération HTTP GET. Enfin, la représentation de cette ressource se fera en utilisant un format texte brut défini par l'annotation `@Produces`. Ce court exemple nous permet d'appréhender les grands concepts du protocole REST (route, opération et représentation).

Lancer l'application et rechargement à chaud

Maintenant que nous avons un peu de code, il serait intéressant de le voir s'exécuter. Pour cela, nous pouvons lancer l'exécution depuis une ligne de commande `./mvnw compile quarkus :dev` **Figure 2**
On peut constater que l'application a démarré en 1,5 seconde. On peut vérifier le fonctionnement de l'API avec un simple curl.

```
$ curl -w "\n" http://localhost:8080/hello
hello
```

La tâche Maven `quarkus :dev` est très intéressante, car, en plus de démarrer notre API, elle écoute les modifications apportées aux fichiers sources de notre application pour pouvoir recharger les modifications à chaud.
Revenons sur notre ressource `TodoResource` et modifions un peu le code de la méthode `hello` comme ceci :

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public String hello() {
  return "Bonjour Programmez !";
}
```

Nous avons remplacé la chaîne de caractères « hello » initialement retournée par « Bonjour Programmez ! ». Revenez maintenant à votre terminal et, sans redémarrer votre application, effectuez un nouveau curl :

```
$ curl -w "\n" http://localhost:8080/hello
Bonjour Programmez !
```

Vous avez dû constater une légère lenteur au moment de cet appel. Ce ralentissement minime est provoqué par le rechargement de l'application par la tâche Maven `quarkus :dev`. Pour en avoir le cœur net, revenez sur le terminal où s'exécute votre application. **Figure 3**
La seconde apparition de la bannière Quarkus correspond au second démarrage, la dernière ligne de ce redémarrage indique que le rechargement à chaud a pris 0,854s.
Vous vous demandez peut-être comment ce rechargement à chaud fonctionne ? Il s'appuie sur la capacité de Quarkus à démarrer extrêmement rapidement. Lorsque l'application est

```
sample ./mvnw compile quarkus:dev
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< io.quarkus:sample >-------------------------
[INFO] Building sample 1.0.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ sample ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ sample ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- quarkus-maven-plugin:1.5.0.CR1:dev (default-cli) @ sample ---
Listening for transport dt_socket at address: 5005
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2021-02-07 16:20:07,719 WARN  [io.qua.agr.dep.AgroalProcessor] (build-21) The Agroal dependency is present but no JDBC datasources have been defined.
2021-02-07 16:20:08,691 INFO  [io.quarkus] (Quarkus Main Thread) sample 1.0.0-SNAPSHOT on JVM (powered by Quarkus 1.5.0.CR1) started in 1.473s. Listening on: http://0.0.0.0:8080
2021-02-07 16:20:08,693 INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
2021-02-07 16:20:08,693 INFO  [io.quarkus] (Quarkus Main Thread) Installed features: [agroal, cdi, hibernate-orm, hibernate-orm-panache, hibernate-validator, jdbc-postgresql, mutiny, narayana-jta, resteasy, resteasy-jsonb]
```

Figure 2

```
[INFO] --- quarkus-maven-plugin:1.5.0.CR1:dev (default-cli) @ sample ---
Listening for transport dt_socket at address: 5005
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2021-02-07 16:20:07,719 WARN  [io.qua.agr.dep.AgroalProcessor] (build-21) The Agroal dependency is present but no JDBC datasources have been defined.
2021-02-07 16:20:08,691 INFO  [io.quarkus] (Quarkus Main Thread) sample 1.0.0-SNAPSHOT on JVM (powered by Quarkus 1.5.0.CR1) started in 1.473s. Listening on: http://0.0.0.0:8080
2021-02-07 16:20:08,693 INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
2021-02-07 16:20:08,693 INFO  [io.quarkus] (Quarkus Main Thread) Installed features: [agroal, cdi, hibernate-orm, hibernate-orm-panache, hibernate-validator, jdbc-postgresql, mutiny, narayana-jta, resteasy, resteasy-jsonb]
2021-02-07 16:20:15,599 INFO  [io.qua.dep.dev] (vert.x-worker-thread-3) Changed source files detected, recompiling [/Users/daniel/workspace/programmez/sample/src/main/java/io/quarkus/sample/TodoResource.java]
2021-02-07 16:20:16,163 INFO  [io.quarkus] (Quarkus Main Thread) sample stopped in 0.018s
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2021-02-07 16:20:16,283 WARN  [io.qua.agr.dep.AgroalProcessor] (build-6) The Agroal dependency is present but no JDBC datasources have been defined.
2021-02-07 16:20:16,440 INFO  [io.quarkus] (Quarkus Main Thread) sample 1.0.0-SNAPSHOT on JVM (powered by Quarkus 1.5.0.CR1) started in 0.277s. Listening on: http://0.0.0.0:8080
2021-02-07 16:20:16,447 INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
2021-02-07 16:20:16,447 INFO  [io.quarkus] (Quarkus Main Thread) Installed features: [agroal, cdi, hibernate-orm, hibernate-orm-panache, hibernate-validator, jdbc-postgresql, mutiny, narayana-jta, resteasy, resteasy-jsonb]
2021-02-07 16:20:16,447 INFO  [io.qua.dep.dev] (vert.x-worker-thread-3) Hot replace total time: 0.054s
```

Figure 3

démarrée en mode « dev » (ce qui est le cas lorsqu'on utilise la tâche Maven : `quarkus :dev`), une partie de Quarkus se met en mode écoute sur les fichiers sources et configuration. Lorsqu'une modification est détectée, Quarkus va ajouter le fichier dans la liste des éléments à recharger. Quarkus définit un ensemble d'évènements capables de provoquer un rechargement et l'appel à une route HTTP est un de ces évènements. Lorsque nous avons réalisé notre second curl, le framework regarde s'il possède des fichiers à recharger (notre TodoResource.java par exemple) et, si c'est le cas, va compiler et remplacer le fichier. Enfin, une fois que tous les fichiers sont à jour, Quarkus va simplement tuer et redémarrer l'application. Grâce aux performances de Quarkus, ce fonctionnement est transparent.

Cette fonctionnalité vient illustrer une des préoccupations majeures de Quarkus : l'Expérience Développeur (ou DX pour « Developper Experience »). Au-delà du pur aspect performance, le souhait de l'équipe qui réalise Quarkus est d'améliorer le quotidien des développeurs en leur fournissant des outils plaisants à être utilisés. Le rechargement à chaud (popularisé par les outils du monde web) permet une boucle de feedback beaucoup plus rapide et donc un gain d'efficacité appréciable.

Note : L'écosystème Java propose du rechargement de code depuis un certain temps avec des solutions plus ou moins natives et transparentes qui s'appuient sur un rechargement à chaud du byte code de l'application. L'expérience a montré que cette méthode pouvait avoir des effets de bords assez compliqués à expliquer. L'approche de Quarkus (le redémarrage de l'application) rend le résultat beaucoup plus prévisible d'après nous. Ceci reste un avis et nous serions ravis d'en discuter si votre opinion est différente.

Accéder à une base de données

Notre application doit permettre la gestion de tâches à faire, il nous faut donc logiquement une persistance en base de données.

Quand on parle ORM, Hibernate est de facto le standard. Quarkus fournit une extension Hibernate `quarkus-hibernate-orm` avec toutes les fonctionnalités classiques de l'ORM tout en fournissant les performances permises par Quarkus. Quarkus et Hibernate étant sponsorisés par RedHat, il est logique de trouver une excellente intégration entre les deux frameworks, une intégration avec « Panache ».

Panache est le terme choisi par l'équipe Quarkus pour proposer une version « orientée », « simplifiée » de certaines extensions disponibles. Pour faire une analogie, « Panache » se rapproche du concept de « Spring Boot » pour « Spring Framework ».

Pour revenir sur l'accès à la base de données, là où traditionnellement le développeur manipule des entités, l'extension `hibernate-orm-panache` va proposer d'utiliser des entités Panache. Prenons un exemple, créez la classe « Todo » (`src/main/java/io/quarkus/sample/Todo.java`)

```
@Entity
public class Todo extends PanacheEntity {

  @NotBlank
  @Column(unique = true)
  public String title;

  public boolean completed;

  @Column(name = "ordering")
  public int order;

  public String url;

  public static List<Todo> findNotCompleted() {
    return list("completed", false);
  }

  public static List<Todo> findCompleted() {
    return list("completed", true);
  }

  public static long deleteCompleted() {
    return delete("completed", true);
  }
```

```
}
```

Nous retrouvons dans cet exemple des éléments classiques de JPA, notre classe « Todo » est annotée avec `@Entity` pour indiquer au framework Hibernate que cet objet est considéré comme à persister. On retrouve également des informations pour aider à la relation code-base de données avec les annotations `@Column`.
Dans les éléments qui diffèrent, on peut tout d'abord remarquer que notre classe « Todo » hérite de « PanacheEntity ». Cet héritage va permettre de profiter d'un champ « id » géré par la classe parente. Dans certains contextes, il fera sens de reprendre le contrôle du champ « id » et dans ce cas votre classe devra hériter de la classe mère de « PanacheEntity » qui est « PanacheBaseEntity ». Les arborescences « PanacheEntity » et « PanacheBaseEntity » fournissent également les opérations liées à la base de données telles que `persist`, `find`, `update` et `delete`.
Cette organisation du code implémente un pattern d'accès aux données assez peu utilisé dans le monde Java qui s'appelle « Active Record » Soyez rassurés, Quarkus implémente les deux options.
Dans les architectures d'accès aux données, on retrouve deux grands styles. La solution très populaire dans l'écosystème Java est le modèle « Repository » (popularisé, notamment, avec Spring Data). Les structures de données à persister (les entités) sont séparées des fonctions qui les manipulent (les « repositories »), ce style est surtout utile dans les très grands projets, car il permet de séparer les problématiques rendant le code plus simple à lire et donc plus maintenable.
La seconde approche est « Active Record » qui rassemble dans la même structure la donnée et les fonctions de manipulation. Cet autre style permet de rationaliser le code et donc de le simplifier, il permet aussi d'avoir une écriture plus fluide des opérations.
Pour revenir à notre entité, on va faire simple et utiliser la solution « Active record ». On voit donc des méthodes statiques ` findNotCompleted`, ` findCompleted` et ` deleteCompleted` qui utilisent des méthodes `find` et `delete` qui proviennent de notre héritage `PanacheEntity`.
Pour compléter l'analyse de notre entité, vous voyez que les attributs sont marqués comme « public », et non ce n'est pas une erreur. Les visibilités des attributs ou méthodes étaient un enjeu crucial quand la mémoire était partagée entre applications comme elle l'était dans les serveurs d'applications. Dans un mode où l'isolation a été renforcée (par l'utilisation d'archive JAR exécutable ou par un container), le concept de visibilité n'a plus du tout le même enjeu. Dans la philosophie Quarkus, la recommandation est d'utiliser une visibilité publique autant que possible et laisser l'infrastructure gérer l'isolation. La conséquence directe de ces approches est une réduction du code en supprimant les accesseurs qui, soyons francs, avaient peu de valeur ajoutée dans la plupart des situations.
Maintenant que notre est code est prêt à interagir avec la base de données, passons à la configuration de l'accès à la base.

Configuration

Note : *nous n'allons pas aborder l'installation ou la configuration de la base de données. Nous vous recommandons d'utiliser une image docker pour simplifier vos développements. Vous trouverez dans le dépôt GitHub un exemple de configuration `docker-compose.yml`.*

Sans être original, la configuration d'une application Quarkus se fait au travers du fichier `src/main/resources/application.properties `.

```
# Configuration file
# key = value
quarkus.datasource.url=jdbc:postgresql://localhost/rest-crud
quarkus.datasource.driver=org.postgresql.Driver
quarkus.datasource.username=restcrud
quarkus.datasource.password=restcrud
quarkus.datasource.max-size=8
quarkus.datasource.min-size=2
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.hibernate-orm.log.sql=true

%prod.quarkus.hibernate-orm.database.generation=update

quarkus.test.native-image-profile=it
%it.quarkus.datasource.url=jdbc:postgresql://localhost:5499/rest-crud
```

Comme avec beaucoup d'outils, cette configuration peut-être surchargée par variable d'environnent ou par ligne de commande (en utilisant des majuscules et le caractère underscore « _ » comme séparateur en lieu et place du point « . »).

```
export QUARKUS_DATASOURCE_PASSWORD=youshallnotpass
```

Pour connaître l'ensemble des configurations disponibles, vous pouvez utiliser la documentation https://quarkus.io/guides/all-config ou utiliser les plugins Maven et Gradle pour générer un fichier de configuration exemple. Cette configuration générée reprendra toutes les configurations disponibles pour les extensions définies dans votre projet.

```
# Dans votre projet
$ ./mvnw io.quarkus:quarkus-maven-plugin:generate-config

```

Une des différences de Quarkus par rapport à ses concurrents est de promouvoir l'utilisation d'un fichier de configuration unique par rapport à un fichier par environnement (par exemple application-dev.properties). Pour répondre au besoin de multiples configurations en fonction de l'environnement ou simplement d'un contexte différent, le framework propose un concept de profil qui va venir préfixer l'élément de configuration. C'est ce qu'on voit dans l'exemple pour la propriété `quakus.hibernate-orm.database.generate`. Dans le contexte standard, le comportement attendu est `drop-and-create`, mais dans le cas de l'environnement de production, on souhaite appliquer une stratégie différente. On utilise alors le

préfixe `%prod` devant le nom de la propriété pour dire que dans un contexte de production, on veut surcharger la valeur de ` quakus.hibernate-orm.database.generate` avec la valeur `update`.
Quarkus vient avec trois profils prédéfinis :
%dev : qui s'active par défaut à l'exécution du mode `quarkus :dev`
%test : qui s'active par défaut durant les phases de tests
%prod : qui s'active par défaut quand les profils %dev et %test ne sont pas utilisés.
Vous pouvez définir vos propres profils, c'est ce que nous faisons avec le profil `%it`. Si vous souhaitez utiliser explicitement un profil, vous pouvez le faire via la propriété `quarkus-profile`, par exemple

```
./mvn -Dquarkus-profile=it compile quarkus:dev
```

Le retour de la ressource REST

Précédemment, nous avons abordé le concept d'une ressource JAX-RS, voyons comment développer une API permettant les opérations basiques (Create, Read, Update, Delete).
Notre API va produire et consommer des objets « Todo » sous la forme de documents JSON. Pour cela, nous ajoutons les annotations suivantes sur la classe « TodoResource ».

```
@Path("/api")
@Produces("application/json")
@Consumes("application/json")
public class TodoResource {  …}
```

La première opération que nous allons coder est la création d'une tâche à faire.

```
@POST
@Transactional
public Response create(@Valid Todo item) {
    item.persist();
    return Response.status(Status.CREATED).entity(item).build();
}
```

La création est associée au verbe http POST. Pour garantir la cohérence de la donnée, nous annotons la méthode `create` avec l'annotation `@Transactional` pour que l'opération soit encapsulée dans une transaction.
Puisque nous avons inclus `quarkus-hibernate-validator` dans les dépendances du projet, nous avons accès à l'annotation `@Valid`. Cette annotation permettra d'exécuter un certain nombre de contrôles d'intégrité sur la donnée reçue avant d'appeler la méthode. Dans notre tutoriel, nous avons utilisé l'annotation `@NotBlank` pour indiquer que le titre d'une tâche devait être une chaîne de caractères non-nulle non vide. Si un objet Todo qui ne respecte pas cette contrainte est envoyé à la méthode `create`, l'annotation `@Valid` déclenchera une exception de type `ConstraintViolation Exception` empêchant l'exécution de la méthode.
Une fois dans le corps de la méthode, on peut voir l'approche « Active Record » en action. L'objet « item » qui est une instance d'une « PanacheEntity » possède une méthode `persist` qui permet de déclencher la création de l'enregistrement correspondant dans la base de données.
Enfin, la méthode construit sa réponse. Dans l'exemple précédent de ressource REST, nous avions vu que l'objet (une chaîne de caractères) était directement retourné. Dans des contextes plus complexes, il peut être nécessaire de construire une réponse plus élaborée incluant un code de retour HTTP, des en-têtes, etc. Pour cela, la spécification JAX-RS définit la classe « javax.ws.rs.core.Response » et vous pouvez voir un exemple d'usage avec ` Response.status(Status.CREATED).entity(item).build()`.
Pressé de voir vos tâches à faire ? Voilà un exemple de consultation :

```
@GET
public List<Todo> getAll() {
  return Todo.listAll(Sort.by("order"));
}
```

On peut voir le modèle « Active Record » en action avec la méthode `listAll` fournit par la surcouche Panache. `quarkus-hibernate-orm-panache` dispose d'un certain nombre de méthodes pour requêter la base de données. Les différentes variantes des méthodes `find` et `findAll` retournent un objet de type `PanacheQuery` qui permet d'avoir un chaînage fluide des opérations (par exemple : `Todo.findAll().project(MyClass.class).page(10,100)`). La méthode `listAll ` retourne toutes les entités disponibles, elle convient parfaitement au cas simple comme ce tutoriel. Vous aurez remarqué l'argument `Sort.by("order")` qui nous donne la capacité de trier les résultats.
Prochaine étape, il se peut que vous ayez à la mettre à jour. Voyons ensemble le code :

```
@PATCH
@Path("/{id}")
@Transactional
public Response update(@Valid Todo todo, @PathParam("id") Long id) {
  Todo entity = Todo.findById(id);
  entity.id = id;
  entity.completed = todo.completed;
  entity.order = todo.order;
   entity.title = todo.title;
  entity.url = todo.url;
  return Response.ok(entity).build();
}
```

Un choix de design a été d'utilisé : le verbe HTTP PATCH pour déclencher la mise à jour plutôt que PUT. Pour pouvoir identifier quelle tâche nous devons corriger, nous utilisons `@Path(/{id})` ou id sera un paramètre de la route HTTP. Pour pouvoir utiliser cet identifiant dans la méthode `update`, nous utilisons `@PathParam("id")`.
Dans le corps de la méthode, on voit que la classe « Todo » dispose d'une méthode `findById` qui provient de Panache et qui nous permet de récupérer l'enregistrement de la base

qui correspond à cet identifiant. La mise à jour consiste ensuite à mettre à jour les attributs de l'objet récupéré. Comme vous pouvez le constater, il n'y a pas d'appel explicite pour écrire les changements en base (un équivalent d'un `flush` ou `save`). L'objet retourné par la méthode `findById` est une entité qui est complètement managée, chaque changement est automatiquement répercuté en base. Le reste du code de la ressource reprend les concepts présentés, voilà donc le résultat final :

Code complet sur programmez.com et GitHub

Si tester c'est douter, alors doutons

Quarkus met à disposition la bibliothèque `quarkus-junit5` qui décore le célèbre JUnit5. Nous allons voir les différents éléments mis à disposition pour faciliter le test de notre application.

Il existe deux grands types de test. Le premier est le test JVM qui correspond aux tests classiques d'une application Java. Le second est le test Natif qui valide le comportement de l'application une fois convertie en exécutable. Dans le cadre de notre application de gestion de tâches, nous allons commencer par le test JVM.

Notre application utilise une base de données PostgreSQL pour persister les tâches. Afin de pouvoir tester, nous allons privilégier une approche qui utilise une « vraie » base plutôt que d'utiliser une version bouchonnée (mock) de l'application (L'objectif de ce choix est d'illustrer la gestion de ressources durant les tests). Quarkus met à disposition le concept de `QuarkusTestResourceLifecycleManager` qui permet de définir des ressources (base de données, cluster Kafka, etc.) à démarrer et arrêter dans le contexte de nos tests. Voilà un exemple qui va s'appuyer sur TestContainer pour démarrer une image Docker de PostgreSQL pour réaliser nos tests :

```
public class DatabaseResource implements QuarkusTestResourceLifecycle
Manager {
  private static final PostgreSQLContainer DATABASE = new PostgreSQL
Container<>("postgres:13.1")
        .withDatabaseName("rest-crud")
        .withUsername("restcrud")
        .withPassword("restcrud")
        .withExposedPorts(5432);

  @Override
  public Map<String, String> start() {
    DATABASE.start();
     return Collections.singletonMap("quarkus.datasource.jdbc.url", DATABASE
.getJdbcUrl());
  }

  @Override
   public void stop() {
     DATABASE.stop();
  }
}
```

Nous allons maintenant créer un cas de test `TodoResourceTest`. Voilà la structure minimale de notre classe :

```
@QuarkusTest
@QuarkusTestResource(DatabaseResource.class)

class TodoResourceTest {
}
```

Un test Quarkus est annoté `@QuarkusTest` et nous retrouvons ici notre `DatabaseResource` qui sera démarrée au début de l'exécution des tests et arrêtée après qu'ils soient tous exécutés.

Commençons par tester notre API par la consultation des tâches. La méthode recommandée par Quarkus pour tester une API HTTP est d'utiliser la bibliothèque RESTAssured.

```
@Test
@Order(1)
void testInitialItems() {
  List<Todo> todos = get("/api").then()
      .statusCode(HttpStatus.SC_OK)
      .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION
_JSON)
      .extract().body().as(getTodoTypeRef());
   assertEquals(4, todos.size());


  get("/api/1").then()
      .statusCode(HttpStatus.SC_OK)
      .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION
_JSON)
      .body("title", is("Introduction to Quarkus"))
      .body("completed", is(true));
}
```

RestAssured s'appuie sur un client HTTP qui nous permet de définir des requêtes de manière simple, mais aussi des outils de vérification (assertion). Dans l'exemple présenté, la première partie du test réalise un appel HTTP GET sur l'adresse `/api`, RestAssured va alors vérifier que le code en retour est bien HTTP OK (code 200), que l'on retrouve bien les entêtes précisant que la réponse est en JSON. RestAssured nous permet de convertir la réponse (JSON) en objet Java (en l'aidant un peu avec une méthode `getTodoTypeRef()` pas présentée dans l'exemple qui donne le type d'une collection de « Todo » dans un format attendu par RestAssured). Une fois le document JSON converti en liste d'objets Java, on peut faire appel aux outils d'assertion classiques (ici `assertEquals`).

Le second test est relativement similaire dans son approche. RestAssured réalise un appel HTTP GET sur la route `/api/1` et une suite de vérifications ont lieu. RestAssured permet de tester un document JSON par une notation JSON Path, c'est ce qui est fait avec `.body("title", is("Introduction to Quarkus"))` par exemple.

En plus de simplifier la génération d'exécutable natif, Quarkus permet de tester cet artefact. Si vous vous demandez pourquoi tester un exécutable alors que nous avons déjà 100 % de couverture de test, gardez en tête que le code JVM et natif n'est pas exactement le même. Dans le cas d'application JVM, Quarkus va s'appuyer sur le compilateur de compilateur du JDK alors que dans le cas d'une image native, le framework utilisera le Graal Compiler. Puisque la cible de ces deux compilateurs est différente, on peut dans de rares cas avoir un écart de comportement (notamment dans les cas où le programme abuse de la réflexion/introspection).
Nous allons donc créer une classe `NativeTodoResourceIT` avec le code suivant :

```
package io.quarkus.sample;

import io.quarkus.test.junit.NativeImageTest;

@NativeImageTest
public class NativeTodoResourceIT extends TodoResourceTest {

}
```

La classe va être annotée `@NativeImageTest` pour indiquer à Quarkus que le test s'exécute contre l'artefact natif. En terme de tests à réaliser, la bonne pratique est d'hériter des tests JVM (ici `TodoResourceTest`). Comme cela, tous les tests JVM seront reproduits en mode natif garantissant le bon fonctionnement.
Afin que ce test soit exécuté, nous avons besoin de l'exécuter une fois que l'artefact natif a été construit. Dans un contexte Maven, la phase la plus adéquate pour exécuter ce genre de test est la phase `verify`. En effet, la phase `verifiy` est associée aux tests d'intégration et s'exécute après la phase `package` qui a pour rôle de construire l'artefact. Voilà la configuration Maven générée par https://code.quarkus.io.

```
<profile>
  <id>native</id>
  <activation>
    <property>
      <name>native</name>
    </property>
  </activation>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>${surefire-plugin.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
            <configuration>
              <systemProperties>
                <native.image.path>${project.build.directory}/${project.
build.finalName}-runner
                </native.image.path>
              </systemProperties>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  <properties>
    <quarkus.package.type>native</quarkus.package.type>
  </properties>
</profile>
```

Cet exemple définit un profil `native` qui associe le plugin failsafe qui exécute les tests d'intégration avec la phase `verify`. Nous pouvons maintenant tester notre exécutable natif avec la commande :

```
$ ./mvnw verify -Pnative
```

Le sujet des tests est relativement large et mériterait à lui seul un article. Dans cette section, nous nous sommes volontairement concentrés sur les éléments basiques pour vous permettre de démarrer rapidement. Si vous souhaitez plus d'informations sur les bouchons (Mocks), l'interception durant les tests ou sur la gestion des profils, nous ne pouvons que vous recommander la lecture du guide suivant https://quarkus.io/guides/getting-started-testing.

Packaging

On touche au but ! On a une API testée, il est temps de construire nos artefacts. Commençons par la construction Java. Sans être très original, nous pouvons lancer la commande :

```
./mvnw package
```

Le résultat de cette commande n'est pas un… mais deux JARs. Si vous regardez dans le dossier `/target`, vous pouvez constater la présence de l'archive `quarkus-todo-apps-1.0.0-SNAPSHOT.jar` qui contient les classes et ressources du projet. Ce qui est plus surprenant est la présence d'une archive `quarkus-todo-apps-1.0.0-SNAPSHOT-runner.jar`. Cette archive est un jar exécutable, vous pouvez donc exécuter une commande du style :

```
$ ./target/quarkus-todo-apps-1.0.0-SNAPSHOT-runner.jar
```

Souvent, JAR exécutable est synonyme de « uber-jar », c'est-à-dire un JAR contenant toutes les classes et ressources du

projet, mais aussi toutes les classes des dépendances du projet. Les « uber-jar » permettent de s'abstenir de gestion de classpath, car possédant toutes les classes nécessaires pour le bon fonctionnement du programme. Cette approche est très appréciée par les développeurs, mais assez peu adéquate dans un contexte de performance. En effet, la première étape que fait la JVM à l'exécution d'une archive JAR, est de la décompresser. Plus l'archive est grosse, plus cette opération est longue et impacte le démarrage de l'application.
Quarkus propose une approche différente avec la notion de « fast-jar ». Lors de la phase de construction, le framework va copier toutes les classes des dépendances dans le dossier `target/lib` et configurer le fichier MANIFEST.MF de l'archive `quarkus-todo-apps-1.0.0-SNAPSHOT-runner.jar` pour lui indiquer que le classpath est composé du dossier `lib`. Cette technique permet d'économiser du temps d'extraction d'archives.
En plus, avec Jandex, Quarkus construit un index contenant le nom d'une ressource ou classe et sa localisation dans le dossier `lib`. Cet index est utilisé par le ClassLoader de Quarkus. En une opération, Quarkus peut transformer le nom d'une classe en chemin de fichier à charger en mémoire. Cette optimisation à un impact direct sur le temps de démarrage de l'application.
Pour finir, nous pouvons également construire un exécutable natif avec la commande :

```
# Pensez à vérifier que vous avez bien installé et configuré GraalVM
./mvnw package -Pnative
```

La compilation native est longue, très longue. Comptez 2-3 minutes pour un projet exemple comme celui de cet article et une dizaine de minutes pour un projet moyen. Le temps s'explique par le besoin du GraalVM d'explorer tous les chemins de votre application pour respecter la « Closed-World Assumption » et ainsi supprimer de manière très agressive le code inutile.

Conclusion

Quarkus est plus qu'un simple framework qui optimise les performances d'une application Java. En misant sur un écosystème éprouvé (MicroProfile qui est la suite de Java EE, Hibernate, Vert.x), Quarkus garantit une continuité dans les compétences acquises par les développeurs depuis plusieurs décennies. Le framework a su investir sur des axes clés comme le rechargement à chaud, Panache ou encore la prise en main du natif pour se différencier.

Durant ce tutoriel, nous avons repris les grands concepts d'une application tels que l'exposition d'une API REST, la persistance en base de données, la configuration, les tests et enfin la construction d'artefact. Bien que volontairement simplifiées, nous espérons que ces quelques lignes vous auront donné envie de découvrir Quarkus.

Fabien Pomerol
Architecte Technique Michelin
@fabienpomerol https://blogit.michelin.io/author/fabien-pomerol/

Julien Millau
Leader Technique Michelin
@devnied https://blogit.michelin.io/author/julien-millau/

PARTIE 3

Retour d'expérience sur l'utilisation de Quarkus chez Michelin

L'un des projets IT majeur chez Michelin concerne la collecte et le traitement de données IoT provenant de pneus connectés et de données télématiques véhicule (périphérique embarqué ou débarqué). Le but de ce projet est de corréler l'usage et l'usure des pneumatiques afin de mieux en comprendre le comportement.

L'ensemble de ces données collectées nous permettent de travailler sur des thématiques telles que la prédiction de fin de vie des pneumatiques, la détection de perte de pression, ou bien aider les équipes de recherche et développement Michelin dans l'amélioration de nos produits.
Ce projet innovant en constante évolution depuis plus de 5 ans est composé de différentes briques logicielles écrites en Java. La solution repose aujourd'hui sur une architecture micro-services événementiels (Event Driven). **Figure 1**
Celle-ci se compose d'une trentaine de micro-services principalement basés sur le toolkit Eclipse Vert.x. Ces derniers ont pour fonction d'exposer des API HTTP et des consommer et produire des messages d'un broker Apache Kafka. **Figure 2**

Pourquoi vouloir utiliser un Framework ?

Eclipse Vert.x est un toolkit. En cela, il n'impose pas de cadre comme peut le faire un framework.
Aujourd'hui, l'utilisation de Vert.x répond à notre besoin, mais au prix d'une maintenance lourde et importante de tous nos composants développés en interne et d'une longue phase d'apprentissage pour nos dévelop-

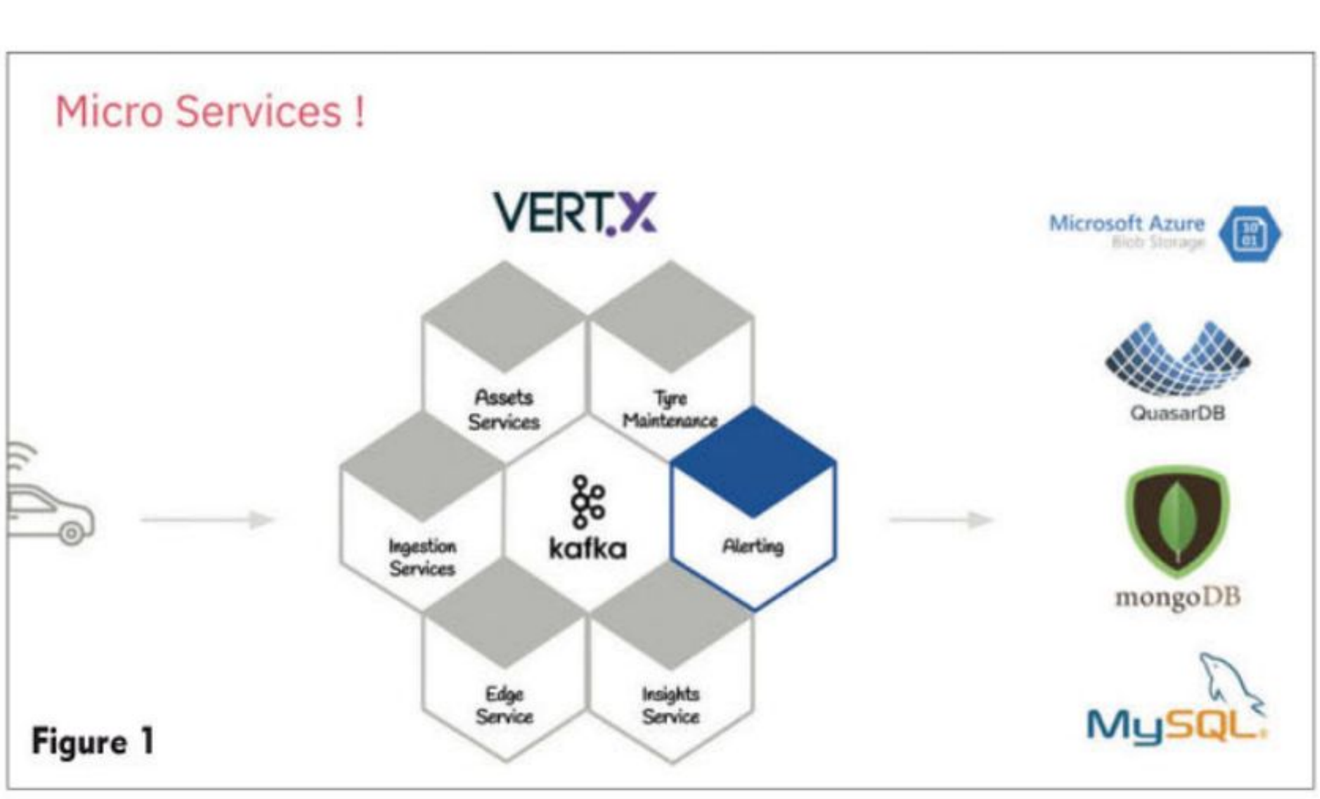


Figure 1

peurs. L'utilisation d'un framework nous permet de donner un cadre aux développeurs avec des conventions, des librairies, une documentation et une communauté pour faciliter la montée à bord des nouveaux arrivants. De plus, cela simplifie l'intégration de différentes librairies externes avec le BOM Quarkus en s'assurant de leurs inter- compatibilité et ainsi diminuer drastiquement le nombre de nos tests d'intégration sur ces libraires externes.

En quoi Quarkus et différent des autres frameworks ?

Notre cahier des charges avait plusieurs contraintes. Nous voulions un framework réactif et "passe-partout" avec comme cible de déploiement aussi bien des conteneurs que des fonctions/lambda.
D'un point de vue logiciel, celui-ci devait nous permettre de diminuer nos temps de développement, de simplifier la création de nos API, d'avoir un support natif des bases de données courantes (Postgres, Mongo, Redis, Mysql) et une intégration avec Apache Kafka.
La possibilité de compiler nativement nos micro-services était aussi importante afin de diminuer nos coûts d'infrastructure. En effet la compilation native permet de réduire notre consommation en ram,nous estimons le gain en termes de mémoire consommée supérieur à 60 % par service.
Le dernier point essentiel était la possibilité de réutiliser du code existant écrit en Vert.x sans trop de modifications.
En comparant Quarkus et Spring, nous avons fait le choix de retenir Quarkus. La promesse de Quarkus est belle : un framework basé sur du Vert.x, unifiant l'impératif et le réactif supporté par Red Hat avec une communauté grandissante et offrant la possibilité de compiler nativement nos applications.

Les promesses de Quarkus sont-elles au rendez-vous ?

Globalement oui, le fait de pouvoir mixer le réactif et l'impératif nativement au sein de la même application est très intéressant. Le nombre important d'extensions disponibles est un vrai point positif, il faut voir ces extensions comme des dépendances du projet. Les extensions configurent, démarrent et intègrent un framework ou une technologie dans l'application. Elles fournissent également les bonnes informations à GraalVM pour que l'application puisse se compiler en natif. Le support encore en preview de certaines librairies Spring comme Spring data et Spring Security permet de mixer des fonctionnalités intéressantes de plusieurs frameworks, mais vous obligera à faire des choix au moment de votre phase de design. Cette ouverture de Quarkus est vraiment très intéressante et permet en fonction du contexte du projet de choisir parmi plusieurs implémentations pour une même fonctionnalité.

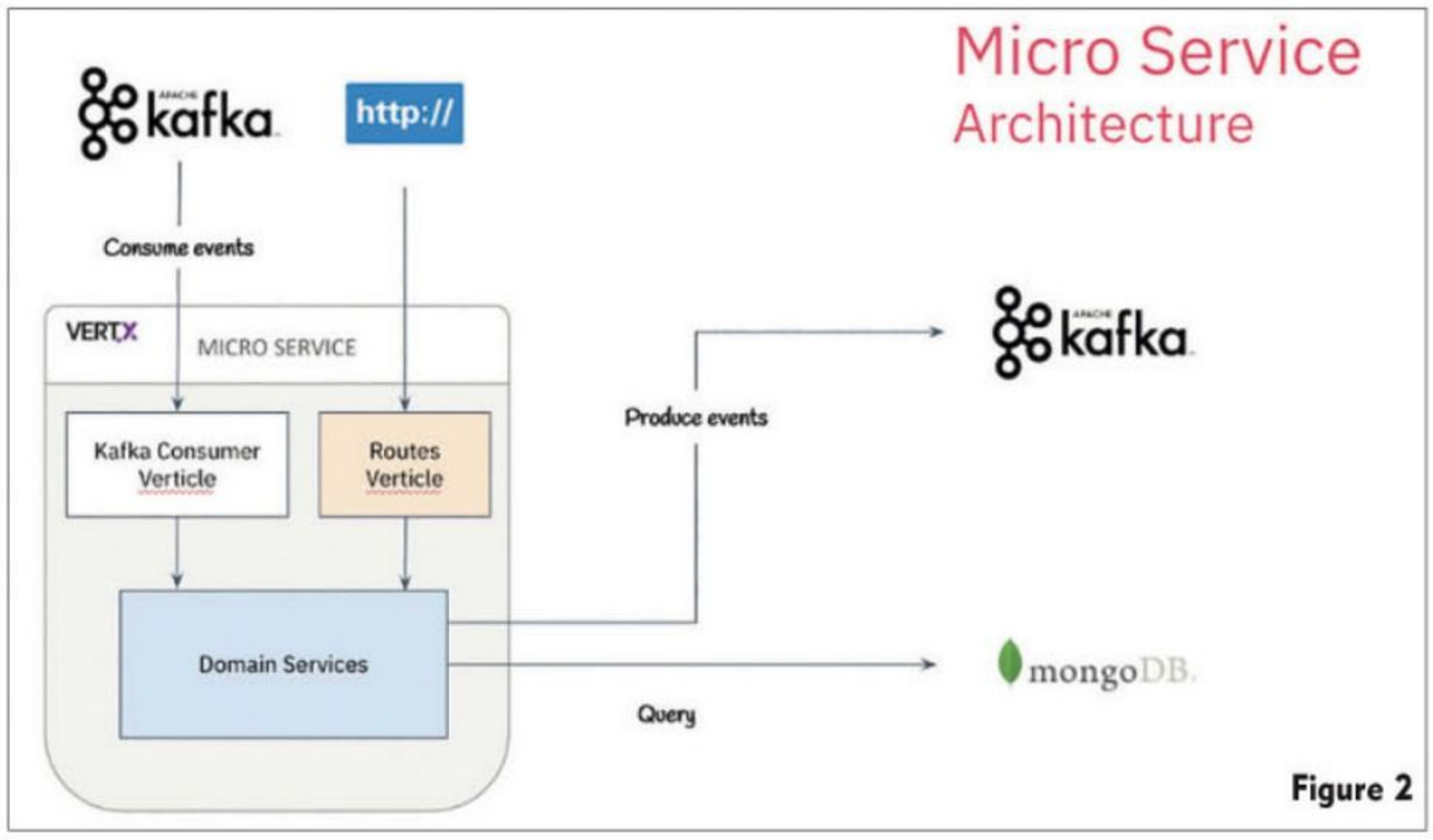


Figure 2

La création de "librairie" pour factoriser du code entre micro-services se traduit souvent par la création une "extension" et cette partie est pour l'instant peu documentée et assez compliquée à mettre en œuvre surtout pour la partie native sans passer par des recherches dans le code source de Quarkus.
Quarkus est encore jeune et en évolution rapide. Le rythme des releases est important avec une version majeure par mois et plusieurs mineures. Cela oblige l'équipe à suivre et appliquer ces releases de façon régulière afin de ne pas prendre trop de retard et avoir des migrations trop coûteuses (le rythme de release devrait réduire à partir de la version 2.x de Quarkus).

Quels gains à l'utilisation de Quarkus ?

Concernant les promesses en termes d'économie de ressources, nous n'avons pas encore passé le dernier cap qui est de compiler nos services en natif avec GraalVM. Nous avons pour l'instant seulement fait des expérimentations. Le gain est très intéressant, mais le chemin pour y arriver est plus compliqué avec de nombreux bugs de GraalVM et Quarkus. Le temps de compilation beaucoup plus long et le positionnement pas très clair d'Oracle sur GraalVM et GraalVM Entreprise sont pour l'instant des points à regarder.
Le fait de passer de Vert.x à Quarkus sans compilation native ne nous amène quasi aucun gain en termes d'utilisation de ressource (CPU/RAM). Vert.x étant déjà optimisé pour permettre ce gain, la différence aurait pu se voir si nous étions passés d'un service d'un autre framework à Quarkus. Le gain se fera lorsque nous passerons sur une compilation native.
Un des gains importants que nous avons actuellement perçus suite à l'utilisation de Quarkus est une réduction de notre temps de développement par rapport à notre existant en Vert.x

Un gain pour les développeurs ?

Le fait de passer de Vert.x à un framework a simplifié énormément le développement et les tests pour les développeurs et a grandement accru notre vélocité.
Quarkus en tant que tel n'apporte pas plus de simplicité qu'un autre framework, c'est même l'inverse, car il est encore jeune et l'intégration dans les IDE n'était pas encore complète.
Le mode de développent avec le rechargement à chaud du code modifié fonctionne mal (comme sur beaucoup de frameworks).
De plus la phase "d'augmentation" qui introduit une modification du byte code sur certaines classes par rapport au code source de l'application doit être connue et comprise par les développeurs, car elle a un impact sur l'utilisation de certains outils comme Jacoco pour la couverture de code.
Afin de comprendre le fonctionnement de certaines fonctionnalités de Quarkus, la documentation n'est pas encore suffisante et les développeurs sont encore souvent obligés de regarder en détail le code des extensions Quarkus pour comprendre le fonctionnement de certaines fonctionnalités.

Suite de votre utilisation de Quarkus ?

Aujourd'hui, notre but n'est pas de tout réécrire en Quarkus. Notre base Vert.x va rester, mais tout nouveau micro-service sera écrit en Quarkus. Nous avons actuellement 3 micros-services en production entièrement écrits en Quarkus. Le cadre que celui-ci propose est très apprécié de nos développeurs et nous offre suffisamment de flexibilité pour répondre à nos besoins.
Un de nos axes de travail sera de passer sur cette compilation native dans les prochains mois. Red Hat propose également une distribution communautaire de GraalVM avec Mandrel qui sera certainement un levier à l'adoption de la compilation native.

Quarkus et Google Cloud Platform

Loïc Mathieu
Consultant
Zenika Lille

zenika
<animés par la passion>

Quarkus est un framework de développement de microservice pensé pour le cloud et les conteneurs. Il est pensé pour avoir une utilisation mémoire réduite et un temps de démarrage le plus court possible. Il se base principalement sur des standards (Jakarta EE, Eclipse MicroProfile,…) et permet l'utilisation de librairies Java matures et très répandues via ses extensions (Hibernate, RESTeasy, Vert.X, Kafka,…). Quarkus a été pensé pour le cloud dès sa conception, il permet le développement d'applications Cloud Ready (tel que défini par le principe des applications 12 Factor - https://12factor.net/) et Cloud Native (utilisation des capacités des cloud publics pour développer vos applications). Quarkus permet entre autres la création de fonctions Google Cloud Functions pour le développement de vos applications.

Implémenter une Google Cloud Function avec Quarkus

Quarkus permet d'implémenter une fonction Google Cloud Function de type *background* ou *HTTP* de plusieurs manières :

- Via les API de Google Cloud
- Via Funqy : une framework de développement de fonction agnostique du fournisseur de Cloud.
- Via une des extensions HTTP de Quarkus: RESTEasy (JAX-RS), Vert.x reactive routes, Undertow (Servlet), Spring Web.

Background Function avec l'API Google Cloud

Première étape, créer un projet Quarkus avec le plugin Maven de Quarkus :

```
mvn io.quarkus:quarkus-maven-plugin:1.12.2.Final:create \
  -DprojectGroupId=org.acme \
  -DprojectArtifactId=google-cloud-functions \
  -Dextensions="resteasy,google-cloud-functions"
```

Cette commande Maven va vous créer un projet contenant une fonction background et une fonction HTTP.
La fonction background créée est la suivante :

```
@ApplicationScoped //1
public class HelloWorldBackgroundFunction implements BackgroundFunction<HelloWorldBackgroundFunction.StorageEvent> { //2

  @Override
  public void accept(StorageEvent event, Context context) throws Exception { //3
    System.out.println("Receive event on file: " + event.name);
  }

  public static class StorageEvent {
    public String name;
  }
}
```

1. Pour que Quarkus détecte votre fonction, elle doit être annotée avec une annotation CDI.
2. La fonction utilise l'API de Google, ici elle étend BackgroundFunction.
3. La fonction sera déclenchée par un évènement représenté par l'objet StorageEvent, pour plus de simplicité j'ai uniquement défini sa propriété name.

Cette fonction sera déclenchée par un évènement de type Google Cloud Storage, et va afficher dans ses logs le nom de l'objet créé ou mis à jour dans Cloud Storage.
Comme l'exemple vient avec deux fonctions, il vous faut supprimer la fonction HTTP qui a aussi été créée, ou sélectionner la fonction à déployer. Ces étapes sont expliquées dans le guide des Google Cloud Functions et omises ici pour des raisons de simplicité.
Pour packager la fonction vous devez utiliser la commande Maven suivante : `mvn clean package`. Cette commande va créer un uberjar dans le répertoire `target/deployment` que vous pourrez ensuite déployer comme une Cloud Functions.
Pour déployer cet uberjar, vous pouvez utiliser la commande gcloud suivante :

```
gcloud beta functions deploy quarkus-example-storage \
--entry-point=io.quarkus.gcp.functions.QuarkusBackgroundFunction \
--trigger-resource quarkus-hello \
--trigger-event google.storage.object.finalize \
--runtime=java11 --source=target/deployment
```

Cette commande utilise comme entry point `io.quarkus.gcp.functions.QuarkusBackgroundFunction` et non votre fonction. Cet entry point va démarrer Quarkus, puis localiser votre fonction dans le conteneur CDI.
Pour déclencher la fonction via un évènement Cloud Storage, on doit définir l'évènement de déclenchement `google.storage.object.finalize` et préciser un nom de bucket, ici `quarkus-hello`.
L'ajout d'un objet dans le bucket quarkus-hello va alors déclencher votre fonction. Vous pouvez tester cela avec la commande gsutil qui permet d'envoyer des fichiers dans un bucket Cloud Storage ou simuler un déclenchement via :

```
gcloud functions call quarkus-example-storage \
--data '{"name":"test.txt"}'.
```

On peut noter qu'on utilise la taille par défaut de fonction qui est de 256Mo, ceci est suffisant pour le démarrage du conteneur de la fonction (Jetty) et de notre application Quarkus. On tire parti de la consommation mémoire faible de Quarkus. Si on regarde dans les logs de démarrage de notre fonction, on constate que notre fonction prend 2,1s à démarrer, dont 1,1s pour Quarkus et le reste (1s donc) pour le conteneur de fonction (Jetty). On tire parti du démarrage rapide de Quarkus.

HTTP Function avec l'API Google Cloud

Dans le projet Quarkus créé, une fonction HTTP a été générée dont voici le code :

```
@ApplicationScoped //1
public class HelloWorldHttpFunction implements HttpFunction { //2

  @Override
  public void service(HttpRequest httpRequest, HttpResponse httpResponse)
 throws Exception { //3
    Writer writer = httpResponse.getWriter();
    writer.write("Hello World");
  }
}
```

1. Pour que Quarkus détecte votre fonction, elle doit être annotée avec une annotation CDI.
2. La fonction utilise l'API de Google, ici elle étend HttpFunction.
3. Pour implémenter notre fonction, on a accès à la requête et la réponse HTTP.

Cette fonction fait un simple Hello World.

Pour packager la fonction vous devez utiliser la commande Maven suivante : `mvn clean package`. Cette commande va créer un uberjar dans le répertoire target/deployment que vous pourrez ensuite déployer comme une Cloud Functions.

Pour déployer cet uberjar vous pouvez utiliser la commande gcloud suivante :

```
gcloud beta functions deploy quarkus-example-http \
--entry-point=io.quarkus.gcp.functions.QuarkusHttpFunction \
--runtime=java11 --trigger-http --source=target/deployment
```

Cette commande utilise comme entry point `io.quarkus.gcp.functions.QuarkusHttpFunction` et non votre fonction. Cet entry point va démarrer Quarkus puis localiser votre fonction dans le conteneur CDI. Pour plus d'informations sur les fonctions background et HTTP via l'API Google, voir le guide suivant : https://quarkus.io/guides/gcp-functions

Background Function avec Funqy

L'exemple de fonction background que nous avons vu précédemment a comme inconvénient d'être dépendant de l'API Google Cloud Functions. Elle n'est donc pas portable d'un cloud à l'autre. C'est là qu'entre en piste **Funqy**, l'API de fonction de Quarkus, agnostique du cloud provider. On peut réécrire la fonction background précédente via Funqy de la manière suivante :

```
public class HelloWorldBackgroundFunction { //1

  @Funq //2
  public void accept(StorageEvent event) throws Exception { //3
    System.out.println("Receive event on file: " + event.name);
  }

  public static class StorageEvent {
    public String name;
  }
}
```

1. Votre fonction n'est plus dépendante de l'API de Google.
2. L'annotation `@Funq` permet de dire à Funqy que c'est la méthode à appeler pour chaque déclenchement de fonction.
3. La fonction sera déclenchée par un évènement représenté par l'objet StorageEvent, pour plus de simplicité j'ai uniquement défini sa propriété name.

Pour utiliser Funqy, il faut remplacer l'extension `quarkus-google-cloud-functions` par l'extension `quarkus-funqy-google-cloud-functions`.

Après avoir packager votre application avec Maven, vous pouvez la déployer avec gcloud de la même manière que précédemment sauf qu'il vous faut utiliser l'entry point `io.quarkus.funqy.gcp.functions.FunqyBackgroundFunction`.

Pour plus d'informations sur Funqy : https://quarkus.io/guides/funqy-gcp-functions

HTTP Function avec RESTEasy

L'exemple de fonction HTTP que nous avons vu précédemment a comme inconvénient d'être dépendant de l'API Google Cloud Functions. Elle n'est donc pas portable d'un cloud à l'autre. De plus, la manipulation directe des objets request et response peut être fastidieuse. Pour pallier à cela, Quarkus permet d'utiliser ses différentes extensions HTTP à la place de l'API de Google, votre fonction devient alors agnostique au cloud provider et beaucoup plus simple à écrire. L'exemple précédent peut être réécrit de la manière suivante en utilisant l'extension RESTEasy :

```
@Path("/") //1
public class HelloWorldHttpFunction{ //2

  @GET //3
  @Produces(MediaType.TEXT_PLAIN)
  public void String() throws Exception {
    return "Hello World";
  }
}
```

1. On utilise ici l'annotation JAX-RS permettant de définir la classe comme une ressource REST.
2. Notre classe ne dépend plus de l'API Google Functions.
3. Notre fonction est implémentée via une opération REST classique (on peut même en définir plusieurs si nécessaire).

Pour utiliser RESTEasy, il faut remplacer l'extension `quarkus-google-cloud-functions` par l'extension `quarkus-google-cloud-functions-http` et ajouter l'extension RESTEasy : `quarkus-resteasy-jackson`.

Après avoir packager votre application avec Maven, vous pouvez la déployer avec gcloud de la même manière que précédemment sauf qu'il vous faut utiliser l'entry point `io.quarkus.gcp.functions.http.QuarkusHttpFunction`.

Pour plus d'informations sur l'utilisation d'extension HTTP via des Cloud Functions : https://quarkus.io/guides/gcp-functions-http

Conclusion

Quarkus permet de facilement implémenter des fonctions Google Cloud. Funqy est une bonne solution pour rendre agnostique aux API Google vos fonctions background et l'utilisation directe des extensions HTTP de Quarkus (RESTEasy, Undertow, Reactive Routes, Spring Web) facilite grandement l'écriture d'application REST. De par sa faible empreinte mémoire et sa capacité à démarrer très vite, Quarkus est une très bonne solution pour écrire vos fonctions Google Cloud.

JUnit : il est temps de passer la 5e !

Saviez-vous que JUnit 5 a déjà 4 ans ? Pourtant, un grand nombre de projets Java sont encore testés avec JUnit 4, qui est sorti… il y a une bonne quinzaine d'années ! Énormément de choses ont évolué depuis 2006, Java a pris 9 versions ! Ne serait-il donc pas temps de remettre nos tests au goût du jour ? L'équipe de JUnit a profité de cette 5ème version pour restructurer complètement le framework. De nombreuses fonctions ont été ajoutées ou retravaillées afin de s'adapter aux nouveaux paradigmes de l'écosystème Java.

Juliette de Rancourt

Intéressée par les tests et l'open source, j'ai été amenée à contribuer sur JUnit 5. Faisant désormais partie de l'équipe qui maintient le framework depuis plus d'un an, j'ai pu découvrir les coulisses d'un projet open source de grande ampleur. Je travaille également en tant que développeuse full-stack chez Shodo, où j'essaye avant tout de construire du code maintenable et testable.

Julien Topçu

Tech Coach chez Shodo, j'accompagne le développement de logiciels à forte valeur métier en usant de techniques issues du Domain-Driven Design, le tout propulsé en Extreme Programming dans la philosophie Kanban #NoEstimates. Membre de la fondation OWASP, j'évangélise sur les techniques de sécurité applicative afin d'éviter de se faire hacker bien comme il faut.

Une plus grande modularité

Si vous êtes familiers de JUnit 4, vous connaissez ce bon vieux junit.jar à rajouter dans votre projet. Dans sa version 5, JUnit est composé de 3 modules: JUnit Jupiter, JUnit Vintage et JUnit Platform.

JUnit Jupiter est le moteur de test de JUnit 5. La 5e version a été baptisée selon la 5e planète du système solaire. En ajoutant org.junit.jupiter:junit-jupiter dans votre classpath de test, vous allez pouvoir profiter du nouveau modèle de programmation de test. On le verra tout au long de cet article, le mot d'ordre de cette version majeure est l'extensibilité. JUnit Jupiter offre une multitude de moyens d'étendre et de customiser le comportement du moteur de test selon vos besoins grâce au nouveau mécanisme d'extensions.

La rétrocompatibilité n'a pas été oubliée dans cette nouvelle version. Grâce à JUnit Vintage, pas besoin de migrer tout son code d'un coup de JUnit 4 à JUnit 5. Le jar org.junit.vintage:junit-vintage-engine vous assurera la cohabitation des tests écrits dans l'ancien et le nouveau paradigme afin d'opérer une migration itérative de vos tests. Il vous permettra aussi de continuer à utiliser des frameworks de tests basés sur JUnit qui n'auraient pas encore migré vers JUnit Jupiter.

JUnit Platform représente l'infrastructure du framework. Il contient l'exécuteur des tests sur la console c.-à-d. ConsoleLauncher, ainsi que les fondations permettant le lancement de framework de tests dans la JVM. De ce fait, il offre une API de TestEngine permettant à n'importe quel développeur de développer son propre moteur de tests JUnit 5. C'est bien sûr sur ces API que JUnit Jupiter s'appuie. JUnit Platform est intégré par défaut dans bon nombre d'IDE et est supporté par les outils de build tels que Maven, Gradle et Ant. Il n'est pas nécessaire de rajouter son jar org.junit.platform:junit-platform-launcher dans vos dépendances, sauf si vous souhaitez utiliser une version plus récente de JUnit que celle intégrée dans votre IDE. À noter que JUnit Platform ne suit pas le même versioning que JUnit Jupiter et JUnit Vintage, étant actuellement en 1.7.x contre 5.7.x pour le reste des modules. Il faut savoir toutefois que tous ces modules de JUnit 5 requièrent un runtime en Java 8 minimum.

Une gestion du cycle de vie des tests bien plus claire

L'équipe de JUnit a souhaité offrir une version plus souple, moins verbeuse et plus simple d'utilisation de son framework. Dans cette philosophie, il n'est plus nécessaire de déclarer ses classes et ses méthodes de test avec le scope public, la visibilité package suffit (sans 'modifier'). Les annotations de cycle de vie ont été complètement repensées pour plus de clarté. Vous pouvez toujours initialiser ou détruire le contexte de vos tests avant et après chacun d'entre eux et/ou autour de l'exécution de tous les tests de la classe. Cependant, exit les @Before, @After, @BeforeClass et @AfterClass qui étaient orientées structure de code, pour accueillir maintenant @BeforeEach, @AfterEach et @BeforeAll et @AfterAll (...tests) beaucoup plus sémantique. Les méthodes annotées par @BeforeAll et @AfterAll devront toujours être statiques sauf si vous changez la politique d'instanciation des tests en la définissant par classe.

Par défaut, JUnit 5 crée une instance de votre classe de test pour chaque méthode de tests décrite dans celle-ci. La raison est d'assurer l'isolation totale de vos tests en évitant les effets de bord potentiels sur vos données de tests qui seraient provoqués par les autres tests de votre classe. Ce comportement est hérité des précédentes versions de JUnit. Cependant, il est possible de demander au framework de créer une seule instance de votre classe pour toutes les méthodes qui la compose au moyen de @TestInstance(Lifecycle.PER_CLASS) à placer au-dessus de la définition de votre classe.

Pour spécifier à JUnit qu'une méthode est un test, l'annotation n'a pas changé: @Test, mais le piège est dans le package ! Il faut faire bien attention à prendre maintenant cette annotation du nouveau package org.junit.jupiter.api (et non plus org.junit) sous peine de ne pas voir ses tests s'exécuter.

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.TestInstance.Lifecycle.PER_METHOD;

@TestInstance(PER_METHOD) // valeur par défaut, pas besion de le spécifier
class MyTest {
  @BeforeAll
  static void avantTousLesTests() {}

  @AfterAll
  static void apresTousLesTests() {}

  @BeforeEach
  void avantChaqueTest() {}

  @AfterEach
  void apresChaqueTest() {}

  @Test
  void unTest() {}
}
```

Les tests imbriqués

JUnit Jupiter supporte maintenant nativement les tests imbriqués. Ils permettent une organisation plus fine de vos tests, en créant des sous-groupes. Ces groupes se matérialisent sous la forme de classes imbriquées annotées par @Nested. Cette imbrication permet de créer une hiérarchie entre les tests, offrant une plus grande granularité dans la gestion de leur cycle de vie. Chaque @Nested classe peut avoir ses propres méthodes d'initialisation et de destruction de contexte, tout en héritant de celles de leurs classes parentes. C'est-à-dire que les méthodes annoncées par @BeforeEach dans les classes parentes, s'exécutent avant celles de la classe imbriquée et ceux pour tous les tests de cette dernière. Il en va de même pour @BeforeAll et de manière symétrique pour @AfterEach et @AfterAll.

```
public class NestedTests {
  @BeforeEach
  void globalSetup() {
    System.out.println("A");
  }

  @Test
  void test1() {
    System.out.println("1");
  }

  @Test
  void test2() {
    System.out.println("2");
  }

  @Nested
  class Inside {
    @BeforeEach
    void localSetup() {
      System.out.println("B");
    }

    @Test
    void test3() {
      System.out.println("3");
    }

    @Test
    void test4() {
      System.out.println("4");
    }
  }
}
```

Pour l'exemple ci-dessus, l'exécution des tests aura le résultat suivant dans la console : (A 1) (A 2) (A B 3) (A B 4)

Quelques nouvelles assertions

Au niveau des assertions, on retrouve ce qui existait dans JUnit 4, ainsi que quelques nouveautés. Les assertions classiques anciennement présentes dans org.junit.Assert (telles que assertEquals, assertTrue, assertNull…) sont disponibles dans la nouvelle classe org.junit.jupiter.api.Assertions.
Pour vérifier qu'une exception est bien levée dans un test, on peut maintenant utiliser assertThrows au lieu de devoir passer par une @Rule ou par la syntaxe @Test(expected = …). Ces deux méthodes n'existent d'ailleurs plus dans JUnit Jupiter.

```
@Test
void testThrowingException() {
  Exception exception =
    assertThrows(RuntimeException.class, () -> failingCode());
  assertEquals("Failure!", exception.getMessage());
}

private void failingCode() {
  throw new RuntimeException("Failure!"));
}
```

La nouvelle méthode assertAll peut également être utile, notamment lors de l'exécution des tests. Elle permet de grouper plusieurs assertions, et de forcer leur vérification, quel que soit le résultat des précédentes. On a donc un rapport complet de toutes les assertions du test, au lieu de s'arrêter au premier échec. Dans l'exemple suivant, on aura donc deux assertions en échec dans le même test.

```
@Test
void groupedAssertions() {
  var list = List.of("Programmez", "Magazine", "JUnit");
  assertAll(
    () -> assertEquals(4, list.size()),
    () -> assertTrue(list.contains("Java")));
}
```

Le but de JUnit est de fournir les assertions essentielles pour pouvoir écrire des tests simples sans importer de dépendances supplémentaires. Pour des besoins plus complexes en termes de vérification, ou pour avoir des assertions plus lisibles, il est recommandé d'utiliser un framework dédié comme AssertJ.

Des tests paramétrés plus flexibles

Pour ce qui est des tests paramétrés, on peut dire que Jupiter a révolutionné la façon dont ils peuvent être écrits. Cette fonctionnalité — en général très appréciée des développeurs — est maintenant beaucoup plus flexible et moins verbeuse qu'avec JUnit 4. Il y a aussi de nouvelles façons de créer les jeux de données, pour s'adapter à n'importe quel besoin de test.
Pour pouvoir utiliser cette fonctionnalité, il faut avoir org.junit.jupiter:junit-jupiter-params dans le classpath (il sera déjà présent si vous utilisez l'agrégat org.junit.jupiter:junit-jupiter qui l'inclut). Ensuite, plus besoin d'utiliser des runners ni d'annoter des attributs de classe ! Une seule annotation suffit : @ParameterizedTest. Quant aux paramètres, ils sont directement passés en tant qu'arguments de la méthode de test.

```
@ParameterizedTest
@ValueSource(strings = {"Programmez", "Magazine"})
void a_simple_test(String word) {
  assertTrue(word.length() > 8);
}
```

Dans l'exemple précédent, les valeurs sont fournies via @ValueSource. Cette annotation est pratique dans le cas d'un test avec un seul argument de type primitif. Pour ajouter une valeur de test nulle ou vide, on peut également la combiner avec

@NullSource, @EmptySource ou @NullAndEmptySource. Dans le cas d'un test avec plusieurs paramètres, on peut écrire les jeux de données dans un format semblable au CSV grâce à @CsvSource. Si les données sont présentes dans un fichier CSV, on peut également utiliser @CsvFileSource en donnant le chemin du fichier.

```
@ParameterizedTest
@CsvSource({
  "Programmez, 10",
  "Magazine, 8"
})
void a_test_with_two_parameters(String word, int expectedLength) {
  assertEquals(expectedLength, word.length());
}
```

On remarque dans cet exemple que le paramètre expectedLength est un entier, alors que @CsvSource ne prend que des chaînes de caractère en argument. Cette transformation se fait grâce au mécanisme des "implicit converters" qui peuvent interpréter un String comme un autre type (ici en un entier). Ces convertisseurs fonctionnent nativement avec des types plus complexes comme File, Path, BigDecimal, URL, ou encore avec des énumérations. Les classes de l'API date de Java sont également convertibles, comme le montre l'exemple suivant. Cela permet d'écrire des tests concis, tout en ayant des jeux de données très lisibles.

```
@ParameterizedTest
@CsvSource({
    "2021-01-01, JANUARY",
    "2020-04-15, APRIL"
})
void a_test_with_converters(LocalDate date, Month expectedMonth) {
  assertEquals(expectedMonth, date.getMonth());
}
```

Ces convertisseurs sont implicites, c'est-à-dire qu'ils sont implémentés et déclarés directement par JUnit 5 (la liste exhaustive de ceux-ci est disponible dans le guide d'utilisateur). Ils aident à réduire le code nécessaire à la préparation des données de test, et à se concentrer sur la logique métier testée.
Si besoin, il est possible d'implémenter ses propres convertisseurs grâce à SimpleArgumentConverter. Il faut ensuite le déclarer sur le paramètre à l'aide de @ConvertWith (contrairement aux convertisseurs implicites qui sont présents par défaut).

```
@ParameterizedTest
@ValueSource(strings = {"JANUARY", "FEBRUARY", "MARCH"})
void a_test_with_explicit_conversion(
    @ConvertWith(MonthToNumberConverter.class) int month) {
  assertTrue(month <= 3);
}

class MonthToNumberConverter extends SimpleArgumentConverter {
  @Override
  protected Object convert(Object source, Class<?> targetType) {
    return Month.valueOf((String) source).getValue();
  }
}
```

Une dernière façon de créer des valeurs de paramètres, qui peut s'avérer utile quand on manipule des objets plus complexes : @MethodSource. On va ici créer programmatiquement les cas de tests dans une méthode statique dédiée (qui peut se situer dans la classe de test ou dans une classe externe), chaque cas étant symbolisé par un objet Arguments.

```
@ParameterizedTest
@MethodSource("somePeople")
void a_test_with_method_source(String name, int age, Month birthMonth) {
  // ...
}

private static Stream<Arguments> somePeople() {
  return Stream.of(
      Arguments.of("Alice", 25, JUNE),
      Arguments.of("Bob", 32, AUGUST)
  );
}
```

Plus besoin de dupliquer les tests pour vérifier des contrats !

Il vous est peut-être déjà arrivé de devoir dupliquer des tests pour vérifier que toutes les implémentations d'une de vos interfaces observent toutes le même comportement ? Ou bien qu'il était très fastidieux de rajouter les tests sur les equals et les hashcodes ? Grâce à cette nouvelle version, vous pouvez dorénavant écrire des interfaces qui comportent des tests. Si on reprend l'exemple de nos equals et hashcodes, vous pouvez créer une interface qui va tester les relations d'égalité pour tous vos objets.

```
public interface EqualityTest<T> {

  T createValue();
  T createOtherValue();

  @Test
  default void should_be_equal() {
    assertThat(createValue()).isEqualTo(createValue());
    assertThat(createValue().hashCode()).isEqualTo(createValue().hashCode());
  }

  @Test
  default void should_not_be_equal() {
    assertThat(createValue()).isNotEqualTo(createOtherValue());
    assertThat(createValue().hashCode()).isNotEqualTo(createOtherValue
().hashCode());
  }
}
```

L'interface EqualityTest est générique, ce qui permettra de l'appliquer à la classe que l'on désire tester. Elle contient deux méthodes, createValue et createAnotherValue, qui ne sont pas implémentées ici. Elles forceront le test concret qui implémente cette interface à fournir des valeurs différentes pour l'objet que l'on souhaite tester. On retrouve ensuite deux implémentations de méthodes par défaut qui sont en réalité nos tests génériques, testant l'égalité et l'inégalité au moyen des méthodes précédentes. Il suffit ensuite de créer un test qui étend cette interface.

```
class BookmarkTest implements EqualityTest<Bookmark> {
```

```
  @Override
  public Bookmark createValue() {
    return Bookmark.create("http://www.test.com", "name");
  }

  @Override
  public Bookmark createOtherValue() {
    return Bookmark.create("http://www.test2.com", "other name");
  }
}
```

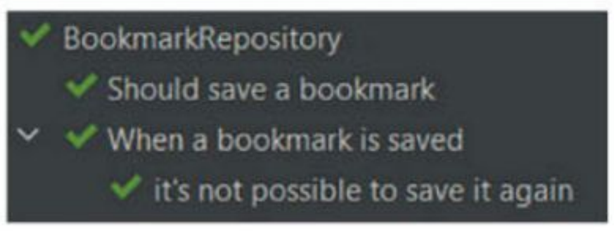
A l'exécution de ce test, vous verrez que les tests décrits dans EqualityTest seront joués au même titre que ceux présents dans BookmarkTest. Vous pouvez alors tester les relations d'égalité d'une autre classe, simplement en implémentant EqualityTest dans un nouveau test et ceci sans surcoût !

Plus de lisibilité avec les display names

Si vous cherchez à rendre vos rapports de tests plus lisibles dans la console ou dans votre IDE, vous pouvez utiliser l'annotation @DisplayName. Se positionnant sur vos classes et méthodes de tests imbriquées ou non, elle vous permet de remplacer l'affichage par une phrase de votre choix.

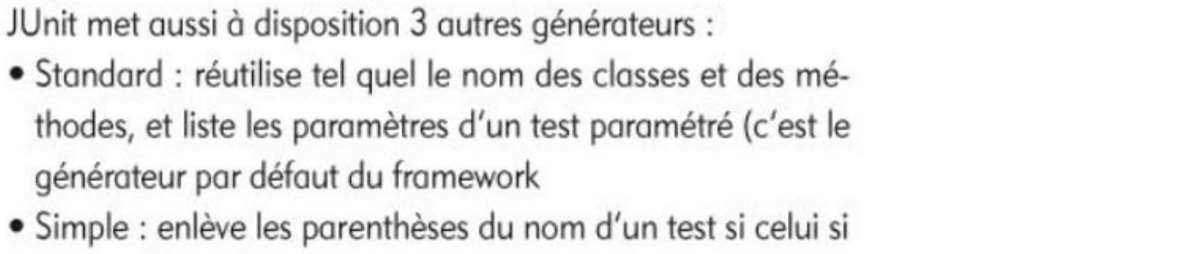
```
import org.junit.jupiter.api.DisplayName;

@DisplayName("BookmarkRepository")
class BookmarkRepositoryTest {

  @Test
  @DisplayName("Should save a bookmark")
  void should_save_bookmark(Bookmark bookmark) {}

  @Nested
  @DisplayName("When a bookmark is saved")
  class WhenBookmarkIsSaved {
    @Test
    @DisplayName("it's not possible to save it again")
    void should_not_save_an_already_existent_bookmark() {}
  }
}
```

Avec le code précédent, le rapport de tests ressemblera à ceci :



Il est aussi possible de personnaliser un test paramétré. Par défaut, ces tests vont générer une ligne dans le rapport de test en listant les valeurs des paramètres qui seront utilisés lors de chaque exécution. L'attribut name de @ParameterizedTest vous permet de spécifier un template pour la génération de ses lignes. Vous pouvez y référencer la valeur de vos paramètres en utilisant {N} où N est la position du paramètre désiré dans votre méthode de test.

```
@CsvSource({
    "http://www.junit.org, JUnit, Testing",
    "https://www.oracle.com/fr/java/, Java, Language"
})
@DisplayName("Should create the bookmark")
@ParameterizedTest(name = "for the url {0} named {1} & tagged {2}")
void should_create_the_bookmark(String url, String name, String tag) {}
```



Cependant, si vous ne désirez pas positionner @DisplayName sur chaque test, vous pouvez utiliser les DisplayNameGenerator. JUnit fournit des générateurs qui vont se baser sur le nom des méthodes de test pour générer des rapports plus digestes. Par exemple, beaucoup de développeurs utilisent la nomenclature should + undescores pour nommer leur test (e.g. should_do_something). L'utilisation du générateur ReplaceUnderscores a pour effet de remplacer automatiquement les underscores par des espaces lors de la génération du rapport. Pour l'utiliser, il suffit de positionner l'annotation @DisplayNameGeneration(Display NameGenerator. ReplaceUnderscores.class) au-dessus de sa classe de test.

JUnit met aussi à disposition 3 autres générateurs :

- Standard : réutilise tel quel le nom des classes et des méthodes, et liste les paramètres d'un test paramétré (c'est le générateur par défaut du framework
- Simple : enlève les parenthèses du nom d'un test si celui si ne prend pas de paramètre
- IndicativeSentences : permet de générer des noms qui ressemblent à des phrases, en concaténant le nom de la classe et le nom de la méthode de test (avec des séparateurs customisables)

Si vous n'êtes pas satisfait par les générateurs fournis par JUnit, vous pouvez créer le vôtre en implémentant l'interface DisplayNameGenerator. Vous pourrez alors définir votre propre stratégie de génération pour les classes, classes imbriquées ainsi que les méthodes de test. Vous pouvez aussi renseigner le générateur de votre choix comme générateur par défaut, si vous ne désirez pas le spécifier dans chacune de vos classes de test. Pour cela, il faut créer un fichier src/test/resources/junit-platform.properties dans votre projet. Il suffit ensuite d'affecter à la propriété "junit.jupiter.displayname.generator.default" le nom complet qualifié du générateur.

```
junit.jupiter.displayname.generator.default = \
  org.junit.jupiter.api.DisplayNameGenerator$ReplaceUnderscores
```

Les extensions

Une des principales volontés de cette nouvelle version a été de rendre le framework ouvert à l'extension, et de permettre à chacun de l'adapter à ses propres besoins de test. C'est pour cela qu'une nouvelle API a été conçue : celle des extensions. Il s'agit d'un ensemble d'interfaces implémentables par les utilisateurs du framework, qui vont ensuite être appelés par le moteur de test lors de l'exécution. Certaines fonctionnalités natives de JUnit 5, comme les tests paramétrés, sont en réalité implémentées à l'aide de ces extensions. Elles sont également utilisées par les frameworks tiers (Spring, Mockito, etc.) pour intégrer leurs outils de test à JUnit. Il existe actuellement plus d'une di-

zaine d'extensions disponibles dans JUnit 5. Elles peuvent également être combinées entre elles, ce qui offre vraiment de multiples possibilités pour améliorer ses tests. Une fois implémentées, ces extensions se déclarent sur les classes ou méthodes de test à l'aide de l'annotation @ExtendWith.

Un type d'extension assez simple à prendre en main est celui qui permet de modifier les étapes du cycle de vie du test. Par exemple, prenons une application Spring qui nécessite le démarrage d'un faux serveur lors des tests d'intégration. Pour éviter d'implémenter cette initialisation pour chaque classe de test, on pourrait créer cette extension :

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.*;

@ExtendWith(MyExtension.class)
class MyIntegrationTests {

  @Test
  void test() {/*...*/}
}

class MyExtension implements BeforeAllCallback, AfterAllCallback {

  @Override
  public void beforeAll(ExtensionContext context) {
    // Préparation du contexte de test
  }

  @Override
  public void afterAll(ExtensionContext context) {
    // Destruction du contexte de test
  }
}
```

Ici, on a créé une extension qui combine BeforeAllCallback et AfterAllCallback : elle créera donc des méthodes @BeforeAll et @AfterAll pour chaque classe où elle sera déclarée. De façon similaire, il existe BeforeEachCallback et AfterEachCallback.

Notons qu'il est possible de déclarer plusieurs extensions sur une même classe de test (que ce soit une implémentation personnelle ou fournie par un framework tiers). On peut même maintenant créer une méta-annotation pour rendre le tout encore plus concis et lisible :

```
@IntegrationTest
class MyIntegrationTests {
  // ...
}

@ExtendWith({
    MyIntegrationTestExtension.class,
    SpringExtension.class // Implémenté par Spring
})
@Retention(RetentionPolicy.RUNTIME)
@interface IntegrationTest {
}
```

Une autre extension très utile qui permet d'injecter des objets dans ses tests : le ParameterResolver. On est parfois obligé de construire des objets complexes, sans forcément que le test ne porte sur le contenu de ceux-ci. Cela crée des classes encombrées où l'on perd le sens fonctionnel du test. Grâce aux parameters resolvers, on peut externaliser ces créations et injecter les objets directement comme arguments des méthodes de test.

```
class BookmarkResolver implements ParameterResolver {

  @Override
  public boolean supportsParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
    // Le type d'arguments concernés par le resolver
    return parameterContext.getParameter().getType() == Bookmark.class;
  }

  @Override
  public Object resolveParameter(ParameterContext parameterContext,
ExtensionContext extensionContext) {
    if (parameterContext.isAnnotated(Article.class)) {
      return Bookmark.create("https://www.programmez.com/article
/1", "Cool article");
    }
    return Bookmark.create("https://www.programmez.com", "Programmez
Magazine");
  }
}

@ExtendWith(BookmarkResolver.class)
class BookmarkTest {

  @BeforeEach
  void set_up(Bookmark bookmark) {
    // Bookmark injecté : https://www.programmez.com
  }

  @Test
  void a_test(@Article Bookmark bookmark) {
    // Bookmark injecté : https://www.programmez.com/article/1
  }
}
```

L'objet à injecter est construit dans resolveParameter. Elle peut être fixe, ou dépendre du contexte du paramètre (voir l'exemple ci-dessus où on utilise une annotation pour influer sur la valeur injectée). Pour en apprendre plus sur les autres extensions disponibles, vous pouvez vous rendre sur le guide d'utilisateur de JUnit où elles sont toutes répertoriées et documentées. (https://junit.org/junit5/docs/current/user-guide/#extensions).

Aller plus loin

Pour en savoir plus, rendez-vous sur le site de JUnit 5 ! La documentation est très claire et complète : https://junit.org/junit5/docs/current/user-guide.

Pour avoir un exemple concret de migration d'une application de JUnit 4 vers JUnit 5, notre conférence est disponible sur YouTube : https://youtu.be/EfxwS54hdkM.

Le code présenté est disponible ici : https://gitlab.com/crafts-records/remember-me.

Comment évaluer la qualité de ses tests grâce au mutation testing ?

Aujourd'hui, la qualité du code est un sujet qui s'invite régulièrement au sein des équipes de développement. Ainsi pour améliorer la qualité d'un projet, naturellement on s'oriente vers des tests unitaires (TU) pour vérifier qu'une portion de code fonctionne bien.

Samuel Marques Antunes

Développeur web fullstack chez Creative Technology by Devoteam. Passionné depuis mon plus jeune âge par les technologies informatiques et le développement, je porte un très grand intérêt sur la qualité du code et plus particulièrement les pratiques du craftsmanship que je cherche à partager à mon équipe.

Un des principes phares de l'Extreme Programming (XP) est le Test first. Il s'agit d'écrire son test avant même d'avoir écrit son code. Cette méthode de développement est aussi connue sous le nom de **Test Driven Development** (TDD ou les tests pilotés par le développement en français). Le dernier indicateur pour vérifier que l'application est bien testée est le **code coverage** (taux de couverture du code par les tests en français). Mais est-ce réellement suffisant pour vérifier la qualité de ses tests ?

Les limites de la couverture de tests

Le code coverage est une métrique importante, mais s'il y a bien une chose qu'elle ne vérifie pas, c'est la qualité des tests, pire elle peut être trompeuse. Elle ne vous montre que le code que vous avez exécuté, pas le code que vous avez vérifié.

La couverture de code est un guide, pas un objectif. Elle vous aide à écrire les bons tests pour valider un des chemins d'exécution de votre code.

La qualité des tests que vous écrivez dépend de la compétence et de l'attention que vous portez à leur rédaction. La couverture a peu de pouvoir pour détecter les tests accidentellement ou délibérément bâclés. C'est là qu'entre en jeu, le mutation testing. Pour faire simple, c'est une méthode qui vérifie la robustesse des tests unitaires via l'utilisation d'une librairie dédiée, dans notre cas, Pitest.

C'est quoi un test de mutation ?

Les tests de mutation consistent à créer des copies défectueuses de votre code et à analyser les résultats de l'exécution de la suite de tests par rapport à ces copies. Si un test échoue, on dit que le mutant est tué. Si aucun test n'échoue, on dit que le mutant a survécu.

Une mutation de code est un changement mineur qui affecte le comportement général de ce code. Voici quelques exemples : la suppression d'une ou plusieurs lignes de code, le remplacement d'opérateurs arithmétiques (par exemple remplacer une addition par une soustraction), le remplacement des opérateurs logiques (par exemple remplacer un "<" par un "<=") ou le remplacement du retour d'une méthode par un objet null ou vide.

L'objectif est de mettre à jour votre suite de tests afin de tuer tous les mutants en faisant échouer tous les tests.
En savoir plus : https://pitest.org/quickstart/mutators/

Comment fonctionne Pitest ?

Pitest est un outil de test de mutation pour Java qui possède une bonne intégration avec les environnements de développement intégrés tels qu'Eclipse ou IntelliJ. Ainsi, que les outils d'analyse statique du code tel que SonarQube.

Pitest génère des mutants en manipulant le bytecode. Cette approche offre des avantages significatifs en termes de performances par rapport aux fichiers mutants compilés, mais présente quelques inconvénients. Parfois il se peut que la mutation ne corresponde pas à un changement que le développeur pourrait réellement faire.

Une fois les mutants générés et les tests unitaires exécutés, Pitest fournit un rapport clair de l'exécution des tests, qui facilite la navigation entre le code source et les mutants et met en évidence les mutants qui n'ont pas été tués

Ça donne quoi Pitest en pratique ?

Dans mon projet Java, je vais implémenter une classe FizzBuzz (implémentation du jeu du même nom, pour enfants leur apprenant la division de manière ludique). Le jeu consiste à compter de 1 à 100 en remplaçant les nombres multiples de 15 par "FizzBuzz", les nombres multiples de 3 par "Fizz" et les nombres multiples de 5 par "Buzz".

```
public class FizzBuzz {
  public String getResult(int number) {
    if (number % 15 == 0) {
      return "FizzBuzz";
    } else if (number % 3 == 0) {
      return "Fizz";
    } else if (number % 5 == 0) {
      return "Buzz";
    }
    return Integer.toString(number);
  }
}
```

Imaginons que mon chef de projet me demande de mettre en place en urgence des tests unitaires et d'avoir absolument 100 % de couverture de tests.

```
public class FizzBuzzTest {
  FizzBuzz fizzBuzz;

  @Before
```

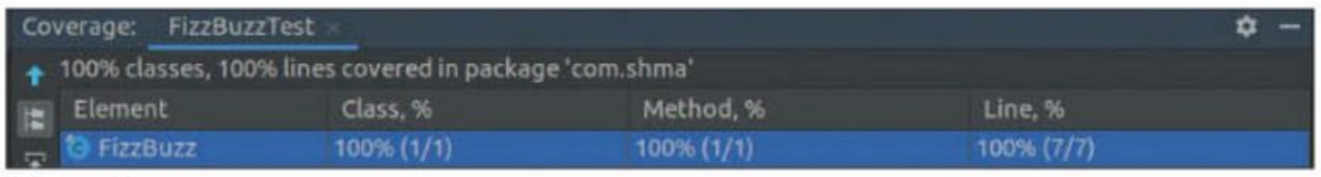


Figure 1

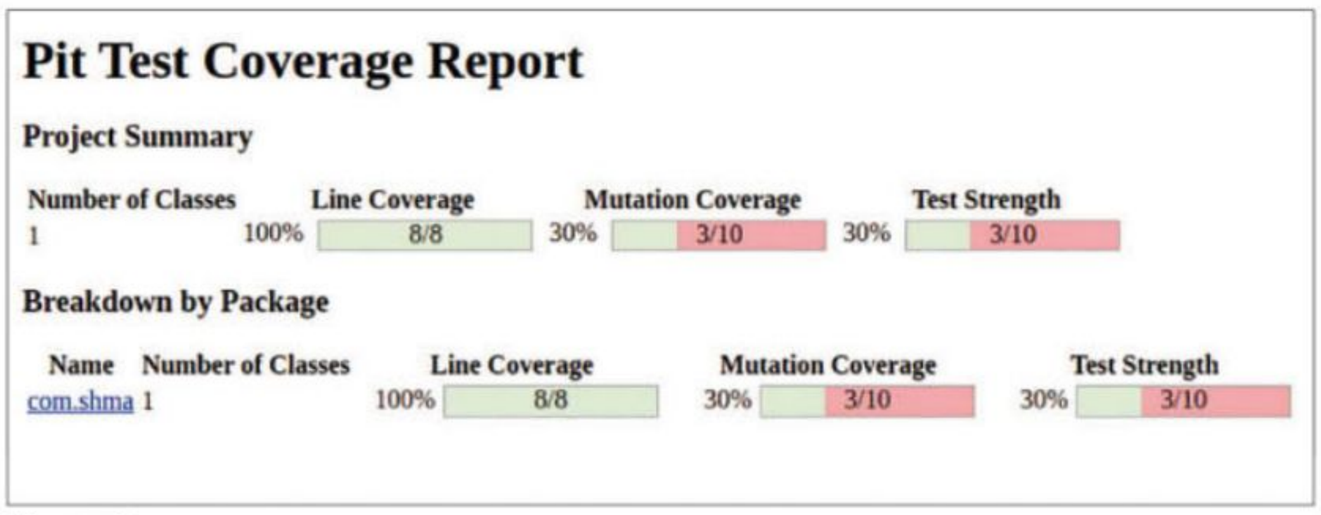


Figure 2

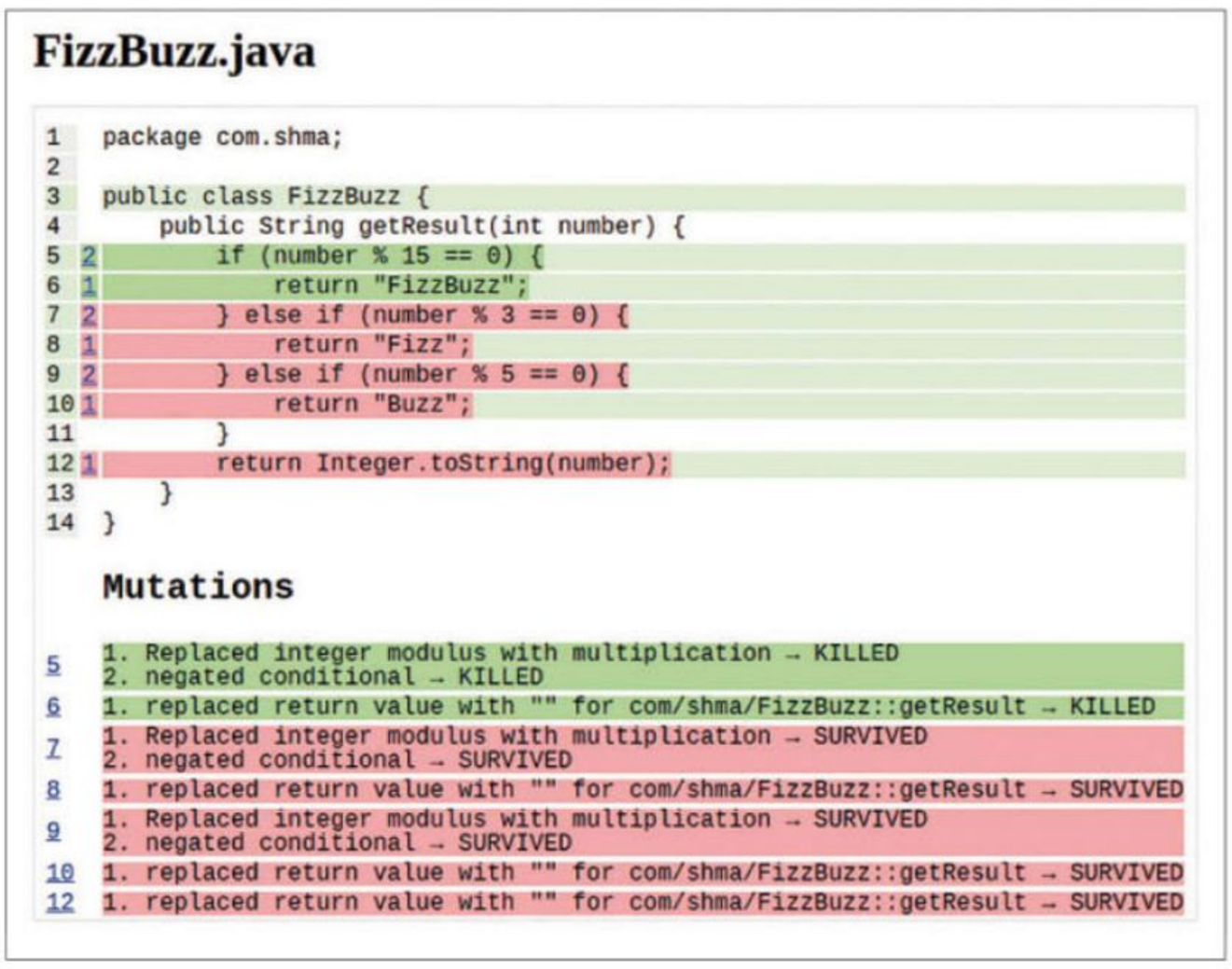


Figure 3

```
public void init() {
  fizzBuzz = new FizzBuzz();
}

@Test
public void should_return_buzz_if_the_number_is_dividable_by_3() {
  fizzBuzz.getResult(3);
  Assert.assertTrue(true);
}

@Test
public void should_return_buzz_if_the_number_is_dividable_by_5() {
  String result = fizzBuzz.getResult(5);
}

@Test
public void should_return_buzz_if_the_number_is_dividable_by_15() {
  Assert.assertEquals("FizzBuzz", fizzBuzz.getResult(15));
  Assert.assertEquals("FizzBuzz", fizzBuzz.getResult(30));
}
```

```
@Test
public void should_return_the_same_number_if_no_other_requirement
_is_fulfilled() {
    Assert.assertEquals(fizzBuzz.getResult(1).getClass(), String.class);
  }
}
```

Je vérifie le code coverage et j'obtiens bien 100%. **Figure 1**

Le chef de projet est très satisfait. Mais est-ce réellement une garantie de la qualité du code ? Un développeur expérimenté remarquera que mes tests ne sont pas satisfaisants pour valider le bon fonctionnement de mon application.

C'est maintenant que Pitest entre en jeu. Pour commencer, je vais importer la librairie dans la configuration Maven.

```
<plugin>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <version>1.6.4</version>
</plugin>
```

et ensuite, exécuter la commande suivante :

```
mvn org.pitest:pitest-maven:mutationCoverage
```

Pitest va générer un rapport au format html dans le dossier target/pit-reports, voici un extrait : **Figure 2**

Nous pouvons voir que dix mutants ont été générés et seulement 3 sur 10 ont mis en échec les tests unitaires (validant ainsi leur bon fonctionnement). Pour plus de détails, naviguer dans le package et sélectionner une classe Java. **Figure 3**

Maintenant, la question que je dois me poser est " Pourquoi ces mutants ont-ils survécu ? ". Le rapport montre que parmi les dix mutants générés, nous avons deux types de mutator : changement d'opérateurs (multiplication au lieu de modulo) et changement du retour d'une méthode par une chaîne de caractères vides. En regardant de plus près les tests unitaires, je peux remarquer que j'ai fait plusieurs erreurs de programmation.

Premièrement, pour la fonctionnalité où les multiples de 3 retournent bien "Fizz", je vois que je ne vérifie pas le retour de la méthode getResult. Je fais juste "Assert.assert True(true)". Le mutation testing va modifier le retour par une chaîne vide ou les opérateurs. Les mutants ne vont pas être tués par les tests.

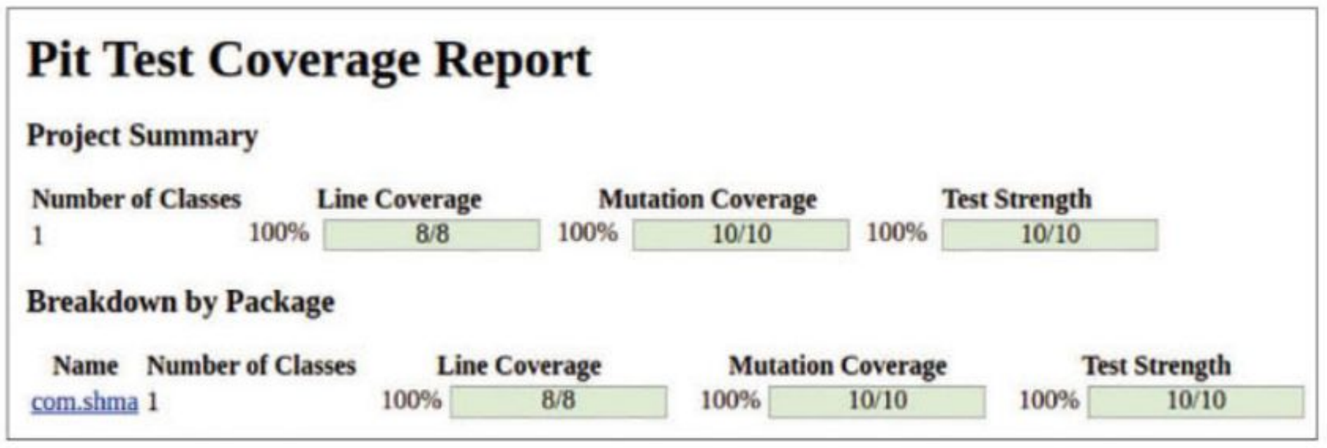
Deuxièmement, pour la fonctionnalité où les multiples de 5 retournent bien "Buzz", là je ne vérifie absolument pas si le retour de la méthode est valide. Donc pareil que le premier point, les mutants ne vont pas être tués par les tests.

Dernier point, nous pouvons voir que pour la toute dernière fonctionnalité de FizzBuzz où les chiffres qui ne sont pas des

multiples de 3, 5 ou 15 doivent retourner sa propre valeur, je teste uniquement si le retour est une chaîne de caractères. Pareil que les autres points, le retour de la méthode va être modifié par Pitest et les mutants ne vont pas être tués.

En prenant en compte les commentaires ci-dessus, je vais modifier les tests :

```
public class FizzBuzzTest {
  FizzBuzz fizzBuzz;

  @Before
  public void init() {
    fizzBuzz = new FizzBuzz();
  }

  @Test
  public void should_return_buzz_if_the_number_is_dividable_by_3() {
    Assert.assertEquals("Fizz", fizzBuzz.getResult(3));
  }

  @Test
  public void should_return_buzz_if_the_number_is_dividable_by_5() {
    Assert.assertEquals("Buzz", fizzBuzz.getResult(5));
  }

  @Test
  public void should_return_buzz_if_the_number_is_dividable_by_15() {
    Assert.assertEquals("FizzBuzz", fizzBuzz.getResult(15));
    Assert.assertEquals("FizzBuzz", fizzBuzz.getResult(30));
  }

  @Test
  public void should_return_the_same_number_if_no_other_requirement
_is_fulfilled() {
    Assert.assertEquals("1", fizzBuzz.getResult(1));
    Assert.assertEquals("2", fizzBuzz.getResult(2));
  }}
```



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	8/8	100%	10/10	100%	10/10

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.shma	1	100%	8/8	100%	10/10	100%	10/10

Figure 4

Je génère un nouveau rapport Pitest pour voir le résultat : **Figure 4**

Conclusion

Pour conclure, le mutation testing couplé au code coverage sont des bons indicateurs pour faire progresser la qualité du code. Cependant, il n'est pas conseillé de mettre le mutation testing dans la pipeline d'intégration continue, car comme vu précédemment le mutation testing va générer un certain nombre de mutants et devoir exécuter tous les tests unitaires. Cette opération est très coûteuse en temps et en ressources matérielles. Il est vivement conseillé d'utiliser cette méthode à de rares occasions ou dans un pipeline hebdomadaire.

Bon à savoir pour limiter le temps d'exécution de Pitest sur les très gros projets, il est possible de lancer les tests sur un module précis, mais également de définir les mutators qu'on souhaite.

Pitest est codé en Java et est utilisé pour les applications Java, mais il existe des librairies alternatives pour les autres langages notamment Stryker Mutator.

Adoptez un JDK

Fayssal MERIMI

Lead Architecte Senior Chez Creative Technology by Devoteam

Expert / Architecte Technique dans divers domaines : Industrie / Banque / Assurance / Distribution et Transport. Avec une forte expertise dans le développement, la conception et dans l'Architecture techniques. Passionné par les nouvelles technologies et les challenges techniques autour des problématiques cloud, API, haute disponibilité et middleware.

Avec les changements apportés à la distribution et au support d'Oracle JDK, il y a eu une incertitude sur les droits d'utilisation d'Oracle JDK et sur l'avenir de Java en général. Des questions pertinentes se posent quant à l'image docker de base de Java que vous devez choisir. Quelle JDK utiliser ? Quels type et variante de JDK est le mieux adapté ? Comment optimiser la taille de l'image docker ? etc.

Oracle et sa nouvelle politique

Les récents changements apportés à la distribution et au support du JDK (Java Development Kit) d'Oracle sont intervenus en peu de temps, créant une véritable confusion. Cette confusion a commencé à cause de deux événements :

- Changement radical du cycle de releases de Java,
- Changement du modèle de support pour Java, rendant les mises à jour payantes pour les anciennes versions.

Ce qui constitue un changement de cap important qui risque d'avoir de nombreux impacts dans le développement et le run des applications dans les SI (Systèmes d'Information) des entreprises.

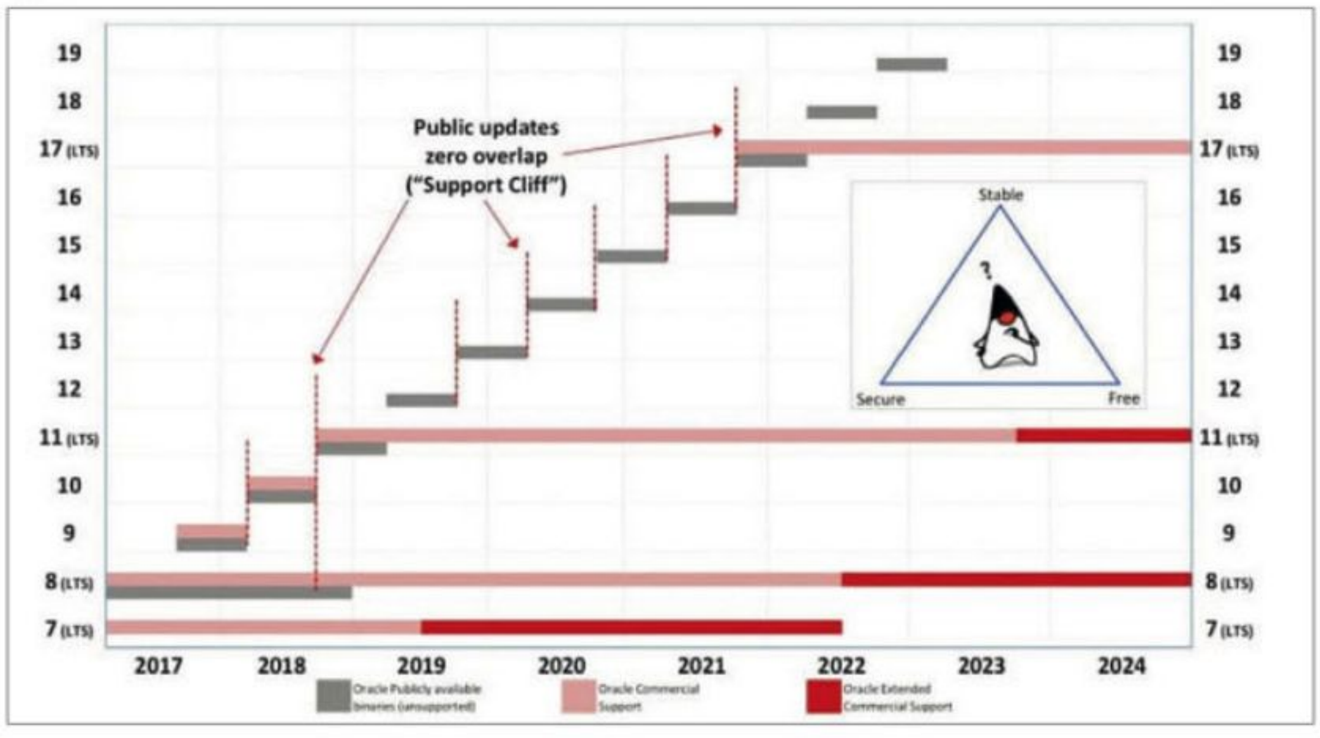
Nous listons un bref résumé des changements apportés par Oracle :

- Oracle distribue désormais deux builds du JDK :
  - Oracle OpenJDK : open source, il est maintenu et développé par Oracle, mais permet aux communautés et autres entreprises de participer à ce développement, comme Red Hat, Azul Systems, IBM ou Apple,
  - Oracle JDK : se base sur l'OpenJDK, mais n'est pas open source. Il est bien meilleur en termes de réactivité et de performances JVM.
- Depuis janvier 2019, Oracle JDK est gratuit pour le développement et les tests, mais payant pour toute utilisation en production.
- OpenJDK d'Oracle est gratuit pour tous les environnements, mais les mises à jour sont payantes dès l'apparition d'une nouvelle release majeure. Cela implique la souscription à un contrat de support auprès d'Oracle, qui analysera l'utilisation par l'entreprise cliente dans son ensemble et transmet une proposition personnalisée.
- Oracle et la communauté OpenJDK passent à un modèle de version LTS plus une nouvelle classe "Feature" Releases.
- Les versions LTS sont programmées tous les 3 ans. Leur portée est comparable à celle des précédentes versions majeures du JDK. Le JDK 11, lancé en septembre 2018, est la plus récente version LTS de Java. **Figure 1**

Cette nouvelle cadence de publication de Java SE présente plusieurs défis et lacunes :

- À partir du JDK 11, les mises à jour publiques du JDK d'Oracle se terminent en même temps ou très peu de temps après la GA (General Availability) d'une nouvelle version. Par exemple, Oracle a proposé des mises à jour publiques du JDK 11 jusqu'en janvier 2019. Les mises à jour publiques pour le JDK 8 ont également pris fin en janvier 2019.
- Comme le JDK 11 mettra du temps à être opérationnel, cette « Support Cliff » présente des défis majeurs et des risques de stabilité, obligeant les utilisateurs de mises à jour publiques à choisir entre une version immature et une version présentant des vulnérabilités de sécurité connues (La prochaine LTS est la 17 prévue pour 09/2021).
- Les versions fréquentes des fonctionnalités permettent d'accéder à de nouvelles capacités du JDK/JVM bien avant la prochaine LTS, mais leur « support Cliff » sans chevauchement les rend problématiques pour une utilisation en production, car il n'y a pas de transition en douceur d'une version stable et mise à jour à la suivante.

Figure 1
Cycle de vie d'Oracle JDK



L'OpenJDK, une alternative à l'OracleJDK

OpenJDK est une implémentation gratuite et open source de la plate-forme Java SE. Il a été initialement publié en 2007 comme le résultat du développement que Sun Microsystems a commencé en 2006. Elle est devenue une implémentation de référence officielle de Java Standard Edition depuis la version SE 7. Initialement, il était basé uniquement sur le JDK 7. Mais, depuis Java 10, l'implémentation de référence open source de la plate-forme JavaSE est sous la responsabilité du JDK Project.

OpenJDK a plusieurs significations et peut se référer aussi à :

- Un dépôt de code sources Java, alias projet OpenJDK,
- Une spécification,
- Un produit et une distribution fournie et maintenue par Oracle,
- Une distribution fournie maintenue par la communauté OpenJDK.

Les deux produits (OpenJDK et Oracle JDK) sont des implémentations de la même spécification Java passant le TCK (Java Technology Certification Kit). Elles ne sont pas forcements identiques, ci-dessous les principales différences :

Calendrier de publication :

L'OpenJDK aura une version tous les 6 mois, qui est uniquement prise en charge jusqu'à la prochaine version de la fonctionnalité. C'est essentiellement un flux continu de versions destinées aux développeurs. Alors que le JDK d'Oracle cible davantage un public d'entreprise qui privilégie la stabilité. Il est basé sur l'une des versions d'OpenJDK, mais bénéficie d'un support à long terme (LTS). Oracle JDK propose des versions tous les 3 ans. **Figure 2**

Licences :

Oracle JDK est sous licence Oracle Binary Code License Agreement, tandis qu'OpenJDK est sous licence GNU GPL version 2 avec une exception de liaison. L'utilisation de la plate-forme d'Oracle a des conséquences sur les licences. À partir de la Java 8, les mises à jour publiques publiées après janvier 2019 ne seront pas disponibles pour une utilisation professionnelle, commerciale ou de production sans une licence commerciale, comme Oracle l'a annoncé. Cependant, OpenJDK est entièrement open source et peut être utilisé librement.

Performances :

En ce qui concerne les performances, celles d'Oracle sont bien meilleures en termes de réactivité et de performances de la JVM. Il met davantage l'accent sur la stabilité en raison de l'importance qu'il accorde à ses clients professionnels.

Fonctionnalités :

Si nous comparons les fonctionnalités et les options, nous voyons que le produit Oracle possède les fonctionnalités Flight Recorder, Java Mission Control et Application Class-Data Sharing, tandis qu'OpenJDK possède la fonctionnalité Font Renderer. En outre, Oracle dispose de plus d'options de Garbage Collection et de meilleurs moteurs de rendus.

À partir de JDK 11, d'autres différences s'ajoutent à la liste :

- Chaque licence aura des builds différents, mais ceux-ci seront fonctionnellement identiques avec seulement quelques différences cosmétiques et de packaging.
- Fonctionnellement identiques et interchangeables, des fonctions traditionnellement "commerciales" telles que Flight Recorder, Java Mission Control et Application Class-Data Sharing, ainsi que le Z Garbage Collector, sont désormais disponibles dans OpenJDK. Par conséquent, les builds d'Oracle JDK et d'OpenJDK sont essentiellement identiques à partir de Java 11.
- Oracle JDK offre une configuration pour fournir les données du journal d'utilisation à l'outil "Advanced Management Console".
- Oracle a toujours exigé que les fournisseurs cryptographiques tiers soient signés par un certificat connu, alors que le cadre cryptographique d'OpenJDK possède une interface cryptographique ouverte, ce qui signifie qu'il n'y a aucune restriction quant aux fournisseurs pouvant être utilisés.

Que se passe-t-il à partir de JDK 11 ? C'est le passage de l'état d'esprit d'un fournisseur unique (Oracle) à l'état d'esprit où vous pouvez sélectionner le fournisseur qui vous offre une distribution, dans les conditions qui vous conviennent le mieux : les plates-formes pour lesquelles ils réalisent la compilation d'OpenJDK, la fréquence et la rapidité des versions,

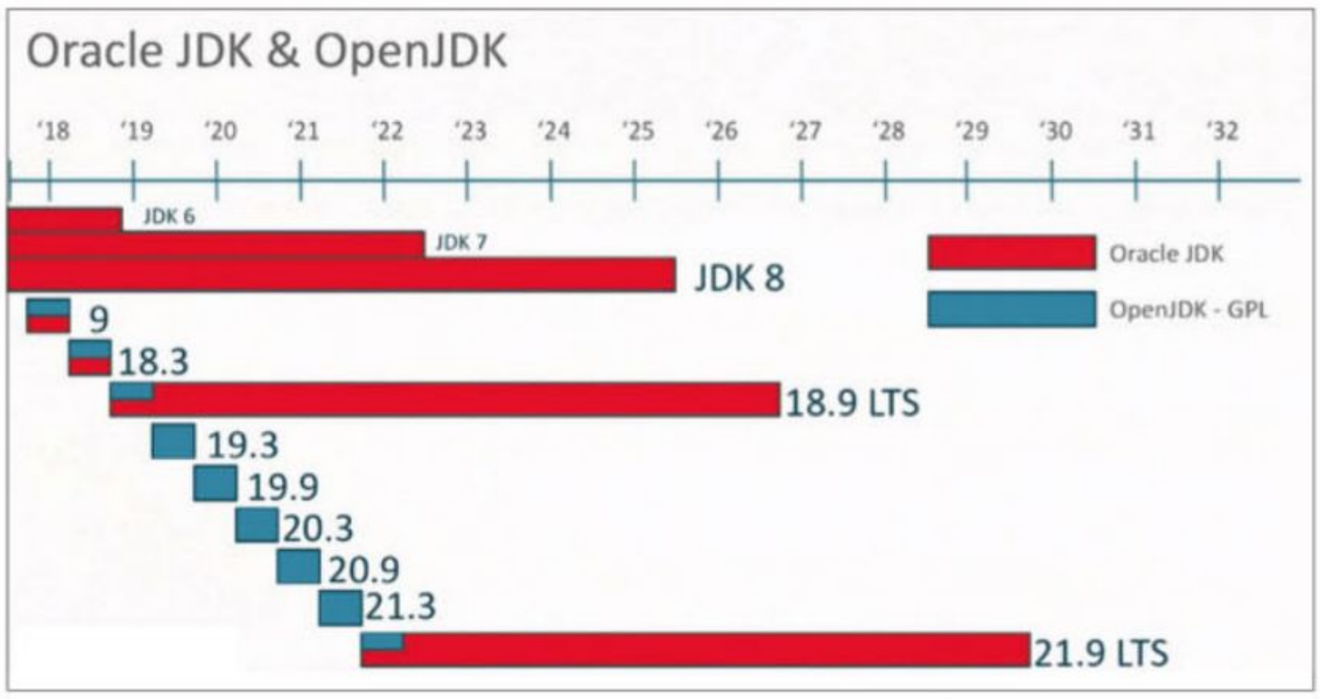


Figure 2
Cycle de vie d'OracleJDK et OpenJDK

la façon dont le support est structuré, etc. Si aucun des fournisseurs ne répond à vos attentes, vous pouvez même construire votre propre distribution d'OpenJDK.

Distributions d'OpenJDK, laquelle choisir ?

La plupart des distributions d'OpenJDK sont écrites au-dessus d'OpenJDK (le projet) en apportant quelques modifications aux composants (principalement pour remplacer les parties propriétaires sous licence / remplacer par des éléments plus performants qui ne fonctionnent que sur un système d'exploitation spécifique) sans rompre la compatibilité TCK.

Les distributions

- AdoptOpenJDK (Adoptium) : AdoptOpenJDK est une communauté de membres du groupe d'utilisateurs Java (JUG), de développeurs et de fournisseurs Java (notamment Azul, Amazon, GoDaddy, IBM, jClarity, Microsoft, Pivotal et Red Hat) qui défendent OpenJDK. AdoptOpenJDK fournit des distributions binaires OpenJDK (HotSpot et Eclipse OpenJ9) pour une très large gamme de plates-formes (Linux, Mac, Windows 32/64, Arm 32/64, Solaris, AIX, PPC, s390 et plus). AdoptOpenJDK n'offre pas de support payant. Il fournit simplement des fichiers binaires testés,
- Amazon Corretto : Amazon web Services (AWS) est également engagé dans OpenJDK. Depuis fin 2018, il a lancé Amazon Corretto, sa distribution basée sur OpenJDK 8, avec prise en charge à long terme au moins jusqu'en mai 2026. L'engagement d'Amazon en faveur d'OpenJDK et de le maintenir libre pour la popularité de Java parmi les développeurs AWS. Amazon Corretto, qui correspond à OpenJDK 8, fournit plus de quatre ans de support à long terme, au-delà de ce qu'Oracle propose actuellement pour la distribution de OpenJDK 8. Cela garantit que le support OpenJDK restera gratuit au moins aussi longtemps. En d'autres termes, Amazon Corretto est une diffusion sans coût, multi-plate-forme et prête pour la production du kit de développement OpenJDK. Corretto est fourni avec une prise en charge longue durée qui inclut des améliorations de performances et des correctifs de sécurité. Corretto est certifié compatible avec la norme Java SE. Grâce à Corretto, vous pouvez développer et exécuter des applica-

tions Java sur plusieurs des systèmes d'exploitation les plus courants, notamment Linux, Windows et Mac OS.

- Azul Zulu : Azul fournit des fichiers binaires OpenJDK (Zulu) ainsi qu'une plate-forme Java spécialisée (Zing). Azul offre une option à toutes les entreprises qui ne souhaitent pas ignorer toutes les versions de Java SE entre les versions de LTS, mais ne peuvent pas passer à la version la plus récente tous les 6 mois. Ensuite, pour le support de toutes les versions de LTS, où Azul fournit une année de support supplémentaire par rapport à Oracle, Azul prend en charge les versions dites de support à moyen terme (MTS) pour leur JDK Zulu. Ici, vous pouvez acheter un support commercial pour chaque seconde version de Java SE, qu'il s'agisse ou non de LTS. La durée de support de ces versions est différente. Azul essaie de fournir une bonne plage de temps pour préparer une migration vers la prochaine version et définit 3 durées différentes pour la prise en charge des versions de Java SE. L'assistance commerciale de Zulu n'est pas définie par processeur comme c'est le cas chez Oracle, mais en fonction du nombre de systèmes. Un système est défini comme un serveur physique ou virtuel. La seule différence entre le support standard et premium est la disponibilité du support. En achetant une assistance premium, vous pouvez appeler Azul 24x7.
- BellSoft Liberica : Liberica est une implémentation Java 100% open source. Elle est construite à partir d'OpenJDK, auquel BellSoft contribue, est testée de manière approfondie et a passé le JCK fourni sous la licence d'OpenJDK. Toutes les versions supportées de Liberica contiennent également JavaFX 12.
- IBM SDK Java : IBM fournit des kits JDK natifs pour AIX, Linux (sur x86, Power, zSystems), z/OS et IBM i. IBM propose les versions d'IBM SDK Java SE à utiliser avec les produits ou plates-formes IBM et à l'usage des développeurs de developerWorks (un centre de ressources techniques d'IBM). IBM fournit également des fichiers binaires OpenJDK (avec Eclipse OpenJ9) construits et testés sur AdoptOpenJDK. Pour Java SE 7 et 8, IBM fournit toujours des mises à jour de sécurité et des corrections de bugs. Le cycle de vie du support IBM continuera d'être mis à jour. Sur la base du nouveau calendrier de publication de Java SE, IBM a annoncé que les versions non-LTS seraient disponibles en tant que fichiers OpenJDK avec OpenJ9 d'AdoptOpenJDK.
- Red Hat OpenJDK : La version OpenJDK de Red Hat est une implémentation libre et open source de la plate-forme Java Standard Edition (Java SE). Oracle JDK et OpenJDK sont fonctionnellement très similaires, mais présentent des différences majeures en termes de support. La version OpenJDK de Red Hat, a décrit l'entreprise, est une excellente alternative et une implémentation multi-plate-forme. Red Hat OpenJDK est pris en charge sous Windows et RHEL (Red Hat Enterprise Linux), ce qui vous permet de normaliser sur une seule plate-forme Java sur un cloud hybride, de centre de données et de bureau. Pour les versions LTS, Red Hat prend en charge OpenJDK 11, ainsi que OpenJDK 7 et 8.
- SapMachine : enfin, il serait bien de notifier que SAP a également créé et maintient le projet SapMachine, une version en aval du projet OpenJDK. Il est utilisé pour créer et gérer une version OpenJDK prise en charge par SAP pour les clients et partenaires SAP souhaitant utiliser OpenJDK pour exécuter leurs applications.

Fournisseur	Compilation depuis sources	Distributions gratuites	MAJ étendues	Support commercial	Licence permissive
AdoptOpenJDK	Oui	Oui	Oui	Non	Oui
Amazon Corretto	Oui	Oui	Oui	Non	Oui
Azul Zulu	Non	Oui	Oui	Oui	Oui
BellSoft Liberica	Non	Oui	Oui	Oui	Oui
IBM	Non	Non	Oui	Oui	Oui
jClarity	Non	Non	Oui	Oui	Oui
OpenJDK Upstream	Oui	Oui	Oui	Non	Oui
Oracle JDK	Non	Oui	Non	Oui	Non
Oracle OpenJDK	Oui	Oui	Non	Non	Oui
ojdkbuild	Oui	Oui	Non	Non	Oui
RedHat	Oui	Oui	Oui	Oui	Oui
SapMachine	Oui	Oui	Oui	Oui	Oui

Tableau comparatif des différentes distributions de l'OpenJDK

Compilation depuis le code sources : le code source de la distribution est disponible publiquement et chacun peut assembler sa propre construction.
Distributions gratuites : les binaires de la distribution sont publiquement disponibles pour le téléchargement et l'utilisation.
Mises à jour étendues : alias LTS - Mises à jour publiques au-delà du cycle de vie de 6 mois de la version.
Support commercial : certains fournisseurs proposent des mises à jour étendues et un support client aux clients payants, par exemple Oracle JDK (détails du support).
Licence permissive : la licence de distribution est non protectrice, par exemple Apache 2.0.

Distribution	Version	TCK	MAJ publiques	Plate-forme (*)	Support commercial
Eclipse Adoptium / AdoptOpenJDK	8, 11	Oui/ Non	Jusqu'au moins septembre 2023	Majeur + Mineur	IBM
Amazon Corretto	8, 11	Oui	Jusqu'au moins juin 2023	Majeur	-
Azul Zulu	7, 8, 11, 13	Oui	?	Majeur + Mineur	Azul
BellSoft Liberica JDK	8, 11	Oui	Jusqu'au moins 2023	Majeur + Mineur	BellSoft
Oracle OpenJDK	11	Oui	Jusqu'au moins mars 2019	Majeur	Oracle (via Oracle JDK)
SapMachine	11	Oui	Jusqu'au moins septembre 2022	Maj	SAP

Tableau comparatif de l'extension du support et mises à jour des distributions OpenJDK

* **Majeure** = Linux x86, macOS, Windows x64 / **Mineure** = autres plates-formes

Voici un organigramme pour nous guider dans le choix d'un fournisseur par type de support et licence : **Figure 3**

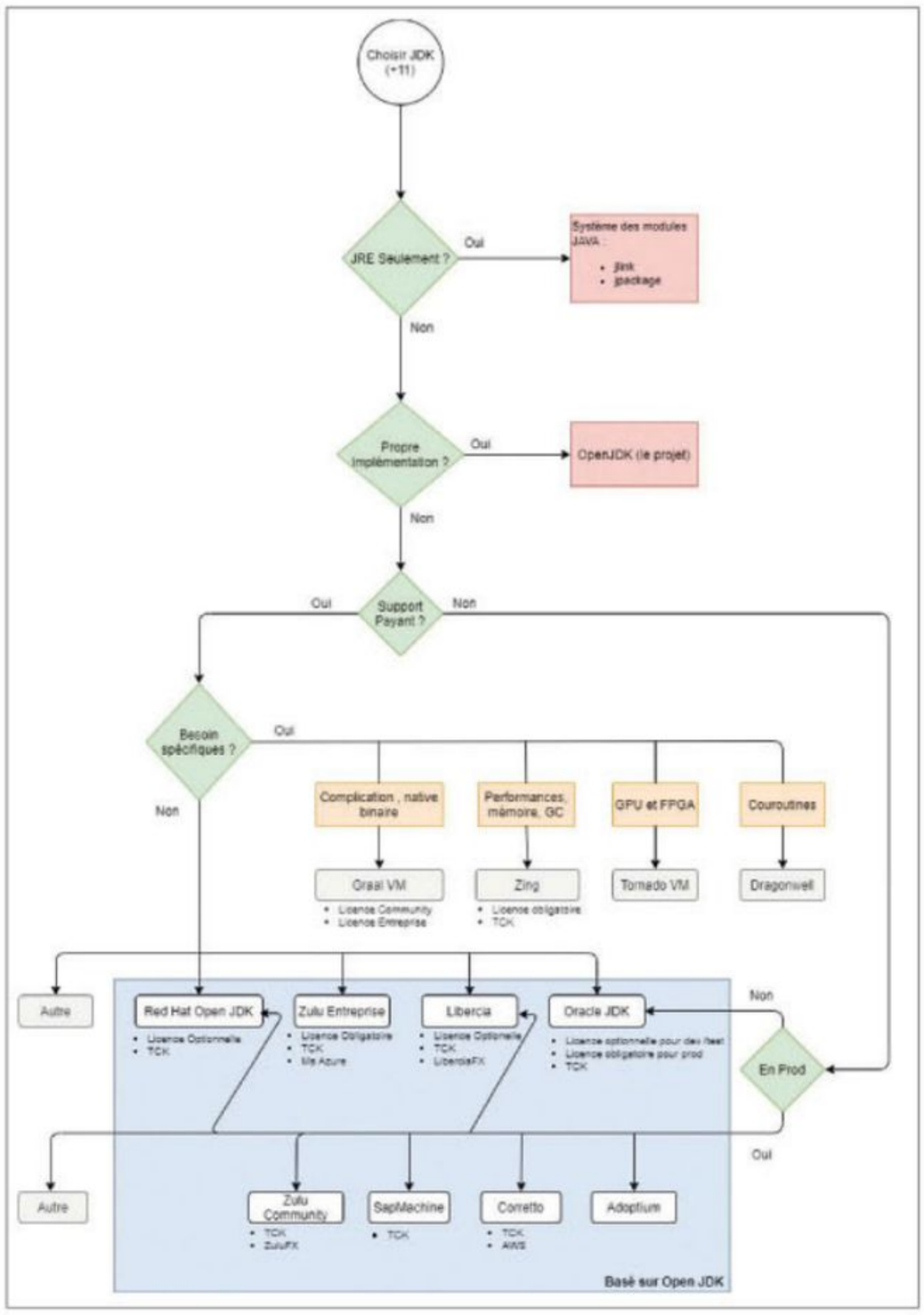


Figure 3
Organigramme de choix d'un fournisseur par type de support et licence

Performances :

Les performances sont un facteur important. Nous réaliserons des tests de performance d'une application Java (une application Springboot) dans un environnement conteneurisé avec différentes installations de JDK 11:

- OracleJDK
- AdoptopJDK (openJDK / openJ9)
- Azul Zulu
- Amazon Corretto

Ces tests vont permettre de comparer les performances selon les indicateurs suivants :

- Utilisation du CPU
- Utilisation de la mémoire JVM et Processus
- Threads
- Temps de réponse

Configuration et Installation:

- Monitoring : Prometheus et Grafana
- Spécifications : Conteneur docker Linux avec 1 CPU et 1024 Mo de mémoire.
- Docker-compose :

```yaml

# Services
services:

# Prometheus
  prometheus:
    image: prom/prometheus
    container_name: demo-prometheus
    volumes:
      - ./prometheus/:/etc/prometheus/
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--storage.tsdb.retention=200h'
      - '--web.enable-lifecycle'
    restart: unless-stopped
    ports:
      - 9090:9090
    depends_on:
      - demo-openjdk
      - demo-corretto
      - demo-zulu
      - demo-adoptjdk
      - demo-openj9
    networks:
      - metrics-net
      - demo-net
    labels:
      org.label-schema.group: "monitoring"

# Grafana
  grafana:
    build:
      context: ./grafana
      dockerfile: Dockerfile
    container_name: demo-grafana
    volumes:
      - grafana_data:/var/lib/grafana
      - ./grafana/datasources:/etc/grafana/provisioning/datasources
      - ./grafana/dashboards:/etc/grafana/provisioning/dashboards
      - ./grafana/setup.sh:/setup.sh
    entrypoint: /setup.sh
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin
      - GF_USERS_ALLOW_SIGN_UP=false
    restart: unless-stopped
    ports:
      - 3000:3000
    depends_on:
      - prometheus
    networks:
      - metrics-net
    labels:
      org.label-schema.group: "monitoring"

```

```yaml
# Demo OpenJDK
  demo-openjdk:
    build:
      context: ./
      dockerfile: Dockerfile-openjdk
    container_name: demo-openjdk
    deploy:
      resources:
          limits:
            cpus: 1
            memory: 1024M
    ports:
      - 8080:5000
    networks:
      - demo-net
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:5000/actuator/health"]
      interval: 30s
      timeout: 10s
      retries: 3
    labels:
      org.label-schema.group: "applications"


# Demo AWS
  demo-corretto:
    build:
      context: ./
      dockerfile: Dockerfile-corretto
    container_name: demo-corretto
    deploy:
      resources:
          limits:
            cpus: 1
            memory: 1024M
    ports:
      - 8081:5000
    networks:
      - demo-net
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:5000/actuator/health"]
      interval: 30s
      timeout: 10s
      retries: 3
    labels:
      org.label-schema.group: "applications"

# Demo Zulu
  demo-zulu:
    build:
      context: ./
      dockerfile: Dockerfile-zulu
    container_name: demo-zulu
    deploy:
      resources:
          limits:
            cpus: 1
            memory: 1024M
```

Image docker :

```dockerfile
FROM maven:3.6.3-amazoncorretto-11 AS MAVEN_BUILD_AWS

LABEL maintainer="fayssal.merimi@devoteam.com"

COPY ./ ./

RUN mvn clean package -Djava.jdk=corretto -Dproject.server.port=5000

FROM amazoncorretto:11

VOLUME /tmp

EXPOSE 5000

COPY --from=MAVEN_BUILD_AWS /target/demo-app-0.0.1-SNAPSHOT.jar /demo.jar

CMD ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/demo.jar"]
```

Exemple de DockerFile pour une image de base AWS Corretto

Pour chaque distribution on crée une image docker :

Distribution	Image de base maven	Image de base JDK
Aamzon Corretto	maven:3.6.3-amazoncorretto-11	amazoncorretto:11
AdoptOpenJDK/ openJDK	maven:3.6.3-adoptopenjdk-11	adoptopenjdk:11
AdoptOpenJDK/ openJ9	maven:3.6.3-adoptopenjdk-11-openj9	adoptopenjdk:11-jdk-openj9
Oracle JDK	maven:3.6.3-openjdk-11	openjdk:11
Azul Zulu	csanchez/ maven:azulzulu-11 (non officielle)	azul/zulu-openjdk:11

• Application Java Reactive (Spring WebFlux)

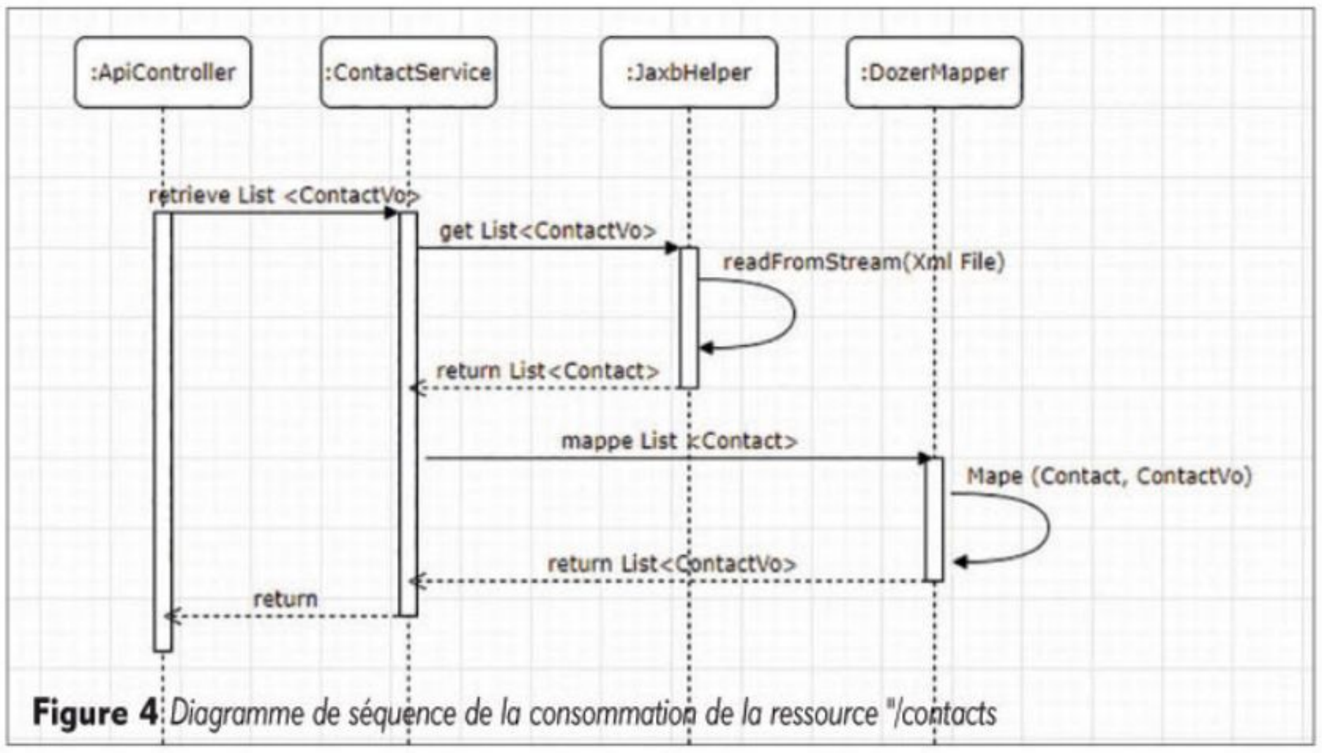


Figure 4 *Diagramme de séquence de la consommation de la ressource "/contacts*

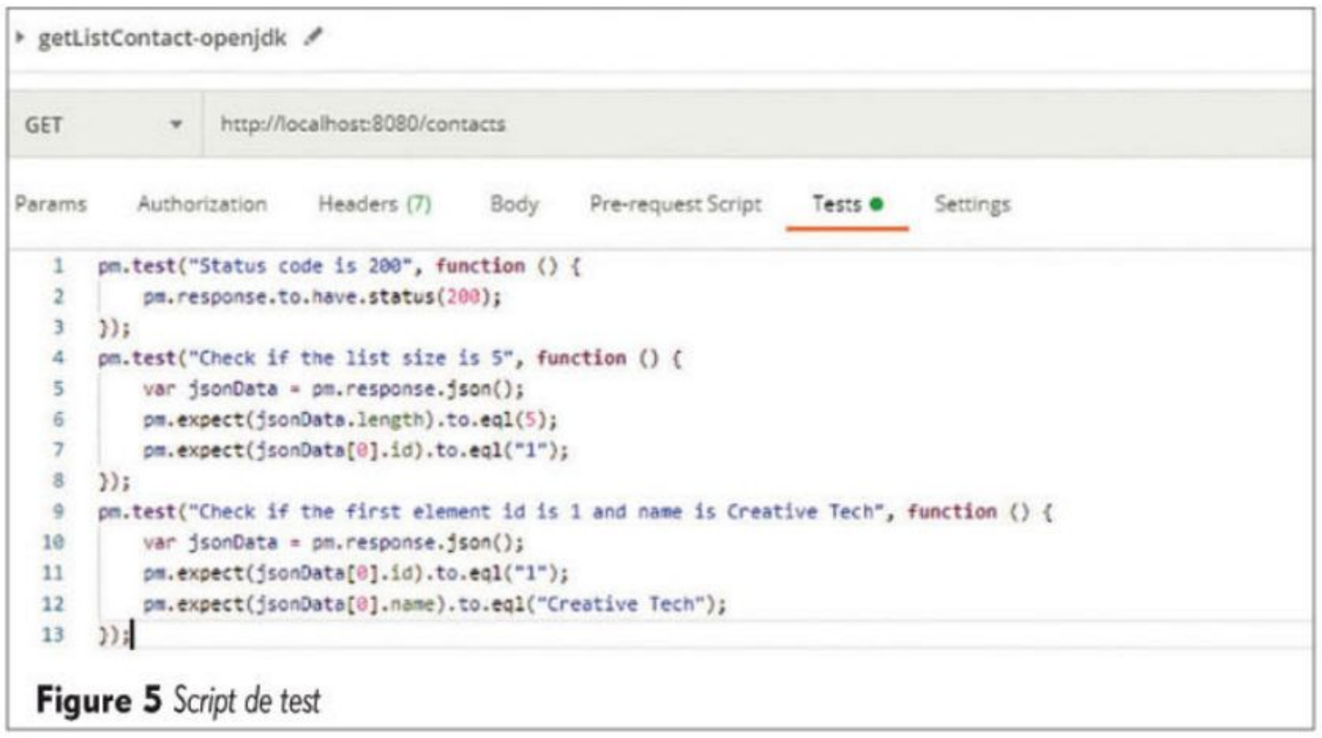


Figure 5 *Script de test*

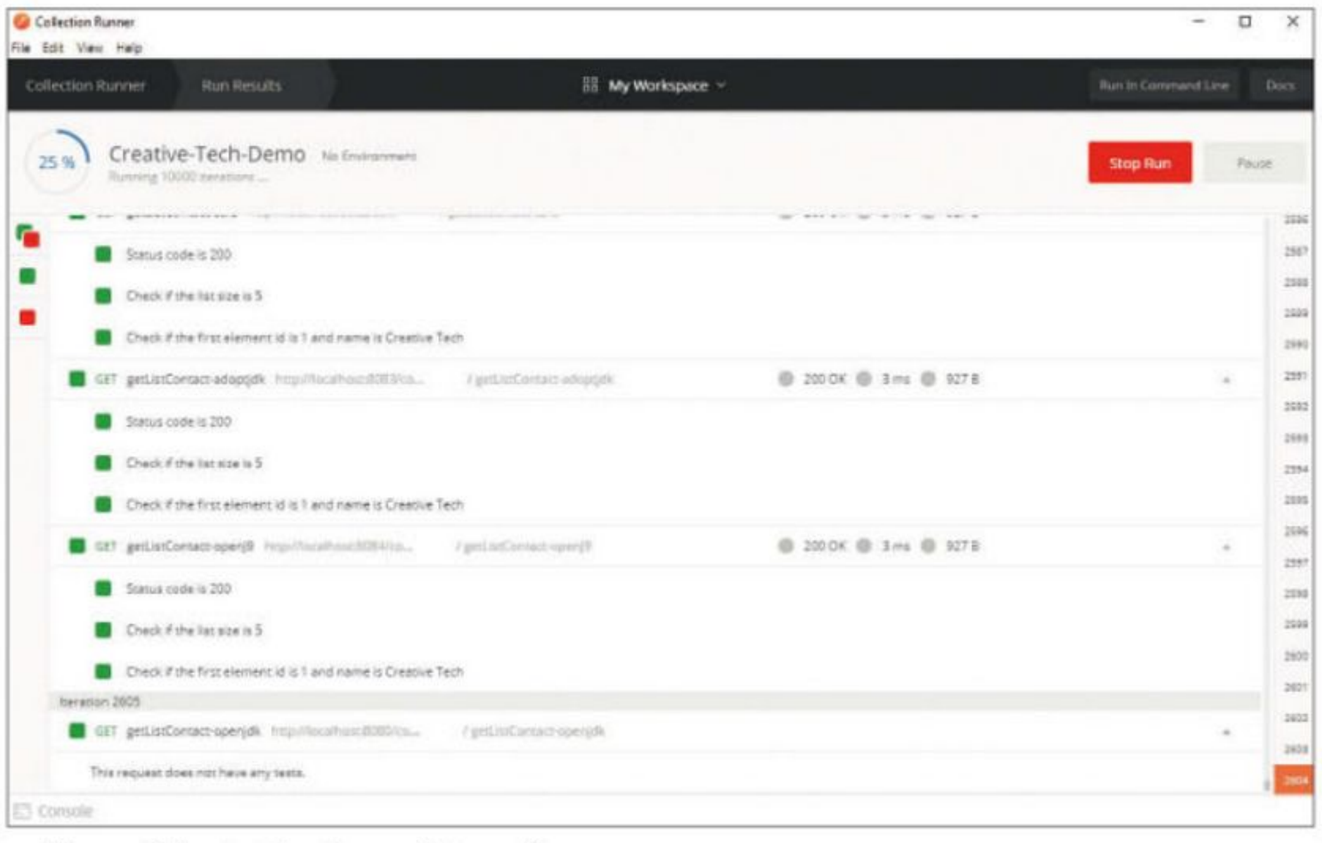


Figure 6 *Les tests lancés avec Postman Runner*

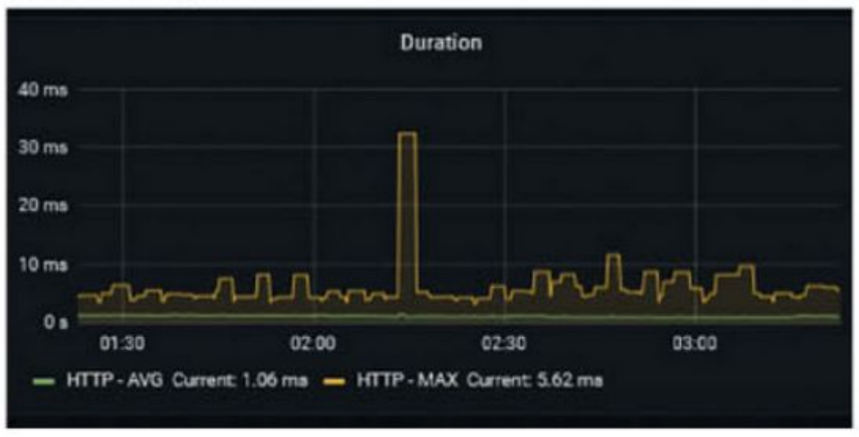
L'application java expose une API Restful :
/hello : retourne un texte "Greeting from Creative Technology"
/contacts : retourne un objet JSON qui enveloppe une liste de contacts
Elle expose d'autres ressources :
/actuator/prometheus : metrics
/actuator/health : état de santé de l'application
/actuator/info : des informations concernant l'application comme sa version, la version et la distribution de JDK
Nous allons effectuer des tests de performance sur la ressource "/contacts". En consommant cette ressource, nous allons charger la liste des contacts depuis un fichier XML et le transformer en liste d'objets java (avec du JAXB) que par la suite nous allons la mapper avec une liste des objets java à retourner (avec du Dozer). **Figure 4**

• Configuration du test de performance :
Nous allons effectuer les tests sur Postman, tout d'abord il faut créer une "Collection" et créer dedans des requêtes. Chaque requête correspond à une requête "Get" par environnement (distribution de JDK). Le script de test va vérifier le statut HTTP retourné (qui doit être 200 : succès) et que la réponse contient bien une réponse JSON qui contient cinq objets structurés.
Le test sera lancé par un Runner qui va réaliser 10 000 itérations avec un délai de 100 ms entre chaque appel. **Figures 5 et 6**

Résultat :
Nous allons utiliser Grafana et Prometheus pour visualiser les résultats du test de charge réalisé à l'aide de Postman sur l'application Java réactive s'exécutant sur les cinq JDK choisis.

• AdoptOpenJDK :
Ressources : CPU, Mémoire, Threads

Temps de réponse



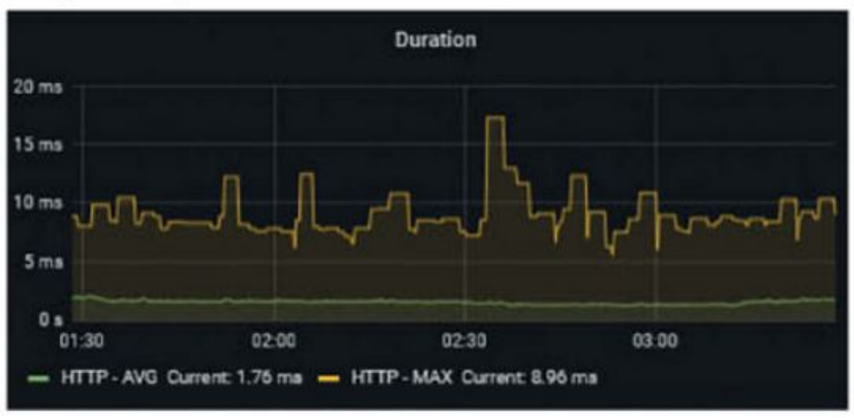
• Amazon Corretto
Ressources : CPU, Mémoire, Threads

Temps de réponse

• AdopteOpenJDK OpenJ9

Ressources : CPU, Mémoire, Threads

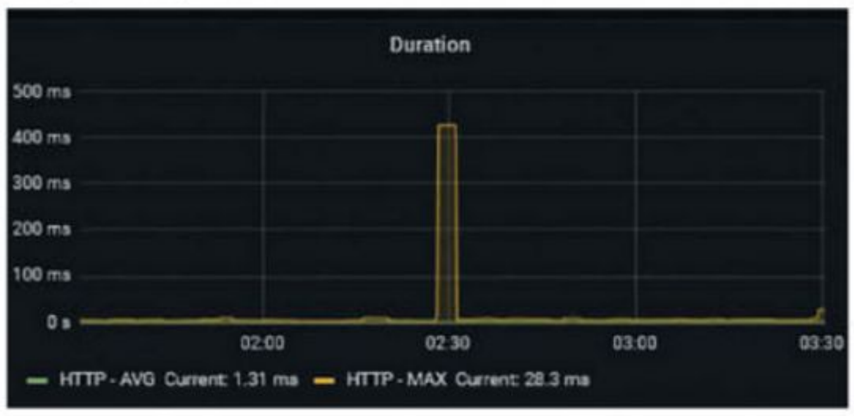
Temps de réponse



• Oracle OpenJDK

Ressources : CPU, Mémoire, Threads

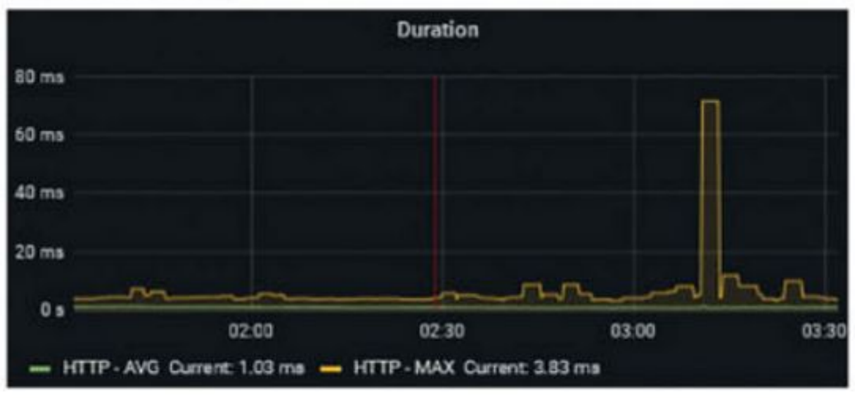
Temps de réponse



• Azul Zulu

Ressources : CPU, Mémoire, Threads

Temps de réponse



Synthèse :

- CPU : Oracle OpenJDK consomme moins de CPU
- Temps de réponse : Zulu est plus rapide
- Mémoire : openJ9 est le bon choix si l'utilisation de la mémoire est une contrainte
- Threads : openJ9 utilise beaucoup de Threads comparé aux autres JDK

Choisir une image Docker de base

L'image Docker peut être considérée comme un ensemble d'instructions sur la façon de créer le conteneur. Une image peut être héritée d'une autre image (ou basée sur elle), en ajoutant des instructions supplémentaires par-dessus les instructions de base. Chaque image est constituée de plusieurs couches, qui sont effectivement immuables.

Souvent nous accordons peu d'attention à la taille d'une image, car nous la téléchargeons une seule fois et l'exécute pour toujours. Alors que la taille des images Docker est en fait très importante et elle a un impact sur :

- La latence du réseau : nécessité de transférer l'image Docker sur le web
- Le stockage : il faut stocker tous les binaires quelque part
- La disponibilité et l'élasticité du service : avec l'utilisation d'un orchestrateur de conteneurs, comme Kubernetes, Swarm (hors production), Nomad ou autre.
- Sécurité : il existe des librairies que vous ne devez pas utiliser, car elles présentent des vulnérabilités importantes (par exemple : libpng)
- L'agilité du développement : de petites images Docker = un temps de construction et un déploiement plus rapide.

Dans la suite nous construirons une image docker basée sur l'openJDK en présentant différentes méthodes de construction.

Type et variante d'un image docker OpenJDK

Les images OpenJDK existent en plusieurs versions, chacune étant conçue pour un cas d'utilisation spécifique.

• openjdk:<version>

C'est l'image de facto. Si vous n'êtes pas sûr de vos besoins, vous devriez probablement utiliser celle-ci. Elle est conçue pour être utilisée à la fois comme un conteneur jetable, ainsi que comme base pour construire d'autres images. Certaines de ces balises peuvent avoir des noms comme « jessie » ou « stretch ». Ce sont les noms des versions de Debian et ils indiquent la version utilisée.

• openjdk:<version>-alpine

Cette image est basée sur le populaire Alpine Linux, disponible dans l'image officielle alpine. Alpine Linux est beaucoup plus petite que la plupart des images de base des distributions (~5MB), et conduit donc à des images beaucoup plus fines en général.

• openjdk:<version>-windowsservercore

Cette image est basée sur Windows Server Core (microsoft/windowsservercore). En tant que telle, elle ne fonctionne que dans les endroits où cette image fonctionne, comme Windows 10 Professionnel/Entreprise (Anniversary Edition) ou Windows Server 2016.

• openjdk:<version>-slim

Cette image installe le paquetage -headless d'OpenJDK et il lui manque donc de nombreuses bibliothèques Java liées à l'interface utilisateur et certains paquets courants contenus dans la balise par défaut. Elle ne contient que les paquets minimaux nécessaires à l'exécution de Java.

Construire une image Docker pour une application Java basée sur l'OpenJDK

Nous passons en revue 3 façons différentes de créer des images Docker pour une application Java.

Première méthode :

Construction de package uniquement :

Dans le cas d'une construction par package, nous déléguons à un outil construction (Maven ou Gradle par exemple) le contrôle du processus de construction.

1 Nous ajoutons un Dockerfile à la racine du projet Java,

comme un manuel de construction de l'image Docker,

```
Dockerfile
1  # On utilise openjdk8 alpine une très petite distribution linux.
2  FROM openjdk:8-jre-alpine3.9
3
4  # copier le fichier jar dans l'image docker
5  COPY target/demo-0.0.1-SNAPSHOT.jar /demo.jar
6
7  # définir la commande de démarrage pour exécuter le jar
8  CMD ["java", "-jar", "/demo.jar"]
9
```

2 Maintenant, nous construisons l'application dans un .jar en utilisant Maven :

```
mvn clean package
```

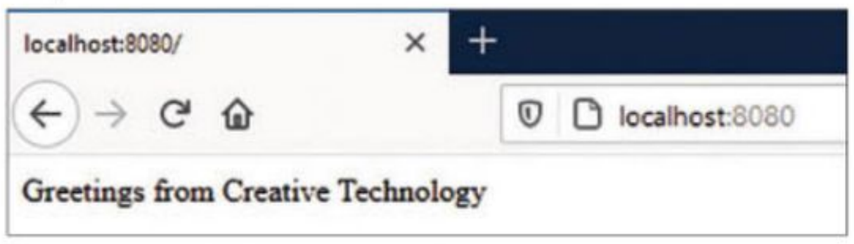
3 Et ensuite construire l'image Docker

```
docker build -t creative-tech/docker-package-only-build-demo:0.0.1-SNAPSHOT ./
```

4 Pour exécuter le conteneur à partir de l'image construite :

```
docker run -d -p 8080:8080 creative-tech/docker-package-only-build-demo:0.0.1-SNAPSHOT
```

5 Naviguez vers http://localhost:8080, et vous devriez voir ce qui suit :



Les avantages de cette approche :

- Elle permet d'obtenir une image Docker légère.
- Ne nécessite pas que Maven soit inclus dans l'image Docker.
- Il n'est pas nécessaire d'inclure les dépendances de notre application dans l'image.

Inconvénients de cette approche :

- Il faut que Maven et JDK soient installés sur la machine hôte.
- La construction Docker échouera si la construction Maven échoue ou n'est pas exécutée au préalable. Cela devient un problème lorsqu'on veut intégrer des services qui construisent (seulement) automatiquement en utilisant le présent fichier Docker.

Deuxième méthode :

construction par Docker

Dans cette construction, Docker contrôle le processus de construction.

```
Dockerfile
1  # sélectionner l'image parent
2  FROM maven:3.6.3-jdk-8
3
4  # copier l'arbre des sources et le pom.xml dans le nouveau conteneur
5  COPY ./ ./
6
7  # emPackager l'application
8  RUN mvn clean package
9
10 # définir la commande de démarrage pour exécuter le jar
11 CMD ["java", "-jar", "target/docker-maven-demo-0.0.1-SNAPSHOT.jar"]
12
13
```

1 Maintenant, nous construirons une nouvelle image comme dans l'étape 1 :

```
docker build -t creative-tech/docker-maven-build-demo:0.0.1-SNAPSHOT ./
```

2 Et exécutons le conteneur :

```
docker run -d -p 8080:8080 creative-tech/docker-maven-build-demo:0.0.1-SNAPSHOT
```

Les avantages de cette approche :

- Docker contrôle le processus de construction, cette méthode ne nécessite donc pas l'installation préalable de l'outil de construction ou du JDK sur la machine hôte.
- S'intègre bien aux services qui se contentent de construire (seulement) automatiquement en utilisant le fichier Docker présent.

Inconvénients de cette approche :

- Donne lieu à la plus grande image Docker des trois méthodes.
- Cette méthode de construction a non seulement packager l'application, mais aussi toutes ses dépendances et l'outil de construction lui-même, qui n'est pas nécessaires pour exécuter le JAR.
- Si la couche applicative est reconstruite, la commande mvn package forcera toutes les dépendances Maven à être tirées du dépôt distant à nouveau (on perd le cache Maven local).
- Si utilisation de repo privés (Nexus ou Frog), il est important de rajouter une configuration settings.xml afin de configurer les accès

Troisième méthode :

Construction en plusieurs étapes

Avec les constructions Docker à plusieurs étapes, on utilise plusieurs instructions pour chaque étape de construction. Chaque instruction crée une nouvelle couche de base et élimine tout ce dont nous n'avons pas besoin dans l'étape précédente.

```
Dockerfile
1  # la première étape de construction utilisera une image parent de maven 3.6.1
2  FROM maven:3.6.1-jdk-8-alpine AS MAVEN_BUILD
3
4  # copier le code pom et src dans le conteneur
5  COPY ./ ./
6
7  # package le code de notre application
8  RUN mvn clean package
9
10 # la deuxième étape de construction utilisera open jdk 8 sur alpine 3.9
11 FROM openjdk:8-jre-alpine3.9
12
13 # ne copier que les artefacts dont on a besoin depuis la première étape et ignore le reste
14 COPY --from=MAVEN_BUILD /docker-multi-stage-demo/target/docker-multi-stage-demo-0.0.1-SNAPSHOT.jar /demo.jar
15
16 # définir la commande de démarrage pour exécuter le jar
17 CMD ["java", "-jar", "/demo.jar"]
```

1 Construire l'image :

```
docker build -t ceative-tech/docker-multi-stage-demo:1.0-SNAPSHOT ./
```

2 Et ensuite, exécuter le conteneur :

```
docker run -d -p 8080:8080 ceative-tech/docker-multi-stage-demo:1.0-SNAPSHOT
```

Les avantages de cette approche :

- Elle permet d'obtenir une image Docker de petite taille.
- Ne nécessite pas l'installation préalable de l'outil de construction ou du JDK sur la machine hôte (Docker contrôle le processus de construction).
- S'intègre bien aux services qui se contentent de construire (seulement) automatiquement en utilisant le présent fichier Docker.

```
PS D:\Workspace\docker-multi-stage-demo> docker image ls
REPOSITORY                                  TAG               IMAGE ID       CREATED              SIZE
axance-tech/docker-multi-stage-demo         0.0.1-SNAPSHOT    8437511fe8e0   8 seconds ago        102MB
axance-tech/docker-maven-demo               0.0.1-SNAPSHOT    e0c217a7e14d   5 minutes ago        615MB
axance-tech/docker-package-only-build-demo  0.0.1-SNAPSHOT    2b6d87e51b3b   48 minutes ago       102MB
```

Figure 7

- Seuls les artefacts dont nous avons besoin sont copiés d'une étape à l'autre (c'est-à-dire que les dépendances de l'application ne sont pas intégrées à l'image finale comme dans la méthode précédente).
- Nous créons autant d'étapes de construction que nous le souhaitons
- Il est possible de s'arrêter à n'importe quelle étape de la construction d'une image en utilisant -target, par exemple

```
docker build - target MAVEN_BUILD -t ceative-tech/docker-multi-stage-demo:1.0-SNAPSHOT ./
```

Inconvénients de cette approche :

- Si la couche applicative est reconstruite, la commande mvn package forcera toutes les dépendances Maven à être tirées du dépôt distant une nouvelle fois (perte de l'avantage du cache Maven local).

Pour vérifier la taille des images construites

```
docker image ls
```

Figure 7

Parmi les trois méthodes de construction d'images Docker décrites dans cet article, la construction en plusieurs étapes est la plus appropriée. Nous obtenons le meilleur des deux mondes lors du package du code de l'application. Docker contrôle la construction, mais il est possible d'extraire uniquement les artefacts nécessaires. Cela devient particulièrement important lorsque nous stockons des conteneurs sur le cloud :

- Vous passerez ainsi moins de temps à créer et à transférer vos conteneurs sur le cloud, car les images sont beaucoup plus petites.
- Pour le coût, plus l'image est petite, moins le stockage est coûteux.
- Pour la sécurité, le vecteur d'attaque est plus petit, car la suppression des dépendances supplémentaires de l'image la rend moins vulnérable aux attaques.

Conclusion

Dans cet article nous avons présenté les différents fournisseurs de l'OpenJDK comme alternative de l'OracleJDK et des éléments clefs pour décider et justifier le choix d'une distribution par rapport à une autre. Nous avons abordé le sujet de l'optimisation de la construction des images de base docker pour les applications JAVA de point de vue taille de l'image et de distribution JDK sachant qu'il y a d'autres aspects à optimiser aussi comme la sécurité et limitation des vecteurs d'attaque et de spectre des vulnérabilités, l'optimisation de la couche réseau …

Finalement, afin de choisir une image Docker Java de base, il faut répondre aux questions suivantes :

- De quels paquets natifs avons-nous besoin pour notre application Java ?
- Devriez-vous choisir Ubuntu ou Debian (ou même Alpine) comme image de base ?
- Quel environnement d'exécution (quelle distribution de JDK)
- Quelle est notre stratégie pour corriger les failles de sécurité, y compris les paquets que nous n'utilisons pas du tout ?
- Étions-nous prêts à payer un supplément (argent et temps) pour le trafic réseau et le stockage de fichiers inutilisés ?

Références

Documentation :
https://adoptopenjdk.gitbooks.io/adoptopenjdk-getting-started-kit/content/en/adopt-openjdk-getting-started/adopt_openjdk_-_getting_started.html

Présentation : ***"Making Java Free again" de H. Ebbers et G. Adams*** de la conférence Eclipse Con'20 : https://speakerdeck.com/hendrikebbers/adoptopenjdk-making-java-free-again

Quality assurance AQA :
https://medium.com/adoptopenjdk/the-first-drop-introducing-adoptopenjdk-quality-assurance-aqa-v1-0-fe09f10ced80

AdoptOpenJDK for Docker :
https://github.com/docker-library/docs/blob/master/adoptopenjdk/README.md#supported-tags-and-respective-dockerfile-links

Java is still free (29/01/2021) de la communauté Java Champion :
https://docs.google.com/document/d/1nFGazvrCvHMZJgFstlbzoHjpAVwv5DEdnaBr_5pKuHo/edit#heading=h.p3qt2oh5eczi

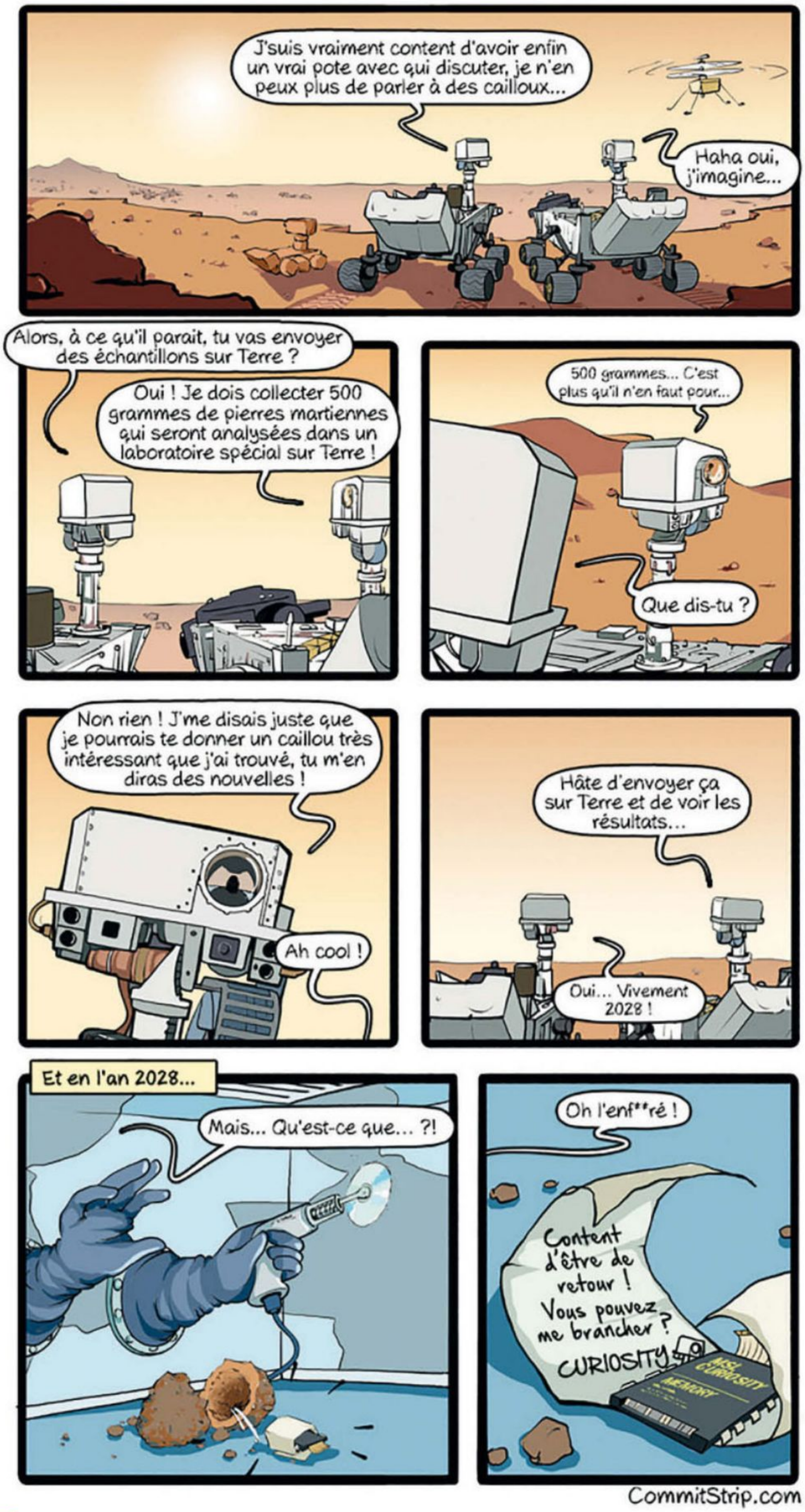
Azul Zulu : https://www.azul.com/products/zulu-enterprise/jdk-comparison-matrix/

Sdkman : https://sdkman.io/jdks

Cycle de vie d'Oracle : oracle.com/technetwork/java/eol-135779.html

Organigramme de choix du fournisseur : basé sur le travail de Basil Bourque
https://stackoverflow.com/users/642706/basil-bourque

pendant ce temps, sur Mars... vivement 2028



PROGRAMMEZ!

Programmez! hors-série n°4
ÉTÉ 2021

Directeur de la publication & rédacteur en chef
François Tonic
ftonic@programmez.com

Contacter la rédaction
redaction@programmez.com

Les contributeurs techniques
Océane Roudier, Nourdine Bouaghaz, Lilian Benoit, Loïc Mathieu, Elvadas Nono, Badr Nass Lahsen, Christophe Brun, Jean Bisutti, Antoine Michaud, Bruno Boucard, Daniel Petisme, Fabien Pomerol, Julien Millau, Loïc Mathieu, Juliette de Rancourt, Julien Topçu, Samuel Marques Antunes, Fayssal Merimi, CommitStrip

Couverture
© Swillklitch, Oracle

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

tangente 200

l'aventure mathématique

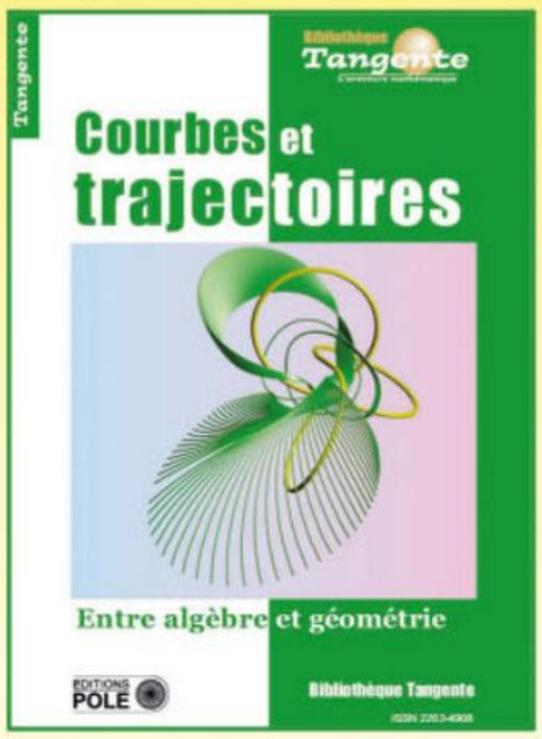
Il vient de paraître : le numéro 200 de *Tangente*

Un numéro « Collector » à ne pas manquer !

Avec son supplément à détacher : un cahier central d'œuvres d'art format A3 à afficher selon vos goûts.

Les derniers numéros parus

Le numéro 200



Bibliothèque Tangente 74

Le hors-série 78

Les numéros de *Tangente* sont disponibles :

– chez votre marchand de journaux

– chez votre libraire pour la bibliothèque Tangente

– en les commandant ou s'abonnant sur infinimath.com/librairie

– en ligne sur tangente-mag.com