REACT / ESP / MAKER / GIT / JAVA / TEST / POWERSHELL

C++ / UX / GITLAB / GITHUB / KUBERNETES / DEVOPS / ONLYOFFICE

PROGRAMMEZ!

Le magazine des développeurs

03/ 2020 N°238 22e année

Go

LANGAGE INCONTOURNABLE en 2020

gopher : © D.R. ville : JIMMY UTHE

M 04319 - 238 - F: 6,50 € - RD

Le seul magazine écrit par et pour les développeurs

Printed in EU - Imprimé en UE - BELGIQUE 7 € - Canada 9,80 $ CAN - SUISSE 13,10 FS - DOM Surf 7,50 € - TOM 1020 XPF - MAROC 55 DH

N°2
DISPONIBLE

Retour vers le passé :

redécouvrez les ordinosaures et les technologies des années 1970 à 2000 !

Commandez directement sur **programmez.com**

6,66 € *(+frais de port*)* **36 pages**

*Revue trimestrielle. Editée par Nefer-IT. *Avec frais de port : 7,66 €*

Faut-il travailler son poker face ?

Que faire quand la situation est… disons… légèrement en red condition ? C'est là que vous devez garder votre calme, même si vous avez cassé la prod en lançant une commande sur le mauvais serveur (non, non, je sais que vous l'avez déjà fait) ou quand on vous vend un super projet mais qu'en réalité c'est la loose totale côté techno et délais (là encore, je sais que vous l'avez déjà subi)…

Par exemple, quand on vous annonce que le super top tendance projet est en réalité la maintenance d'un projet tout moisi de +15 ans avec IE6, VB6 et Access (Access c'est cadeau) :

En réalité **Poker face**

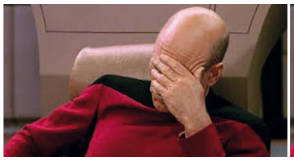
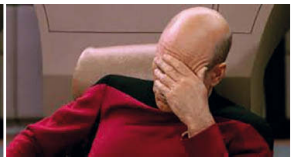

Quand votre compil ne veut pas se faire après la 42e tentative !

En réalité **Poker face**

Bon ok, là, on peut pas. Allez! Je relance une 43e build ! Avec un peu de chance…

Ou quand vous entendez parler technique durant les réunions entre clients et commerciaux et que vous êtes obligés d'entendre ça. Marche aussi en code review, en découvrant les tickets de support, en regardant un utilisateur utiliser la nouvelle interface, etc.

En réalité **Poker face**

Bref, dans tous les cas, restons zen.

François Tonic - ftonic@programmez.com

SOMMAIRE

MESSAGE PRIORITAIRE :
Programmez! n°239, prochain numéro
date stellaire : 31.03.2020

Quantique / Quand le développeur doit être éco-compatible / DevOps partie 2

Obsolescence programmée : Apple prend sa prune

La DGCCRF a annoncé début février avoir infligé une amende de 25 millions € à Apple. L'amende vient sanctionner le comportement d'Apple, qui n'a pas averti ses utilisateurs que ses mises à jour d'iOS10 pouvaient ralentir le fonctionnement de certains modèles d'iPhones. Apple avait déjà été condamné pour ces pratiques en Italie en 2018.

Piratage de Jeff Bezos : l'Arabie Saoudite pointée du doigt

Le dirigeant d'Amazon a bien vu son smartphone piraté pendant l'année 2018. Selon une enquête menée par des proches du milliardaire et dont les conclusions ont été publiées par le Guardian, ce piratage aurait eu pour principal vecteur l'envoi d'un SMS piégé par le prince héritier d'Arabie Saoudite, Mohammed Bin Salman. La piste d'un piratage par les autorités saoudienne avait déjà été évoquée en 2019 par un consultant en sécurité, engagé par Bezos pour enquêter sur l'affaire. Le piratage visait à faire pression sur Bezos, alors propriétaire du New York Post. Le quotidien américain offrait régulièrement des tribunes à Jamal Khashoggi, l'opposant au régime assassiné par les services secrets saoudiens en octobre 2018.

Bouygues construction se fait attraper par un ransomware

La société Bouygues Construction a fait savoir à la fin du mois de janvier qu'elle était victime d'une attaque au ransomware ayant paralysé son système d'information pendant plusieurs jours. Bouygues reste discret sur la nature de l'attaque, mais les opérateurs du ransomware Maze ont revendiqué l'opération sur un site dédié. Ce groupe, spécialisé dans les attaques visant de grosses entreprises, est connu pour faire du chantage aux données volées en plus de simplement chiffrer les machines de ses cibles. Et en profiter pour demander d'importantes rançons aux victimes.

5G : Orange opte pour Nokia et Ericsson

Après Free, au tour d'Orange de formaliser ses choix de partenaires industriels pour le déploiement de son futur réseau 5G en France. L'opérateur historique a misé sur la stabilité en optant pour Nokia et Ericsson. Exit donc Huawei, accusé par Washington de collusions avec le régime chinois. Alors que le cas du constructeur chinois devrait rapidement être tranché par l'ANSSI et Matignon, seuls Bouygues Telecom et SFR doivent encore se positionner pour rendre publics leurs futurs partenaires.

Nvidia mise sur le Cloud gaming avec GeForce Now

Nvidia se lance dans la bataille du cloud gaming. Le géant californien a annoncé le lancement de son service GeForce Now auprès du grand public. Au contraire de Stadia ou de Playstation Now, les services de ses concurrents Google et Sony, GeForce Now sera compatible sous Windows, Mac, Shield TV et Android et prendra en charge les services Steam, Epic Games Store, Battle.net et Uplay. De quoi ravir les gamers, qui devront toutefois débourser 5,49 euros par mois pour avoir accès à des sessions de six heures renouvelables et à des jeux proposés en version RTX sur ce nouveau service.

ProtonVPN joue la carte de la transparence

Dans un effort de transparence, ProtonVPN a publié le code source de ses différentes applications Windows, macOS, Android et iOS publiquement sur Github. La société suisse, qui propose déjà la solution d'emails chiffrés Protonmail, indique que cette décision vise à se démarquer de la concurrence en jouant la carte de la transparence, mais aussi de permettre à des chercheurs tiers d'auditer le code pour remonter d'éventuelles failles de sécurité.

Le coronavirus sème la pagaille au MWC

L'épidémie du Coronavirus fait trembler les organisateurs du Mobile World Congress, qui doit se tenir à Barcelone du 24 au 28 février. L'infection mortelle qui balaie actuellement la Chine a conduit certains acteurs majeurs des télécommunications comme Sony, LG et Ericsson à annuler purement et simplement leur venue à la grand-messe de l'industrie mobile. D'autres acteurs, tels que ZTE et Xiaomi ont réduit leur délégation au strict minimum tandis que l'organisateur du salon, la GSMA, a pris un ensemble de mesures drastiques pour limiter les risques. Les organisateurs ont finalement annulé le salon.

Equifax : les États-Unis désignent Pékin

Au cours d'une conférence de presse, le ministère américain de la Justice a officiellement mis en examen quatre ressortissants chinois, accusés d'être à l'origine du piratage de la société de crédit Equifax. Les faits se sont produits en 2016 et ont permis aux attaquants de mettre la main sur les données personnelles de plus de 140 millions d'Américains.

9e édition

Conférence & Expo

9 & 10 mars 2020

Palais des Congrès • Paris

BIGDATA PARIS BY Corp

18 000 + Visiteurs

300 + Partenaires

100 + Speakers

Rejoignez l'événement Big Data de la scène européenne sur **www.bigdataparis.com**

LES MEETUP PROGRAMMEZ

17 Mars : meetup#9

Comparaison Kotlin Multiplatform & Flutter

Par Valentin Michalak (développeur)

21 avril : meetup#10 *(Sujet à venir)*
19 mai : meetup#11 *(Sujet à venir)*
16 juin : meetup#12 *(Sujet à venir)*

A PARTIR DE 18H30

Où : Infeeny 5 rue d'Uzès Paris
Métro : station Grands Boulevards (lignes 8 & 9)

Informations & inscription : programmez.com

Mars

3 : NYAN / Paris

Criteo organise sa nouvelle conférence technique : Not Your Average .Net Conference ! Objectif : plonger au cœur de .Net. On va parler JIT, .Net GC, exceptions, debug asynchrone… Bref du très lourd !
Site : https://nyanconference.splashthat.com

17 : AWS Summit Paris/Paris

Amazon Web Services propose sa grande journée française pour présenter les services, les supports et les partenaires. De nombreuses sessions et des keynotes !
Site : https://aws.amazon.com/fr/events/summits/paris/

25-27 : BreizhCamp/Rennes

Le BreizhCamp, ce sont 3 jours de conférence à Rennes. Créé en 2011 à l'initiative du BreizhJUG, nous sommes en cours de planification de la 10e édition. Le BreizhCamp propose de rencontrer une communauté de développeurs et d'experts, avec un contenu à la carte sur plus de 100 thèmes présentés. Chaque participant est libre de suivre les sujets qui ont retenu son attention ou de préférer les ateliers pour mettre en pratique les connaissances acquises.
https://www.breizhcamp.org

Avril

7 : JFTL / Montrouge

La journée des tests logiciels revient pour la 12e année. Le but est de montrer, expliquer les tests, et pourquoi ils sont si importants. 1 000 personnes sont attendues. La journée du 6 est réservée aux ateliers et sessions avancées. Site : http://www.cftl.fr/JFTL/accueil/

15-17 : Devoxx France/Paris

Devoxx France c'est du 15 au 17 avril 2020. La 9e édition s'annonce très bien. La première nouvelle c'est qu'il n'y a plus de places à vendre... depuis l'ouverture. Jeudi et vendredi matin, il y aura 5 à 6 plénières, sur un format de 20mn, proche de ce que font les conférences TED. Le thème de cette année est "Tech 4 good, Tech 4 evil". Comment notre métier et la technologie peuvent-ils améliorer (ou non) la vie au quotidien de nos clients ? Une personne de la Fondation de l'Abbé Pierre, un chercheur, une CEO dans l'humanitaire... Sans dévoiler le programme organisé par Benoit Lafontaine, Katia Aresti et Charlotte Abdelnour, on peut s'attendre à de belles choses ! +920 sujets reçus pour 240 sessions ! Bref, c'est l'évènement incontournable des dévs.

24 : Serverless Days / Paris

L'architecture serverless se répand rapidement dans les entreprises et les infrastructures. Une journée complète pour discuter avec les meilleurs experts du domaine :
https://paris.serverlessdays.io

29 & 30 : MixIT/Lyon

De nombreux thèmes sont abordés : design, technologie, makers, éthique dans l'IT, style de vie, travail en équipe, etc. https://mixitconf.org

Mai

6-7 : GitHub Satellite/Paris

La conférence européenne de GitHub s'installe pour la première fois en France. Durant 2 jours, l'écosystème GitHub fait le show et l'éditeur propose des dizaines de sessions et d'ateliers. Un évènement à ne pas rater.
Site : https://githubsatellite.com

13-15 : RivieraDev/Sophia Antipolis

La conférence développeurs du sud revient. 42 conférences, 18 ateliers et +500 développeurs présents. Le 13 est réservé aux sessions intensives de 3h avec des experts niveau 300 ! Les 14 et 15 sont les journées classiques des sessions.

28-29 : newcrafts / Paris

La conférence newcraft revient à Paris pour deux jours dédiés aux méthodes de développement, à la programmation et à la technologie ! Site : https://ncrafts.io

Juin

2 : FlutterConf Paris / Paris

La première grande conférence autour de Flutter arrive début juin. Vous y découvrirez les dernières avancées en matière d'applications mobiles natives multiplateforme (Android et iOS), pour le web ou encore le desktop (Windows, Mac et Linux).
Site : https://flutter-conf.paris/fr_FR/

LES CONFÉRENCES DOT EN FRANCE

2 mars : dotPy/Paris

Une des plus importantes conférences en France sur le langage Python !
https://www.dotpy.io

30 mars : dotGo/Paris

Comme son nom l'indique, la conférence dot 100 % langage Go !

4-5 : Best of web 2020/Paris

Best of web, c'est un évènement sur deux jours. Une journée de workshops et une journée de conférences préparée par 16 meetups web parisiens. La journée-conférence est composée à 50 % du Best Of des talks meetups de l'année et 50 % d'inédits provenant d'une CFP. Les tickets sont spécifiques à chaque journée.
Site : http://bestofweb.paris

12 : DevFest Lille/Lille

La DevFest revient à Lille. L'appel aux speakers est lancé. C'est le plus grand évènement dans le nord, avec +800 personnes attendues.
Site : https://devfest.gdglille.org

Octobre

30 : Agile Tour Brussels/Bruxelles

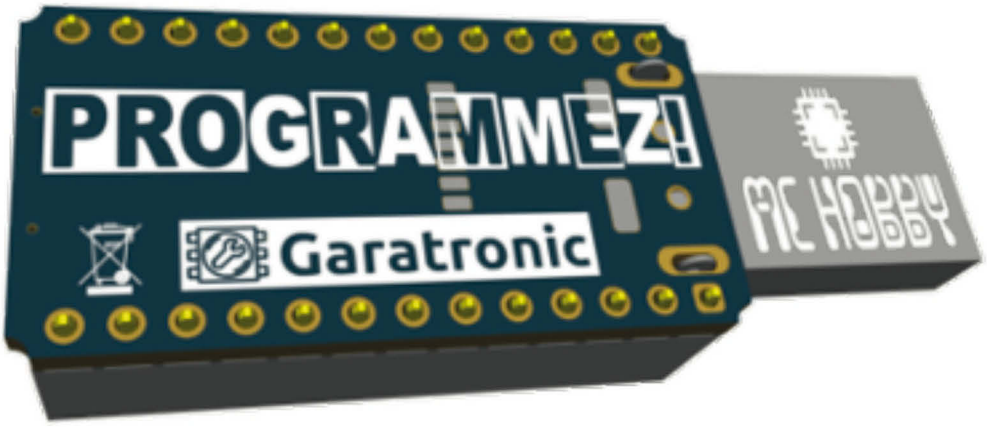
C'est l'évènement agilité en Belgique. +200 personnes attendues. Site : http://www.agiletourbrussels.be

Nutanix, le spécialiste du cloud computing d'entreprise organise une série d'événements .Next on Tour dans cinq villes à travers la France dont Toulouse (5 mars), Lilles (17 mars), Nantes (19 mars), Aix en Provence (24 mars), Lyon (26 mars). Pour plus de renseignements et inscriptions https://www.nutanix.com/next/on-tour

Merci à Aurélie Vache pour la liste 2020, consultable sur son GitHub :
https://github.com/scraly/developers-conferences-agenda/blob/master/README.md

Printemps 2020

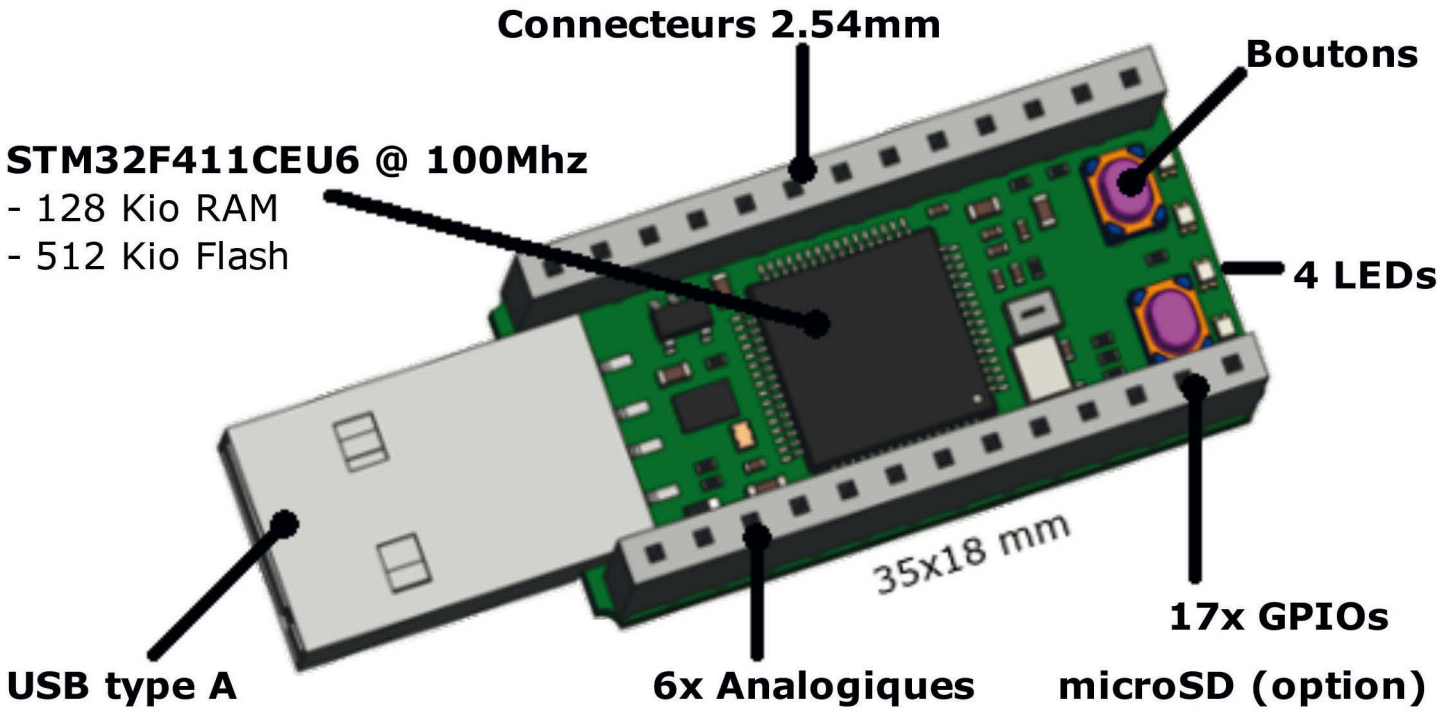
Programmez en C et Python avec la PYBStick



Arduino

MicroPython

Programmez!, Garatronic et MC Hobby s'associent pour vous offrir une plateforme d'apprentissage **surpuissante**, **compacte**, **abordable** et **documentée en Français**!



Ressource 3D

Open hardware

Tutoriels

Fabriqué en FRANCE

Roadmap des langages

Chaque mois, Programmez! vous propose un panorama des sorties des versions des langages, frameworks, etc.

Symfony

Le framework PHP évolue tranquillement, mais sûrement. Actuellement, on peut utiliser la version 4.4 et 5.0.

Côté support :

4.4.x : sortie en novembre 2019, support actif jusqu'en novembre 2022
5.0.x : support jusqu'en juillet 2020
5.1 : sortie prévue en mai et fin support janvier 2021
Parmi les nouveautés disponibles, on notera les nouvelles méthodes Dom Crawler et la gestion des secrets cryptés.

Flutter 1.12

Cette version propose une belle panoplie de nouveautés et améliorations. Citons :

- Le support du mode sombre iOS 13
- Version alpha de Flutter pour macOS
- Flutter pour le web arrive en bêta
- La version de DART évolue aussi. Flutter supporte la v2.7
- Meilleure stabilité des API pour Java, Kotlin, Objective-C, Swift

Cette version est sortie en décembre. Depuis, de nombreuses mises à jour ont été déployées. La dernière en date est la 1.12.13 : https://flutter.dev/docs/development/tools/sdk/release-notes/release-notes-1.12.13#breaking-changes

Python 3.9.0

Eté / automne

Les premières versions alpha de la 3.9 ont été distribuées fin janvier. La phase bêta est attendue vers mai / juin. Désormais, il faudra s'attendre à une mise à jour annuelle du langage. Bref : moins de nouveautés mais plus d'évolutions et d'améliorations. Parmi les nouveautés attendues : des améliorations sur le garbage collector et la résurrection d'objets (pouvant provoquer des blocages), évolution de la stabilité ABI pour éviter les incompatibilités sur les différents systèmes.

PHP 8.0

Fin 2020 ?

La prochaine grande version de PHP doit arriver fin 2020 si tout va bien. Tout n'est pas encore fixé mais nous connaissons les premières nouveautés et évolutions du langage :

- Forte possibilité d'une casse de compatibilité avec les version 7.x : sur ce point, on attend les précisions ;
- Les types Union vont arriver avec PHP 8, qui permettront de définir plusieurs types pour les arguments reçus par une fonction, ainsi que pour la valeur qu'elle retourne. En plus du type self, le type static va devenir un type de valeur retournée valide ;
- Le compilateur JIT de PHP 8 promet d'importantes améliorations de performances.

Rust 1.41

Disponible

En ce qui concerne le langage lui-même, une des nouveautés les plus importantes est l'assouplissement de certaines restrictions lors de l'implémentation des traits. On trouve aussi des améliorations sur le gestionnaire des packages Cargo. Avec cargo install, vous pouvez installer des crates binaires dans votre système. La commande est souvent utilisée par la communauté pour installer des outils CLI populaires écrits en Rust.

Java 14

Mars

Java 14 proposera assez peu de choses à en croire les premières builds. On notera le JFR Event Streaming (consommation des données JFR). Il permettra de collecter des données de profilage et d'analyse d'une application Java en exécution. On aura aussi les expressions switch. Elles doivent simplifier le codage. NullPointerException aura droit à quelques améliorations. Java supportera aussi les mémoires NVM.

Kotlin 1.3.70

Cette future version annonce de nouvelles choses :

- Meilleur support de gradle.kts. Les développeurs veulent améliorer le support des fichiers et réduire la charge CPU durant la synchronisation ;
- Sur Kotlin/JavaScript : on attend une optimisation sur les navigateurs. Des tâches vont changer de nom comme browserWebPack. Donc méfiance.

Lien : https://discuss.kotlinlang.org/t/kotlin-1-3-70-early-access-preview/15876

TypeScript 3.8

Actuellement, la 3.8 est en bêta. Cette mouture se fait principalement remarquer par une nouvelle fonctionnalité permettant d'importer/exporter des types seuls.
On notera aussi le support de l'ECMAScript Private Fields.
Lien : https://devblogs.microsoft.com/typescript/announcing-typescript-3-8-beta/

C++20

Février.

Cette version du langage met en avant deux nouveautés : les modules et les coroutines. Les modules constituent une nouvelle alternative aux fichiers d'en-tête qui apportent un certain nombre d'améliorations clés, notamment en isolant les effets des macros et en permettant des compilations évolutives, explique Herb. Il ajoute qu'en 35 ans c'est la première fois que C ++ ajoute une nouvelle fonctionnalité permettant aux utilisateurs de définir une limite d'encapsulation nommée. Les coroutines sont elles aussi une fonctionnalité à remarquer. Une coroutine est une unité de traitement qui s'apparente à une fonction (ou routine), avec cette différence que si une sortie du corps d'une fonction met fin à l'exécution de celle-ci, la sortie de la coroutine suspend seulement son traitement qui peut ensuite reprendre, l'état du traitement à la sortie étant conservé. Une coroutine peut être vue comme un morceau de programme qui conserve son état, mais qui n'a pas de thread d'exécution. Les coroutines peuvent notamment être utilisées pour des itérateurs et des générateurs.

.Net 5

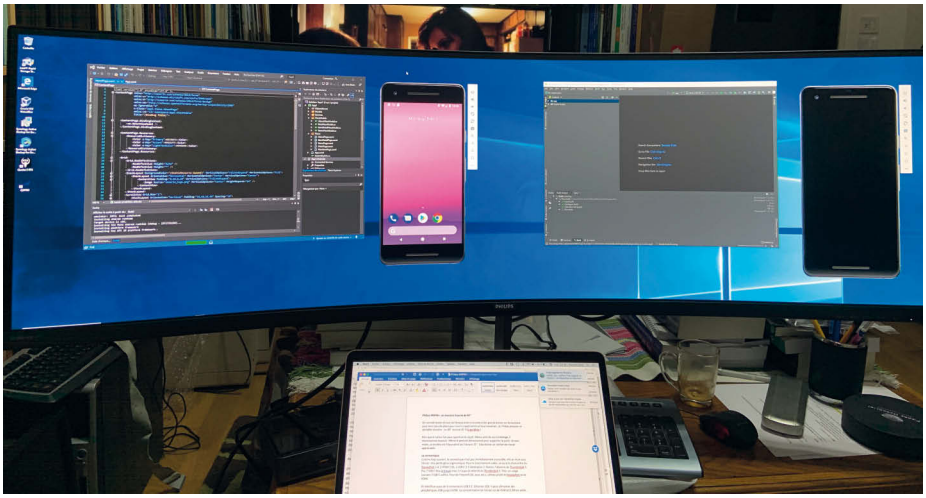
Novembre 2020.

L'annonce en a été faite à la dernière conférence BUILD. Il s'agit de .Net Core vNext, donc au-delà de .Net Core 3.0. Il s'agit de réunifier les noms. L'ambition est d'être disponible sur Windows, Linux, macOS, iOS, Android, tvOS, webassembly, etc.

François Tonic

Philips 499P9H : un monstre incurvé de 49''

On connaît toutes et tous les fameux écrans incurvés et les grands écrans sur les bureaux pour avoir plus de place afin d'ouvrir x applications et tout visualiser. Là, Philips propose un véritable monstre : un 49'' incurvé 32 : 9 SuperWide !



Rien que le carton fait peur quand on le reçoit. Même sorti de son emballage, il impressionne toujours. Le pied est dimensionné pour supporter le poids et apporter une bonne stabilité. Grosso modo, ce modèle est l'équivalent de 2 écrans 27''. Cela donne un confort de travail appréciable.

La connectique

Comme trop souvent, la connectique n'est pas immédiatement accessible, elle se situe sous l'écran. Une petite gêne ergonomique. Pour le branchement vidéo, on aura le choix entre du DisplayPort 1.4, 2 HDMI 2.0 b, 1 USB-C 3.1 Génération 2. Notons l'absence du Thunderbolt 3. Oui, l'USB-C fera le travail, mais il n'a pas la vélocité du Thunderbolt 3. Pour un usage courant, l'USB-C suffira. Pour de l'intensif (3D, jeux, etc.), utilisez plutôt le DisplayPort ou le HDMI.

On bénéficie aussi de 3 connecteurs USB 3.1, Ethernet 1Gb. Il peut alimenter des périphériques USB jusqu'à 65 W (bien mais 85 aura été encore mieux). La consommation de l'écran est de 45W et 0,3 W en veille.

Le design

Le design est plutôt agréable. Le pied est bien pensé et stable. Le réglage de la hauteur est facile et demande peu d'efforts. L'écran a une très belle finition. À noter que la caméra est rétractable. Par défaut, elle est rentrée. Philips propose un modèle réussi, agréable visuellement, avec une belle qualité de fabrication et de finition.

L'écran affiche une belle dynamique et luminosité. L'angle de vision est très bon. Nous ne sommes pas sur des écrans professionnels ni calibrés, mais honnêtement, l'affichage est agréable, sans scintillement visible ou gênant.

Par défaut, l'écran est livré avec le pied, un câble HDMI, DP, USB-A & USB-C, le cordon d'alimentation.

Côté résolution, le constructeur annonce une résolution maximale de 5 120 x 1440 à 60 MHz. À noter que l'écran utilise Adaptive-Sync.

Attention : sur Mac, macOS Mojave gère (très) mal cette résolution. Utilisez des versions plus récentes.

La caméra escamotable est compatible avec Windows Hello, si votre système/machine sait le gérer.

À l'usage

Nous avons voulu utiliser cet écran avec un Intel NUC de dernière génération sous Windows 10. Pas de souci de reconnaissance de la dalle et de la résolution. On prend immédiatement possession de l'espace. Seul étonnement, du moins au début, les trajets de la souris ! Il y a de la distance entre les deux extrémités ! Sinon, l'écran réagit bien, nous n'avons pas vu de gros défauts ou de saccades. Bien entendu, la puissance de votre GPU sera un facteur important pour un écran d'une telle dimension, surtout si vous passez la qualité de l'affichage…

Si vous êtes gamer, l'immersion est très réussie et le jeu prend tout son sens, surtout si c'est un jeu d'action. On a ressorti un vieux Quake optimisé, et le résultat est super fluide.

Pour le développement, plus difficile de déterminer son usage au quotidien avec les IDE. Car c'est l'écran qui permet d'organiser son espace de travail avec plusieurs IDE, émulateurs et outils. Pour les longs fichiers sources, avec la difficulté de splitter verticalement l'affichage, l'intérêt d'un tel écran peut rapidement être limité, un écran plus classique sera suffisant.

Par contre, avoir la main sur plusieurs émulateurs ou versions de navigateur pour tester une app web, le Philips prend tout son sens. Cela deviendra réellement un confort de travail appréciable, finalement plus souple que plusieurs écrans indépendants.

Bien entendu, vous pouvez créer plusieurs bureaux virtuels : chaque bureau aura ses outils. Pratique à l'usage, pour un tel écran, nous sommes plus sceptiques même si c'est un des usages courants.

Pour ou contre ?

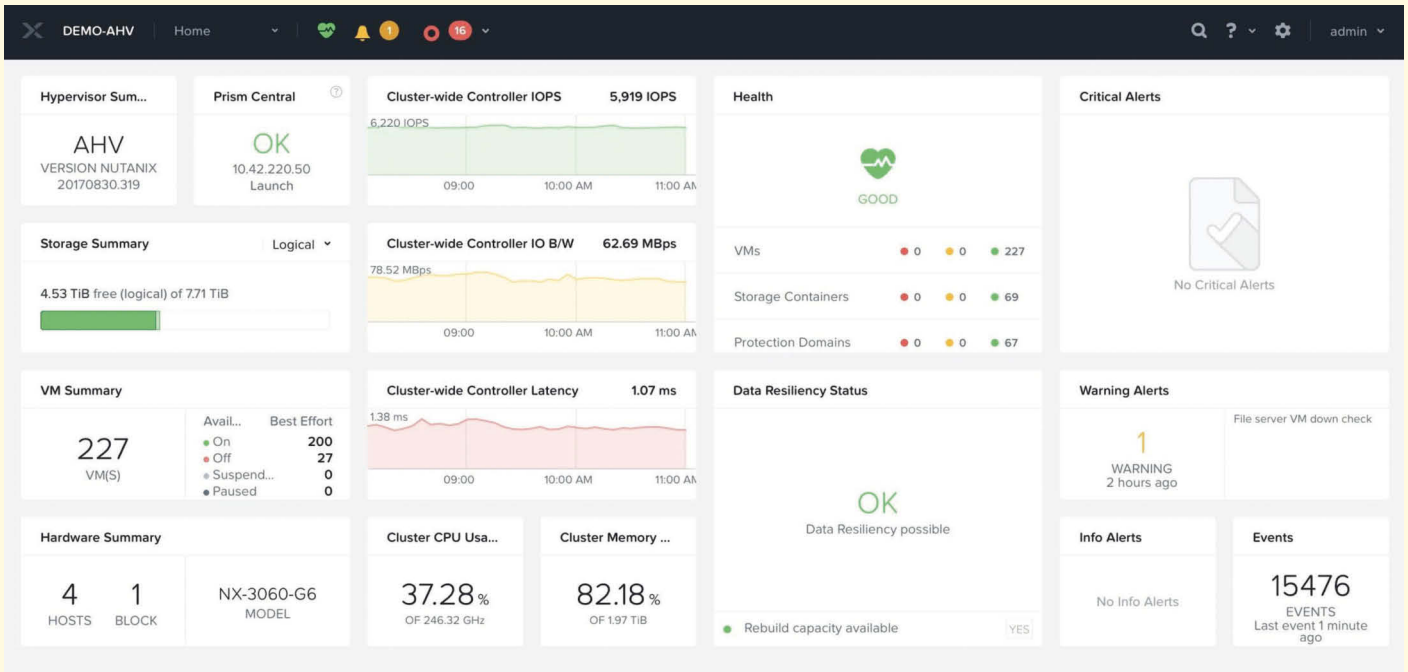
Imposant, design réussi, Philips produit un matériel de qualité. La dalle est de belle facture, avec des couleurs vives sans tomber dans l'excès. Finalement, la question est toujours la même : quelle utilité ? 1 seul écran ou plusieurs ?

Le 499P9H est une solution intéressante pour les développements web et mobile, 3D, les jeux. Pour des projets plus classiques, le choix de plusieurs écrans s'avère plus intéressant, notamment par le format de l'écran et les résolutions supportées.

Ce modèle est vendu environ 999 €. Un bon écran 32'' 4 ou 5K revient souvent à 600-700 €.

Nutanix et les 5 piliers d'une infrastructure DevOps

L'approche DevOps a beau être avant tout une philosophie, une démarche organisationnelle, une culture de développement et de déploiement, elle n'en demeure pas moins irréalisable sans une infrastructure moderne. Champion de l'infrastructure agile, Nutanix simplifie considérablement la démarche.



Pour répondre aux besoins de réactivité et d'efficacité des métiers, de nombreuses DSI optent désormais pour une réorganisation de leurs équipes IT selon le modèle DevOps. Objectif : réduire les frictions qui existaient autrefois entre équipes de développement et les équipes des opérations IT, et plus encore entre les développeurs et le client, qu'il soit interne ou externe.

Mais, si le modèle DevOps préconise une réorganisation des équipes IT et une nouvelle culture permettant de développer, tester et livrer des logiciels et services de façon rapide, fréquente et fiable, il présuppose également une infrastructure agile, automatisée et automatisable, avec des mécanismes de self-service pour monter des environnements de développements et de tests à la demande, industrialiser les tests et les mises en production, sans oublier une supervision des opérations centralisée et simplifiée. De manière générale, toute approche DevOps repose sur cinq piliers essentiels. Chacun d'eux peut bénéficier des fonctionnalités intrinsèques de Nutanix Enterprise Cloud pour gagner en fluidité, simplicité et robustesse.

Pilier n°1 : une infrastructure flexible, évolutive, répartie et immuable

Toute architecture DevOps doit pouvoir s'appuyer sur une infrastructure offrant à la fois cohérence dans le déploiement, simplicité d'administration et de supervision, montée en charge et performances prédictibles. Nutanix Enterprise Cloud procure à l'entreprise toutes ces fonctionnalités d'autant plus simplement que son approche Software Defined et ses automatisations rendent l'infrastructure complètement invisible. Par son approche hyperconvergée à la fois du stockage, des serveurs, du réseau, elle offre en interne une agilité similaire à celle d'un cloud public : elle permet un déploiement facile et rapide des VMs, un déploiement rapide et automatisé de clusters Kubernetes, un stockage évolutif, fiable et multifacettes (blocs, fichiers, objets) ainsi qu'une observabilité des applications qu'elles soient en VMs ou en containers.

Pilier n°2 : L'infrastructure as code

En DevOps, développement et opérationnel sont intimement liés, ce qui implique que la configuration de l'infrastructure puisse être pilotée de manière programmatique. Le DevOps impose en effet une « infrastructure as code », indispensable pour permettre une forte automatisation de tout le processus allant du développement à la mise en production, mais également essentielle pour rendre l'infrastructure à la fois reproductible et sans silos.

L'interface d'administration Nutanix Prism brise les silos opérationnels et offre une simplicité « one click » sans équivalent. En outre, toutes les opérations réalisables dans Prism sont également accessibles au travers d'API faciles à appréhender et permettant une automatisation complète de toute la stack Nutanix.

Pilier n°3 : une infrastructure en self-service

Les développeurs doivent bénéficier d'une autonomie maximale et accéder aux ressources qui leurs sont nécessaires, au moment où ils en ont besoin avec une réactivité totale,

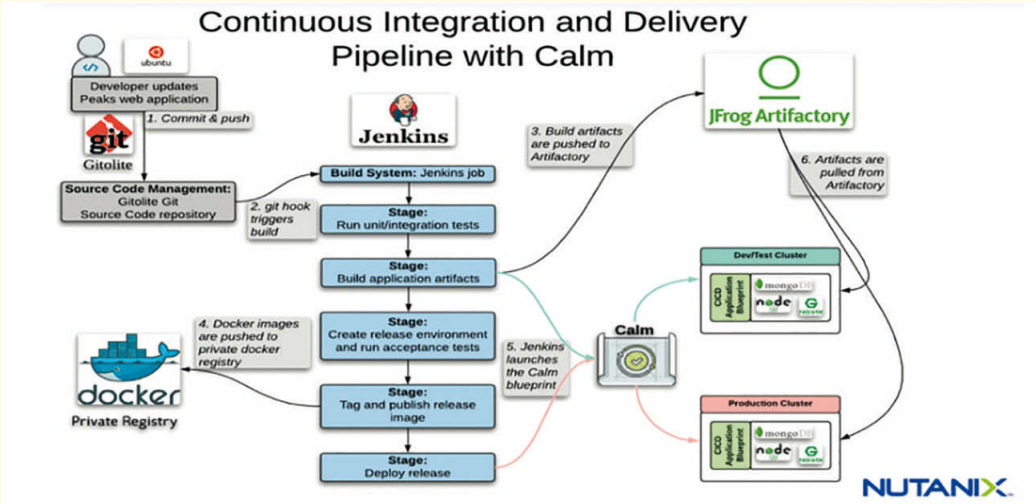


Fig 1 : Schéma de déploiement d'un processus d'intégration continue pour une application web avec Calm.

sans freins organisationnels et techniques. Nutanix Prism incorpore un portail self-service qui permet aux développeurs de provisionner et gérer directement leurs environnements de développement et de tests avec les ressources d'infrastructures associées.
Au-delà, l'idée même d'une infrastructure en self-service ne peut se concrétiser sans une fondation capable de monter en charge à la demande et qui garantit performances et résilience. Ce sont autant de caractéristiques qui font partie de l'ADN de l'infrastructure hyperconvergée Nutanix. *[Fig 1]*

Pilier n°4 : Le CI/CD autrement dit l'intégration, la livraison et le déploiement en continu

Toute approche DevOps doit s'appuyer sur une chaîne de bonnes pratiques, d'outils et de fondations d'infrastructures qui assurent qu'un code source est sans cesse dans un état déployable. C'est le concept même de l'intégration continue. Pensé pour automatiser le cycle de vie des applications, Nutanix Calm simplifie la mise en place de stratégies CI/CD. Cette solution d'automatisation de la gestion du cycle de vie applicatif apporte de nombreux outils aux développeurs, notamment dans la mise en place de modèles de développement DevOps et d'intégration continue. Elle offre également une marketplace pour accéder de façon normalisée et sécurisée à toutes les briques nécessaires à la construction d'environnements de développement.

Pilier n°5 : Une sécurité intégrée

Enfin, la gestion des réseaux et l'application des politiques de sécurité peuvent rapidement devenir extrêmement complexes dans les environnements DevOps.
Simplicité, sécurité et observabilité… Nutanix Enterprise Cloud est conçu pour simplifier l'orchestration du réseau et permettre de l'adapter dynamiquement aux besoins des applications, mais également pour le sécuriser sans ajouter de complexité (au travers de politiques de sécurité propres à chaque application et décorrélées du réseau physique) tout en offrant une visualisation centrée sur les applications.
Par ailleurs, Nutanix Flow offre aussi des fonctionnalités de pare-feu distribué (microsegmentation) qui protègent chaque VM au sein du cluster Nutanix. Simples et intuitives, les fonctions de microsegmentation délivrées par Flow sont particulièrement adaptées aux approches DevSecOps.

À ces cinq piliers s'ajoute encore l'instrumentation complète dopée au machine learning de Nutanix Insights offrant une vision globale de toute l'infrastructure étendue et un suivi proactif de l'état des systèmes et des applications. L'adoption de Nutanix Enterprise Cloud peut donc fournir une fondation solide et permettre d'accélérer la mise en oeuvre des initiatives DevOps.

3 questions à JAMES KARUTTYKARAN

Directeur technique de Nutanix Europe du Sud

1 On voit beaucoup de nouvelles solutions arriver chez Nutanix. Comment peuvent-elles aider les développeurs à mettre en place de nouveaux services ?

Aujourd'hui, une de nos premières préoccupations concerne la modernisation des applications et nous avons mis en place beaucoup d'outils qui peuvent servir aux développeurs. Nous avons notamment lancé Era, solution qui permet de déployer, de dupliquer, de rafraîchir ou de déplacer rapidement des bases de données. En un clic, il est ainsi possible de copier une base de données, avec ses paramètres et sa VM. Cela permet de faciliter de nouvelles solutions en simplifiant la mise à disposition de copie de base de données. Era peut également servir à migrer les bases de données des environnements de développement vers ceux de production. Parmi les autres produits qui vont faciliter le travail des développeurs, on peut citer Karbon. Ce service managé Kubernetes officiellement certifié peut être allié à Era et Buckets, et permet la mise en place d'infrastructures cloud natives entièrement automatisées sur des environnements Nutanix. Cette architecture permet notamment de développer nativement les applications rapidement sur des environnements On Premise pour les pousser ensuite sur le cloud sans avoir à en modifier le code, ou l'inverse. Et nous venons tout juste de lancer la version 2.0 de Karbon qui prend notamment en charge les environnements air gapped.

2 Et qu'en est-il de l'IoT et du Edge computing qui devient un sujet majeur aujourd'hui

Avec notre offre Xi IoT, on propose aujourd'hui tout un ensemble de services qui permettent d'apporter et de déployer massivement de l'intelligence sur le Edge. Les développeurs peuvent accéder à une plateforme de services managés regroupant des conteneurs, des connecteurs IA, ou des data services comme Kafka ou Elastic Search. En s'appuyant sur nos services, ils peuvent développer leurs propres applications sur le Edge en bénéficiant des solutions de management et de gestion du cycle de vie des applications de Nutanix.

3 Satyam Vaghani votre vice-président en charge de l'IoT et de l'IA parlait aussi de simplifier la création des applications dans ce domaine.

C'est aussi un de nos objectifs. Pour mettre en œuvre de la reconnaissance faciale au niveau d'une caméra c'est compliqué. Notre idée est de permettre aux utilisateurs de développer des applications IoT et IA plus facilement sans qu'ils aient à rentrer en profondeur ou à maîtriser la technologie. Du coup nous faciliterons la mise en place des solutions, mais aussi leur déploiement à l'échelle avec nos solutions Nutanix. C'est ce qu'a fait notre partenaire Hardis Group avec ses solutions dédiées à l'optimisation de la supply chain, notamment avec la reconnaissance cognitive dans les entrepôts.

SQLI DIGITAL EXPERIENCE

Mélanie GUILLOT - *Directrice artistique. Mon rôle au sein de SQLI est d'assurer la direction artistique sur divers projets digitaux : interface applicative métier, site vitrine, application mobile... J'accompagne aussi nos clients sur des plans de communication avec la réalisation de supports vidéo et autres. Je travaille également en étroite collaboration avec nos équipes de développement, nos experts SEO, chef de projets etc.*

Héloïse BLIN DRAY - *Ingénieur en conception et développement. Mon rôle au sein de SQLI consiste à réaliser techniquement des sites et des applications web. Spécialisée en développement front-end, j'apporte des solutions techniques adaptées aux besoins de nos clients à travers des interfaces graphiques réalisées par nos équipes de création.*

Comment l'étroite collaboration entre le développement et les expertises UX/UI améliore-t-elle la qualité d'un projet ?

La qualité d'une interface repose sur plusieurs critères au-delà des exigences techniques. Elle doit répondre à un besoin utilisateur précis par le biais d'un design ergonomique, facilement utilisable et manipulable. Donner du sens graphiquement à cette ergonomie permet de minimiser l'effort cognitif et d'avoir un impact positif sur l'expérience utilisateur. Dans une équipe digitale, il est essentiel que chaque métier soit impliqué de manière forte afin de répondre à ces objectifs. Une forte cohésion entre les équipes UX/UI et l'équipe de développement permet de s'aligner autour des mêmes enjeux de production.

Pour illustrer notre propos, nous faisons un retour d'expérience sur le développement d'une application métier à destination des conseillers de vente d'une maison de joaillerie. Leur processus de vente mondiale, alors complexe et peu optimisé, avait besoin d'être totalement revu et retravaillé. L'objectif a été de le transposer au travers d'une interface facile à utiliser et auto-apprenante.

Gestion de projet : le choix de l'agilité

L'agilité est une approche de gestion de projet qui vient s'opposer à des approches plus traditionnelles (comme les méthodes de cycle en V ou en cascade). Elle place le commanditaire et le livrable au centre de la conception et de la production technique et graphique. Cette approche permet d'éviter l'effet tunnel des autres méthodes dans lesquelles le commanditaire exprime et valide un besoin en amont du projet. On parle d'effet tunnel lorsque l'on a peu de visibilité en ce qui concerne l'avancement d'un projet et par conséquent qu'on ne peut pas rebondir pendant cette phase d'avancement.

Avec cette méthode, nous avons pu mettre en place une structure de projet propice à la collaboration car elle implique de faire évoluer le design et le développement en parallèle, afin de répondre au besoin évolutif.

Lors de chaque nouvelle étape du projet (un sprint en méthode agile Scrum), toute l'équipe se réunissait pour préparer la production de nouvelles fonctionnalités. L'objectif était de répondre à des problématiques de conception en ayant à la fois le point de vue de l'équipe de développement et de l'équipe de création. À l'issue de chaque sprint, d'une durée de 2 semaines, nous avons mis en place des rétrospectives permettant d'échanger sur les aspects positifs et négatifs survenus sur cette période. Cette manière de procéder s'inscrit dans un des principes essentiels de la philosophie agile : l'amélioration continue.

Un objectif commun : la qualité

Dès le début du projet, nous avions l'ambition de créer une interface très qualitative. Tout d'abord parce qu'il fallait rester dans les codes du luxe empruntés par la marque et les conseillers de vente, ces derniers étant nos utilisateurs cibles. Il fallait donc réaliser une interface claire et facile à utiliser, tout en y insufflant un univers luxueux.

Cette ambition a été gérée de plusieurs manières. Il y a d'abord eu une réunion de présentation du concept et de l'univers graphique à tous les intervenants du projet. L'idée était de s'aligner à la fois sur notre ambition en termes de design, mais également au niveau du design d'interaction. Le design d'interaction permet de rythmer l'enchaînement des contenus et de rendre plus attrayant le déroulé des pages en valorisant les contenus et messages clés. Il offre également une logique de guide visuel et confère à l'utilisateur le sentiment d'une expérience qualitative et sans couture.

Pour répondre techniquement à ce besoin, nous avons utilisé la bibliothèque **GreenSock (GSAP)**. Notre choix s'est porté sur cet outil car il permet de travailler sur des mouvements complexes tout en conservant des performances optimales et en étant compatible avec tous les navigateurs récents. Il permet de séquencer les mouvements, ce qui génère un rendu naturel et fluide. Nous avons utilisé cette bibliothèque avec **Angular 7**.

Design collaboratif

Au niveau du design, nous avions également à cœur d'utiliser des outils collaboratifs afin de faciliter le travail de chacun. Pour cela nous avons réalisé l'intégralité de notre design sous **Sketch**. Ce logiciel maintenant devenu incontournable possède bon nombre de plugins qui donnent la possibilité de faire des passerelles entre les designers et les développeurs.

Concernant notre projet, nous avons utilisé **MarvelApp** qui, pluggé à Sketch permet de visualiser via un mode « handoff » le détail de chaque composant de l'écran. L'utilisation de MarvelApp nous a fait gagner beaucoup de temps au niveau du

développement front sur tous les détails du type tailles des éléments, margin, couleurs, typographie utilisée et ses variations de graisses qui sont très accessibles. Cela nous a également fait gagner du temps côté design car les phases de recette ont été vraiment minimes. Nous avons également réalisé de nombreux tests, parfois en design, parfois en développement front, afin de répondre aux demandes de chacun et d'aboutir à un résultat à la hauteur de nos ambitions.

Stack technique : le développement modulaire

Atomic design

L'atomic design est une approche modulaire qui permet d'appréhender une nouvelle manière de concevoir des interfaces d'application. Cette approche s'oppose aux processus de création linéaires, et permet de penser l'interface comme un ensemble cohérent d'éléments. Le découpage de chaque élément d'une interface permet de les assembler dans un contexte et de leur donner du sens.
Afin d'optimiser la qualité de notre production, nous avons fait le choix de cette approche car elle satisfait notre besoin de développer notre interface comme un ensemble graphique cohérent. En adéquation avec la méthode agile, l'atomic design a permis une évolution constante de l'interface en réponse à l'ajout de chaque nouvelle fonctionnalité. Nous avons ainsi développé une application visuellement constante et homogène.

Développement en modules

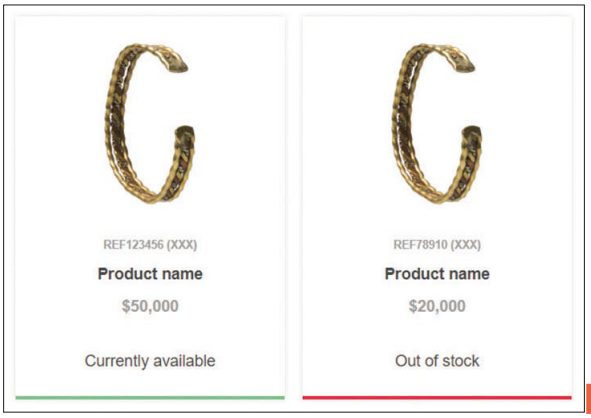
Les **frameworks JavaScript** comme React.js (2013), Vue.js (2014) et Angular 2+ (2016) ont permis l'évolution de nos manières de structurer le développement front-end des interfaces. Ils proposent une approche par composant qui conçoit l'application comme un ensemble d'éléments réutilisables. C'est ce concept, similaire à celui du design moléculaire, qui nous a permis de faire cohabiter Angular 7 et l'atomic design dans notre stack technique.

Un exemple en pratique

Cette manière de découper l'interface en différentes particules a permis d'isoler des éléments récurrents de notre application afin de les rendre génériques et de les réutiliser dans différents contextes.
Concrètement, en prenant l'exemple d'une carte produit, on constate que c'est un élément que l'on retrouve dans le catalogue mais également dans des écrans de wishlists, de gestion d'événements, de commandes, etc.
Cette carte devient alors un composant que l'on contextualise en fonction du besoin tout en conservant le même design. 1

La synergie avant tout

La synergie entre les équipes passe d'abord par un bon état d'esprit et surtout par une bonne communication ! Malgré un projet assez complexe et multi-acteurs, le fait d'avoir été identifiées comme les interlocutrices principales pour nos équipes, a rendu les échanges fluides et efficaces sur les différentes problématiques.
Au-delà des rituels principaux liés à la méthode Agile, nous avons créé un slack dédié au projet afin d'avoir un espace commun où échanger. Cela nous permettait à la fois de demander rapidement des sources comme des pictogrammes par exemple, mais également de prévoir des points de synchronisation en réel. Nous faisions également des points recette design/intégration au fur et à mesure de l'intégration des écrans afin de toujours garder un œil sur la qualité du livrable.
La synergie entre les designers et les développeurs est très souvent délaissée lorsqu'il s'agit de constituer un environnement de projet. C'est pourtant la réponse à de nombreuses problématiques de qualité et de productivité. En mettant en place différentes méthodes afin d'impliquer chaque domaine d'expertise tout au long du développement, chacune des compétences sont mutualisées, et ce au service de la qualité.



1

```
<app-card
    #productCard
    *ngFor="let product of products"
    class="m-card"
    [product]="product"
></app-card>
```

```
.m-card {
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    width: calc(100% / 3 - #{$gutter});
    min-width: 240px;
    margin: $gutter;
    box-shadow: 0 0 6px $grey-shadow;

    &-img-container {
        position: relative;
        overflow: hidden;
        width: 100%;
        height: 162px;
        margin: 15px;

        .a-img {
            position: absolute;
            top: 50%;
            left: 50%;
            transform: translate(-50%, -50%);
            max-height: 100%;
            max-width: 100%;
        }
    }

    .a-text {
        margin: 6px 0;
    }

    .m-status {
        width: 100%;
    }
}
```

```
.a-text {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 0.88em;
    color: $black;

    &.grey {
        color: $grey;
    }

    &.bold {
        font-weight: bold;
    }

    &.small {
        font-size: 0.63em;
    }
}
```

Christophe PICHAUD
Architecte Microsoft chez Infeeny
christophepichaud@hotmail.com
www.windowscpp.com

Le monde du développement selon Microsoft

Si vous voulez construire un système d'information, il faut choisir une technologie et naturellement vous allez vous tourner vers Windows, Linux, NET, Java, PHP ou Node.JS et JavaScript : c'est la tendance du moment. Si vous avez de l'expérience dans le monde Windows, vous serez tenté de partir sur la technologie NET. C'est un bon choix qui a fait ses preuves.

Le développement selon Microsoft

Microsoft envoie des messages parfois complexes sur le développement. En voici les fondamentaux :

- La technologie est le Microsoft .NET Framework
- L'environnement de développement est Visual Studio
- Le langage phare est C#
- Le développement mobile se fait avec Xamarin
- Les API pour le développement Desktop sont WinForms et WPF
- Les API pour l'accès aux données sont ADO.NET et Entity Framework
- La technologie web est ASP.NET MVC et ASP.NET web API
- Microsoft décline .NET en .NET Core et ASP.NET Core pour Windows, Linux et macOS
- Le futur du développement est le Cloud Azure et l'intégration des services managés dans les applications

L'exemple officiel, eShopOnContainers

eShopOnContainers est l'exemple officiel du monde du développement Microsoft. Il met en œuvre une application Mobile, une application web, une app PWA, une application ASP.NET MVC, des web API et des micro-services et une API Gateway. La communication entre les micro-services est assurée par Event Bus ou RabbitMQ, au choix.

L'envers du décor

Le développement *from scratch* est plutôt évident à comprendre, mais que faire des dizaines d'applications existantes ? Doit-on les réécrire (scénario impossible) ou les adapter.

Que devons-nous faire de nos vieilles applications WinForms, SilverLight (RIA ou Rich Internet Applications) ? Dans certains scénarios, on va fabriquer des API au-dessus des systèmes existants. Des API REST pour que tous les langages puissent cohabiter. Le JSON est moins verbeux que le XML ; c'est du pareil au même dans la philosophie. L'ouverture aux multiples langages est la tendance actuelle. La révolution JavaScript est passée par là, à la fois côté client et côté serveur avec NodeJS.

Application web ou application Desktop ?

Il est à la mode de faire des applications web et même full JS/TS web avec Angular, React ou Vue.JS. Ce n'est pas la panacée, car ces applications sont longues et coûteuses à développer pour une complexité exponentielle. Le langage TS utilisé par Angular est OOP, mais les développeurs font beaucoup de copier/coller et rendent ce code spaghetti. C'est le principal problème des technologies JavaScript. À titre personnel, j'émets des réserves sur les applications full JavaScript, car elles sont difficilement maintenables et truffées de copier/coller. De plus, dès que le back-end web API change, c'est souvent la galère pour répercuter les divers changements. Pour finir, Angular est d'une complexité délirante et le coût en formation est non négligeable pour une technologie qui « bouge » tous les ans. Je préfère de loin les applications web Server de type ASP.NET MVC qui permettent de séparer le code et les pages dans des modules distincts et qui ne sont pas des amas de code statique. ASP.NET MVC est beaucoup plus puissant que les applications de type Angular/React et n'adresse pas les mêmes problèmes. Une autre solution est de réaliser des applications Desktop WinForms ou WPF. La technologie WinForms n'est pas dépréciée, n'en déplaise à certains, car elle est rapide. WPF est toujours présent. UWP existe toujours malgré le faible taux d'adoption. Il y a moins de 2% des applications faites en UWP (source Télémétrie Visual Studio de Microsoft). Un détail qui a son importance. Pour tout vous dire, il n'y a que Microsoft qui fait des applications UWP, et encore, c'est C++, comme le Windows Terminal ou les accessoires (Calc, Paint3D, Courrier, etc.).

La révolution XAML avec WinUI

Malgré le fait que Windows n'utilise aucune de ses deux technologies WinForms et WPF, il existe un nouveau venu dans le domaine qui se nomme WinUI qui représente les contrôles XAML natifs de Windows. Il est fortement conseillé par Microsoft d'utiliser ces contrôles natifs dans les applications WinForms ou WPF ou même C++. Le style minimaliste est de mise. L'ancien nom Métro vous rappelle quelque chose ?

Microsoft abandonne progressivement WPF, car trop lent, trop lourd et le modèle WPF est complexe pour un ratio élégance/productivité très limité. Si vous le connaissez déjà, vous ne perdez pas vos connaissances XAML, mais vous ferez du WinUI. XAML n'est plus associé à Silverlight, RIA ou WPF, mais à Windows.

La revanche de Windows sur NET/WPF

C'est encore la guerre Windows Division / Developer Division qui se traduit avec WinUI. Le traumatisme de Longhorn est toujours là. Cette initiative qui consistait à vouloir introduire du C#/NET dans Windows avec WinFS, un système de fichier orienté objet, Avalon (WPF) et WCF (Windows Communication Foundation) et WF (Windows Workflow). Microsoft a décidé de terminer cette expérience et n'a

jamais utilisé de C#/NET dans Windows. Le seul module dans Windows qui utilise du .NET est une partie du serveur web IIS pour la gestion ASP.NET. Sinon rien du tout ! Sur 4000 dlls dans System32, tout est natif. Mieux, le code a évolué du C++03 au C++ Moderne et les modules Shell standard des accessoires et du control panel sont faits en XAML avec les contrôles natifs, mais toujours en C++.

WinUI 3.0

Revenons à WinUI, voici la roadmap disponible sur GitHub : https://github.com/microsoft/microsoft-ui-xaml/blob/master/docs/roadmap.md

WinUI 3 étendra considérablement la portée de WinUI pour inclure la plateforme d'interface utilisateur native Windows 10 complète, qui sera désormais entièrement découplée du SDK UWP.

Nous nous concentrons sur l'activation de trois cas d'utilisation principaux :

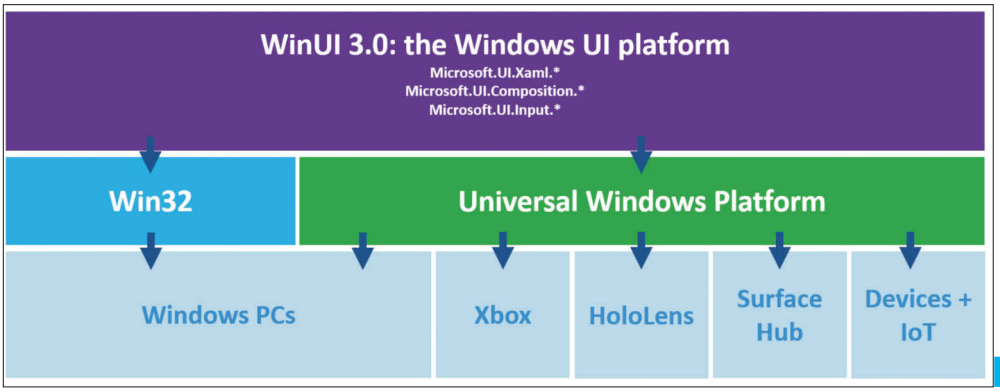
- Modernisation des applications Win32 existantes :
  Vous permettant d'étendre les applications Win32 (WPF, WinForms, MFC ..) existantes avec une interface utilisateur Windows 10 moderne à votre rythme en utilisant la dernière version à venir de Xaml Islands
- Création de nouvelles applications Windows :
  Vous permettant de créer facilement de nouvelles applications Windows modernes "à la carte" avec votre choix de modèle d'application (Win32 ou UWP) et de langue (.NET Core ou C ++)
- Activation d'autres Frameworks :
  Fournir l'implémentation native pour d'autres frameworks comme React Native lors de l'exécution sur Windows.
  WinUI 3.0 est en release Alpha (novembre 2019)

Présentation conceptuelle de WinUI 3 1

Plateforme WinUI 3

Vous pouvez regarder la session de conférence Build 2019 State of the Union : The Windows Presentation Platform. Pour plus de détails : https://mybuild.techcommunity.microsoft.com/sessions/77008

Les API UWP Xaml existantes livrées avec le système d'exploitation ne recevront plus de nouvelles mises à jour de fonctionnalités. Elles recevront toujours des mises à jour de sécurité et des correctifs critiques en fonction du cycle de vie de la prise en charge de Windows 10. La plateforme Windows universelle contient plus que le Framework Xaml (par exemple, le modèle d'application et de sécurité, le pipeline de médias, les intégrations de shell Xbox et Windows 10, la prise en charge étendue des appareils) et continuera d'évoluer. Toutes les nouvelles fonctionnalités Xaml seront développées et livrées à la place dans WinUI.



1

Avantages de WinUI 3

WinUI 3 offrira un certain nombre d'avantages par rapport au Framework UWP Xaml actuel, WPF, WinForms et MFC, ce qui fera de WinUI le meilleur moyen de créer l'interface utilisateur de l'application Windows :

1° La plateforme d'interface utilisateur native de Windows

WinUI est la plateforme d'interface utilisateur native hautement optimisée utilisée pour créer Windows lui-même, désormais plus largement disponible pour tous les développeurs à utiliser pour atteindre Windows. Il s'agit d'une plateforme d'interface utilisateur entièrement testée et éprouvée qui alimente l'environnement du système d'exploitation et les expériences essentielles de plus de 800 millions de PC Windows 10, XBox One, HoloLens, Surface Hub et d'autres appareils.

2° La dernière conception Fluent

WinUI est l'objectif principal de Microsoft pour l'interface utilisateur et les contrôles Windows natifs et accessibles et est la source définitive pour le Fluent Design System sur Windows. Il prendra également en charge les dernières innovations de composition et de rendu de niveau inférieur telles que les animations vectorielles, les effets, les ombres et l'éclairage.

3° Développement de bureau "à la carte" plus facile

WinUI 3 vous permettra de mélanger plus facilement et de faire correspondre la bonne combinaison de :

- Langage : .NET (C #, Visual Basic), C ++
- Modèle d'application: UWP, Win32
- Emballage : MSIX, AppX pour le Microsoft Store, non emballé
- Interop : utilisez WinUI 3 pour étendre les applications WPF, WinForms et MFC existantes avec une interface utilisateur Fluent moderne

4° Compatibilité descendante pour les nouvelles fonctionnalités

Les nouvelles fonctionnalités WinUI continueront d'être rétro-compatibles avec un large éventail de versions de Windows. Vous pouvez commencer à créer et à envoyer des applications avec de nouvelles fonctionnalités dès qu'elles sont publiées, sans avoir à attendre que vos utilisateurs mettent à jour Windows.

5° Prise en charge du développement natif

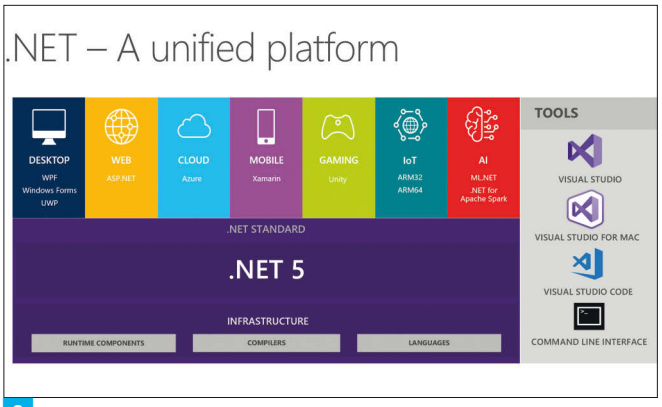
WinUI peut être utilisé avec .NET, mais ne dépend pas de .NET : WinUI est 100% C ++ et peut être utilisé dans des applications Windows non gérées, par exemple en utilisant le standard C ++ 17 via C ++ / WinRT.

6° Mises à jour plus fréquentes

WinUI continuera à livrer de nouvelles versions stables 3 fois par an, avec des versions préliminaires mensuelles.

7° Développement open source et engagement communautaire

WinUI continuera d'être développé en tant que projet open source sur GitHub. WinUI 2 est déjà open source dans le dépôt, et il est prévu d'ajouter le framework WinUI 3 Xaml complet.



2

Vous pouvez vous engager directement avec l'équipe d'ingénierie principale de Microsoft et contribuer aux rapports de bogues, aux idées de fonctionnalités et même au code : consultez le Guide de contribution pour plus d'informations. Vous pouvez également essayer les versions préliminaires mensuelles pour voir les nouvelles fonctionnalités en cours de développement et aider à façonner leur forme finale.

8° Une cible Windows native pour les frameworks web et multiplateforme

WinUI 3 est mieux optimisé pour les bibliothèques et les frameworks sur lesquels s'appuyer.

Par exemple, il est prévu de baser la nouvelle implémentation C ++ React Native Windows hautes performances sur WinUI 3.

La roadmap .NET 5

On nous annonce NET 5 pour fin 2020. C'est la fusion entre .NET Framework 4.8 et .NET Core 3.x. Pourquoi pas. Avec en plus, le multiplateforme sous Linux, Mac et Windows et pour le mobile sous Android et iOS. Mono permettant toujours de faire du GTK sous Gnome. NET 5 est avant tout une plateforme unifiée. On y trouve WinRT qui est disponible depuis Windows 8, les compilateurs Roselyn C# et VB.NET, les langages C#, VB.NET et F# et C++/CLI et on y trouve plusieurs domaines :

- Le desktop avec WPF, WinForms et UWP
- Le web avec ASP.NET
- Le cloud avec Azure
- Le mobile avec Xamarin
- Le gaming avec Unity
- L'IoT
- L'AI avec ML.NET

On dispose des outils Visual Studio (Windows et Mac) et Visual Studio Code. Visual Studio est l'outil dans lequel on fait tout. Si vous avez la possibilité, passez à la version Pro (ou Enterprise). 2

L'accès aux données

L'accès aux données selon Microsoft a été pendant 20 ans MDAC (Microsoft Data Access Components) composé de ADO, RDO, OLEDB et ODBC. Avec .NET, on a vu apparaître ADO.NET puis Entity Framework (EF). EF est un ORM : attention. Cela peut être une catastrophe. Les requêtes générées sont complexes et parfois horribles. C'est un ORM et donc comme tout ORM, c'est une technologie client qui essaie de s'affranchir du SQL, ce qui est une hérésie. Les jeunes développeurs pensent que c'est cool, car ils ne font que du C# et pas de SQL. Le résultat est souvent décevant et catastrophique. Anecdote de terrain : une entreprise a failli mettre la clé sous la porte après avoir fait son SI en ASP.NET/EF, car les pages mettaient 10 secondes à s'afficher. Un audit a été réalisé et les requêtes EF étaient fautives. Le DSI a été viré et les développeurs licenciés. De nombreux architectes bannissent l'emploi de EF sur les projets et proposent plutôt de l'ADO.NET avec Dapper, une librairie open source fortement typée. On conseillera l'utilisation traditionnelle des projets SQL Server ou Oracle avec des traitements serveur… SQL ou procédures stockées. Amis développeurs, apprenez le SQL, c'est important. Linq to SQL c'est bien quand il y a dix enregistrements qui se battent en duel dans une table sinon ça ramène toute la base en local et les ingénieurs systèmes vont vous détester.

Le monde web ASP.NET

ASP.NET Core introduit de nouveaux paradigmes. En effet, la plateforme NET Core tourne sous Windows, Linux et macOS. Quand on dit Linux, cela ouvre le champ des serveurs Nginx Alpine, CentOS, RHEL, Debian, etc. Bref, le web version « Microsoft seulement sur IIS » est révolu.

C'est un sacré pavé dans la marre pour les développeurs Windows. Il va falloir installer Windows Services for Linux v2 (WSL2) sous Windows 10 au mieux, avoir une seconde machine sous Linux ou bien jouer avec des containers Docker. Un développeur a besoin de confort donc je serais vous, je préparerais moralement votre patron à vous payer un second laptop… C'est indispensable.

Cela implique que les développeurs Windows doivent maintenant apprendre Linux (le bash, TCP/IP, la sécurité, etc.). Celui qui reste que sous Windows sera dans une voie de garage à terme. Le Back-end sera de plus en plus sous Linux, car plus léger que Windows Server. SQL Server existe sous Linux donc vous ne serez pas perdu. Amis développeurs, faites votre plan de formations et avancez vos pions. Ne passez pas à côté de ce virage.

Côté nouveautés, ASP.NET Core introduit web API Core, ASP.NET MVC Core et gRPC ainsi que les briques pour l'authentification. ASP.NET Core & MVC Core sont des technologies d'avenir pour le développeur Microsoft. Vous pouvez miser dessus et vous certifier.

L'aspect économique

Ne vous y trompez pas, les bijoux de Microsoft ce sont Windows, Office 365 et Azure. Tout est bon pour vous faire venir sur Azure. On vous parle de VM Linux, de containers, de Docker, de Kubernetes et du support de beaucoup de technologies Open-Source. La réalité c'est que vous venez comme vous êtes (comme au Mc Do) et que vous allez un ou l'autre utiliser et consommer de l'Azure ! C'est une question de temps. Le virage technologique est enclenché. Le Cloud s'impose.

La modernisation des applications

Il est un domaine dans lequel les entreprises de services se sont engouffrées : la modernisation des applications. Sous ce nom exotique, on trouve la mise en œuvre de technologies Cloud comme la télémétrie, le stockage et l'appel à des services managés de type web API en REST. Une application Desktop devient connectée au service de l'entreprise, mais aussi à Azure pour son monitoring, son logging et utiliser de plus en plus de services.

Cela passera par du middleware comme Service Bus ou RabbitMQ, SQL Azure ou peut-être Service Fabric ou même de la technologie Serverless comme Azure Functions. Il paraît que 2020 est l'année du Serverless… À suivre.

Microservices, Docker et Kubernetes

Les architectures microservices sont l'avenir du développement en entreprises. Les microservices sont indépendants, autosuffisants et possèdent leur base SQL ou NoSQL et communiquent via un Bus de Message comme Azure Service Bus ou RabbitMQ. Les microservices ouvrent la voie à l'utilisation des plateformes d'Api Managements ou Gateways d'API et de cluster de pods Docker avec un orchestrateur comme Kubernetes. L'environnement Azure propose AKS et Microsoft fait l'éloge de AKS et Kubernetes matin, midi et soir sauf que k8s (son petit nom) ne tourne pas sous Windows, mais sous Linux… Si vous voulez apprendre k8s, prenez un PC sous Linux Ubuntu 19.10 et installez Docker et MicroK8s pour maîtriser la bête. C'est gratuit et ça fait 2x30 MB et c'est gratuit.
Je vous donne rendez-vous le mois prochain pour un article spécial k8s sous Linux.

Azure Hybrid

Avec les scénarios d'entreprise hybrides, Microsoft nous prépare Azure Arc qui consiste à mettre le monde du développement et de l'infrastructure dans un rôle hybride. Les connexions se font *on-premises* et sur le Cloud, voire chez différents vendeurs de Cloud. Les outils de développement évolueront, c'est certain et Visual Studio aura de nouvelles fonctionnalités à n'en pas douter. 3

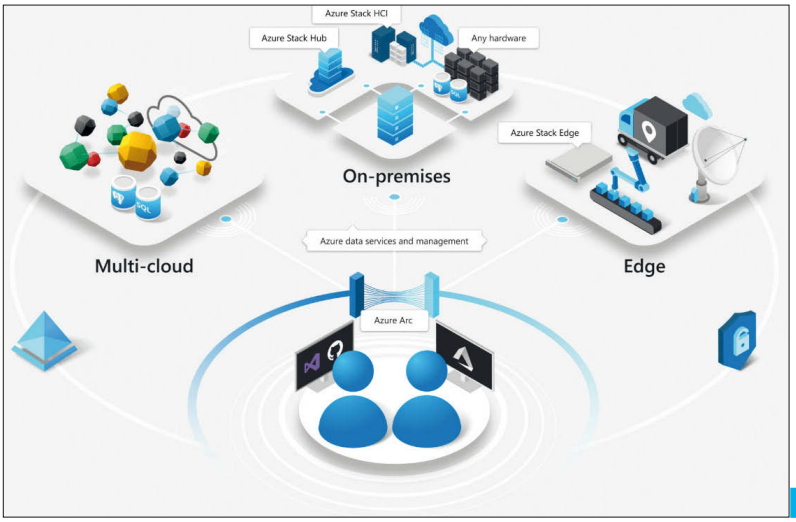
Conclusion

Le développement selon Microsoft c'est la plateforme .NET, Azure et Windows. Ce qui est troublant c'est que Microsoft fait 95% de ses produits en C++ et rien en .NET à part Visual Studio et ses composants NE…
Donc là où Microsoft il y a 20 ans faisait du Dog-fooding en construisant des SDK et en les utilisant dans ses produits, on est passé à l'ère du Marketing où l'on pousse des technologies que l'on n'utilise pas en interne ou si peu.
Est-ce bien raisonnable ? Oui la technologie est fiable, mais Windows ne contient aucun composant NET à part la redistribution du Framework dans le répertoire Windows. À quand le grand saut ?
Pas pour si tôt. En tant que fervent adepte du natif, je souhaite que Windows continue de s'ouvrir avec les API WinRT pour laisser le choix aux développeurs d'utiliser le langage de leur choix.
Si NET pouvait être modifié pour que les modules soient en mode back-end natif, cela permettrait de clore de nombreux débats.
C'est le modèle de Go, Apple avec Objective-C, Rust, etc. Ils ont des back-end LLVM et c'est du natif. Si le C#/NET pouvait faire du natif avec autre chose qu'une gestion de la mémoire via un Garbage Collector, je suis certain que Microsoft Windows Core s'ouvrirait à C#/NET. C'est un souhait personnel, mais qui a peu de chance d'aboutir, car cela casse l'existant. Pourtant cela irait dans le sens des autres acteurs du marché. Microsoft n'invente rien : il suit et s'adapte. Le message est passé.

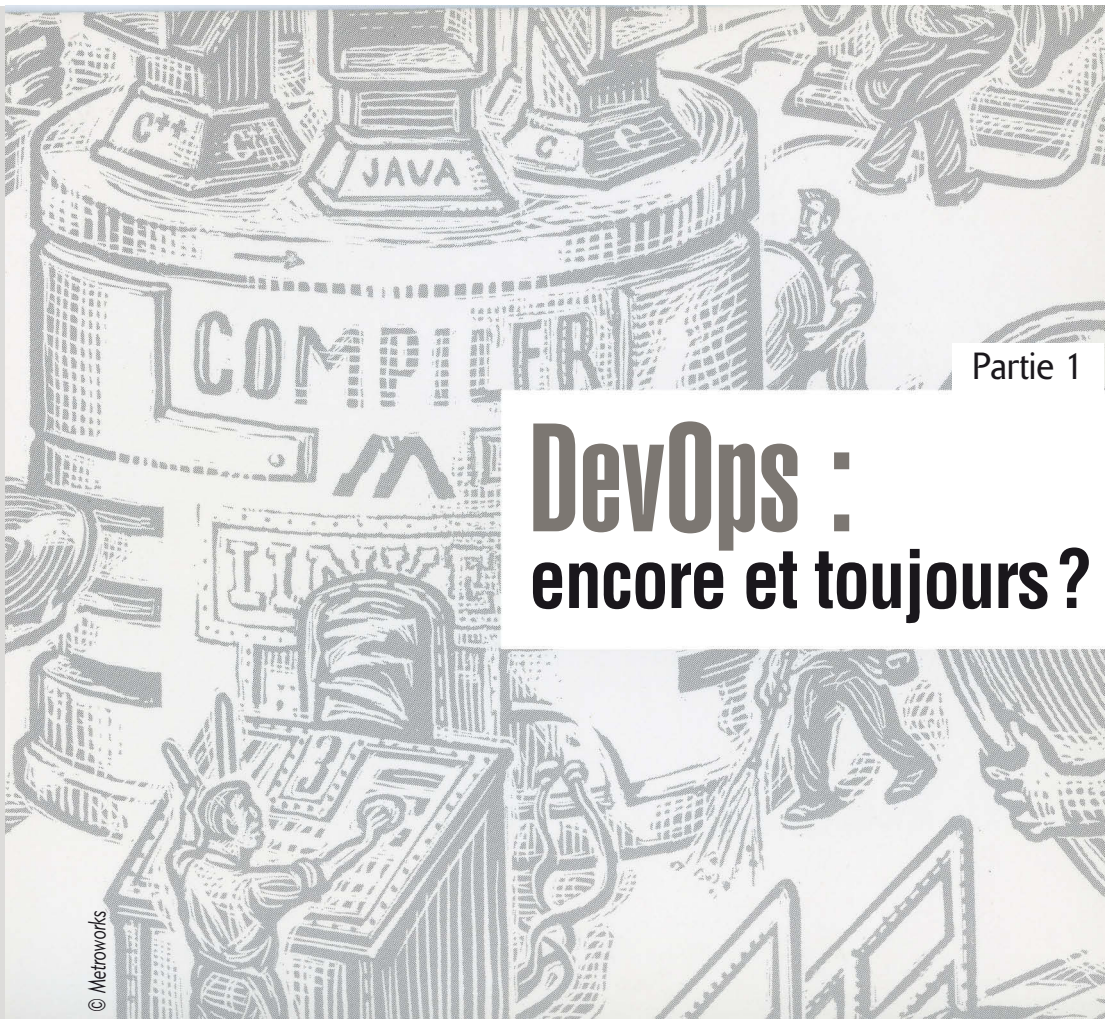
Timothé LARIVIERE | *Tech Lead chez Infeeny*
timothe.lariviere@infeeny.com | timothelariviere.com

LE DÉVELOPPEMENT MOBILE AVEC XAMARIN

La promesse de Xamarin est alléchante : réutiliser vos compétences .NET pour réaliser des applications mobiles en C# (ou F#) et en partageant une majeure partie du code entre les plateformes, possible grâce à Mono qui est disponible sur les OS d'Apple et Linux ! Cela donne un avantage certain face au développement natif. Vous n'êtes pas contraint de redévelopper vos applications plusieurs fois pour les différentes plateformes (Android + Java, iOS + Swift) avec tous les risques que cela implique (bugs, pas équivalent niveau fonctionnalités, etc.).
Vous pouvez partager votre code dans des DLLs (.NET Standard 2.0 par ex.) et bénéficier de tout l'existant .NET (notamment NuGet) pour vous faciliter la vie, réduire les temps de développement et le nombre de développeurs nécessaires.
Cependant, cela ne vous dispense pas d'apprendre les spécificités des plateformes que vous ciblez.
Si vous faites du Xamarin dit natif, vous aurez à créer les interfaces graphiques en utilisant la manière de faire de la plateforme (Activity / XML, UIController / Storyboard, etc.). Il faudra alors du temps aux équipes pour se former, car il n'est pas aisé d'apprendre une nouvelle plateforme.
Xamarin.Forms est un peu plus flexible sur ce point. Le développement des interfaces graphiques se faisant à travers un XAML très semblable à WPF et UWP. Xamarin.Forms se chargera lui-même de faire la correspondance entre le XAML et les contrôles natifs de la plateforme au runtime.
Le pourcentage de code partagé variera considérablement selon le type d'application que vous faites. Une application orientée métier pourra maximiser son partage en centralisant les règles métiers. Une application B2C aura des besoins graphiques beaucoup plus importants, pas forcément partageables. Dans la pratique, cela sera plus de l'ordre de 60-40, assez loin des 80-20 promis par le marketing.
Pour conclure, le développement mobile avec Xamarin est très intéressant, mais il faut bien étudier vos besoins et vos équipes avant de décider de l'utiliser au lieu du développement natif.



3



© Metroworks

Partie 1

DevOps : encore et toujours ?

Encore DevOps ? Vous avez l'impression que l'on en parle trop ? Que vous connaissez déjà tout dessus ? Il faut dire que l'on a l'impression d'écrire toujours un peu la même chose. Et pourtant, la notion même de DevOps évolue par les outils, les architectures et les nouvelles pratiques comme le Kanban et le Craftmanship. Sans oublier la sécurité avec le DevSecOps.

Le Devops recouvre, au-delà de sa philosophie d'origine (de la préhistoire pour certains), plusieurs réalités dans les process, les méthodes de développement et de déploiement, la mise en production ainsi que les outils : des cycles de développement doublés d'un déploiement (beaucoup) plus fréquent, réduire les défauts de conceptions, amélioration (parfois drastiquement) de la collaboration entre les équipes, détecter les bugs et les régressions plus rapidement (notamment avec les notions de CI / CD).

Une des notions sous-jacentes du DevOps est l'automatisation : tout ce qui est automatisable, on l'automatise. Côté déploiement : comment avoir une souplesse de déploiement, comment automatiser et fiabiliser la pile de déploiement, mais aussi la plateforme d'exécution. Le déploiement continu est là pour aider, soutenu par l'Infrastructure as Code et les conteneurs.

Quelles tendances, ou plutôt confirmations, pour 2020 ?

Nous allons être d'une originalité folle :

- Serverless va continuer à s'étendre d'une manière ou d'une autre
- Les architectures micro-services : elles sont dans le paysage applicatif et serveur.
- Automatisation : oui, comme dit plus haut, l'automatisation est une des fondations du DevOps et la tendance ne va pas s'arrêter là. Bon OK, l'automatisation, il faut aussi la concevoir et la mettre en place
- Les Assembly Lines

Assembly Lines pour unifier ce chaos

Le DevOps tendance Assembly Lines n'est pas une nouveauté ni une révolution. On en parle ici et là depuis 3-4 ans. L'assembly lines est là pour automatiser et connecter les activités réalisées par x équipes, dans l'intégration continue pour les dévs, l'infrastructure et sa configuration pour les ops, l'automatisation des tests pour les testeurs, la sécurité pour le responsable sécurité, la gestion des versions, le déploiement sur x cibles, etc.

Quand on regarde la cartographie DevOps, nous avons de gros blocs et à l'intérieur les outils dédiés, le tout relié par les chaînes d'automatisation, de processus, etc. Mais on multiplie les outils et les environnements. Bref, on fait un peu l'inverse de l'idée de départ : on fragmente au lieu d'unifier. Vous voyez l'ironie ? Un peu comme les approches ALM et la multitude de briques logicielles.

Bref, Assembly Lines est là pour faire la glue (et une glue qui tient bien) entre tout ça. On unifie les processus et les outils. On dit souvent que l'Assembly Lines est le pipeline des pipelines. Une des facilités promises par cette approche de haut niveau est d'être capable de traverser l'ensemble des pipelines, réutiliser les différents flux, être le plus agnostique possible, etc., etc. Bref, on réunit l'ensemble en tentant de fluidifier les usages entre les différents morceaux du DevOps.

Philippe Charrière
(pcharriere@gitlab.com),
Technical Account Manager chez GitLab

Partie 1

Cueillette des champignons avec GitLab CI

(ou comment avoir du feedback sur son code source avec un runner)

Dans cet article nous allons découvrir que l'on peut utiliser un GitLab runner installé sur son poste en local pour exécuter la CI d'un projet distant hébergé sur GitLab.com (donc sans utiliser les shared runners, mais vos propres ressources), nous verrons aussi comment utiliser ses propres outils dans une image Docker et se servir de l'API GitLab pour apporter du feedback au développeur sur d'éventuelles vulnérabilités présentes dans son code. Dans cet article les vulnérabilités seront des 🍄.

Prérequis :

- Avoir un compte sur GitLab.com (ou vous pouvez utiliser une version "self-hosted")
- Avoir un client Docker installé + git

Remarques :

- Tout ce qui est décrit dans cet article est reproductible avec un plan free sur GitLab.com ou avec une version core de GitLab "self-hosted"

GitLab runner : qu'est-ce que c'est ?

Le projet GitLab Runner est un projet open source qui est utilisé pour exécuter des jobs de CI sur vos projets et qui ensuite envoie les résultats à votre instance GitLab ou à GitLab.com. Il est utilisé avec GitLab CI, la fonctionnalité de GitLab qui permet de coordonner le fonctionnement du ou des runners. Le ou les runners peuvent être installés n'importe où: VM, Container, bare metal, Kube… et même votre propre laptop. 1

Le runner poll via HTTP/HTTPS votre repository et en cas de modification sur une branche (1)(2) il va aller lire le fichier .gitlab-ci.yml (2) présent à la racine de votre repository et exécuter les tâches décrites dans ce fichier (3)(4) comme du build, du test… et ensuite le ou les runners vont envoyer les résultats à l'instance GitLab. 2

Les runners peuvent être affectés à une instance (on parle de shared runners disponibles pour l'ensemble des projets), à un groupe ou à un projet (et bien sûr il peut y avoir plusieurs runners par groupe et par projets). Et les runners proposent différents types d'"executors" (Shell, Docker, VM,…) pour exécuter ces tâches (dans cet article nous utiliserons le Docker executor).

Le truc sympa, c'est que l'on peut facilement (et simplement) utiliser un runner localement sur son laptop qui va effectuer des tâches de CI pour un projet hébergé sur GitLab.com.

Je vous disais que l'on pouvait utiliser des "executors" docker, l'avantage c'est que vous pouvez utiliser une image avec les outils dont vous avez besoin pour votre CI (continuous integration), votre CD (continuous deployment) par exemple une CLI pour déployer sur un PaaS, curl pour utiliser une API,… **Et moi, j'aime bien utiliser NodeJS pour faire des traitements sur des fichiers. Préparons donc nos outils…**

Création (et 1re utilisation) d'une image docker

Note : vous pourrez retrouver mes images par ici https://gitlab.com/cook-books/have-fun-with-gitlab-ci/ci-docker-images

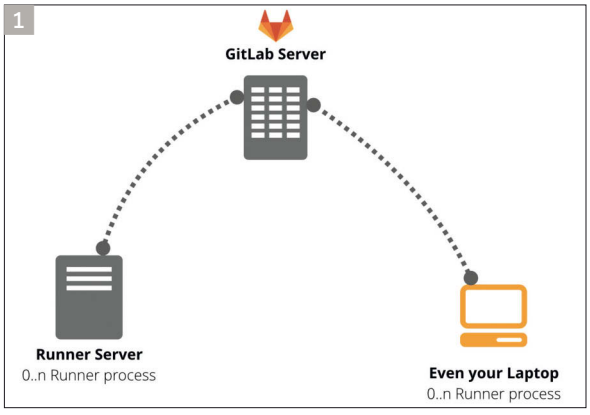
Aujourd'hui, pour faire les scripts de CI, j'aurai besoin de NodeJS et curl (de grep aussi, mais c'est intégré dans tout bon Linux qui se respecte), je me suis donc créé un Dockerfile tout simple:

```
# My little node ci tools
FROM node:12.7.0-alpine
LABEL maintainer="philippe.charriere@bots.garden"

RUN apk --update add --no-cache \
 bash curl git
```

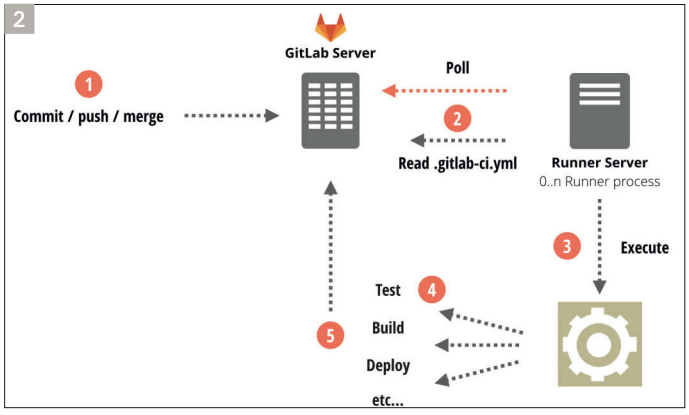
Et j'ai tout aussi simplement construit et publié mon image sur https://hub.docker.com/ de cette manière:

```
docker build --no-cache -t my-little-node-ci-tools .
docker tag my-little-node-ci-tools:latest k33g/my-little-node-ci-tools:latest
docker push k33g/my-little-node-ci-tools
```

1



2



Et vous pouvez me retrouver ici:
https://hub.docker.com/repository/docker/k33g/my-little-node-ci-tools/general

Testons notre image

Pour jouer avec l'image, il suffit de créer un container de cette façon:

```
docker run -it --name node-ci-tools k33g/my-little-node-ci-tools:latest /bin/sh
```

Ainsi vous arrivez sur un prompt qui vous permet de lancer différentes commandes (vérifions que nous avons bien nos outils):

```
/ # node --version
v12.7.0
/ # git --version
git version 2.20.1
/ #
```

Jouons un peu : la cueillette des champignons

Toujours dans le shell de notre container, tapez ceci pour créer un fichier note.txt:

```
/home # cat >> note.txt <<EOL
> hello I ♥ the 🍄
> this is a test
> oh oh another 🍄
> have a nice day
> EOL
```

Et enfin un 2e fichier README.md:

```
/home # cat >> README.md <<EOL
> ### this is a title
>
> I need to find some 🍄
> EOL
```

Note : la difficulté est d'ajouter les emojis, sous macOS c'est très facile, sous Ubuntu c'est natif avec les applications GTK, et sous Windows, j'avoue, je ne sais pas, mais je chercherais pour un prochain article.

Et maintenant, tapez cette commande **magique**:

```
/home # RESULTS=$(grep -rwni " 🍄 " * || echo "")
```

Qui en langage humain, signifie *"sois gentil, trouve-moi tous les champignons dans les fichiers et mets-moi tout ça dans une variable `RESULT`"*. Et donc vous obtiendrez ceci si vous faites un `echo $RESULTS`:

```
/home # echo $RESULTS
README.md:3:I need to find some 🍄 note.txt:1:hello I ♥ the 🍄 note.txt:3:oh oh another 🍄
```

Donc la liste des fichiers où l'on peut trouver des champignons et à quelle position dans le fichier (c'est à ce moment-là que j'ai commencé à aimer grep). Eh bien maintenant, je voudrais que ma CI me recherche tous les champignons vénéneux qu'il pourrait y avoir dans mon code source.
Voyons donc comment faire du **DevSecOps** avec des champignons

Tout d'abord il faut créer un projet sur GitLab.com

Note : *vous pouvez bien sûr faire cela avec une instance GitLab "self hosted"*

Vous allez créer un projet vide sur GitLab.com (le mien est ici https://gitlab.com/cook-books/have-fun-with-gitlab-ci/mushrooms-picking) et le cloner sur votre poste. Par exemple si vous voulez cloner le mien pour gagner du temps, vous faites:

```
git clone git@gitlab.com:cook-books/have-fun-with-gitlab-ci/mushrooms-picking.git
```

Et vous allez ajouter des fichiers dans votre projet avec des champignons 🍄 dedans: 3

Ensuite, installer (et tester) un GitLab Runner sur votre poste

C'est assez facile à mettre en œuvre:
Sous Mac: https://docs.gitlab.com/runner/install/osx.html

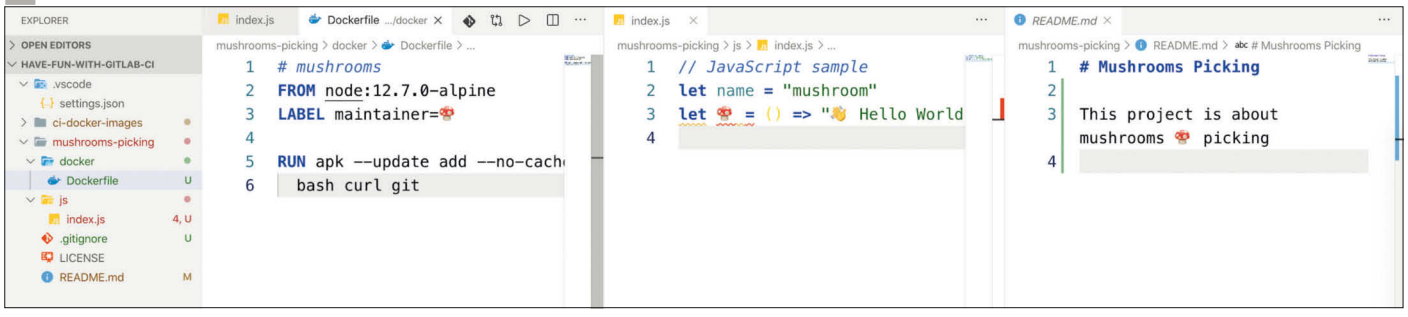
```
sudo curl --output /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amaz
onaws.com/latest/binaries/gitlab-runner-darwin-amd64
sudo chmod +x /usr/local/bin/gitlab-runner
cd ~
gitlab-runner install
gitlab-runner start
```

Sous Linux: https://docs.gitlab.com/runner/install/linux-manually.html

```
# Linux x86-64
sudo curl -L --output /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amaz
onaws.com/latest/binaries/gitlab-runner-linux-amd64
sudo chmod +x /usr/local/bin/gitlab-runner
sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash
sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner
sudo gitlab-runner start
```

Sous Windows: https://docs.gitlab.com/runner/install/windows.html
Pour la suite, il faut aussi que vous ayez un client Docker installé sur votre poste.

3



Un runner GitLab a besoin d'un livre de recettes pour fonctionner, et ces recettes, on les trouve dans le fichier .gitlab-ci.yml qui doit être présent à la racine de votre projet. (on parle aussi de **pipeline**)

Création du fichier .gitlab-ci.yml

Ajoutez à votre projet un fichier .gitlab-ci.yml (la recette de votre projet pour votre runner) avec le contenu suivant:

Note : *vous n'êtes pas obligés de "coller" des emojis dans le fichier yaml ou dans les commandes shell, mais je trouve que cela fait joli*

```
# Mushrooms Picking
stages:
 -   forest-trip

## My tools
.find-mushrooms: &find-mushrooms |

 function mushrooms_markdown_report() {
  RESULTS=$(grep -rwni " 🍄 " * || echo "") node --eval '
   let mushrooms = process.env.RESULTS.split("\n")
    .map(item => item.split(":"))
    .map(item => `- File: ${item[0]} (line ${item[1]}) Extract: \`${item[2].trim()}\``)
    .join("\n")
   let report = `### Mushrooms Picking \n${mushrooms}`
   console.log(report)
  ' > mushroom.report.md
 }

before_script:
 - *find-mushrooms

# gitlab-runner exec docker 🍄 mushrooms-picking
🍄 mushrooms-picking:
 stage:  🎄 forest-trip
 image: k33g/my-little-node-ci-tools:latest
 only:
  - merge_requests
  - master
 script: |
  mushrooms_markdown_report
  # Display the mushrooms report
  cat mushroom.report.md
 artifacts:
  paths:
  - mushroom.report.md
  expire_in: 1 week
```

Explications:

.find-mushrooms: &find-mushrooms me permet de définir un ensemble de fonctions shell que je pourrais réutiliser dans mes jobs. Par exemple la fonction **mushrooms_markdown_report** qui me permet d'exécuter un **grep** pour rechercher mes 🍄, coller le résultat dans une variable d'environnement **RESULTS** et de le retraiter grâce à NodeJS (**node --eval**) et de l'enregistrer dans un fichier markdown (**> mushroom.report.md**)

La section:

before_script:
- *find-mushrooms

me permet de mettre à disposition la fonction **mushrooms_markdown_report** pour l'ensemble de mes jobs

🍄 **mushrooms-picking**: c'est le job qui sera exécuté dans le pipeline et vous pouvez voir que dans sa section **script** j'appelle ma fonction **mushrooms_markdown_report**, ensuite je fais un **cat** du fichier généré pour afficher le résultat.

La section **artifacts** me permet de définir quels éléments générés je conserve comme artifacts.

Maintenant vous pouvez "commiter":

```
git add .
git commit -m " 🚧 add ci script"
```

"N'envoyez" pas tout de suite vers GitLab.com, il est en effet possible de tester notre script de CI localement puisque nous avons un runner installé en local (avec quelques limitations que vous trouverez ici: https://docs.gitlab.com/runner/commands/#limitations-of-gitlab-runner-exec)

Par exemple pour exécuter notre job de cueillette, exécutez la commande suivante :

```
gitlab-runner exec docker 🍄 mushrooms-picking
```

Et vous devriez voir s'afficher ceci (ou quelque chose de similaire):

```
### Mushrooms Picking
- File: README.md (line 3) Extract: `This project is about mushrooms 🍄 picking`
- File: docker/Dockerfile (line 3) Extract: `LABEL maintainer= 🍄 `
- File: js/index.js (line 3) Extract: `let 🍄 = () => " 👋 Hello World 🌍"`
```

Vous obtenez donc la liste des champignons et des endroits où vous pouvez les cueillir dans le code source.

Sympa, non ? Vous venez de créer **votre analyseur de vulnérabilité de recherche de champignons vénéneux** dans du code source.

"Raccrocher" notre runner à notre projet sur GitLab.com

Avant de tout "pousser" vers GitLab.com, nous allons enregistrer notre runner dans notre projet sur GitLab.com. Pour cela,

- Allez dans les "**Settings > CI/CD**" du projet.
- Sélectionnez "**Expand**" de la rubrique Runners
- Désactivez les shared runners en cliquant (à droite) sur "**Disable shared runners**"
- Puis notez les informations (à gauche) qui vous permettront d'enregistrer votre runner 4

Pour enregistrer votre runner, exécutez la commande suivante (sur votre laptop):

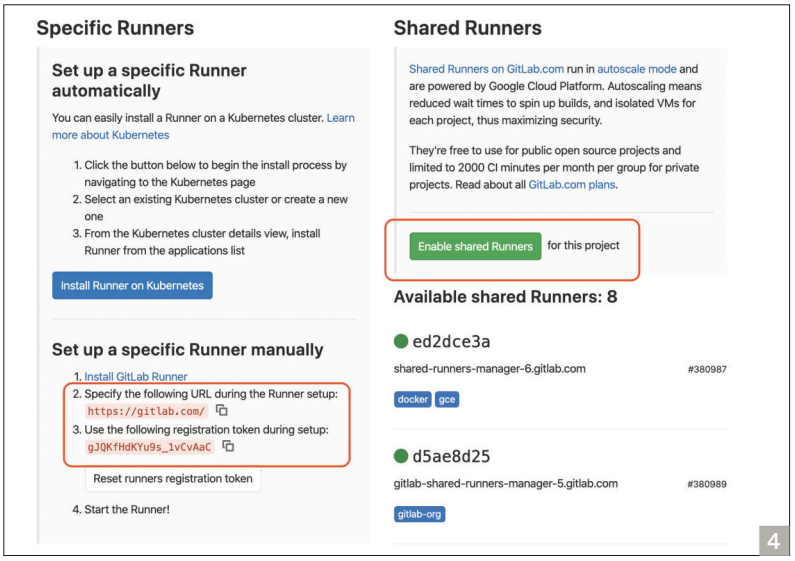
```
gitlab-runner register \
 --non-interactive \
 --registration-token gJQKfHdKYu9s_1vCvAaC \
 --locked=false \
 --description runner-of-k33g \
 --url https://gitlab.com/ \
 --executor docker \
 --docker-image docker:stable
```

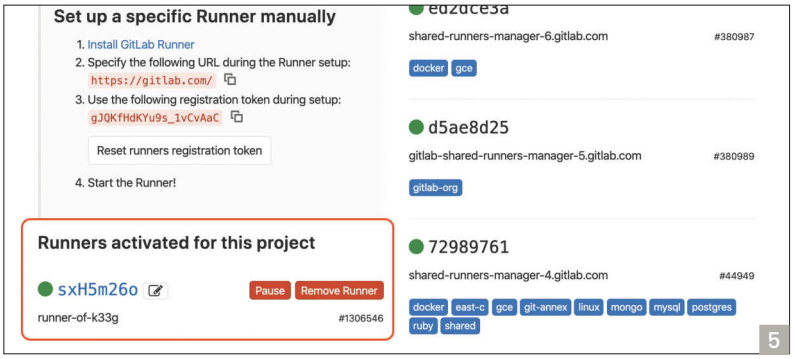
Dans votre terminal, vous devriez obtenir quelque chose comme ceci:

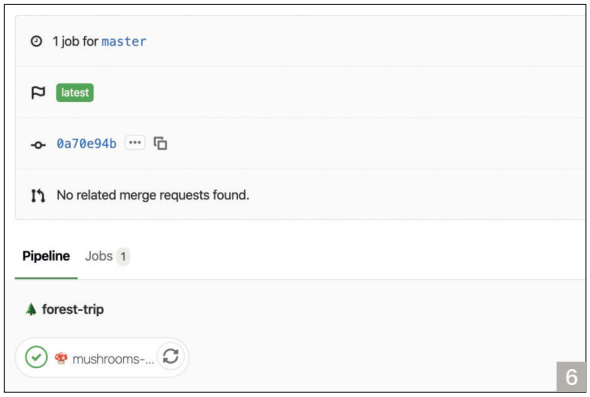
```
Runtime platform arch=amd64 os=darwin pid=60911 revision=05161b14 version=12.4.1
WARNING: Running in user-mode.
WARNING: Use sudo for system-mode:
WARNING: $ sudo gitlab-runner...

Registering runner... succeeded  runner=gJQKfHdK
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!
```

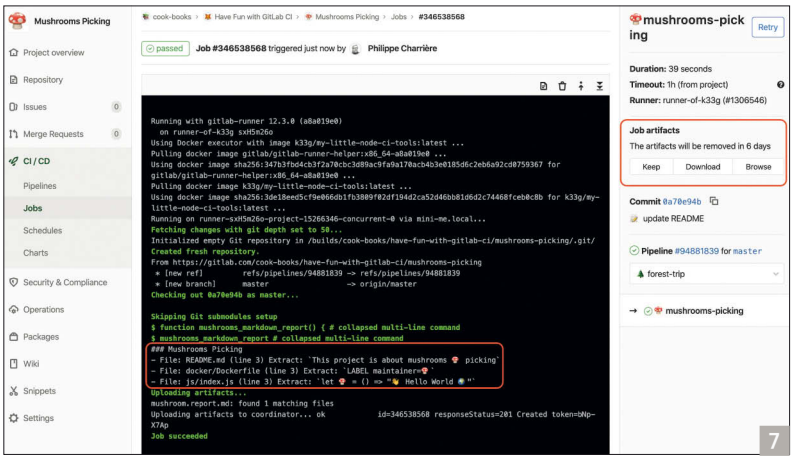
Et si vous actualisez votre page de settings, vous devriez voir apparaître votre runner: 5
Donc maintenant, vous pouvez "faire" un **git push** de votre projet.
Et vous pouvez découvrir qu'un pipeline a été exécuté: 6
Et si vous allez voir la sortie du job (cliquez sur le champignon) vous pouvez voir que le rapport a bien été généré (et à droite qu'il est accessible en téléchargement: cf **Job Artifacts**) 7
Vous pouvez donc télécharger l'artefact et le visualiser: 8
Maintenant, allons un peu plus loin. Ce qui est important pour un développeur, c'est le feedback sur son code, et c'est encore mieux si ce feedback est facilement accessible. Et pour cela, la **Merge Request** est un bon endroit (selon moi).

La suite dans le numéro 239









Le DevOps mainframe, clé de la satisfaction client

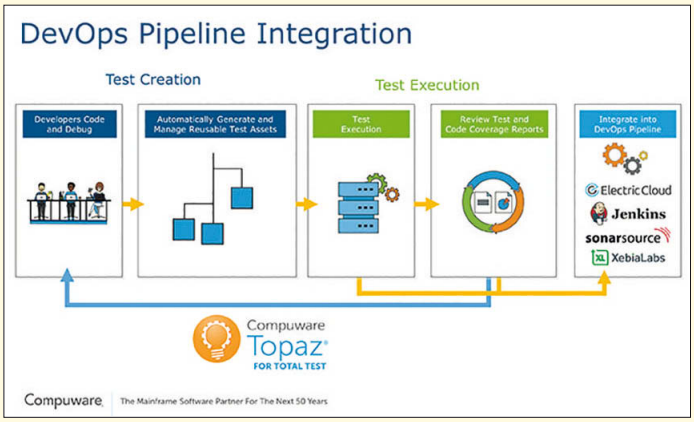
par Véronique Dufour-Thery, Vice President South Europe & MENA, Compuware

La transformation digitale et les nouveaux usages qui en découlent imposent aux départements informatiques d'optimiser les cycles de développement et de déploiement d'applications continus tout en intégrant des méthodes agiles dans les pratiques de développement. La culture DevOps a pour objectif d'améliorer la communication et la collaboration entre les équipes de développement et les opérations pour casser les silos. Cela permet à l'entreprise de développer sa capacité d'innovation tout en accélérant les cycles de livraison.

Au cœur du mainframe

Le mainframe, puissant système transactionnel dans les entreprises, intègre aujourd'hui les pratiques DevOps. Loin d'être en voie de disparition, les mainframes voient leur charge de travail augmenter partout dans le monde. 57% des entreprises dotées d'un environnement mainframe exécutent à l'heure actuelle plus de la moitié des applications stratégiques. De plus, 72% des systèmes d'engagement clients dépendent entièrement ou en grande partie de la puissance de traitement de l'environnement mainframe (étude Forrester Consulting pour Compuware, février 2018).

Or aujourd'hui, les méthodes de travail et les outils de développement utilisés sur mainframe sont souvent obsolètes. Par ailleurs, les logiciels de mise en qualité logicielle comme SonarSource sont souvent utilisés pour mesurer la qualité du code



dans toute l'entreprise. Ils intègrent cependant très rarement le mainframe. C'est également le cas pour les tests et l'intégration continue. Pire encore, les professionnels les plus expérimentés sont progressivement en train de partir à la retraite. Ils emportent avec eux la connaissance mainframe critique pour l'entreprise. En plus, seulement 1/3 du personnel qualifié est remplacé. En 2020, ce sont plus de 84 000 postes qui seront à pourvoir dans le secteur.

Les conditions de marché et la problématique du renouvellement générationnel impliquent pour les grandes entreprises d'attirer une nouvelle génération de développeurs avec un environnement de travail ressemblant aux IDE qu'ils ont l'habitude d'utiliser. De surcroît, le mainframe doit intégrer des outils de développement modernes et appliquer les bonnes pratiques DevOps. Les avantages d'une approche Agile sur mainframe sont les mêmes que pour les autres plateformes ; gains de temps pour les équipes, accélération du déploiement applicatif, réduction des risques, réduction du nombre et du temps de résolution moyen des incidents, … Ceci pour, au final, optimiser la satisfaction clients.
Une plateforme agile sur mainframe permet aujourd'hui à tout développeur même débutant de comprendre rapidement les applications les plus anciennes, les plus complexes et/ou les moins documentées de plus de 20, 30 ou 40 ans. Au travers d'une interface intuitive, il pourra visualiser et travailler sur les données à la fois mainframe et non mainframe sans écrire de programmes ou de scripts, coder en SQL ou utiliser plusieurs utilitaires. Elle donne également accès aux principaux outils de productivité pour l'analyse & le débogage, l'édition, le développement et la gestion des codes sources, les tests et l'optimisation des performances.
Encourager les développeurs à s'inscrire dans une véritable approche DevOps sur le mainframe pourrait paraître compromis dans un environnement où les tests sont encore en grande majorité effectués manuellement.
Pourtant, avec une plateforme agile sur mainframe, les développeurs pourront automatiser les tests unitaires, fonctionnels, d'intégration et de non-régression d'un programme. Voire même d'un groupe de programmes à partir du même scénario de tests réalisé à l'aide d'un éditeur et d'une boîte de dialogue d'exécution.
Et ce, au sein d'un environnement unique. Ils pourront mettre en place une méthode de tests plus agile reposant sur l'approche « shift left ».

Enfin, une plateforme de développement mainframe agile s'intègre parfaitement dans la toolchain DevOps des entreprises grâce à des plug-ins Jenkins et API REST. Les développeurs vont pouvoir créer, tester et déployer plus rapidement du code mainframe en toute sécurité et le gérer tout au long de son cycle de vie. Des indicateurs de performance tels que le temps moyen de résolution (MTTR), la couverture du code et le nombre d'anomalies détectées en phase de test vs en production feront émerger des métriques de qualité. Cela aidera les équipes à améliorer le développement tout au long du cycle de vie des applications.

Un changement d'état d'esprit au sein des entreprises favorisant la modernisation des processus et des plateformes mainframe permettra aux équipes de collaborer de manière plus transparente, indépendamment de leur plateforme. Mais en 2020, combiner DevOps et mainframe ne suffira pas ! Les entreprises doivent reconnaître que leurs équipes de développement sont la clé de la satisfaction client. Elles doivent leur donner des méthodes de travail modernes, ainsi que des outils et des processus qui amélioreront continuellement leurs capacités et satisferont toujours plus le client.

Grégory Guillou
Leader technique de l'équipe Easyteam DevOps. Grégory travaille avec Kubernetes depuis 2016 et développe en Go depuis 2018. Il a d'abord construit et maintenu des clusters, avant d'intégrer des solutions de gestion opérationnelle incluant le déploiement continu, la supervision des applications et des solutions de ChatOps avec Slack, Teams et GitHub. Il est un contributeur régulier du blog https://natives.easyteam.fr.

Étendre Kubernetes avec Kubebuilder

Partie 1

Kubernetes est un orchestrateur de containers. Mais c'est surtout une API extensible et riche ! Les fournisseurs cloud ont intégré leur réseau, leurs outils de supervision, leur solution de stockage, leurs équilibreurs de charge niveau 4 et niveau 7 pour offrir des services à valeur ajoutée à Kubernetes. Les éditeurs de logiciels comme Elastic, Confluent ou Redis Labs proposent des opérateurs qui gèrent leurs solutions sur vos clusters. Des projets comme Linkerd, Knative ou Argo-Flux étendent Kubernetes avec un "Service Mesh", une infrastructure "Serverless" et une solution "GitOps". Kubernetes est décrit comme une solution "API-centric" et des centaines d'organisations utilisatrices et de fournisseurs développent des composants qui intègrent ou étendent Kubernetes. Les composants de Kubernetes deviennent eux-mêmes, peu à peu, des extensions de l'API. Vous allez découvrir comment étendre Kubernetes et une application avec un opérateur appelé Emojis, programmé avec Kubebuilder et Go.

Pourquoi programmer Kubernetes ?

Cet article s'inscrit dans un dossier consacré au DevOps et sans doute qu'il se passera 365 jours à travailler avec Kubernetes avant que vous arriviez à la conclusion que développer vos infrastructures est indispensable à votre travail. Pourtant, et d'une certaine manière, c'est la plus haute marche du DevOps aujourd'hui. Entendons-nous :

- Les Ops ne devraient plus consacrer trop de temps à construire et mettre à jour des clusters Kubernetes. Les fournisseurs cloud, les éditeurs de distributions et même certaines solutions open source comme Kops contribuent largement à rendre cette tâche relativement atteignable ; quitte à vous faire aider.
- Les écosystèmes sont de plus en plus matures : intégrer la supervision, sécuriser les environnements, mettre en place des capacités d'adaptation ou opérer les systèmes devrait également être de moins en moins compliqué. C'est en tout cas ce que veut dire la communauté quand elle dit que Kubernetes devient "ennuyeux".
- Les opérations principales du DevOps devraient désormais consister à offrir des services et des qualités aux applications. Pour fixer les idées, le DevOps pourra offrir des outils pour tester, valider ou documenter une application. Il pourra créer des outils pour simplifier l'expérience des développeurs, du produit et des métiers. Il pourra intégrer des solutions pour améliorer les temps de réponse, sécuriser les données ou mesurer les retours des utilisateurs. Idéalement, il faudrait que 3/4 du temps des équipes DevOps soit consacré à soutenir les projets et après quelques mois, c'est généralement le cas.

La deuxième année que vous passerez après avoir construit et opéré Kubernetes, alors que vous aurez également développé des outils, intégré d'autres outils et aidé vos collègues à construire la meilleure application, l'un des événements suivants va très probablement se produire:

- Vous aurez envie de contribuer, vous aussi, à une communauté qui vous donne sans attendre en retour.
- Vous utiliserez un outil qui s'appuie sur Kubernetes et auquel il manque juste une petite fonctionnalité ou qui a juste un petit défaut que vous pourriez améliorer si vous programmiez.
- Vous vous rendrez compte que si certains composants applicatifs savaient eux-mêmes utiliser l'infrastructure, des possibilités infinies s'offriraient à l'application et à vous.
- Vous voudrez devenir développeur front-end parce que si tout le monde voit vos erreurs, personne ne se rend compte à quel point vous avez transformé le projet.

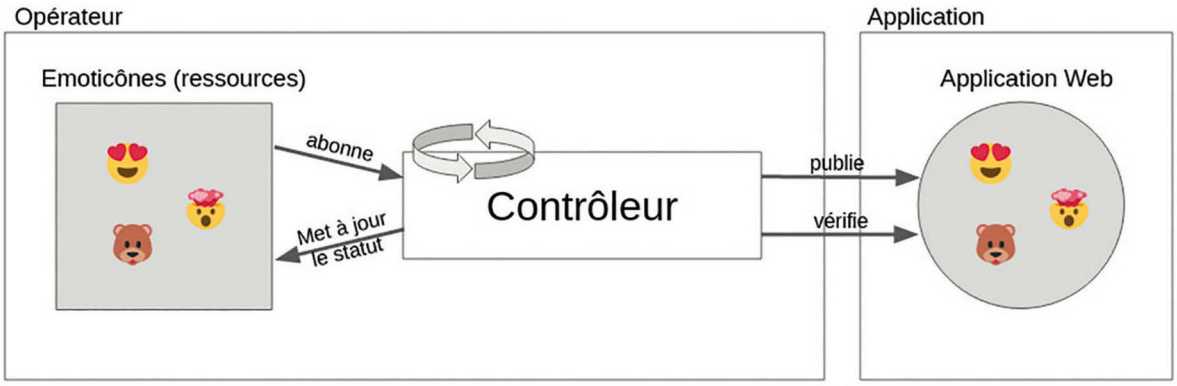
À moins que vous n'ayez réellement envie de changer de métier, créer des extensions à Kubernetes deviendra une évidence. Par exemple, votre application pourra allouer elle-même des ressources pour certaines opérations. Elle pourra, si elle est multi-tenant, créer des nouveaux environnements client. Elle pourra, elle-même, créer des environnements de formation. Les possibilités sont infinies ; pensez juste à ce qu'on vous demande de faire et qu'un opérateur virtuel ou l'application pourraient faire pour vous. Vous aurez dépassé l'ère du script, de l'infrastructure as code, de la construction et de la validation d'artefacts. Vous serez en train d'améliorer l'expérience de vos collègues et d'augmenter les capacités de vos applications.

Ne soyez pas bête : oubliez les 365 premiers jours et commencez maintenant !

Étendre Kubernetes

Il existe différents types d'extensions pour programmer Kubernetes. Ils dépendent du niveau d'intégration:

- Les plug-ins sont des programmes exécutables qui sont directement appelés par les composants. Elles permettent d'étendre le client kubectl. Vous trouverez de nombreux plug-ins dans le projet kubernetes-sigs/krew-index sur GitHub. Elles permettent également d'étendre les kubelets notamment pour gérer les devices, les réseaux, le runtime de container ou le stockage. Ces dernières sont très intégrées aux clusters et sont plutôt développées par des éditeurs spécialisés ou les fournisseurs cloud.
- Les webhooks sont des composants qui sont appelés par le serveur d'API à certaines étapes et influent généralement sur celles-ci. La majorité des webhooks permettent de modifier le



1 *Interactions entre les ressources, le contrôleur et l'application*

comportement de l'API. Ils sont utilisés pour étendre l'authentification et les autorisations. Ils permettent de modifier des ressources à leur création ou modification. Ils permettent de valider les ressources à leur création ou modification. Ils permettent de modifier les règles d'ordonnancement des pods.

- Les "Custom Resource Definitions" ou CRD sont généralement décrites à l'aide d'un manifeste et permettent de définir des ressources personnalisées que l'API pourra prendre en charge comme les ressources natives de Kubernetes. Ainsi vous pourrez définir une "fonction serverless" ou une "règle pour déployer une version applicative sans qu'elle soit livrée aux utilisateurs".
- Les "Controllers", appelés contrôleurs ci-dessous, sont des programmes qui observent un type de ressource, lit ses métadonnées ou sa spécification, réalise un certain nombre d'opérations auprès de Kubernetes ou d'un système tiers et, lorsqu'il est associé à une CRD met à jour le statut de la ressource. Il existe les contrôleurs internes, les contrôleurs pour le fournisseur cloud et les contrôleurs personnalisés
- L'"Aggregation Layer" est un mécanisme par lequel les appels à l'API peuvent être entièrement re-routés à une API extérieure. Si ce mécanisme permet de réaliser des prouesses comme le fait de faire tourner une stack compose sur Kubernetes avec le projet GitHub docker/compose-on-kubernetes, vous préférerez éviter cette approche pour x raisons et surtout parce que vous risquez de perdre la plupart des bénéfices de Kubernetes à la mise en œuvre.

D'autre part, si vous êtes intéressé par Kubernetes, vous avez certainement entendu parler des "Operators". Les opérateurs sont en fait un ensemble de CRD et de contrôleurs regroupés pour intégrer une application "nativement" à Kubernetes. Ainsi un opérateur pourra faciliter l'installation et la mise à jour de l'application. Il pourra assurer de nombreux autres services tels que l'intégration à la supervision, la mise en place de sauvegarde et de restauration, la capacité à reprendre en cas de crash ou la possibilité de créer des environnements. Ainsi si les possibilités offertes par les opérateurs n'ont pas d'autres limites que l'imagination, développer un opérateur c'est développer des CRD, des contrôleurs et un paquet pour le gérer.

Langages et Frameworks

Avant de démarrer se pose la question des langages et des frameworks pour étendre Kubernetes. En théorie, Kubernetes étant basé sur OpenAPI et YAML, vous pouvez utiliser n'importe quel langage et d'ailleurs, vous trouverez des contrôleurs en Java ou en Rust. Néanmoins c'est largement plus facile d'utiliser un framework et d'accéder au code de Kubernetes ainsi qu'aux nombreux "helpers" disponibles. Go reste le langage préféré pour programmer Kubernetes. Parmi les bénéfices immédiats, la possibilité d'utiliser un cache de l'API kubernetes ou les différents outils de tests. Vous pouvez imaginer qu'avec le temps cela soit de moins en moins vrai et certains langages voient apparaître des frameworks comme Python avec Kopf et Ansible avec operator-sdk.

Avant d'aller plus loin, regardez si un outil haut niveau comme "Kubernetes Universal Declarative Operator ou KUDO" et le "Metacontroller" ne conviennent pas mieux à votre besoin. Si vous utilisez Go, il existe 2 frameworks de développement : Kubebuilder et Operator-SDK. Ces 2 frameworks sont complémentaires, Kubebuilder se concentrant sur "Go". Les équipes travaillent en commun comme expliqué dans "Deep Dive into the Operator Framework for Kubernetes"[*]. Bien sûr, vous pouvez aussi développer sans outil, auquel cas, vous regarderez le projet kubernetes/sample-controller sur GitHub.

L'exemple de cet article est basé sur Kubebuilder. Chacun des outils cités plus haut a ses avantages. À vous d'étudier celui qui vous convient le mieux. 1

[*] "Deep Dive into the Operator Framework for Kubernetes" - https://youtu.be/8_DaCcRMp5I?t=2767

API et programmation

Avant de plonger dans le développement, il est important de comprendre la philosophie de ce que vous allez réaliser. D'abord l'API :

- Sa structure de données est fixe. Elle est contrainte par des metadonnées comme le groupe d'API, la version, la spécification ou encore le statut.
- La liste des verbes qui s'appliquent à ces données est fixe ; elle contient CREATE, UPDATE, GET, DELETE, PATCH, LIST et WATCH.

Ce modèle restrictif est souvent appelé "modèle de ressources". Il facilite largement l'utilisation de l'API par des programmes, des utilisateurs et même les programmeurs. Quand vous connaissez les verbes et les structures de données, il ne reste plus qu'à connaître les propriétés spécifiques à un type de ressources.

Le modèle de programmation est, lui aussi, contraint; il s'appuie essentiellement, sur les webhooks et les contrôleurs personnalisés. Les autres modèles ne s'appliqueront que rarement à vos usages.

La structure d'API adresse la complexité de gestion tandis que la structure de programmation adresse la complexité d'état.

Ensemble, elles créent des cadres abordables pour les développeurs et permettent de faire fonctionner un système largement distribué en bénéficiant de toutes ses propriétés. Évidemment il existe de nombreux cas aux limites, y compris le verbe SCALE ou les mécanismes pour faire évoluer une version d'API. Kubernetes et Kubebuilder couvrent aussi ces cas.

Opérateur Emojis : Conception

Dans les sections qui suivent, vous allez développer un opérateur Kubernetes qui gère des émoticônes. Cet opérateur interagit avec une application qui affiche la liste des émoticônes ainsi que les noms "friendly" que vous leur associerez. Le "design" de l'opérateur en question est présenté ci-dessus : 1

Opérateur Emojis : Développement

Dans les sections suivantes, vous trouverez les étapes principales pour développer un opérateur. Le code associé est dans le projet **easyteam-fr/emojis** disponible sur GitHub. Cloner le projet sur votre ordinateur ou une instance Linux. La branche master du projet contient uniquement l'application sans l'opérateur. Elle est située dans le répertoire app. Il s'agit de 2 API REST : l'une sur le port 8080 affiche les émoticônes publiées si vous faites un GET sur / ; l'autre sur le port 8081 permet de faire un GET sur /emojis, mais aussi des GET et PUT et DELETE sur /emojis/:emoji pour vérifier, ajouter et supprimer des émoticônes. Cette application est inspirée de l'exemple fourni par Linkerd et supporte les 100 émoticônes disponibles dans buoyantio/emojivoto.

Kubebuilder ne fonctionne que sur Linux ou MacOSX ; vous devez en outre avoir une configuration sur laquelle sont installés gcc, make, git, go, docker et kubectl. Vous devez avoir accès à un cluster Kubernetes et une registry Docker. Pour ce dernier point et pour aller vite, n'hésitez pas à utiliser kind - mon coup de cœur-, minikube ou, sur Mac, la version Kubernetes de Docker for Desktop. Chacune des sections qui suit correspond à une étape de développement. Pour vous aider dans la progression, partez de la branche master. Chaque section fait référence à une branche qui contient le code de l'étape. Vous pouvez ainsi suivre l'évolution du code en changeant de branche à la fin de chaque étape.

Créer un environnement de développement

branch: operator/01-initialization

Pour créer un environnement de développement, vous devrez préalablement installer gcc, git, go, docker, kubectl et kubernetes connecté à une registry docker. Vous pourrez alors installer kubebuilder, cloner le projet easyteam-fr/emojis qui contient uniquement l'application sur la branche master et créer un squelette pour l'opérateur :

```
# installer kubebuilder
mkdir -p $HOME/bin
export PATH=$HOME/bin:$PATH
cd $HOME/bin
export KBUILDER=https://github.com/kubernetes-sigs/kubebuilder/releases/download
export VERSION=2.2.0
curl -L $KBUILDER/v${VERSION}/kubebuilder_${VERSION}_$(go env GOOS)_amd64.tar.gz \
| tar -zxvf - -C . --strip=2 \
  kubebuilder_${VERSION}_$(go env GOOS)_amd64/bin/kubebuilder

# cloner le projet
git clone https://github.com/easyteam-fr/emojis.git

# créer un squelette d'opérateur dans le répertoire operator
cd emojis
mkdir -p operator
cd operator
kubebuilder init --domain natives.easyteam.fr \
      --repo=github.com/easyteam-fr/emojis/operator
```

Attention !

gcc doit être installé sur votre instance et kubebuilder n'est pas disponible sur Windows

Restreindre l'opérateur à un namespace

branch: operator/02-namespace-operator

Un opérateur peut gérer des données dans un ou plusieurs namespace. Par défaut, kubebuilder crée une configuration qui gère l'ensemble des "namespace". Vous pouvez ajouter les lignes suivantes en haut de la fonction main du fichier main.go pour ne gérer qu'un unique namespace.

```
namespace := "default"
if os.Getenv("EMOJIS_NAMESPACE") != "" {
    namespace = os.Getenv("EMOJIS_NAMESPACE")
}
```

Ajoutez ensuite la propriété Namespace dans les options de ctrl.NewManager :

```
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
    [...]
    Namespace: namespace,
    })
```

Créer la "Custom Resource Definition" et le "Controller"

branch: operator/03-crd-and-controller

Vous pouvez désormais créer l'API Emojis avec : la "Custom Resource Definition" grâce au paramètre --resource ; le contrôleur grâce au paramètre --controller ; un exemple grâce à --example. Exécutez la commande ci-dessous :

```
# créer une CRD, un exemple de CR et le contrôleur associé pour Emoji
kubebuilder create api \
  --group app --version v1alpha1 --kind Emoji \
  --controller --example --namespaced --resource
```

Pour tester immédiatement la CRD et le contrôleur, lancez les commandes ci-dessous :

```
# Déployer le CRD sur Kubernetes
make install

# Démarrer le contrôleur
make run
```

La suite dans le numéro 239

Julien Lepine
Manager, EMEA
Specialist SA / AWS

DevOps côté Dev, les outils, les mises en pratique, les tendances !

Cet article propose un tour d'horizon des impacts de l'adoption des pratiques DevOps sur les équipes de développement. Encore trop souvent, en entreprise, le point de focalisation est l'éternel jour/homme du développeur qui produira de la valeur. En accord avec la théorie des contraintes (au centre des approches DevOps et Lean), *toute optimisation hors du chemin critique et n'impactant pas la contrainte principale de votre chaîne de production (logicielle ici) est inutile*. Il est donc important d'identifier quelles sont les étapes de votre chaîne logicielle puis d'identifier les points de contention. Si l'éditeur et le langage sont souvent traités, les développeurs ne passent pas tout leur temps à « pondre du code », et nous abordons ici certains autres domaines. S'il vous faut trois mois pour mettre en production un changement fait en une journée, accélérer le développement des fonctionnalités ne sert à rien, car l'impact sera juste plus de fonctionnalités en attente, et non des utilisateurs satisfaits.

Les trois voies du DevOps

Bien que la démarche DevOps ne se concentre pas sur la méthode de développement, et s'applique à tout type d'environnements matériels et logiciels, ***la première voie du DevOps traite du contrôle et l'accélération du « travail en cours »*** (*WIP / Work in Progress* en Anglais) à travers la notion de flux descendant, et s'associe en cela parfaitement avec les méthodes agiles. L'idée est simple, une fonctionnalité développée et non déployée ne fournit pas de valeur à l'entreprise. La raison principale est que tout travail commencé implique un investissement (matériel ou humain), donc un coût pour l'entreprise, et ne produit de la valeur qu'une fois livré. Nous retrouvons cela dans les principes *lean*, et les outils de type *kanban* sont très utilisés pour rendre le WIP visible par tous. Les démarches agiles sont un allié de poids pour le DevOps, en permettant de prioriser les fonctionnalités développées sur les *stories* à forte valeur métier. Une question importante dans les équipes agiles est de définir ce qu'est un *travail fini*, est-ce que cela veut dire développé, testé, intégré, validé, packagé, déployé ? Dans les équipes suivant les principes DevOps, cela veut très souvent dire déployé en production et validé.

La gestion des *environnements* est souvent déléguée à une équipe externe, voire ignorée, et trop peu documentée. C'est aussi régulièrement un point de conflit entre les équipes de développement et de production en raison du coût de ceux-ci, de l'impossibilité de les maintenir à jour, ou d'avoir des données fraîches. Une connaissance profonde des besoins, et pratiques de l'équipe permet d'adapter au mieux les réponses techniques et améliorer la productivité des développeurs. Il n'est pas acceptable par exemple pour de nouveaux développeurs rejoignant une équipe de devoir patienter des jours avant d'avoir un environnement fonctionnel. En adoptant des principes DevOps, certaines entreprises font déployer un changement en production à tous les nouveaux développeurs rejoignant l'entreprise dans leur première journée en poste !

Programmez! donne régulièrement la main aux testeurs, et dans un monde DevOps, la qualification logicielle et la qualité ont un rôle primordial, en phase avec la ***deuxième voie du DevOps, centrée sur la création de feedback rapides*** à toutes les étapes de la chaîne de valeur. Les démarches *Test Driven Development* permettent aux développeurs d'intégrer des tests directement dès l'écriture de leur code, mais ne sont pas à elles seules suffisantes. Il est crucial d'intégrer des vérifications et des tests à toutes les étapes du processus, par exemple des code review et de l'analyse statique de code sur chaque *pull request*, des tests automatisés à chaque compilation, un déploiement automatique sur un environnement d'intégration qui sera lui aussi testé automatiquement (*continuous integration*). Ceci peut aller jusqu'aux environnements de pré-production (*continuous delivery*), voir mener à un déploiement en production (*continuous deployment*). L'étendue des tests (sécurité, performance, systèmes, …) est grande, et doit être prise en considération tout au du cycle de vie de l'application.

Le monitoring et les tests en production sont en forte croissance, car nous le savons, les problèmes intéressants se passent toujours en production. Les canaris par exemple sont des tests automatiques qui valident que les environnements en production répondent toujours de manière appropriée. Ils sont exécutés périodiquement et automatiquement. Les tests A/B permettent eux de sélectionner des utilisateurs et de les envoyer vers un environnement sur lequel certaines fonctionnalités spécifiques ont été activées afin de valider l'impact pour les clients, les entreprises les plus avancées ont des centaines de tests en parallèle actifs à tout moment. Une pratique relativement récente est le *Chaos Engineering*, ou comment tester la résilience des applications lors d'incident par la pratique : des erreurs et incidents sont injectés sur les environnements de manière régulière lors de *Game Days*, ou continuellement. Ceci valide que les applications et les équipes ont la capacité à surmonter ces erreurs de manière automatique et à apprendre. ***La maîtrise de l'environnement permet la prise de risque, qui, avec une expérimentation continue et une répétition, sont au cœur de la troisième voie du DevOps***. En accord avec le principe « si ça fait mal, faisons-le plus souvent », des entreprises comme Netflix déploient en production plusieurs milliers de fois par jour. Cela change des déploiements mensuels ou trimestriels encore trop présents.

1 (https://twitter.com/ben11kehoe/status/1209951169000464391)

Les techniques et outils

Sur la gestion des sources, Git mène toujours la danse, avec un écosystème florissant d'outils et solutions en faisant la solution incontournable dans la communauté et dans les entreprises. Un modèle très utilisé est le *trunk based development*, où les fonctionnalités sont intégrées dans la branche master au plus tôt, et en utilisant des *feature flags* pour ne pas activer les fonctionnalités trop tôt en production. Le but assumé ici est d'effectuer des modifications les plus fines possibles, en suivant le principe des *small batches* où chaque tâche (et donc WIP) est la plus petite possible pour en accélérer la production. L'utilisation de plateformes automatisées pour l'ensemble de la chaîne d'intégration, de contrôle, de test, et de déploiement est l'épine dorsale de votre stratégie DevOps. Des plateformes telles qu'AWS CodePipeline (et l'ensemble des services AWS Code*) vous fournissent des solutions à toutes les étapes de votre travail. En DevOps, les équipes d'opérations utilisent aussi les mêmes outils. Tout changement (applicatif, configuration, système…) suit le même processus de déploiement de bout en bout, formant le modèle *GitOps*.

Un retour fréquent des développeurs est une réticence envers les fichiers YAML ou JSON. Il semble bien que l'aspect « human readable » n'ait jamais voulu dire « facile à comprendre, manipuler et modifier ». Les plateformes Infrastructure as Code sont puissantes et comparables à un assembleur pour le cloud. Heureusement, elles évoluent aussi pour que vous puissiez programmer vos environnements. Des outils comme Pulumi ou AWS Cloud Development Kit (CDK) expriment directement en « vrai » code (CDK supporte JavaScript, TypeScript, Python, C#/.NET Core, et Java) ce que vous voulez effectuer, avec des valeurs par défaut simples et une syntaxe plus courte. Cet exemple CDK crée une table Amazon DynamoDB, une fonction AWS Lambda pour le traitement des données, et donne le droit à cette fonction de lire et écrire dans cette table. Résultat ? Un environnement totalement élastique, payé à la requête (gratuit pour le premier million de requêtes par mois), documenté, et sécurisé.

```
const itemsTable = new Table(this, 'Items', {
  billingMode: BillingMode.PAY_PER_REQUEST,
  partitionKey: { name: 'itemId', type: AttributeType.STRING }
});

const itemHandler = new lambda.Function(this, 'ItemHandlerFunction', {
  runtime: Runtime.NODEJS_10_X,
  code: new AssetCode('my-code'),
  timeout: Duration.seconds(300),
  handler: "Items.handler",
  environment: {
    ITEMS_TABLE: itemsTable.tableName
  }
});

itemsTable.grantReadWriteData(itemHandler);
```

L'essor des plateformes cloud offre la possibilité de bénéficier des meilleurs outils dans l'état de l'art, de manière totalement automatisée. Vous avez besoin d'une base de données pour une application ? Il vous est possible d'en avoir une en quelques minutes, suivant les meilleures pratiques de haute disponibilité, mise à jour automatique, sauvegarde, et sécurité. Vous pouvez aussi choisir parmi plusieurs types de bases de données, utiliser une base de données orientée graphe pour votre moteur de recommandation, une base NoSQL pour vos applications à fort trafic, un journal de transactions sécurisé pour vos applications financières. Vous choisissez le meilleur outil pour votre besoin : le langage de programmation le plus approprié, la base de données idéale, les services de machine learning de pointe, tout en bénéficiant de l'expérience combinée de millions de clients.

Les plateformes cloud full-stack prennent de plus en plus d'essor. Vous pouvez vous concentrer sur le développement de votre application, et utiliser l'ensemble des patterns pré-configurés pour l'environnement. AWS Amplify permet par exemple de développer des applications Web et mobiles en construisant sur des briques préparées pour vous. Le « serverless » est un mouvement de fond sur l'hébergement d'applications modernes. Vous n'avez pas à prendre en charge l'infrastructure sous-jacente. L'impact ? Une société comme iRobot qui fournit des robots à usage domestique a migré son infrastructure sur les solutions Serverless AWS, et peut soutenir des pics de charge très importants, au niveau mondial, quasiment sans impact. 1

L'extension de la responsabilité des équipes leur demande aussi de comprendre les comportements en production. L'objectif ici n'est pas de leur donner un accès complet aux environnements, loin de là. L'*observabilité* regroupe plusieurs techniques permettant d'avoir accès à des données dites a forte cardinalité (pour lesquelles une simple agrégation n'est pas suffisante, et sur lesquelles on ne connaît souvent pas la question au préalable) et donc très utilisé pour l'analyse des logs par exemple. En profiling, des plateformes comme Amazon CodeGugu analysent le comportement réel en production des applications, présentant les variances par rapport aux tests de montée en charge, les chemins principaux du code, vous laissant vous concentrer sur l'optimisation du code dont l'impact est le plus important.

Les transformations sur le long terme

Avec l'adoption des pratiques DevOps, la sécurité devient (ainsi qu'elle aurait toujours dû l'être) la responsabilité de l'équipe dans son ensemble, et donc des développeurs. Il n'est plus possible de déléguer cette responsabilité à une équipe sécurité lointaine qui dépend des ops. Les développeurs doivent plus que jamais prendre ce problème à bras le corps, de la validation des entrées, la gestion des dépendances (détection des CVE sur les bibliothèques externes par exemple), tests OWASP sur les applications, et toutes les pratiques qui permettent de réduire les risques efficacement.

En accord avec la loi de Conway, les systèmes sont souvent un miroir des organisations qui les ont produits. Pour avoir des équipes agiles, il faut les laisser déployer sans dépendre d'autres équipes, et les micro-services fournissent des solutions technologiques de pointe pour cela. Qu'elles soient sur des conteneurs, complètement serverless, ou même à base de machines virtuelles, ces applications offrent un couplage faible et par contrat entre les services.Elles laissent les équipes s'organiser en fonction de leurs besoins. Cela ne veut pas dire que les plateformes « monolithiques » n'ont plus d'avenir. Les outillages disponibles rendent la création de ces applications efficace (Progiciels, Enterprise Resource Planning, Frameworks, Object Relational Mappers, Environnements LowCode ou Rapid Application Development). Le point clé est de pouvoir prendre des décisions informées sur les avantages et inconvénients de chaque modèle.

Bien que les pratiques DevOps n'imposent ni langage, ni plateforme, ni technologie, certaines pratiques d'architectures logicielles permettent de se préparer au mieux à une approche DevOps, telles que les architectures hexagonales (présentées dans les numéros de janvier et février). Ces pratiques isolent les différentes parties de l'application, rendant aisé le changement du mode de communication, du système, ou de la base de données, en concentrant le code développé sur le cœur métier de l'application. Ces mouvements s'accompagnent d'une croissance de la programmation fonctionnelle. Ces applications, parfois *polyglottes* car elles vont utiliser plusieurs langages selon les besoins, laissent le développeur choisir encore une fois l'outil le plus approprié au besoin. Les équipes adoptent aussi une approche « craft » de développement, utilisant diverses techniques pour apprendre et collaborer (mob, kata, pair programming).

Loin du « framework d'entreprise » qui impose partout une uniformité des pratiques, les démarches vues jusqu'ici laissent de la place à l'innovation et à la responsabilité. Ceci en demandant des garanties (sécurité, opérabilité, performance, maintenabilité) sur les solutions choisies. La culture DevOps amène l'idée d'avoir des plateformes évolutives, centrées sur l'amélioration continue. Ceci est un changement important dans la mentalité des équipes informatiques, qui pour beaucoup encore ont été formées à la création de la « plateforme parfaite » du premier coup. Les technologies, les besoins métiers, les contraintes réglementaires et organisationnelles, tout est en évolution constante. La capacité des applications à évoluer avec elles est un enjeu fort pour le contrôle de la dette technique des organisations.

En conjonction avec l'adoption de pratiques DevOps et des méthodes agiles, un changement profond est en cours dans la culture actuelle du « projet » dans les entreprises. Les équipes DevOps gèrent des produits (elles ont souvent plusieurs produits en parallèle), et en ont la responsabilité de bout en bout, en accord avec le principe *You build it you run it*. De ce fait, au lieu d'avoir des grandes organisations hiérarchiques par fonction (développement, environnements, QA, sécurité), les entreprises ont des équipes pluridisciplinaires, et les fonctions support deviennent des équipes elles aussi, fournissant des services internes, souvent en mode DevOps.

Les avantages du DevOps sur le cloud pour les développeurs

L'adoption des principes DevOps amène une culture de partage, d'excellence technologique, centrée sur l'apprentissage des bonnes pratiques et des erreurs entre les équipes. Dans ce modèle, les experts peuvent évoluer en changeant d'équipe et en augmentant leur impact (avec des titres de « staff », « principal » ou « distingué ») sans avoir à devenir chef de projet, ou manager.

- Le cloud AWS propose une boîte à outils de plus de 180 services qui répondent aux besoins réels et mesurés de millions de clients. Cela leur permettant de supporter leurs applications critiques, et de se transformer au travers de la technologie. Le DevOps est au cœur de la stratégie d'AWS, les services étant disponibles par API, facturés à l'usage, et élastiques.
- Avoir une vision claire des besoins en termes d'environnements, de composants, de données, et une chaîne automatisée assurant la qualité du travail effectué donne aux équipes un sentiment contrôle. Elles ont les outils nécessaires pour effectuer leur travail efficacement, prendre des risques dans un environnement agile et sécurisé, et cela impacte positivement leur motivation.
- Pouvoir livrer rapidement les fonctionnalités à plus forte valeur ajoutée pour l'entreprise est un objectif derrière lequel les directions informatiques et les directions générales peuvent facilement s'aligner. L'adoption DevOps est un chemin, qui touche beaucoup de domaines, mais qui offre énormément d'opportunités.

Sources

The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win : https://www.amazon.fr/Phoenix-Project-Helping-Business-English-ebook/dp/B078Y98RG8

The Unicorn Project: A Novel about Developers, Digital Disruption, and Thriving in the Age of Data : https://www.amazon.fr/Unicorn-Project-Developers-Disruption-Thriving-ebook/dp/B07QT9QR41

Werner 10 Lessons from 10 years of Amazon Web Services [2016]: https://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html

Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations : https://www.amazon.fr/Accelerate-Software-Performing-Technology-Organizations-ebook/dp/B07B9F83WM

Thierry BOLLET

Powershell DSC : fonctionnalités avancées

Et si Powershell était en fait le langage parfait pour une pratique DevOps ? Automatisation de bout en bout ? Oui, de plus en plus, et encore plus, avec la solution Powershell DSC.

Powershell DSC est une solution gratuite et totalement intégrée de maintien en condition opérationnelle pour les machines Windows Server et également, au prix de quelques acrobaties, pour les machines Linux. Mieux même, Microsoft propose depuis quelques années un ressource kit dédié qui prolonge l'utilisation et permet, en plus de la conformité, l'installation de produits ou fonctionnalités. Il est par exemple possible d'installer ses contrôleurs de domaine Windows, ses machines SQL ou ses fermes Sharepoint de manière totalement automatisée. C'est un produit en évolution constante avec des mises à jour toutes les 6 semaines pour les modules d'installation. Je ne parle ici que des modules validés Microsoft. La bibliothèque communautaire publiée sur le site https://www.powershellgallery.com/ est riche de plusieurs centaines d'autres modules et fonctions. Une mine d'or.

Compatible Microsoft Azure au travers de la solution Azure Automation et son option State configuration (DSC), c'est une solution de code parfaite pour l'homogénéisation d'un parc de machines On Premise mais aussi dans le nuage. Dernier avantage, un serveur de rapports intégré pour assurer le suivi des actions.

Présenté et expliqué dans numéro 228 de la revue Programmez! dans son mode le plus courant, Powershell DSC offre également des fonctions avancées. Il est question aujourd'hui de fichiers de nœuds et de configurations partielles.

Le fichier de nœuds :

Lors de la compilation de la configuration DSC, c'est à dire le passage d'un format PS1 au format Mof accepté par le moteur de configuration, chaque fichier est "dédié" à une machine distante cliente de la configuration.

Par exemple, un contrôleur de domaine recevra un ensemble de paramètres de sécurité commun à toutes les machines puis ses paramètres spécifiques. Pour préparer une machine d'hébergement internet, le fichier sera copié, les paramètres communs conservés, les paramètres spécifiques au contrôleur de domaine retirés et ceux spécifiques au serveur internet ajoutés. Alors oui, cela fonctionne, mais ce n'est pas toujours très simple et cette démarche oblige à gérer de nombreux fichiers avec un ensemble de données communes (les fameux paramètres de sécurité). Beaucoup de fichiers donc, avec un tronc commun.

Heureusement, il existe une autre solution, bien plus efficace et bien plus élégante : le fichier de nœud. Ce fichier embarque l'ensemble des paramètres dans un seul et même script. En début de script, une variable $DemoProgrammez de type Hashtable est créée avec l'ensemble des serveurs concernés par la configuration. Mais chaque serveur est attaché à un ou plusieurs paramètres. Exemple ici, le serveur SRV1 recevra un bloc de configuration Role1 là ou SRV4 recevra les blocs Role1, Role2 et Role3.

```
$DemoProgrammez = @{
    AllNodes = @(
        @{  NodeName = "SRV1"
            Role = "Role1"
        }
        @{  NodeName = "SRV2"
            Role = "Role1","Role2"
        }
        @{  NodeName = "SRV3"
            Role = "Role1","Role3"
        }
        @{  NodeName = "SRV4"
            Role = "Role1","Role2","Role3"
        })}
```

On retrouve dans le corps du script une déclaration classique de Powershell DSC, avec une légère différence puisque chaque paramètre est filtré par une commande Where. Ainsi, la configuration du registre RegistryKeyRole1 présentée ce dessous sera appliquée au nœud (node) filtré et égal à "Role1", Node $AllNodes.Where{$_.Role -eq "Role1"} alors que la configuration RegistryKeyRole2 sera, elle, appliquée au nœud égal à "Role2", Where{$_.Role -eq "Role2"}. Cette table isole chaque machine, le fichier généré sera différent, il est propre à chacune des machines.

```
Node $AllNodes.Where{$_.Role -eq "Role1"}.NodeName
    {Registry RegistryKeyRole1
        {
        Ensure = "Present"
        Key = "HKEY_LOCAL_MACHINE\SOFTWARE\Role1"
        ValueName = "Role1"
        ValueData = "ProgrammezData1"
        }}
Node $AllNodes.Where{$_.Role -eq "Role2"}.NodeName
    {Registry RegistryKeyRole2
        {
        Ensure = "Present"
        Key = "HKEY_LOCAL_MACHINE\SOFTWARE\Role2"
        ValueName = "Role2"
        ValueData = "ProgrammezData2"
        }}
```

Dernière différence, la compilation. Le passage du fichier PS1 au fichier Mof se fait normalement par un simple appel qui porte le nom de la configuration globale. Le script est présent dans un bloc configuration TestProgrammez {*lecodepowershellDSC*} puis est compilé par la commande TestProgrammez. Pour utiliser la variable $DemoProgrammez et ainsi construire une configuration pour chaque type de serveur, la compilation utilise le paramètre -ConfigurationData sous cette forme :

TestProgrammez -ConfigurationData $DemoProgrammez.

Résultat, 4 fichiers Mof dont la taille et donc le contenu ne sera pas forcément la même.

```
    Répertoire : C:\temp\TestProgrammez


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        09/02/2020     08:16           2128 SRV1.mof
-a----        09/02/2020     08:16           3008 SRV2.mof
-a----        09/02/2020     08:16           3008 SRV3.mof
-a----        09/02/2020     08:16           3888 SRV4.mof
```

SRV1 appliquera uniquement les paramètres pour les serveurs "Role1" là ou SRV3 appliquera les paramètres "Role1" mais également "Role3". Un seul fichier, mais qui génère différentes configurations à la compilation. Une très bonne pratique ! Fichier unique n'est pas forcément égal à un contenu Mof unique.

Les configurations partielles.

Une fois DSC bien assimilé, il est possible de déléguer l'écriture d'une partie des configurations. Pourquoi en effet ne pas laisser chaque équipe technique gérer elle-même un fichier de configuration adapté à son produit et certainement beaucoup plus fin en terme de configuration ? Les équipes de base de données constituent un fichier adapté aux contraintes des bases de données, là où les équipes internet gèrent le spécifique à leur produit. Là aussi, cette bonne pratique peut se gérer manuellement par la récupération de l'ensemble des configurations puis la copie dans un fichier global. Mais ce n'est pas très efficace. La configuration partielle est une option avancée qui agit sur deux paramètres.

- La configuration du LCM, ou moteur de configuration DSC. Le comportement du moteur est paramétrable, il existe une trentaine d'options pour l'utiliser plus efficacement (comportement lors du redémarrage, validité des configurations…, etc.).
- La publication des configurations.

Pour bien comprendre le fonctionnement, il est possible d'utiliser un serveur ou même un poste de travail qui n'est pas piloté par DSC et d'afficher la configuration de son moteur LCM. Pour rappel, ce moteur est embarqué dans les produits Microsoft W10 ou Windows Serveur (2008 et Powershell 4 minimum).

Dans une fenêtre Powershell avec élévation de privilège, commande Get-DscLocalConfigurationManager.

La configuration du moteur est affichée avec l'ensemble des paramètres. Ici, une configuration courante sans option particulière.

```
ActionAfterReboot                : ContinueConfiguration
AgentId                          : FA9AD4C5-F3D1-11E9-B41B-9C7BEF76D9FC
AllowModuleOverWrite             : False
CertificateID                    :
ConfigurationDownloadManagers    : {}
ConfigurationID                  :
ConfigurationMode                : ApplyAndMonitor
ConfigurationModeFrequencyMins   : 15
Credential                       :
DebugMode                        : {NONE}
DownloadManagerCustomData        :
DownloadManagerName              :
LCMCompatibleVersions            : {1.0, 2.0}
LCMState                         : Idle
LCMStateDetail                   :
LCMVersion                       : 2.0
StatusRetentionTimeInDays        : 10
SignatureValidationPolicy        : NONE
SignatureValidations             : {}
MaximumDownloadSizeMB            : 500
PartialConfigurations            :
RebootNodeIfNeeded               : False
RefreshFrequencyMins             : 30
RefreshMode                      : PUSH
ReportManagers                   : {}
ResourceModuleManagers           : {}
PSComputerName                   :
```

La valeur PartialConfigurations est vide.

Le code suivant modifie le comportement du moteur avec de nouveaux paramètres pour la valeur PartialConfigurations. La commande finale Set-DscLocalConfigurationManager applique cette nouvelle configuration LCM.

```
[DSCLocalConfigurationManager()]
    configuration PartielleConfigProgrammez
        {Node localhost
            {
                PartialConfiguration ConfigBDD
                {
                Description = 'Configuration BDD localhost'
                RefreshMode = 'Push'
                }
                PartialConfiguration ConfigInternet
                {
                Description = 'Configuration Internet localhost'
                RefreshMode = 'Push'
                }}}
PartielleConfigProgrammez
Set-DscLocalConfigurationManager -Path C:\Users\tbollet\PartielleConfigProgrammez -Force
```

Puis la configuration du moteur est affichée de nouveau, commande de Get-DscLocalConfigurationManager.

```
MaximumDownloadSizeMB            : 500
PartialConfigurations            : {[PartialConfiguration]ConfigBDD, [PartialConfiguration]ConfigInternet}
RebootNodeIfNeeded               : False
```

Attention, le fichier de configuration a deux significations avec Powershell DSC. La configuration des paramètres, c'est-à-dire une déclaration de ce qui est attendu et la configuration LCM, c'est-à-dire la configuration du comportement du moteur. Ce n'est pas toujours très clair lorsque l'on débute avec DSC. Ici, après la configuration LCM, il ne reste qu'à préparer un fichier de configuration pour déclarer de nouveaux paramètres de conformité.

Le fichier est écrit normalement avec une seule contrainte, le nom de la configuration. L'équipe BDD prépare un fichier configuration ConfigBDD {codeDSCdedéclaration} et l'équipe internet configuration ConfigInternet {codeDSCdedéclaration}. Ces deux valeurs sont attendues par le moteur.

Après compilation, les configurations sont publiées sur la machine cliente par la commande Publish-DscConfiguration "chemindesconfigurationscompiléesmof" -ComputerName localhost -Verbose.

Sur la machine cliente des configurations, ces « morceaux » de fichiers Mof sont stockés temporairement dans le répertoire C:\Windows\System32\Configuration**PartialConfigurations**. C'est une différence majeure et très importante avec le fonctionnement normal de DSC. Un fichier compilé est unique, c'est-à-dire que le moteur de configuration reçoit un fichier Mof qu'il stocke dans le répertoire C:\Windows\System32\Configuration\ puis qu'il traite depuis cet emplacement. Il est impossible pour lui d'avoir deux fichiers Mof courants à cet endroit. Il y a dans ce répertoire 3 fichiers Mof de configuration. Le courant, traité par le moteur, un pending, nouveau fichier en attente si nécessaire, et le previous qui est l'ancien fichier. Plus d'autres Mof destinés à la configuration du moteur.

Avec la configuration partielle, les Mof sont donc stockés dans le sous répertoire **PartialConfigurations** et ne sont pas traités par le moteur. Du moins, pas avant d'en recevoir l'ordre par la commande Start-DscConfiguration -UseExisting -ComputerName localhost -verbose -Wait.

A cet instant, les « morceaux » de configurations partielles sont compilés de nouveau pour ne former qu'un seul fichier Mof. Le moteur renomme le fichier courant, il devient previous.mof puis le fichier nouvellement créé prend sa place et le moteur le traite comme un fichier unique. Il devient le nouveau fichier de configuration de référence pour la machine.

C'est une belle façon de déléguer certaines parties de code à d'autres équipes. Il ne reste pour le responsable de la solution Powershell DSC que le lancement de la commande de démarrage.

Ces deux exemples sont représentatifs de ce que peut amener ce gestionnaire de configuration dans ce mode avancé. Avec encore plus de simplicité et d'efficacité. Il existe d'autres avantages liés à l'utilisation avec par exemple un serveur de rapport ou une intégration parfaite et complète avec le Cloud Microsoft Azure comme expliqué en début d'article. Il serait vraiment dommage de se priver de Powershell DSC !

Pour en savoir plus sur DSC : https://docs.microsoft.com/fr-fr/powershell/scripting/dsc/overview/overview?view=powershell-7

Didier Danse
IT Manager | IT Architect | DevSecOps and Agile Coach

DevSecOps : Les devs et la sécurité, le début d'une belle histoire ?

Partie 1

La sécurité informatique est largement discutée, et ce, depuis les débuts de l'informatique. Les idées reçues au sujet de cette sécurité sont nombreuses et sont bien souvent obsolètes tant les besoins, mais aussi les moyens, évoluent. De ce fait, les objectifs des équipes sécurité ou encore les méthodes appliquées n'ont globalement que peu évolué depuis de nombreuses années.

Pourtant, les risques d'un vol ou d'une altération de données grandissent au quotidien. L'intérêt porté à ces vols de données est également en croissance et les méthodes frauduleuses d'accès à l'information ont évolué en parallèle. A cela s'ajoutent de nouvelles problématiques liées aux nouvelles architectures, et la manière de concevoir des solutions et des applications. Ainsi, l'approche composite de solutions basées sur de nouvelles techniques de développement, de nouvelles architectures, de nouveaux langages mais aussi l'exploitation de solutions open-source introduit de nouvelles failles et de nouveaux risques liés aux licences. De plus, l'accroissement de la fréquence de déploiement réduit le temps disponible pour des tests de sécurité classiques.

En être conscient est une chose, modifier la situation en est une autre. Divers freins au changement sont présents, notamment la dette technique. La perception que le changement doit venir des équipes sécurité, bien souvent sous-dimensionnées, fait que ce changement prend du temps, voire n'est même pas initié. Pourtant, opérer ce changement a bien des intérêts et ce, y compris pour le développeur. Ce même développeur est d'ailleurs très bien placé pour mener ces changements et en profiter pour convertir les problématiques précitées en opportunités qu'il s'agit d'exploiter au mieux.

La sécurité, c'est quoi finalement ?

La notion de sécurité est large. Il est donc nécessaire de prendre un certain nombre d'orientations. Tout en restant relativement traditionnel, avec une approche où le développeur est responsable des applications uniquement, le périmètre s'avère déjà étendu. L'Open Web Application Security Project (OWASP), qui se targue d'être indépendant de tout fournisseur de solutions, propose ainsi de la documentation sur les bonnes pratiques et même des solutions à proprement parler. Au terme d'un long travail, l'OWASP a ainsi publié en 2017 une liste de 10 risques qui sont toujours d'application aujourd'hui :

1. **Injection de données** : l'injection consiste à fournir des scripts, commandes SQL, voire même des commandes au niveau de l'OS non fiables au travers de formulaires, de code, ou encore d'URLs modifiées afin d'exploiter des failles du système.
2. **Authentification brisée et gestion de session** : ce type d'incident est lié à une mauvaise gestion de l'identité de l'utilisateur en exposant des informations au travers des cookies utilisateur ou parfois même dans le code d'une page web. Dans une telle situation, il est possible d'utiliser une session utilisateur de quelqu'un d'autre même si celle-ci a été fermée entretemps.
3. **Exposition de données sensibles** : les données sont présentes tant sur le serveur que sur les applications clientes et transitent entre les deux sur le réseau. Elles peuvent être interceptées si celles-ci ne sont pas correctement protégées, directement en lisant ce qui transite sur le réseau ou encore simplement grâce aux captures d'écran provoquées par une application tierce.
4. **XML External Entity (XXE)** : Un dérivé de l'injection de données est le code XML malicieux. L'XML est analysé par le système et exécute les opérations malicieuses présentes dans le code XML.
5. **Contrôle d'accès brisé** : il permet d'accéder à des ressources qui ne devraient pas l'être, telles que des pages ou des bases de données. Lorsqu'il ne s'agit pas d'une configuration trop permissive, des techniques d'injections permettent ainsi d'exploiter une authentification brisée pour donner accès à ces ressources.
6. **Mauvaise configuration de la sécurité** : en laissant les paramètres par défaut ou en utilisant des mots de passe faibles ou ceux par défaut, des informations inutiles telles que le nom du serveur de rendu applicatif sont dévoilés à tout un chacun. Il s'agit certainement d'un des premiers risques connus et pourtant de nombreuses informations restent toujours exposées.
7. **Cross-Site Scripting (XXS)** : à nouveau, il s'agit d'un type d'injection spécifique qui permet à l'attaquant d'exécuter des scripts pour accéder aux différentes informations présentes dans les cookies ou encore dans le code source de la page. Ainsi, il est possible d'envoyer ces informations vers un site web tiers, bien souvent à des fins frauduleuses en exploitant ce à quoi l'utilisateur accède, sans attaquer le serveur à proprement parler.
8. **Désérialisation non sécurisée** : certaines applications stockent de l'information du côté client. En modifiant ce contenu bien souvent non protégé, il est possible de modifier le comportement de l'application en injectant du code malicieux.

9 **Utilisation de composants avec des vulnérabilités connues** : des composants qui comportent des failles connues ouvrent la porte à tout attaquant qui pourra alors, soit accéder à de l'information, soit altérer le comportement d'une application pour récupérer cette information autrement ou rediriger l'utilisateur vers un autre site.

10 **Logs et surveillance insuffisants** : quelles que soient les mesures mises en place, il se peut qu'une attaque, ou tout du moins une tentative, génère un incident ou un comportement suspect. Ces situations peuvent être identifiées grâce à l'enregistrement et au suivi des activités. Les alertes permettent alors de réagir rapidement, de manière automatique ou non.

Rentrer dans le détail serait trop long. Je vous renvoie vers le *A Developer's Guide to the OWASP Top 10 2017* qui reprend de manière exhaustive le pourquoi et le comment. Par ailleurs, l'*OWASP API Security Top 10* est dérivé de cette liste tout en répondant aux nouvelles architectures orientées services. Il est recommandé d'en tenir compte dès lors que vous êtes amené à fournir des API REST.

Pour limiter l'impact lié à chacun des scénarios décrits ci-dessus, de nombreux tests peuvent, ou doivent, être effectués. Certains de ces risques peuvent aisément être identifiés sur base du code source tandis que d'autres se doivent d'être testés en situation de fonctionnement.

Concrètement, combien de risques listés aviez-vous identifiés ? Et combien sont traités au quotidien ?

En quoi cela me concerne-t-il en tant que développeur ?

Seule, l'équipe sécurité ne peut qu'identifier les risques et les failles sans pour autant les résoudre. Le développeur est en effet le seul à pouvoir le faire. De ce fait, il est important pour les équipes sécurité mais aussi pour le développeur lui-même de pouvoir agir au plus tôt et avec un maximum d'autonomie tout en évitant un impact fort sur son travail.

Durant des années, les équipes sécurité effectuaient des tests sur une application avant de rendre un rapport d'audit plusieurs jours ou plusieurs semaines après cet audit. Ainsi, l'applicatif lui-même était bien souvent déjà remplacé par une nouvelle version. Dans cette situation, le développeur se trouvait face à un dilemme avec, d'une part, le besoin de livrer plus de fonctionnalités, et, d'autre part, de livrer une application plus sécurisée. Souvent, le développeur et même les équipes dirigeantes ont fait le choix de privilégier la partie visible de l'applicatif. Ceci notamment à cause de la durée nécessaire à la mise en œuvre des correctifs, bien trop longue. La situation ne fait donc qu'empirer puisque la dette technique ne fait qu'augmenter. Avant de démarrer, faisons une petite introspection au travers de quelques questions :

- Savez-vous quels composants sont utilisés dans vos solutions ?
- Faites-vous entièrement confiance à l'ensemble des composants de votre application ?
- Des standards sont-ils définis au sein de votre organisation ? Sont-ils respectés par ces composants ?
- Le type de licence utilisé ne met-il pas l'organisation dans une situation risquée ?
- Les composants sont-ils suffisamment sécurisés ? Protègent-ils d'attaques ?

Oui, les architectes se doivent de répondre à certaines de ces questions, tout comme les équipes sécurité. Il n'en demeure pas moins que c'est bel et bien le développeur qui manipule et introduit ces éléments dans les architectures. Le développeur a donc un rôle fort dans le contexte de la sécurité. Il est probable que la réponse à l'ensemble des questions est identique : « non ».

Et l'équipe sécurité là-dedans ?

Les équipes sécurité ne disparaissent pas pour la cause. Certains regretteront que ce ne soit pas le cas mais il n'existe aucune raison valable que cela en soit ainsi. Les équipes sécurité restent l'autorité en charge de la sécurité dans sa globalité tandis que les développeurs et les opérationnels sont quant à eux en charge de réaliser le nécessaire pour fournir des solutions sécurisées.

Dans ce contexte, la sécurité se doit donc de fournir un cadre de travail dans lequel chacun peut se sentir partie prenante et favoriser le *security by design*. Pour y parvenir, il s'agit de former les développeurs en continu, étudier les nouveaux risques, aider l'ensemble des équipes à valider des hypothèses mais aussi à identifier les nouveaux risques informatiques et à communiquer au mieux.

A quel moment tester la sécurité ?

La réponse est simple : À tout moment ! L'objectif est d'identifier et de résoudre le plus rapidement possible. En effet, plus tôt le problème est identifié, moins il coûte. Lorsque l'on parle de coût, il ne s'agit pas uniquement du coût financier mais aussi du coût lié à l'urgence qui peut impacter les personnes impliquées et principalement les développeurs qui seront chargés de corriger le problème. Ainsi une faille identifiée en production peut coûter 100 fois plus qu'un risque identifié durant les phases de développement ou de tests.

Evidemment, tout ne peut pas être identifié directement et les tests se doivent d'être effectués de nombreuses fois. C'est d'ailleurs une des limitations de l'audit : celui-ci s'effectue de manière sporadique. Pour permettre d'avoir des résultats de tests adéquats, il s'agit d'automatiser tout en identifiant la bonne balance entre les tests et la capacité de livrer. A cela s'ajoute le besoin de définir ce qui est testé à quel moment. En effet, un test complet de l'ensemble du code et des applications lors de chaque commit augmente considérablement le temps de compilation et réduit ainsi la capacité à délivrer. Le processus en place doit, en plus, permettre éventuellement de bloquer un composant mais aussi de le libérer rapidement.

Mais au-delà de tester les applications par rapport à de potentielles failles, il s'agit avant tout de connaître et de comprendre son environnement. Pour cela, la documentation initiale s'avère fort utile mais elle ne fournit qu'une vue de la réalité. Disposer d'informations en temps réel au travers de tableaux de bord s'avère d'autant plus important. Ceux-ci peuvent ainsi fournir de l'information sur les composants utilisés, les risques identifiés mais aussi la dette technique. Le développeur lui-même peut ainsi être informé qu'une faille a été identifiée dans un composant qu'il utilise au sein de ses applications.

Si malgré toutes ces mesures, il se peut que des incidents surviennent. Que ce soit parce que l'environnement a évolué, ou encore suite à un comportement inattendu d'un utilisateur qui, peut-être sans le savoir, a ouvert un risque.

La suite dans le numéro 239

Villeret Foyang Keuko,
Software Engineer chez
Margo

Kubernetes et la sécurité des conteneurs

Kubernetes est un système en open-source qui permet d'automatiser le déploiement, la mise à l'échelle et la gestion d'applications conteneurisées. En regroupant les conteneurs d'applications sous forme d'unités logiques, ce système en facilite à la fois la gestion et la découverte. Alors que ce mode d'accès aux ressources de calcul commence à devenir omniprésent du fait de sa vaste adoption par le marché, à laquelle s'ajoutent 15 années d'expérience de l'exécution de charges de travail en mode production auprès de Google, ce succès est contrebalancé par la découverte d'une quantité croissante de vulnérabilités au niveau des conteneurs. Dès lors, la sécurisation et le maintien de normes de conformité deviennent des enjeux extrêmement importants dans ces environnements. Nous évoquerons, dans cet article, la sécurité et les conteneurs de Kubernetes, en nous penchant plus précisément sur les types de solutions que privilégient aujourd'hui les grandes entreprises pour se prémunir contre les problèmes de sécurité visant les conteneurs, ainsi que les outils en open-source auxquels ils accordent leur confiance. Quelles sont les meilleures pratiques techniques pour éviter les failles de sécurité des conteneurs ? Quels sont les outils en open-source disponibles pour détecter ces vulnérabilités ? C'est ce que nous allons voir.

Vulnérabilités de Kubernetes et des conteneurs

Avant d'aborder les stratégies employées par les entreprises pour déjouer les vulnérabilités, il convient de revenir un instant sur les types de failles les plus courants. Le recensement donné sur le site de CVE (Common Vulnerabilities Exposures) cite principalement les cas suivants :

- **Attaques par déni de service, ou DoS** : elles permettaient auparavant à des utilisateurs habilités à adresser des demandes de modifications au serveur d'API de Kubernetes d'envoyer un morceau de programme spécialement fabriqué qui consommait des ressources excessives lors du traitement, entraînant une attaque DoS sur le serveur d'API.
- **Faille des escalades de privilèges** : permettait à tout utilisateur disposant du niveau d'accès requis d'exercer l'ensemble des privilèges d'administrateur sur la moindre unité de matériel informatique ou le moindre nœud exécuté sur un cluster Kubernetes. Il devenait alors possible à un pirate d'élever ses privilèges de manière à acquérir des droits d'administrateur complets sur n'importe quel nœud de calcul exécuté dans un « pod » Kubernetes, autrement dit un ensemble de nœuds partageant les mêmes ressources et le même réseau local.
- **Extraction de flux d'informations** : permettait à un groupe de conteneurs partageant des ressources de stockage et de réseau identiques (pod) sur un nœud donné d'extraire les fichiers journaux depuis n'importe quel autre pod membre de ce nœud, dans la mesure où les requêtes de localisation des journaux de pod ne faisaient l'objet d'aucune vérification. Un attaquant distant pouvait alors exploiter cette faille pour afficher des informations sensibles via des journaux de pod théoriquement inaccessibles.

Ces quelques exemples ne représentent qu'une petite partie des vulnérabilités qui ont été identifiées, rendues publiques et ont fait l'objet de correctifs. Pour autant, chacun sait qu'entre la découverte d'une vulnérabilité, sa publication et l'établissement de rapports et de correctifs, les pirates informatiques ont tout loisir de mener une multitude d'offensives telles que des attaques inter-conteneurs en phase de déploiement, ou encore la prise de contrôle de systèmes à des fins malveillantes.

Comment répondre à ces vulnérabilités au niveau de l'entreprise ?

Les entreprises et services informatiques disposent aujourd'hui d'une pluralité d'options pour prémunir leur organisation contre les faiblesses de leurs logiciels. En règle générale, ces mesures consistent à mettre en place une nouvelle méthodologie de travail et à automatiser les procédures de sécurité.

Modification des processus pour plus d'agilité dans le domaine de la sécurité

L'une de ces opportunités consiste à mettre sur pied une équipe de développement, de sécurité et d'opérations (« DevSecOps ») chargée de mettre en œuvre la sécurité durant l'intégralité de la phase de conception des applications. Certes, les entreprises aimeraient pouvoir confier la sécurité à leurs équipes de développeurs ou de DevOps, mais ce domaine exige des connaissances très spécifiques, telles qu'une bonne connaissance des réseaux ou une solide expertise de la remédiation aux menaces, dont la maîtrise échappe a priori à tout développeur. La préhension de l'ensemble du pipeline depuis la phase de conception jusqu'à l'exécution, ainsi que la garantie d'avoir sous la main une expertise technique de haut niveau, entrent donc systématiquement en ligne de compte. Dans une certaine mesure, une équipe de DevSecOps incite les équipes de sécurité à collaborer avec les équipes de DevOps pour mettre au point l'automatisation des processus de sécurité. De fait, ce mode d'organisation contribue à tenir les développeurs informés de la visibilité de la sécurité, des retours d'information et des menaces connues. Au final, la mise en place d'une DevSecOps permet d'instaurer entre les développeurs, DevOps et équipes de sécurité un langage commun qui favorise la compréhension mutuelle en termes d'application.

Automatisation

L'automatisation est le seul moyen d'harmoniser le rythme de progression entre les équipes de sécurité et de DevOps. La mise à l'échelle des conteneurs au niveau de l'entreprise ne peut se contenter de politiques et de listes de contrôle de sécurité statiques ; le pipeline a besoin de services de sécurité renforcés. Grâce à l'automatisation, il devient possible de déployer des

conteneurs évolutifs et des politiques de conformité sécuritaire à une échelle massive. Pour automatiser un flux de travaux dans le pipeline alors que les conteneurs sont morcelés au niveau du réseau, les entreprises averties spécifient le mode de comportement et d'interaction des services au moment de l'exécution, afin de décider quelle politique de sécurité il convient d'appliquer pour les automatiser.

Pour que l'efficacité de la sécurité soit garantie à l'échelle d'une organisation, celle-ci doit s'automatiser jusqu'au stade où elle devient capable de générer des règles de sécurité de façon autonome, puis elle doit définir un vocabulaire commun permettant d'évoquer l'infrastructure en termes de code. Ainsi, la configuration doit, par exemple, être délivrée sous la forme d'un morceau de code source ou de sécurité en tant que code. Imaginez qu'une atteinte à la sécurité soit automatiquement détectée, consignée, corrigée, testée et déployée pendant que votre personnel informatique est en plein sommeil… Votre système pourrait alors se réparer lui-même, en rassemblant toutes les informations pertinentes pour découvrir si une attaque est survenue, ainsi que sa provenance et ce, sans perdre le moindre temps de disponibilité. Par ailleurs, l'automatisation de la sécurité signifie un temps moyen de réaction raccourci de plusieurs semaines à quelques jours, quelques heures, voire quelques minutes seulement, dans la mesure où le temps de déploiement par les développeurs est réduit.

Règles de bonne pratique en matière de sécurité des conteneurs

Si les années récentes nous ont appris quelque chose, c'est que les failles de sécurité informatique sont inévitables. Les vulnérabilités ont beau être signalées, cela n'empêche pas de nombreuses organisations de continuer à adopter le prochain produit, la prochaine mise à niveau ou le correctif suivant. « Cette fois-ci, c'est sécurisé pour de bon », espèrent-elles. Or, à ce jour, il n'en a jamais été ainsi. La seule manière de faire efficacement des affaires dans un monde où règne l'insécurité consiste à mettre en place des processus qui reconnaissent l'insécurité inhérente aux produits. Pour protéger les ressources des entreprises contre les pirates qui cherchent à tirer profit des vulnérabilités, et pour réduire leurs risques d'exposition, quels que soient les produits ou correctifs concernés, les organisations doivent mettre en œuvre des règles de bonne pratique en matière de sécurité. Examinons-en quelques-unes :

1• Confiance et restriction

Ayez toujours à votre disposition des images réalisées à partir de sources fiables, via une authentification d'image par une signature numérique ou par le biais d'un système de stockage à registre privé. Ces images doivent être analysées afin d'évaluer les vulnérabilités courantes et connues (CVE), les configurations non sécurisées, les fuites d'informations d'identification, les droits d'accès insuffisants aux fichiers sensibles, la conformité aux normes sectorielles (PCI, NIST, SIC), ainsi que les licences de logiciels non prises en charge. Ce passage au crible doit permettre de déceler les vulnérabilités contenues dans toutes les couches de l'image, et non uniquement dans sa couche de base.

Cette analyse des images est indispensable, mais elle n'est encore pas suffisante : si les images des conteneurs sont immuables, ce n'est pas le cas de leur mode d'exécution. Lorsqu'un conteneur arrive en phase de production, il se peut que de nouvelles vulnérabilités soient découvertes, que les logiciels présentent des défauts, que des modifications soient intervenues dans les données et les droits d'accès, ou que des fuites de mémoire se soient produites. Les outils de contrôle de configuration automatisés sont alors capables de scanner l'ensemble de l'environnement Kubernetes afin d'évaluer sa conformité aux lignes directrices établies, telles que les évaluations comparatives ou « benchmarks » de Kubernetes.

2• Inspection et sécurisation du réseau

Le réseau représente l'aspect le plus critique en matière de sécurité des conteneurs. Certaines capacités de protection telles que les pare-feu et mesures de sécurité au niveau des nœuds finaux doivent également être étendues aux conteneurs et à l'ensemble du trafic est-ouest. Kubernetes, de même que tout autre outil d'orchestration de conteneurs, repose sur une configuration de sécurité par défaut qu'il est généralement très important de passer en revue, comprendre et paramétrer en fonction des besoins.

3• Réduction de la surface d'attaque au strict minimum

La surface d'attaque correspond à la somme des différents points à partir desquels un utilisateur non autorisé peut saisir des données ou les extraire de votre environnement. S'agissant des conteneurs, la surface d'attaque est généralement plus restreinte, dans la mesure où votre Docker vous permet de sélectionner une image de distribution allégée telle qu'Alpine, qui est conçue dans une perspective de sécurité, de simplicité et de performances des ressources. Sur ce type de conteneur, la surface d'attaque est réduite puisque seules les bibliothèques essentielles sont présentes.

Recommandations de sécurité en phase de pré-déploiement

Cette liste de meilleures pratiques de sécurité est loin d'être exhaustive et peut différer d'une organisation à l'autre d'après les exigences spécifiques rencontrées. Au demeurant, il est recommandé aux équipes DevOps d'observer certaines mesures de sécurité préalablement au déploiement, de manière à garantir la création d'images efficaces, protéger chaque nœud et prévenir les menaces telles que les configurations à risque, fuites d'informations d'identification ou droits inadéquats sur les fichiers sensibles. Citons notamment les recommandations et lignes directrices générales suivantes :

- Isoler les conteneurs au moyen d'espaces-noms afin de prévenir les attaques par escalade de privilèges.
- Limiter les capacités de Linux de manière à éviter l'exécution de processus en tant qu'utilisateur root. Pour les conteneurs dont les processus doivent obligatoirement être exécutés en tant qu'utilisateur racine à l'intérieur du conteneur, vous pouvez affecter à cet utilisateur un niveau de privilèges moins élevé.
- Activer SELinux, qui est une version à sécurité renforcée de Linux mise au point par la NSA appliquant une politique de sécurité supplémentaire à tous les processus.
- Activer le mode Secure Computing (seccomp), une fonctionnalité du noyau Linux destinée à restreindre les actions disponibles à l'intérieur du conteneur.
- Configurer les Cgroups de manière à limiter une application à un ensemble spécifique de ressources.
- Utiliser un système d'exploitation hôte

réduit au minimum, en privilégiant des images minimales contenant uniquement les outils et bibliothèques systèmes indispensables à votre projet, afin de réduire la surface d'attaque au strict minimum.
- Mettre à jour les correctifs du système.
- Exécuter des tests de sécurité avec des comparatifs CIS.
- Ne jamais insérer de code secret dans un fichier Docker.
- Ne jamais transférer le port privilégié (n° 22).
- Ne pas forcer la directive User.
- Ne pas effectuer de copier/coller.
- Utiliser une image de base digne de confiance.
- Éviter les packages non indispensables.
- Disposer d'images immuables reproductibles, pour éviter d'obtenir des résultats variables entre différentes exécutions.
- Ne pas utiliser la « toute dernière » version en production. Dans le cas contraire, vous ne pourrez pas effectuer de déploiement régressif. Il n'est pas possible de revenir au « dernier niveau -1 ». Mieux vaut donc libeller vos images avec des balises porteuses de sens.

Outils de sécurité en open-source de Kubernetes

Bien que les outils disponibles dans le commerce proposent une protection et une visibilité à portée multivectorielle, certains projets en open-source continuent d'évoluer et de se doter de fonctionnalités de sécurité nouvelles. Voici quelques solutions à envisager pour les projets moins stratégiques en production :

- **Falco** : Falco est un projet en open-source né de la volonté de comprendre le comportement des conteneurs et de protéger votre plateforme contre les activités malveillantes potentielles. Le moteur de règles peut détecter une activité anormale dans les applications, les conteneurs, l'hôte sous-jacent et la plateforme de conteneurs. Falco fait partie intégrante de l'architecture du moteur de réponse Kubernetes, qui traite les événements de sécurité en contrant les atteintes à la sécurité au moyen de réponses tactiques.
- **Istio** : Istio crée un maillage de services pour gérer les communications, y compris le routage, l'authentification et le chiffrement, mais ce logiciel n'est pas conçu comme un outil de sécurité destiné à détecter les attaques et les menaces.
- **Anchore** : Anchore est une solution d'analyse de conteneurs dotée de fonctions de génération de rapports et de mise en conformité sur la base de règles.
- **Clair** : Clair est un projet en open-source qui procède à une analyse statique des vulnérabilités dans les conteneurs applicatifs.

Conclusion

La sécurité des conteneurs émerge parmi les préoccupations importantes, voire essentielles exprimées dans les enquêtes récentes, car elle concerne désormais la phase de production des services réels. Les entreprises prennent conscience de la nécessité de sécuriser le flux de travaux complet au sein de leur pipeline afin d'éviter de s'exposer à une diversité de menaces. À l'instar de n'importe quel autre logiciel, Kubernetes n'est pas à l'abri des incidents de sécurité. L'adoption massive des applications conteneurisées sur le marché ajoute la vulnérabilité des images et des conteneurs à la liste des problèmes de sécurité courants, ainsi qu'à celle des nouvelles menaces découvertes. Pour contrer ces failles, les entreprises averties doivent automatiser et rendre plus agiles leurs processus, garantir leur conformité par des meilleures pratiques de sécurité, sécuriser leur plateforme de déploiement et maîtriser les problèmes de sécurité à grande échelle.

Liens de référence :

https://www.cvedetails.com/product/34016/Kubernetes-Kubernetes.html?vendor_id=15867
Fix of directory traversal on kubectl
https://sysdig.com/blog/oss-container-security-runtime/#understandingresponseengineandsecurityplaybooks
https://learn.cisecurity.org/benchmarks
https://neuvector.com/container-security/kubernetes-security-guide/

DevOps : objectif agilité & sécurité

Alain Hélaïli
Principal Solutions Engineer, GitHub

L'heure est à l'accélération des projets de transformation numérique, à la généralisation des architectures Cloud et à l'accroissement de la transparence, notamment avec l'open source. Le déploiement de solutions informatiques doit donc être rapide, tout en respectant les exigences de sécurité. À ce titre, DevOps favorise une collaboration entre développeurs, ingénieurs de production et responsables métiers, indispensable pour garantir agilité business et sécurité aux clients.

DevOps : buzzword ou transformation en profondeur ?

Il y a encore quelques années, une démarche DevOps se résumait, grosso-modo, à donner aux développeurs l'accès direct à l'environnement de production. Cette première étape a permis d'extraire le développeur de son silo et l'associer à la mise en œuvre des opérations IT ; ce fut le début d'un consensus mutuel entre « Ops » et « Devs » sur la nécessité d'automatiser les tâches répétitives de mise en production. Accélération numérique oblige, la démarche DevOps a profondément modifié l'organisation même du travail. Dans sa définition la plus simple, DevOps est un ensemble de pratiques permettant de livrer des applications, des logiciels ou d'autres outils informatiques en production, de manière plus rapide et plus fiable. Les fondamentaux sont l'automatisation, mais également la collaboration et le partage. Là où auparavant les équipes se passaient la main, elles doivent maintenant travailler ensemble et se faciliter mutuellement le travail.

Une nouvelle approche

DevOps est une approche et non pas une famille d'outils. Ce vocable n'est donc pas juste une mode ou le buzzword de ces 10 dernières années, malgré certains abus de langage et la généralisation des titres « ingénieur DevOps ». Il serait plus juste de parler du concept de Site Reliability Engineer (SRE) qui reflète bien mieux la nouvelle répartition des responsabilités. Le rôle des Ops est de créer l'autoroute permettant le déploiement rapide et simple (donc automatisé) du code depuis le repository vers la production et d'identifier les solutions architecturales. À ce titre, l'évolution la plus symptomatique de l'environnement de travail des Ops a été le recul de l'importance de l'outil de service desk/ticketing et l'adoption des outils de gestion de version. Les Ops ont ainsi adopté des techniques de Dev.

Une évolution indispensable pour garantir la sécurité

L'approche DevOps est devenue une nécessité dans un environnement économique en constante accélération qui fait la part belle aux start-ups et aux géants de la tech. Pour rester dans la course, les entreprises doivent opter pour le cloud, le SaaS et l'ouverture de leurs systèmes d'information via des Apps mobiles ou des APIs. Mais cette médaille a un revers. Les organisations font face à des risques sécuritaires majeurs car leurs applications sont exposées sur le web. Elles sont composées majoritairement de briques logicielles open source qui érigent, à raison, la transparence comme argument principal de la sécurité et de la confiance. Dès lors, la communauté informatique a dû remettre en question les dogmes mêmes sur lesquels elle avait bâti son fonctionnement. Elle a été obligée de renoncer à placer l'environnement de production dans une bulle de protection au nom de la sacro-sainte croyance : « moins on y touche, plus on garantit la sécurité ».

Remettre sans cesse le métier sur l'ouvrage

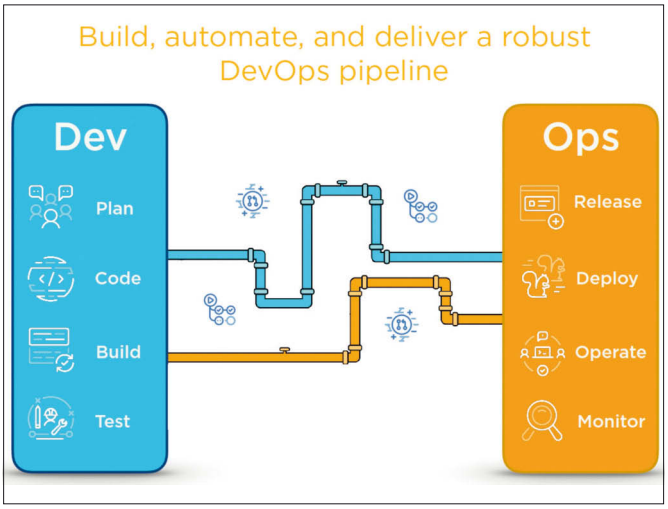
Voilà pourquoi l'impératif de réactivité, essentiel à la survie des entreprises, doit s'imposer dans l'agenda des DSI ! Par exemple, ne pas mettre à jour ses environnements de production régulièrement, voire de manière réactive et à la demande, c'est laisser la possibilité aux hackers, même novices, d'exploiter des failles, sachant qu'elles sont connues et documentées car déjà corrigées dans les versions récentes. En 2017, avec un coût (amendes, remédiations, indemnités…) estimé à 1,4 Milliard de dollars[i], une entreprise comme Equifax a payé au prix fort cette passivité. Les différentes études sur DevOps (State of DevOps par DORA notamment) montrent que, contrairement aux habitudes et croyances héritées du passé, ce sont les équipes qui déploient le plus fréquemment qui obtiennent la meilleure stabilité. Même les applications sont concernées, une fois finalisées : la découverte d'une vulnérabilité telle que « Zip Slip », par exemple, nécessite un examen immédiat du code applicatif, ainsi que la correction et le déploiement d'une nouvelle version pour lui éviter d'être prise pour cible.

Une collaboration interdisciplinaire indispensable

Pour aller plus loin dans la démarche DevOps, les entreprises ne doivent plus attendre que le code soit considéré comme « prêt pour la production » avant d'être déployé. En d'autres termes, elles doivent être capable de tester en production afin de valider des hypothèses métier et technique. Cette approche requiert des fonctions d'observabilité, d'aiguillage/filtrage d'accès aux fonctionnalités (feature toggle) et d'isolation du code. Celles-ci sont déjà disponibles sur le marché mais pas assez répandues. Grâce à ces techniques, on dissocie l'action de mise en production du code d'avec l'évènement de mise à disposition d'une fonctionnalité. La granularité des déploiements est plus fine, avec des impacts moindres et plus facilement identifiables. Mais, au-delà des bénéfices techniques, ce niveau de maturité DevOps apporte véritablement un nouveau levier d'agilité. La notion de sprint disparaît. De plus, un réel partenariat entre le métier et les développeurs peut se mettre en place. Il est ainsi possible d'expérimenter rapidement, tester des implémentations ou dissocier des expériences en fonction des utilisateurs.

Une communication fluide et transparente

Depuis une dizaine d'année, les Dev sont décisionnaires en matière de choix d'outils et d'environnement informatique, auparavant du ressort des Ops, qu'il s'agisse de



serveurs applicatifs, puis de bases de données et maintenant, avec l'essor des microservices et des conteneurs, d'architectures d'exploitation complètes. Paradoxalement, alors que dans l'idéal les équipes Ops auraient dû faire disparaître l'infrastructure et les problématiques de déploiement du quotidien des développeurs, ces derniers s'engouffrent massivement dans la vague Kubernetes. Le processus de développement doit donc, plus que jamais, être un système qui favorise et unifie les échanges entre développeurs, métier, sécurité et Ops. Ce n'est plus ce quelque chose de mystérieux qu'on alimente avec des pizzas et des post-it. Les principes de visibilité et traçabilité du code, qu'il soit applicatif ou d'infrastructure, doivent également s'appliquer aux décisions architecturales et fonctionnelles car leur impact est omniprésent. Au-delà de la technique, une communication fluide et transparente est le signe extérieur ultime de la maturité DevOps.

Bien plus qu'un mot à la mode, DevOps est devenu une véritable nécessité. Véritable levier d'agilité, cette approche représente une opportunité unique de faire bouger les lignes dans la relation entre métier et technique, au-delà des pratiques agiles stéréotypées habituelles. Néanmoins, DevOps ne peut pas faire l'impasse sur la sécurité qu'il contribue d'ailleurs à améliorer en favorisant son intégration tout au long du cycle de développement. La sécurité doit donc faire partie intégrante de la collaboration entre développeurs, opérationnels et experts de ce domaine.

[i] https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473

Maria Pashkina
maria.pashkina@gmail.com

Intégrer ONLYOFFICE à une application GED en Python

ONLYOFFICE est une suite bureautique en ligne à code source ouvert. Elle propose un logiciel de traitement de texte, un tableur et un outil de présentation.

Cette suite permet de toujours garder le contrôle sur tous vos fichiers. Cela assure la prise en charge rapide et efficace des documents complexes de tous les formats courants en fournissant les outils performants d'édition et co-édition en temps réel. La suite est accessible en ligne, en version bureau (Linux, Mac et Windows) ou en applications mobiles pour iOS et Android. Il s'agit d'une alternative à G Suite, Office 365, etc.

ONLYOFFICE permet de gagner en productivité et offre une expérience utilisateur fluide à travers :

- La lecture et édition des fichiers .docx, .xslx et .pptx. Le format OOXML utilisés comme base, assure une haute compatibilité avec les fichiers Microsoft Word, Excel et PowerPoint.
- L'édition des autres formats courants (.odt, .rtf, .txt, .html, .ods., .csv, .odp) via leur conversion aux formats OOXML.
- Une interface familière en français organisée sous forme d'onglets. 1
- Outils de collaboration : deux modes de co-édition (rapide et stricte), suivi de modifications, commentaires et chat intégré. 2
- La gestion des droits d'accès : accès complet, lecture seule, révision, remplissage de formulaire et commentaire. 3
- Le développement de vos propres modules complémentaires à base des API fournis par ONLYOFFICE.
- 250 langues disponibles et un nombre des alphabets hiéroglyphiques.

La suite bureautique peut être intégrée à des sites web ou des applications écrites en différents langages. Elle fournit les API permettant aux développeurs d'ajouter l'édition de documents à leurs plateformes de stockage, de partage et de collaboration.

Pour intégrer les éditeurs de la suite, nous avons besoin d'un connecteur (une application d'intégration) qui va servir de pont reliant ONLYOFFICE Document Server et le service tiers (site ou application). Le connecteur ajoute les outils de rédaction à la plateforme tierce, permettant de travailler sur les fichiers directement depuis son interface et de paramétrer les éditeurs.

Pour rendre l'intégration possible, il faut accorder à ONLYOFFICE les autorisations suivantes :

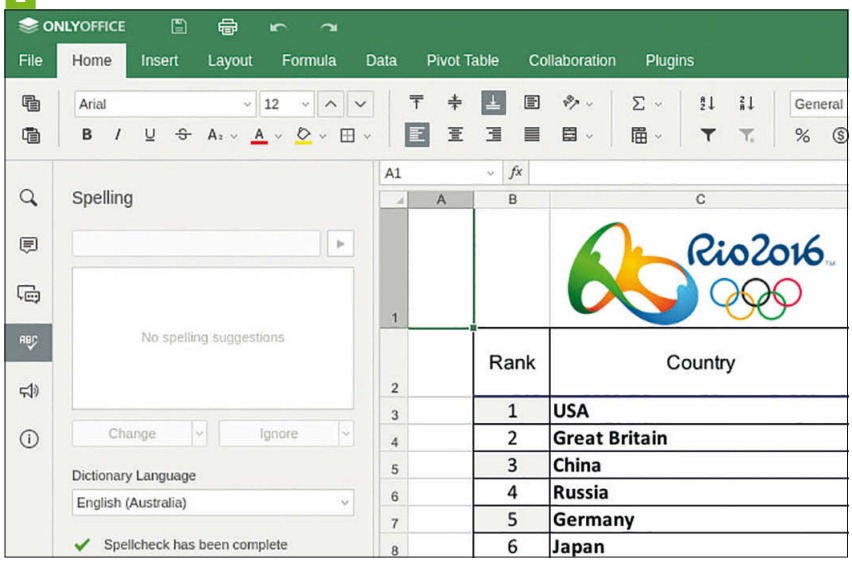
- ajouter et exécuter le code :

La suite ne modifie pas le code source. En effet, pour effectuer l'intégration, le service doit permettre d'ajouter les modules utilisateur, les modules complémentaires, les packs ou les applications.

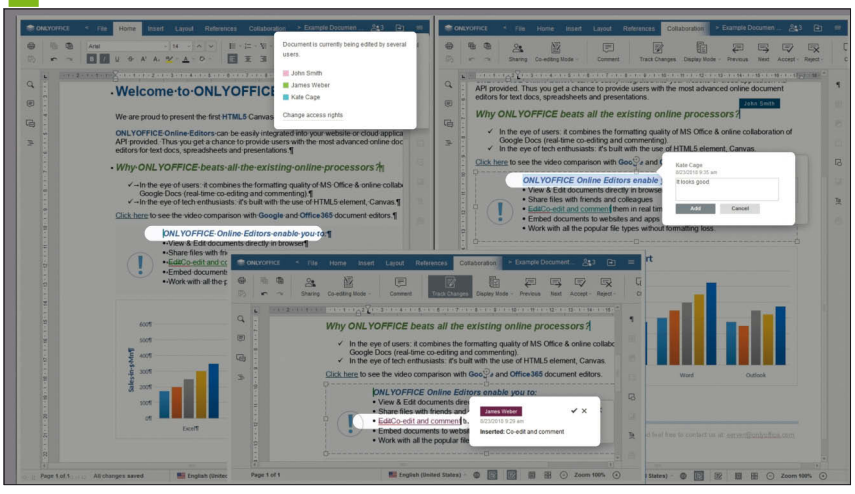
Afin d'éviter toute modification des paramètres accidentelle ou indésirable, Document Server chiffre les demandes par une signature numérique sous la forme de token. La sécurité y est assurée au moyen de la technologie JWT. Les données sont vérifiées à l'aide d'une signature numérique. Le Document Server ajoute JWT à la demande en utilisant le mot de passe de son fichier de configuration. A son tour, le service le vérifie en utilisant le mot de passe de son fichier de configuration. La fonction de rappel exécute la demande seulement dans le cas où la signature est correctement validée.

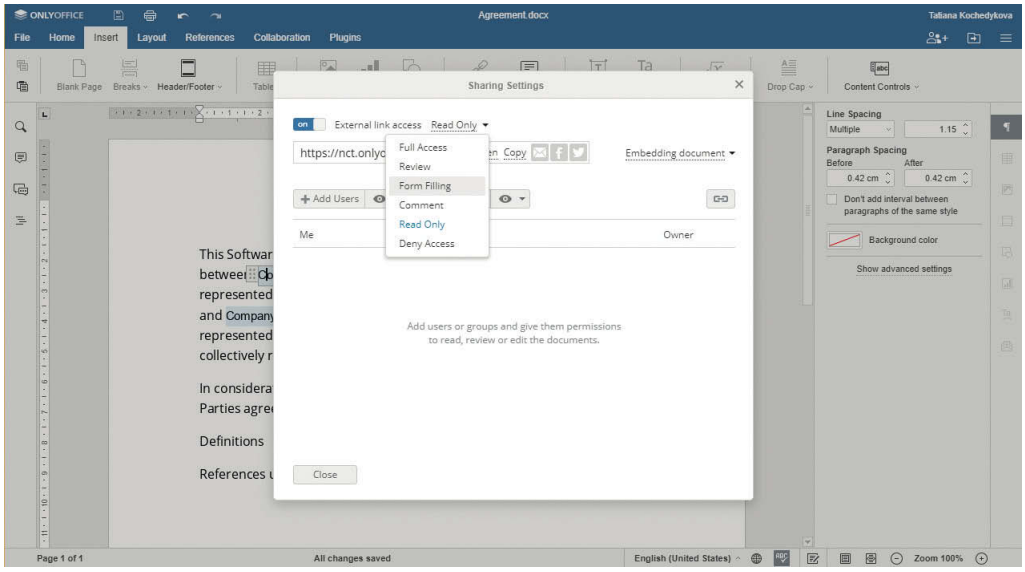
- accéder aux fichiers pour les télécharger et enregistrer sans demander aucune donnée d'utilisateur (comme, par exemple, les

1



2





3

cookies du navigateur),

- ajouter des boutons à l'interface utilisateur,
- ouvrir une nouvelle page où ONLYOFFICE puisse exécuter le script pour ajouter un éditeur,
- paramétrer l'application.

Il existe plusieurs cas d'intégration réussie avec les plateformes populaires de stockage et partage de fichiers et de collaboration. La suite fournit aux utilisateurs des connecteurs officiels prêts à l'emploi pour Nextcloud, ownCloud, Alfresco, Confluence, SharePoint. En même temps il existe plusieurs connecteurs développés par les développeurs tiers comme Pydio, eXo, Xwiki, Jalios, etc., qui peuvent vous servir d'exemple.

Le cas de l'intégration le plus fréquent c'est le déploiement des éditeurs en ligne avec la plate-forme de collaboration du même concepteur écrite en C#. Cette plateforme de collaboration inclut la gestion des documents et des projets, le CRM, l'agrégateur de mail, le calendrier, la base de données des utilisateurs, les blogs, les sondages, la base de connaissances avec le balisages de wiki, la messagerie instantanée.

Dans cet article nous visons à montrer le mécanisme de l'intégration dans la Gestion Électronique des Documents (la GED) développée en Python, un des langages de programmation les plus populaires. Afin de rendre les éditeurs opérationnels depuis l'interface de la GED, nous devons les doter de fonctionnalités suivantes : ouverture du fichier pour la lecture, édition du document, enregistrement du document. Voyons de plus près chacune de ces étapes.

Installation des éléments nécessaires à l'intégration

Préparons les composants clés de l'intégration : Document Server et l'application (la GED) en Python.

- Installons Document Server avec les éditeurs. Vous pouvez utiliser Docker ou les packs snap, .deb ou .rpm. Il est recommandé d'installer Document Server et toutes les dépendances à l'aide du Docker :

```
docker run -itd -p 80:80 onlyoffice/documentserver-de
```

- Si vous avez une application GED en Python, assurez-vous qu'elle correspond aux conditions suivantes :
  - l'application contient plusieurs fichiers à ouvrir pour la lecture ou l'édition,
  - l'application permet de télécharger les fichiers.

Sinon, on peut la créer en utilisant le framework Bottle : *pip install bottle* .

Ensuite, il nous faut créer les fichiers suivants : *main.py* (le code de l'application) et *index.tpl* (le modèle de fichier), et ajouter le code ci-dessous dans le fichier *main.py* :

```
from bottle import route, run, template, get, static_file # connecting the framework and the necessary components
@route('/') # setting up routing for requests for /
def index():
    return template('index.tpl') # showing template in response to request
run(host="localhost", port=8080) # running the application on port 8080
```

Une fois que l'application est lancée, une page vide s'affiche à l'adresse IP *http://localhost:8080* .

Pour que le Document Server puisse générer de nouveaux documents, nous devons créer les fichiers par défaut et faire la liste de leurs titres dans le fichier modèle. Nous créons un dossier *files* avec trois fichiers dedans (.docx, .xlsx et .pptx).

La méthode *listdir* permet de lister tous les noms de fichiers d'un répertoire.

```
from os import listdir
```

Créons maintenant une variable pour tous les noms de fichiers du dossier *files* :

```
sample_files = [f for f in listdir('files')]
```

Nous allons utiliser la méthode *template* pour utiliser cette variable dans le fichier modèle :

```
def index():
  return template('index.tpl', sample_files=sample_files)
```

Pour voir la variable dans le fichier modèle :

```
%for file in sample_files:
  <div>
    <span>{{file}}</span>
  </div>
% end
```

Une fois que l'application est redémarrée, la liste des noms s'affiche sur la page. Dès maintenant nous pouvons mettre ces fichiers à disposition de tous les utilisateurs.
Voici la méthode pour le faire :

```
@get("/files/<filepath:re:.*\.*>")
def show_sample_files(filepath):
  return static_file(filepath, root="files")
```

Maintenant nous avons tous les composants pour passer à l'intégration.

Comment ouvrir les documents depuis la GED écrite en Python

Avant tout, connectons l'API des éditeurs dans le fichier modèle :

```
<script type="text/javascript" src="editor_url/web-apps/apps/api/documents/api.js">
</script>
```

editor_url c'est le lien vers les éditeurs des documents.
Le bouton pour ouvrir chaque fichier pour la lecture :

```
<button onclick="view('files/{{file}}')">view</button>
```

Ensuite nous ajoutons un élément * * avec *id* :

```
<div id="editor"></div>
```

L'éditeur du document s'ouvrira dans ce * * après l'appel à la fonction suivante :

```
<script>
function view(filename) {
  if (/docx$/.exec(filename)) {
    filetype = "text"
  }
  if (/xlsx$/.exec(filename)) {
    filetype = "spreadsheet"
  }
  if (/pptx$/.exec(filename)) {
    filetype = "presentation",
    title: filename
  }
  new DocsAPI.DocEditor("editor",
    {
      documentType: filetype,
      document: {
        url: "host_url" + '/' + filename,
        title: filename
      },
      editorConfig: {mode: 'view'}
    });
}
</script>
```

Il y a deux arguments pour la fonction de l'éditeur du document : *id* de l'élément où les éditeurs seront ouverts et un *JSON* contenant les paramètres de l'éditeur.
Dans cet exemple, nous utilisons les paramètres par défaut tels que :

- *documentType* est le type du fichier qui se détermine par son format (.docx, .xlsx, .pptx pour les documents texte, les feuilles de calcul et les présentations respectivement).
- *document.url* est le lien vers le fichier à ouvrir.
- *editorConfig.mode*.

Ajoutons aussi *title* qui est le nom du fichier qui s'affichera dans les éditeurs.
Maintenant nous avons tous les éléments nécessaires à la lecture des fichiers depuis l'interface de notre application en Python.

Comment éditer les documents ?

Ajoutons le bouton " Éditer " :

```
<button onclick="edit('files/{{file}}')">edit</button>
```

Nous avons besoin de créer une nouvelle fonction pour ouvrir les documents à l'édition. La création de cette fonction est semblable à celle que nous venons d'effectuer pour la lecture.
Maintenant nous avons 3 fonctions :

```
<script>
  var editor;
  function view(filename) {
    if (editor) {
      editor.destroyEditor()
    }
    editor = new DocsAPI.DocEditor("editor",
      {
        documentType: get_file_type(filename),
        document: {
          url: "host_url" + '/' + filename,
          title: filename
        },
        editorConfig: {mode: 'view'}
      });
  }
  function edit(filename) {
    if (editor) {
      editor.destroyEditor()
    }
    editor = new DocsAPI.DocEditor("editor",
      {
        documentType: get_file_type(filename),
        document: {
```

```
            url: "host_url" + '/' + filename,
            title: filename
          }
        });
  }
  function get_file_type(filename) {
    if (/docx$/.exec(filename)) {
      return "text"
    }
    if (/xlsx$/.exec(filename)) {
      return "spreadsheet"
    }
    if (/pptx$/.exec(filename)) {
      return "presentation"
    }
  }
</script>
```

destroyEditor ferme un éditeur ouvert.
Le paramètre *editorConfig* a par défaut une valeur {"*mode*": "*edit*"}, c'est pourquoi la fonction *edit()* n'y figure pas.
Maintenant nous pouvons ouvrir les fichiers pour les éditer depuis l'interface de la GED développée en Python.

Enregistrer les documents

Quand nous travaillons sur le document, ONLYOFFICE enregistre toujours toutes les modifications. Une fois que l'éditeur est fermé, le Document Server crée la version du fichier à enregistrer et envoie une demande à l'adresse *callbackUrl*. Cette demande contient *document.key* et le lien vers un fichier récemment créé.
Nous utilisons *document.key* pour accéder à une version précédente du fichier et la remplacer par une version la plus à jour. Comme nous n'avons pas de base de données, nous envoyons juste le nom du fichier en utilisant *callbackUrl*.
Spécifions *callbackUrl* dans les paramètres de *editorConfig.callbackUrl*. Une fois ce paramètre ajouté, la méthode *edit()* s'affiche de la manière suivante :

```
function edit(filename) {
    const filepath = 'files/' + filename;
    if (editor) {
      editor.destroyEditor()
    }
    editor = new DocsAPI.DocEditor("editor",
      {
        documentType: get_file_type(filepath),
        document: {
          url: "host_url" + '/' + filepath,
          title: filename,
          key: filename + '_key'
        }
```

```
        ,
        editorConfig: {
          mode: 'edit',
          callbackUrl: "host_url" + '/callback' + '&filename=' + filename  // add file name
as a request parameter
        }
      });
  }
```

Maintenant nous avons besoin d'écrire une méthode pour enregistrer le fichier après la requête POST à l'adresse de */callback* :

```
@post("/callback") # processing post requests for /callback
def callback():
  if request.json['status'] == 2:
    file = requests.get(request.json['url']).content
    with open('files/' + request.query['filename'], 'wb') as f:
      f.write(file)
  return "{\"error\":0}"
```

 # status 2 est le fichier créé.
Après la fermeture de l'éditeur la version du fichier la plus à jour est enregistrée.
En suivant les étapes de ce tutoriel, vous pouvez désormais créer votre propre application d'intégration qui permet à ONLYOFFICE d'accéder aux fichiers de la GED, les ouvrir pour l'édition et enregistrer. De plus, le connecteur permet d'établir les paramètres avancés de la rédaction collaborative tels que la gestion des permissions d'accès des utilisateurs aux fichiers et co-édition des documents en temps réel.

A propos de la licence

ONLYOFFICE est distribuée sous une licence double. La licence open source et gratuite est sous GNU AGPL v3. Si les éditeurs sont utilisés en tant qu'une partie de votre solution cloud ou auto-hébergée, la licence commerciale est requise. Il y a aussi les exemples de l'usage commercial. Par exemple, PowerFolder, éditeur de solutions cloud recourt à ONLYOFFICE pour intégrer des éditeurs bureautiques en ligne.

Liens utiles

Site officiel : https://www.onlyoffice.com/fr/
Documentation : https://api.onlyoffice.com/editors/basic
Les dépôts des applications d'intégration sur GitHub :
https://github.com/ONLYOFFICE/onlyoffice-nextcloud
pour ownCloud https://github.com/ONLYOFFICE/onlyoffice-owncloud
pour Alfresco https://github.com/ONLYOFFICE/onlyoffice-alfresco
pour Confluence https://github.com/ONLYOFFICE/onlyoffice-confluence
pour SharePoint https://github.com/ONLYOFFICE/onlyoffice-sharepoint

PROGRAMMEZ! Le magazine des développeurs

Offres printemps 2020

Nos classiques

1 an 11 numéros	49€*
2 ans 22 numéros	79€*
Etudiant 1 an - 11 numéros	39€*

** Tarifs France métropolitaine*

Abonnement numérique

PDF ..... 35€
1 an - 11 numéros

Option : accès aux archives 15€

Souscription uniquement sur
www.programmez.com

1 an
11 numéros
+ Python et l'analyse forensique (éditions ENI)
50€*

2 ans
22 numéros
+ Python et l'analyse forensique (éditions ENI)
80€*

** Offres limitées à la France Métropolitaine*

Toutes nos offres sur www.programmez.com

Oui, je m'abonne

ABONNEMENT à retourner avec votre règlement à :
Service Abonnements PROGRAMMEZ, 4 Rue de Mouchy, 60438 Noailles Cedex.

☐ **Abonnement 1 an :** 49 €
☐ **Abonnement 2 ans :** 79 €
☐ **Abonnement 1 an Etudiant :** 39 €
Photocopie de la carte d'étudiant à joindre

OFFRES PINTEMPS 2020
☐ **Abonnement 1 an :** 50 €
☐ **Abonnement 2 ans :** 80 €

PROG 238
Valable jusqu'au 31 mars 2020

☐ Mme ☐ M. Entreprise : Fonction :

Prénom : Nom :

Adresse :

Code postal : Ville :

email indispensable pour l'envoi d'informations relatives à 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**

Diana Ortega
Diana est une développeuse back-end avec plusieurs années d'expérience dans l'écosystème JVM et le langage Go. Organisatrice du meetup Women Who Go Paris, elle a vécu de nombreux projets cloud & data et est passionnée par le développement logiciel, la data et le craftsmanship

Stanislas Michalak
Stanislas est développeur backend chez Molotov TV à Paris. Il contribue à des projets Open Source tels que Buffalo, un écosystème Web en Go. Autodidacte passionné, fan d'ergonomie, d'automatisation et d'optimisation, il pratique le Go depuis 3 ans et cherche à partager sa passion

Go : comment développer votre application ?

Dans le numéro 196 de Programmez, nous avons commencé à parler du langage Go. Depuis, de l'eau a coulé sous les ponts et Go a continué d'évoluer. C'est donc le bon moment pour refaire un point sur ce langage et faire le tour des dernières nouveautés !

C'EST QUOI LE LANGAGE GO ?

Un peu d'histoire

Go est un langage **compilé et fortement typé**, conçu par Robert Griesemer, Rob Pike et Ken Thompson au sein de Google. La première version publique est sortie le 10 novembre 2009.

Ses 10 ans ont été célébrés l'an passé et ses parents peuvent être fiers ! Il est actuellement utilisé dans des projets très connus comme Docker, Kubernetes ou Terraform. Sa simplicité, sa facilité de prise en main et sa capacité à compiler rapidement en font un outil de choix pour créer des (micro-)services à l'heure du Cloud.

La communauté Go est très active : au rythme d'une release tous les six mois, il en est à sa **version 1.14**. Une équipe chez Google est dédiée à sa maintenance. C'est aussi un langage Open Source qui accueille de nombreux contributeurs.

Enfin, Go a une mascotte : le Gopher !

Qui l'utilise ?

Une liste des projets utilisant Go est accessible sur Github https://github.com/golang/go/wiki/GoUsers.

Nous comptons parmi ces projets Docker, Kubernetes et Terraform que nous avons déjà cités, mais aussi des entreprises telles que Uber, Datadog… et également en France : Zenly, Molotov TV ou encore Scaleway.

Pourquoi ?

Go est un langage rapide à apprendre et à comprendre. Des outils standards comme *gofmt* permettent de mettre son code en conformité avec des conventions de style strictement établies.

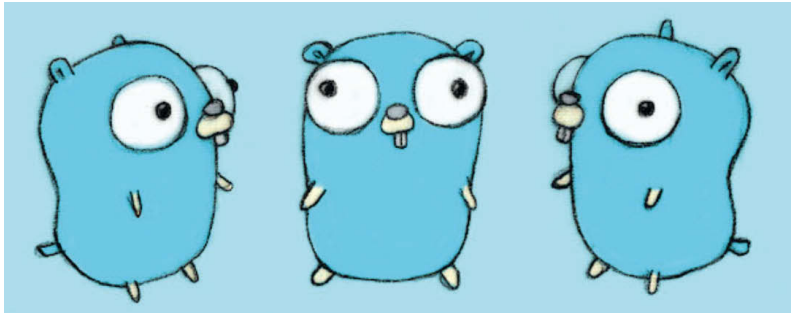
Go gère nativement la concurrence avec les *goroutines* et les *channels*. C'est un langage fortement typé qui se compile vite. Les temps de réponse peuvent se compter en microsecondes (pour du code correctement optimisé) et l'empreinte mémoire est très faible.

Le langage possède une librairie standard qui évolue régulièrement sans casser la rétro-compatibilité. La communauté est conséquente, elle entretient de nombreux projets et des bibliothèques variées.

Go possède désormais un outil standard de gestion de dépendances, dont nous parlerons plus loin. Il supporte nativement la compilation vers de multiples plateformes et produit des binaires auto-suffisants. Ceci signifie qu'il n'y a pas besoin d'installer de dépendances, un interpréteur ou une machine virtuelle.

Cependant, il vous faudra un environnement capable d'exécuter des binaires (du VPS aux conteneurs Docker).

On pourrait continuer cet éloge longtemps, mais le mieux est de le constater par vous-même via un peu de pratique. Que diriez-vous de créer votre premier projet en Go ?



Le Gopher, dessiné par Renee French.

CRÉER MON PREMIER PROJET GO

Installer Go

Ça y est, vous allez développer en Go ! La première chose à faire est d'installer Go : pour cela, rendez-vous sur le site https://golang.org/ et téléchargez la version qui correspond à votre système. En installant le paquet téléchargé, vous disposerez du compilateur et des outils (ou *toolchain*) qui vont vous permettre de travailler avec Go.

Si vous voulez apprendre le langage sans installer Go tout de suite, vous pouvez utiliser le playground (https://play.golang.org/). Ce site web permet de compiler votre code directement depuis le navigateur web et donc de le tester sans n'avoir rien à installer !

Cependant, pour suivre correctement cet article, nous vous conseillons d'installer Go sur votre machine.

Outillage et Écosystème

Choisir un éditeur de code

Vous pouvez écrire du Go dans n'importe quel éditeur, mais voici quelques choix populaires au sein de la communauté :

- **Visual Studio Code** : l'éditeur **gratuit** fourni par Microsoft est l'un des plus utilisés par les Gophers. Il propose une extension pour le langage Go avec tout l'outillage nécessaire et quelques intégrations (pour lancer ses tests depuis l'éditeur, par exemple). Rendez-vous sur https://github.com/Microsoft/vscode-go/wiki/All-Settings-&-Commands-in-Visual-Studio-Code-Go-extension pour plus d'informations sur la configuration de cette extension.
- **GoLand** : un IDE **payant** fourni par Jetbrains (l'éditeur d'IntelliJ IDEA, un IDE très populaire dans le monde *Java*). C'est un environnement dédié à Go, qui fournit une intégration assez

avancée avec le langage. Consultez https://www.jetbrains.com/help/go pour plus d'informations sur cet éditeur.
- **vim** : si vous êtes déjà habitué à vim, il est possible de l'utiliser. Un plugin tel que fatih/vim-go (https://github.com/fatih/vim-go) propose des fonctionnalités avancées pour travailler avec Go.

Peu importe l'éditeur que vous choisissez, il est probable qu'il utilise un outil commun aux autres éditeurs : *gopls*. C'est ce que l'on appelle un **serveur de langage** : une brique logicielle qui gère de manière centralisée l'autocomplétion, la recherche de définitions et d'autres fonctionnalités intelligentes spécifiques au langage. L'intérêt d'un tel outil est de concentrer les efforts des développeurs d'éditeurs de code sur une seule brique centrale, partagée par langage. Vous pouvez trouver plus d'informations sur ce sujet sur https://langserver.org/.

Mon premier programme

***Note** : par convention, nous utiliserons le symbole $ pour signaler le début d'une commande Si nous parlons de saisir la commande « $ echo », vous devrez donc saisir « echo » dans un terminal.*

Exploration de l'outil « Go »

Avant de commencer notre projet, nous allons explorer l'outil de gestion de code Go.
Pour vérifier que tout s'est bien passé lors de l'installation de Go, ouvrez un terminal de commandes et entrez :

```
$ go help
```

Vous obtenez la liste des commandes disponibles et leur description, ainsi qu'une liste de topics. Si vous entrez :

```
$ go version
```

Vous devez voir une ligne indiquant la version de Go installée. À la date d'écriture de cet article, il s'agit de la version 1.14.

Exploration des variables d'environnement

***Note** : historiquement, certaines variables ont été un sujet de discussion sur plusieurs versions du compilateur. Nous nous permettons d'aborder le sujet des variables d'environnement avant même de vous expliquer comment créer votre premier projet Go.*

Une variable d'environnement est une variable déclarée dans le contexte du système d'exploitation. Elle est en dehors du logiciel et peut être partagée avec d'autres programmes (c'est le cas de la variable *PATH*, le chemin d'accès aux exécutables). C'est une alternative assez commune aux fichiers de configuration.
Pour lister les variables d'environnement utilisables avec Go, vous pouvez saisir dans votre console :

```
$ go env
```

Cette commande vous montre une liste de variables et leurs valeurs. Habituellement, un développeur Go n'est pas amené à changer la plupart de ces variables, à moins de vouloir utiliser des fonctionnalités plus avancées.
Parmi les variables les plus utilisées, vous pouvez trouver :

- *GOROOT* qui contient le chemin d'installation de votre version de Go.
- *GOPATH* qui contient le chemin de votre espace de travail par défaut. Dans les **anciennes versions** de Go, il était nécessaire de lui donner une valeur vous-même, et il fallait placer son code dans un sous-répertoire de cet espace de travail.
- *GOARCH* et *GOOS* qui spécifient l'architecture et le système d'exploitation pour lequel vous compilez. Nous allons en parler plus en détail dans la prochaine section.
- D'autres variables telles que celles préfixées par CGO servent à la compilation du code C dans du code Go. Certains drivers de bases de données (*sqlite* par exemple) nécessitent l'utilisation de ce mode de compilation.

Création du projet

Initialisation du projet

Depuis sa version 1.11, Go est livré avec un système de gestion de dépendances intégré : les modules. À partir de la version 1.13, c'est le système utilisé par défaut.
Pour créer votre premier projet, vous allez utiliser la commande *go* :

- Ouvrez un terminal de commandes et naviguez vers le répertoire où vous voulez créer votre projet. Cela peut être n'importe où dans votre ordinateur, mais **évitez** de le faire dans **le GOPATH**, cela activerait le mode de compatibilité. Vous pouvez vérifier la valeur de votre GOPATH avec la commande :

```
$ go env GOPATH
```

Créez un répertoire avec le nom de votre projet (par exemple *monapp*), placez-vous dans ce nouveau dossier. Nous appelons ce dossier la racine ou ***root (en anglais)* de votre projet**.
Puis tapez :

```
$ go mod init monapp
```

Ça y est ! Vous venez de créer votre premier projet Go en utilisant les modules.
Dans le répertoire *monapp*, un nouveau fichier ***go.mod*** a été créé. Si vous ouvrez ce fichier et regardez son contenu, vous constatez qu'il y a un en-tête avec le nom de votre projet et la version de Go présente sur votre ordinateur.

```
module monapp
go 1.14
```

Cette version de Go sera utilisée pour compiler le projet. Il sera nécessaire de la changer si vous souhaitez utiliser des fonctionnalités proposées qu'à partir d'une certaine version.

Passons au code

Dans un projet Go, les fichiers de code sont répartis à l'intérieur de répertoires. Chaque répertoire représente un *package*. Le nom du *package* est déclaré dans l'entête de chaque fichier contenant du code (comme le code *Java*). Un logiciel exécute en premier le *package main* qui représente le point d'entrée du programme. Il ne peut donc y en avoir qu'un seul par programme.
Créons maintenant le répertoire *hello*. À l'intérieur de celui-ci, nous allons créer un nouveau fichier *main.go*. Nous avons désormais la structure suivante :

```
monapp
  |-hello
    |-main.go
```

Le fichier *main.go* contient le code de notre premier programme. Nous allons créer notre premier *hello world* :

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

Il est temps de tester ce programme ! Pour cela :

- Ouvrez un terminal de ligne de commandes.
- Naviguez jusqu'à la racine (*root*) de votre projet (répertoire *monapp*).

Lancez la commande :

```
$ go run ./hello/main.go
```

En principe, vous obtenez le résultat suivant :

```
hello world
```

La commande go run compile le code source puis l'exécute. Cette méthode est utile pour tester le code au fur et à mesure, mais elle ne permet pas de produire un exécutable pour votre environnement de production.

Décomposons le code

Comme nous l'avons mentionné précédemment, la première ligne de code est :

```
package main
```

Tous les fichiers de code débutent par la directive « package » suivie d'un nom (sans chemin d'accès, contrairement au *Java*). La norme consiste à prendre comme nom de package le nom du répertoire où le fichier est stocké.

Viennent ensuite les imports de dépendances.

```
import "fmt"
```

Une dépendance est du code fourni par d'autres développeurs. Ici nous utilisons le package « fmt » fourni par la librairie standard de Go qui permet, entre autres, d'afficher du texte à l'écran. Les dépendances importées sont précédées du mot-clé *import* et chaque librairie est écrite entre guillemets.

Finalement nous avons déclaré une fonction de nom *main*

```
func main() {
    fmt.Println("hello world")
}
```

Pour déclarer une fonction il faut écrire le mot-clé func suivi de la définition de la fonction. *main* est le nom de la fonction principale de votre application.

Importer des dépendances

Nous avons utilisé la dépendance « fmt » dans notre programme, elle est fournie par la librairie standard. Si vous voulez inclure du code en dehors de la librairie standard, vous allez devoir utiliser un gestionnaire de dépendances : les modules.

Nous avons déjà initialisé notre projet en utilisant les modules : vous vous souvenez du fichier *go.mod* ? Grâce à ce fichier, Go sait quelles sont les dépendances externes de votre projet. Pour le reste, il suffit d'importer dans votre code les dépendances qui vous intéressent et Go se chargera du reste.

Voici les commandes les plus utilisées pour manipuler les modules :

- go mod init : initialiser un projet avec les modules.
- go mod tidy : scanner le projet à la recherche des dépendances manquantes dans le fichier *go.mod* et retirer celles inutiles.
- go mod download : télécharger les dépendances dans le cache local (qui se trouve dans le dossier *$GOPATH/pkg/mod*).

> ***Note :*** *si votre connexion à Internet est derrière un proxy, vous aurez besoin de configurer votre environnement pour télécharger les dépendances. Visitez* https://github.com/golang/go/wiki/GoGetProxyConfig *pour plus d'informations.*

Pour vous entraîner avec les dépendances, nous allons transformer notre hello world en une API REST. L'idée de cet exercice est de créer un service qui, au lieu d'afficher du texte sur la console, répond un message HTTP. Cela a l'avantage de faciliter la communication avec le monde extérieur, comme c'est le cas pour le monde du *Cloud*.

Commençons par importer les packages nécessaires à notre code :

```
import (
    "github.com/gorilla/mux"
    "log"
    "net"
    "net/http"
)
```

Nous avons éliminé l'import du package « fmt », car Go ne compile pas si nous laissons une dépendance non utilisée. Ensuite, nous avons importé une librairie externe, « github.com/gorilla/mux », très utilisée pour créer des *routers*. Un *router* est le code qui permet d'acheminer les demandes (*requests*) des utilisateurs en faisant le lien entre un chemin d'URL et le code à exécuter.

Ensuite, nous utilisons plusieurs packages de la librairie standard de Go : « log » pour gérer les logs (écrire en console ou dans un fichier tout ce qui se passe dans notre application). Les dépendances « net » et « net/http » gèrent la couche réseau du service REST.

À ce stade, vous pouvez lancer la commande suivante dans la console, sous la racine de votre projet (*monapp*) :

```
$ go mod tidy
```

Si vous regardez le fichier *go.mod*, vous verrez une nouvelle ligne pour « github.com/gorilla/mux » avec sa version.

> ***Note :*** *les commentaires simples ou d'une seule ligne en Go, commencent par // . Nous utiliserons des commentaires dans le code, pour vous aider à comprendre son contenu.*

Maintenant, créons une fonction prenant deux paramètres :
- un *http.ResponseWriter* pour écrire une réponse HTTP aux clients du service.
- une *http.Request* pour lire les paramètres de la requête HTTP.

```
func ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // écrire dans le header de la réponse le code 200
    w.WriteHeader(http.StatusOK)
    w.Header().Set("Content-Type", "application/json")

    // écrire dans le body un message Json
    _, err := w.Write([]byte(`{"message": "hello world"}`))

    // gestion d'erreur
    if err != nil {
        log.Fatal(err)
    }
}
```

Finalement, passons à la fonction main :

```
func main() {
    //1
    log.Println("initializing hello world api")

    //2
    router := mux.NewRouter()
    //3
    router.HandleFunc("/", ServeHTTP)

    //4
    server := http.Server{
        Addr:    net.JoinHostPort("", "8080"),
        Handler: router,
    }

    //5
    err := server.ListenAndServe()
    if err != nil {
        log.Fatal(err)
    }
}
```

Voici le détail de ce que nous venons d'écrire :

1 • Utilisation de la librairie « log » pour afficher dans la console le message "initializing hello world api".

2 • Création d'une variable de type *mux.router*. La notation := implique la déclaration et l'initialisation d'une nouvelle variable.

3 • La méthode *HandleFunc* reçoit un chemin pour la méthode REST et une fonction. Nous allons utiliser ici la fonction *ServeHTTP* que nous venons de créer. En Go, les fonctions sont des objets de première classe : cela veut dire que nous pouvons créer des fonctions qui reçoivent d'autres fonctions (pour gérer des *callbacks*, par exemple) ou qui renvoient des fonctions.

4 • Création d'une variable de type *http.Server*. Il s'agit d'une structure qui définit un serveur HTTP. Nous allons lui passer *Addr* avec le port d'écoute et *Handler* avec le router déclaré dans le point 2.

5 • Démarrage du serveur, avec la méthode *ListenAndServe*.

Bravo ! Vous avez codé un Service REST, écrit en Go, qui répond à un appel HTTP GET. Vous pouvez le lancer de la même manière que précédemment, avec la commande *go run*. Vous pouvez tester le service en ouvrant un navigateur web à l'url "http://localhost:8080".

Mais allons plus loin. Regardons comment générer un binaire de votre programme qui ne dépend pas de Go pour fonctionner et qui soit exécutable sur d'autres machines.

Compiler mon projet

Go est un langage compilé, cela veut dire qu'il est nécessaire de passer par un processus de compilation pour traduire le code écrit et générer un binaire exécutable. Comme chaque plateforme est différente, il faut compiler le code pour chaque plateforme où il va s'exécuter.

L'exécutable d'un programme Go n'a pas de dépendance pour tourner dans un autre système : tout est inclus dans le binaire final. Il suffit de copier l'exécutable et voilà, notre code fonctionne sur une autre machine avec le même OS et la même architecture.

Nous avons vu comment lancer du code Go (avec *go run*), voici maintenant comment le compiler. Depuis la racine de votre projet, exécutez :

```
$ go build -o bin/ ./hello/
```

Ou

```
$ go build -o bin/ ./...
```

Ces commandes génèrent une version exécutable du code ou binaire. Le flag -o permet de spécifier le dossier où les binaires seront générés : dans l'exemple ci-dessus, il y aura un seul exécutable qui se trouvera dans le répertoire *bin* de votre projet. Maintenant, vous pouvez exécuter le binaire, et le tester en ouvrant un navigateur web avec l'url "localhost:8080".

Parmi les modifications apportées avec la version 1.13, il est possible de compiler plusieurs packages avec l'option -o.

Les binaires générés sont, par défaut, compatibles avec la plateforme de votre ordinateur. Par exemple, si vous utilisez un système d'exploitation *linux* sur *amd64*, votre code sera compilé pour cette architecture.

Cependant, il est aussi possible de créer un exécutable pour une machine différente. Go prend en charge la compilation croisée (*cross-compilation*) : les variables d'environnement *GOOS* et *GOARCH* détaillées précédemment permettent de choisir l'environnement pour lequel le binaire sera compilé. Voici par exemple comment générer un binaire compatible avec *linux* sur *amd64* :

```
$ GOOS=linux GOARCH=amd64 go build -o ./bin/ ./...
```

Pour obtenir la liste complète des combinaisons système d'exploitation / architecture supportées par votre compilateur, vous pouvez exécuter :

```
$ go tool dist list
```

WebAssembly, une promesse à venir de Go ?

Si vous avez exécuté l'instruction précédente, vous avez

probablement remarqué que parmi la liste des combinaisons supportées par le compilateur, il y en a une qui est très particulière : js/wasm.
Depuis la version 1.11 de Go, il est possible de compiler nos programmes Go pour *WebAssembly*.
WebAssembly (WASM) définit un format d'instruction pour une machine virtuelle. C'est un standard qui est supporté par la plupart des navigateurs web, dans le but de permettre au code *Javascript* d'exécuter du *bytecode* ou code compilé.
Comme le standard WASM définit une cible de compilation, ce *bytecode* peut être généré par des langages de haut niveau comme C, C++, *Rust* et maintenant Go.
Bien qu'il soit possible de développer des applications WASM avec Go, cette fonctionnalité n'est pas encore assez robuste pour être utilisée en production.

Trouver les bugs dans mon code

Delve

Comme pour les autres langages de programmation, il y a plusieurs manières de trouver des *bugs* présents dans son code.
La première méthode est **celle du log** : vous placez des *fmt.Println ("test")* pour afficher « test » (ou autre chose, suivant votre état d'énervement) dans la console et vérifier par où le programme passe avant le bug. L'avantage de cette méthode, c'est qu'elle est rapide à mettre en place, et qu'en général, vous trouverez des pistes rapidement. L'inconvénient, c'est qu'il faut recompiler le code à chaque fois pour rajouter des traces et penser à les retirer après (ça ne fait pas propre et ça ralentit l'exécution).
La seconde méthode est **celle du debugger** :

- Vous utilisez un outil dédié qui va instrumenter votre code.
- Vous pouvez placer des points d'arrêt pour arrêter l'exécution du code à des endroits définis.

Cette méthode a le gros avantage de montrer l'état des variables à l'endroit où vous avez arrêté le code, et donne la possibilité de continuer l'exécution pas à pas si vous tâtonnez sur la position du bug.
Delve (https://github.com/go-delve/delve) est le débogueur le plus utilisé par la communauté Go. C'est un équivalent à *gdb* (en C/C++) mais adapté à Go. Cet outil est généralement intégré dans les éditeurs supportant Go. Consultez https://github.com/Microsoft/vscode-go/wiki/Debugging-Go-code-using-VS-Code pour son utilisation dans *Visual Studio Code*, ou https://blog.jetbrains.com/go/2019/02/06/debugging-with-goland-getting-started/ pour GoLand.. Il suffit souvent de cliquer dans la marge à côté du code pour positionner un *breakpoint* et de lancer votre programme en mode debug. Suite à des modifications récentes du compilateur, l'information obtenue par le débogueur est plus précise, ce qui permet un meilleur debug des binaires.

Tests

Une bonne manière de se prémunir des *bugs* est d'écrire des tests. En Go, le support des tests est natif : il n'est pas nécessaire d'inclure une dépendance pour les gérer, comme *JUnit* en *Java* par exemple. Pour écrire un test, vous pouvez commencer par créer un fichier avec le suffixe « _test.go ». Cela indique au compilateur de l'ignorer lors de la compilation de votre exécutable. En général vous utilisez le nom du fichier contenant la fonction à tester, et vous ajoutez le suffixe indiqué (par exemple, pour tester *gopher.go* vous créez *gopher_test.go*).
Nous allons maintenant écrire un test ensemble. Voici le contenu du fichier *gopher.go* :

```
package gopher

func Plural(s string, count int) string {
    if count == 1 {
        return s
    }
    return s + "s"
}
```

Le but de cette fonction est d'ajouter un « s » à la fin d'une chaîne de caractères, si l'argument count est différent de 1.
Nous allons créer le fichier *gopher_test.go* pour écrire notre test avec le contenu suivant :

```
package gopher

import "testing"
```

Comme vous pouvez le voir, nous avons importé les outils de test (de la bibliothèque « testing »). Maintenant, nous définissons une fonction qui prend en paramètre un type **testing.T*. Ce paramètre permet de notifier le *framework* de test de l'état du test pendant l'exécution. Si vous écrivez par exemple :

```
func TestPlural(t *testing.T) {
    t.Fail()
}
```

Votre test sera marqué comme échoué. Vous pouvez exécuter les tests avec la commande suivante et vérifier leur état :

```
$ go test ./...
--- FAIL: TestPlural (0.00s)
FAIL
FAIL    gopher    0.002s
FAIL
```

Il est temps de terminer notre test. Nous allons définir le résultat attendu pour différents paramètres d'entrée et le faire échouer si le résultat ne correspond pas :

```
func TestPlural(t *testing.T) {
    // tableau qui contient les cas de test
        cases := []struct{
            Input string
            Count int
            Output string
        }{
            {Input: "tomate", Count: 1, Output: "tomate"},
            {Input: "tomate", Count: 2, Output: "tomates"},
        }

        // exécution des test
        for _, c := range cases {
            res := Plural(c.Input, c.Count)
```

```
            if res != c.Output {
                t.Errorf("expected %s, got %s", c.Output, res)
            }
        }
}
```

Cette manière d'écrire un test avec un tableau est très courante. Elle permet d'ajouter des cas de test rapidement et de garder le test lisible. Si vous exécutez les tests, ils devraient maintenant passer avec succès :

```
$ go test ./...
ok  gopher   0.003s
```

Vous pouvez tester avec d'autres paramètres, et vérifier que le test passe toujours !

Benchmarking

Imaginons qu'un de vos collègues, après une revue de code, vous dise que vous pouvez améliorer la fonction *Plural* et la rendre plus rapide, en utilisant la fonction *strings.Builder* pour créer de manière efficiente des *strings*. Cette fonction est disponible à partir de la version 1.10 de Go.

Vous implémentez une nouvelle fonction que vous appelez *Plural2*, ayant la même signature. Son implémentation utilise cette fois *strings.Builder*. Pour vérifier que votre collègue a raison, vous mesurez la performance de ces deux implémentations de la fonction pour choisir celle que vous convient le plus : vous faites un *benchmark*.

Go dispose d'un outil pour faire du *benchmarking* : cet outil fait aussi partie du package de testing.

Créer un benchmark est très similaire à créer un test et vous devez les écrire dans vos fichiers «*_test.go».

Reprenons notre exemple : pour *benchmarker* vos fonctions, ajoutez dans votre fichier *gopher_test.go* deux fonctions, une pour l'exécution de *Plural* et une autre pour l'exécution de *Plural2*. Voici un exemple :

```
func BenchmarkPlural(b *testing.B) {
 for i := 0; i < b.N; i++ {
   Plural("gopher",3) //changer par Plural2
 }
}
```

À l'instar des tests, la fonction de *benchmark* doit commencer par le mot « Benchmark » suivi par le nom du test. Le paramètre de la fonction est du type **testing.B*.

Le benchmark sera exécuté *b.N* fois. La valeur de *b.N* est incrémentée automatiquement jusqu'à ce que le benchmark converge vers un résultat stable.

Pour lancer le *benchmark* vous utilisez la commande suivante :

```
$ go test ./… -bench=.
```

Cela exécute tous les tests et tous les *benchmarks* dans les fichiers de test. Dans notre cas nous obtenons des résultats similaires à la ligne suivante :

```
BenchmarkPlural-8     100000000          21.2 ns/op
```

- La première colonne contient le nom de la fonction de *benchmark* suivie par "-C". C' est le nombre de coeurs du processeur de votre machine (8) ;
- La deuxième colonne indique le nombre de fois que la fonction s'est exécutée (100000000) ;
- La troisième colonne contient le temps de chaque itération de votre benchmark (21.2 nanosecondes par boucle).

Avec ces résultats vous pouvez constater quelle fonction est la plus performante : dans notre cas, celle qui prend le moins de temps par itération. Vous pouvez également écrire un *benchmark* orienté consommation mémoire, si c'est la ressource que vous cherchez à optimiser. En général, vous ne pouvez pas optimiser toutes les ressources, il faudra faire des choix.

> ***ATTENTION :*** *lorsque vous optimisez vos applications, il est nécessaire de faire des compromis. Parfois le gain n'est pas si important. Généralement, vous échangez des améliorations de performances avec de la lisibilité. N'oubliez pas que réduire la lisibilité du code entraîne des problèmes de maintenabilité.*

Détection de conflits de concurrence

Lorsque vous travaillez avec du code concurrent (par exemple avec l'utilisation des *Goroutines*), vous pouvez rencontrer des conflits de concurrence. C'est un problème qui arrive lorsque vous essayez de modifier et/ou de lire une valeur en même temps, dans deux contextes différents.

Par exemple, si vous définissez une variable globale qui compte le nombre de requêtes HTTP et qu'elle est incrémentée à chaque requête dans une fonction qui tourne en parallèle (*Goroutine*), il arrive un moment où deux utilisateurs du serveur vont envoyer une requête et tenter de modifier le compteur en même temps.

Pour détecter ce type de problèmes, Go propose de manière native une option à passer au compilateur : *-race*. Cette option va demander au compilateur d'instrumenter le code pour détecter les *race conditions*, c'est-à-dire les possibilités d'accès concurrent dans notre code. Si une telle chose se produit, notre programme va planter avec une erreur indiquant le problème d'accès concurrent.

Utilisation

- Pour éviter de rencontrer le problème lorsque le code est compilé ou en production, vous pouvez utiliser *-race* avec la commande de test. À partir de ce point, il suffit d'écrire vos tests comme d'habitude. Ils échoueront à l'exécution si un accès concurrent est détecté.

```
$ go test ./... -race
```

- Vous pouvez aussi lancer votre code avec l'option *-race*. Vous verrez le rapport de *race condition* pendant l'exécution.

```
$ go run -race ./hello/main.go
```

- Vous pouvez aussi compiler votre projet avec cette même option *-race*, mais il sera moins performant et consommera plus de mémoire. Évitez donc d'utiliser cette option en production et réservez-la pour vos environnements de test.

```
$ go build -race ./hello/main.go
```

Tracing

Avec Go, il est possible de diagnostiquer des problèmes de CPU, mémoire ou concurrence de vos applications, grâce aux outils de collecte des statistiques d'utilisation de ressources. Ces statistiques sont générées en format lisible par *pprof*, l'outil de Go pour les visualiser.

Pprof

Pprof est un outil de visualisation, mais fournit aussi des librairies pour faciliter la génération et l'accès à ces métriques. La plus connue est « net/http/pprof », une librairie haut niveau qui se branche directement en HTTP sur votre application. Elle facilite l'accès à vos statistiques.
Il suffit d'importer dans votre code le package « net/http/pprof » et d'ajouter les endpoints :

```
router.HandleFunc("/debug/pprof/", pprof.Index)
router.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
router.HandleFunc("/debug/pprof/profile", pprof.Profile)
router.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
```

Vous pouvez alors interroger ces URLs pendant l'exécution de l'application, en utilisant l'outil *pprof* de la manière suivante :

```
$ go tool pprof http://localhost:8080/debug/pprof/profile
```

L'outil *pprof* va collecter des statistiques, afin d'analyser votre application en tournant de manière interactive. Vous pouvez utiliser des commandes comme *top* ou *web* pour visualiser l'information récupérée par l'outil.

Tracing avec benchmarks

Vous pouvez aussi obtenir des métriques à partir de vos benchmarks. Pour ce faire, exécutez les benchmarks en utilisant les options « -cpuprofile », « -memprofile », « -blockprofile » ou « -mutexprofile » :

```
$ go test -bench=. -cpuprofile=./bin/cpuprofile.out \
gopher/gopher_test.go gopher/gopher.go
```

La commande précédente exécute les benchmarks et écrit la sortie des statistiques du CPU dans le fichier ./bin/*cpuprofile.out*. Pour explorer ces statistiques, vous utilisez l'outil *pprof* :

```
$ go tool pprof -http=:5000 ./bin/cpuprofile.out
```

Depuis la version 1.13 de Go, il est possible d'obtenir des métriques personnalisées pour vos *benchmarks*, grâce à l'*api* « Benchmark. ReportMetric ». Pour cela, vous devez coder le comportement de votre métrique à l'intérieur de votre *benchmark*. Vous aurez alors comme résultat une nouvelle colonne qui contient l'information demandée.

LE FUTUR DE GO

Go 2

En 2018 a été lancée la construction de Go 2 : un langage plus puissant, plus simple, contenant tout ce dont la communauté aurait besoin au niveau du langage et de l'outillage. Bien qu'encore en travaux, Go 2 inspire déjà Go : de grands changements ont commencé à sortir avec Go 1.11 et l'idée d'avoir un langage complètement différent a été abandonnée très tôt, pour garantir un certain niveau de rétrocompatibilité entre les différentes versions du langage.
Grâce à l'existence d'un seul compilateur et une seul équipe chez Google focalisée dans l'implémentation du langage, une version stable sort tous les six mois. Les dernières versions ont apporté des améliorations dans le runtime et des nouvelles fonctionnalités, notamment dans la gestion de dépendances avec *go modules*.
Voici les modifications les plus récentes et intéressantes :

- Go a désormais une manière standard de décorer les erreurs avec leur trace. Cela facilite la gestion des erreurs en cascade, vu que jusqu'à présent, nous ne pouvions avoir que la trace de la dernière erreur. Pour activer ce nouveau système, Go met à disposition un nouveau type de format possible pour les erreurs : %w.
- Depuis la version 1.13, TLS 1.3 est activé par défaut, et la version 1.14 améliore ce support. SSLv3 est quant à lui déprécié à cause des risques de sécurité.
- Des améliorations dans le système de modules, notamment dans le support des répertoires *vendor*.
- Des améliorations dans la librairie des logs pour ajouter des préfixes aux messages d'erreur.
- Dans la version 1.15, des améliorations des outils de testing sont prévues.

Et beaucoup de travail dans le runtime et le tooling !

Conclusion

Go est un langage puissant et simple. Il permet de construire des systèmes robustes de manière standard et lisible, car beaucoup de complexité est cachée par le runtime. C'est un langage très facile à apprendre, avec une communauté bienveillante. Un développeur venant du monde *Java*, *Python* ou C pourra le prendre en main très facilement.
Go, c'est aussi un langage en évolution. Les dernières versions ont changé de manière très significative l'expérience de développement : les modules, la gestion d'erreurs, mais aussi les avancées dans le *tooling* et le *runtime* vont dans ce sens. Il y a un grand engagement de l'équipe de Go, mais aussi de la communauté en général qui travaillent ensemble pour construire un meilleur langage.

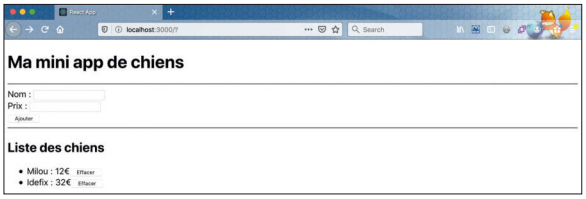
Thierry LERICHE
Architecte et tech lead
@ThierryLeriche

React Hooks : State Vs Reducer

Les Hooks sont arrivés avec React 16.8. Ils permettent de bénéficier d'un état local (et pas seulement) sans avoir à écrire de classe. Le Hook le plus utilisé est certainement useState mais d'autres méritent également votre attention, à commencer par useEffect et useReducer…

Le projet d'exemple

Je vous propose une app très simple(1) qui manipule des chiens. Elle est composée d'un mini formulaire pour créer un chien, en haut, et de la liste des chiens créés, en bas. La liste des chiens déjà existants est chargée au démarrage. Enfin un bouton, sur chaque ligne, permet de supprimer le chien correspondant.



L'app utilise une API sur un serveur pour gérer les chiens. Pendant l'appel, elle affiche un sablier (*spiner*). Et en cas d'erreur, l'app affiche le message correspondant.

Composant

Je vous propose d'abord une version écrite à l'ancienne, avec des classes héritant de *Component*, qui servira de base de départ. La classe *Dogs* est le point central. Elle utilise, sans surprise, les classes *DogForm* pour le formulaire et *DogList* pour afficher la liste des chiens. La liste des chiens, l'état du sablier (*loading*) et l'éventuelle erreur sont conservés dans le *state* et passés en *props* aux sous-composants.

```
class Dogs extends React.Component {

  state = {
    dogs: [],
    loading: false,
    error: null
  }

  componentDidMount() {
    ...
  }

  addDogHandler = dog => {
    ...
  }

  removeDogHandler = dogId => {
    ...
  }

  render() {
    return (
      <div>
        {this.state.error && <ErrorBox errorMessage={this.state.error} />}
        <h1>Ma mini app de chiens</h1>
        <hr />
        <DogForm onAdd={this.addDogHandler} />
        <hr />
        <DogList
          dogs={this.state.dogs}
          onRemove={this.removeDogHandler}
          loading={this.state.loading} />
      </div>
    );
  }
}
```

Je place le chargement de la liste initiale des chiens dans la fonction *componentDidMount* mais il y a d'autres choix possibles. C'est donc là que je lance l'appel à l'API. Mais avant, je change le *state* de *loading* pour que le sablier s'affiche. La fonction *apiFetch* est une surcouche qui simplifie l'utilisation de *fetch*. Une fois la réponse reçue(2), je stocke la liste des chiens dans le *state* et je désactive le sablier. Et bien entendu, si jamais il y a une erreur, je passe dans le *catch* et je change le *state* en conséquence pour l'afficher.

```
componentDidMount() {
  this.setState({
    loading: true
  });

  apiFetch('https://some-website.com/api/dogs', {
    method: 'GET'
  })
    .then(dogs => {
      this.setState({
        dogs,
        loading: false
      });
    })
    .catch(error => {
      this.setState({
        error,
        loading: false
      });
    });
}
```

La fonction *addDogHandler* est celle que je passe au sous-composant *DogForm* et qui sera appelée en retour lorsque l'utilisateur clique sur le bouton d'ajout.
Pour commencer, je change le *state* de *loading* pour que le sablier s'affiche. Puis j'appelle l'API avec le verbe *POST*, en passant le chien dans le *body*. Une fois la réponse reçue, je mets à jour la liste des chiens dans le *state* en ajoutant le chien reçu(3) et je désactive le sablier. Ici, il y a une différence importante par rapport à ce que je fais dans *componentDidMount*, dans la mesure où je crée une copie de la liste. Je réutilise le nom de variable *dog* dans le *then* (ligne 11) mais sentez-vous libre de changer si vous trouvez cela compliqué.

```
addDogHandler = dog => {
  this.setState({
    loading: true
  });

  apiFetch('https://some-website.com/api/dogs', {
    method: 'POST',
    body: JSON.stringify(dog),
    headers: { 'Content-Type': 'application/json' }
  })
    .then(dog => {
      this.setState({
        dogs: [...this.state.dogs, dog],
        loading: false
      });
    })
    .catch(error => {
      this.setState({
        error,
        loading: false
      });
    });
}
```

La fonction *removeDogHandler* marche plus ou moins de la même façon que *addDogHandler*. Cette fois l'id du chien à supprimer est passée dans l'URL et j'utilise le verbe *DELETE*. Je n'attends pas de réponse spécifique. Cette fois encore, je mets à jour la liste des chiens, par copie, en filtrant l'id du chien supprimé.

```
removeDogHandler = dogId => {
  this.setState({
    loading: true
  });

  apiFetch('https://some-website.com/api/dogs/' + dogId, {
    method: 'DELETE',
    headers: { 'Content-Type': 'application/json' }
  })
    .then(() => {
      this.setState({
        dogs: [...this.state.dogs].filter(dog => dog.id !== dogId),
        loading: false
      });
    })
    .catch(error => {
      this.setState({
        error,
        loading: false
      });
    });
}
```

La classe *DogForm* va monopoliser une bonne partie de notre temps. Les champs du formulaire sont directement reliés au *state* et organisés dans la sous-variable *dog*. Les changements de valeur sont gérés par la fonction *changeHandler*, qui fait une copie de *dog*, et qui change l'état du chien en entier, ce qui est pratique mais pas optimisé. Cela dit, ce mini formulaire ne nécessite pas qu'on s'attarde trop dessus.

```
class DogForm extends React.Component {

  state = {
    dog: {
      name: '',
      price: ''
    }
  };

  submitHandler = event => {
    event.preventDefault();

    const newDog = {
      name: this.state.dog.name,
      price: this.state.dog.price
    };

    this.props.onAdd(newDog);

    this.setState({
      dog: {
        name: '',
        price: ''
      }
    });
  };

  changeHandler = event => {
    event.preventDefault();
    const dog = { ...this.state.dog };
    dog[event.target.id] = event.target.value;
    this.setState({ dog });
  }

  render() {
    return (
      <section className="dog-form">
        <form onSubmit={this.submitHandler}>
          <div>
            <label>Nom : </label>
            <input
              type="text"
              id="name"
              value={this.state.dog.name}
              onChange={this.changeHandler} />
          </div>
          <div>
            <label>Prix : </label>
            <input
              type="text"
              id="price"
              value={this.state.dog.price}
              onChange={this.changeHandler} />
          </div>
          <div>
            <input type="submit" value="Ajouter" />
          </div>
        </form>
      </section>
    );
  }
};
```

La classe *DogList* est relativement simple. C'est clairement un *dumb component* qui sera facile à *refactorer*. En effet, tout passe par les *props* et la fonction *removeHandler* n'est qu'un sas. C'est ici que j'ai placé le sablier (*Spiner*), qui s'active donc par *props* également.

```
class DogList extends React.Component {

  removeHandler = dog => event => {
    event.preventDefault();
    this.props.onRemove(dog.id);
  }

  render() {
    return (
      <section className="dog-list">
        <h2>Liste des chiens</h2>
        {this.props.loading && <Spiner />}
        <ul>
          {this.props.dogs.map(dog => (
            <li key={dog.id}>
              <span className="name">{dog.name} : </span>
              <span className="price">{dog.price}€</span>
              <button onClick={this.removeHandler(dog)}>Effacer</button>
            </li>
          ))}
        </ul>
      </section>
    );
  }
};
```

D'ailleurs, je vous propose de régler son compte à *DogList* dès maintenant. Comme ça, on n'en parle plus. Il n'y a presque rien à faire, si ce n'est de transformer la classe en fonction, de mettre la fonction en constantes et de virer le mot clé *this*.

```
const DogList = props => {

  const removeHandler = dog => event => {
    event.preventDefault();
    props.onRemove(dog.id);
  }

  return (
    <section className="dog-list">
      <h2>Liste des chiens</h2>
      {props.loading && <Spiner />}
      <ul>
        {props.dogs.map(dog => (
          <li key={dog.id}>
            <span className="name">{dog.name} : </span>
            <span className="price">{dog.price}€</span>
            <button onClick={removeHandler(dog)}>Effacer</button>
          </li>
        ))}
      </ul>
    </section>
  );
};
```

State

L'idée est maintenant de ne plus utiliser de classe. Je vous propose de commencer par *DogForm* avec qui on va reproduire la même modification que pour *DogList*. Il s'agit surtout de transformer les classes en fonctions.

En remplacement du *state*, j'utilise *useState* qui renvoie un tableau avec la valeur courante et sa fonction de mise-à-jour. Je précise simplement la structure d'un chien vide en paramètre pour l'initialiser.

```jsx
const DogForm = props => {

  const [dog, setDog] = useState({ name: '', price: '' });

  const submitHandler = event => {
    event.preventDefault();

    const newDog = {
      name: dog.name,
      price: dog.price
    };

    props.onAdd(newDog);

    setDog({ name: '', price: '' });
  };

  const changeHandler = event => {
    event.preventDefault();
    const updatedDog = { ...dog };
    updatedDog[event.target.id] = event.target.value;
    setDog(updatedDog);
  }

  return (
    <section className="dog-form">
      <form onSubmit={submitHandler}>
        <div>
          <label>Nom : </label>
          <input
            type="text"
            id="name"
            value={dog.name}
            onChange={changeHandler} />
        </div>
        <div>
          <label>Prix : </label>
          <input
            type="text"
            id="price"
            value={dog.price}
            onChange={changeHandler} />
        </div>
        <div>
          <input type="submit" value="Ajouter" />
        </div>
      </form>
    </section>
  );
};
```

Note

Il y a une différence fondamentale de fonctionnement entre setState qui permet de changer le state et setDog qui permet de changer le chien. La fonction setState effectue un merge des valeurs spécifiées avec l'ensemble des autres valeurs du state. La fonction setDog remplace l'ensemble des valeurs par les nouvelles, et les valeurs absentes ne sont pas reprises.

```jsx
const DogForm = props => {

  // const [dog, setDog] = useState({ name: '', price: '' });
  const [name, setName] = useState('');
  const [price, setPrice] = useState('');

  const submitHandler = event => {
    event.preventDefault();

    const newDog = {
      name: name,
      price: price
    };

    props.onAdd(newDog);

    setName('');
    setPrice('');
  };

  return (
    <section className="dog-form">
      <form onSubmit={submitHandler}>
        <div>
          <label>Nom : </label>
          <input
            type="text"
            id="name"
            value={name}
            onChange={event => {
              setName(event.target.value);
            }} />
        </div>
        <div>
          <label>Prix : </label>
          <input
            type="text"
            id="price"
            value={price}
            onChange={event => {
              setPrice(event.target.value);
            }} />
        </div>
        <div>
          <input type="submit" value="Ajouter" />
        </div>
      </form>
    </section>
  );
};
```

Pour aller plus loin, je voudrais passer une fonction, et non une valeur, à *setDog* pour gérer les éventuels rendus multiples avec des *snapshots* de *state* périmés… L'idée est que *prevDog* contient la valeur précédente du chien, au moment de l'appel. Ici, on définit la fonction et non la valeur. Cela ressemblerait à ça :

```jsx
const changeHandler = event => {
  event.preventDefault();
  setDog(prevDog => {
    const updatedDog = { ...prevDog };
    updatedDog[event.target.id] = event.target.value;
    return updatedDog;
  });
}
```

Note

Les appels à setName et setPrice successifs vont être regroupés par React. Ils ne vont donc pas provoquer des cycles de rendu distincts.

Voyez-vous le problème synthétique que cela pose ? React lance une erreur expliquant que ça réutilise l'événement. Et pour cause… La solution est de nouveau très simple, bien que je ne la trouve pas très élégante, et consiste à sauver temporairement la valeur.

```jsx
const changeHandler = event => {
  event.preventDefault();
  const targetId = event.target.id;
  const newTargetValue = event.target.value;
  setDog(prevDog => {
    const updatedDog = { ...prevDog };
    updatedDog[targetId] = newTargetValue;
    return updatedDog;
  });
}
```

Si vous trouvez ça trop complexe, vous pouvez séparer les différents champs. Je ne suis pas toujours super fan de cette approche mais je dois admettre qu'elle fonctionne bien ici. En effet, il n'y a plus besoin de *merger* soi-même le chien, ce qui permet de faire un peu de ménage.

Je refais grosso modo le même travail pour *Dogs*, ce qui rend immédiatement le code plus simple.

```jsx
const Dogs = props => {

  const [dogs, setDogs] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState();

  // init
  // ...

  const addDogHandler = dog => {
    setLoading(true);
    apiFetch('https://some-website.com/api/dogs', {
      method: 'POST',
      body: JSON.stringify(dog),
      headers: { 'Content-Type': 'application/json' }
    })
      .then(dog => {
        setDogs([...dogs, dog]);
        setLoading(false);
      })
      .catch(e => {
        setError(e);
        setLoading(false);
      });
  }

  const removeDogHandler = dogId => {
    setLoading(true);
    apiFetch('https://some-website.com/api/dogs/' + dogId, {
      method: 'DELETE',
      headers: { 'Content-Type': 'application/json' }
    })
      .then(() => {
        setDogs([...dogs].filter(dog => dog.id !== dogId));
        setLoading(false);
      })
      .catch(e => {
        setError(e);
        setLoading(false);
      });
  }

  return (
    <div>
      {error && <ErrorBox errorMessage={error} />}
      <h1>Ma mini app de chiens</h1>
      <hr />
      <DogForm onAdd={addDogHandler} />
      <hr />
      <DogList
        dogs={dogs}
        onRemove={removeDogHandler}
        loading={loading} />
    </div>
  );
}
```

J'en profite pour passer une fonction à *setDogs* à la place de la valeur.

```
const addDogHandler = dog => {
  setLoading(true);
  apiFetch(...)
    .then(receivedDog => {
      setDogs(prevDogs => [...prevDogs, receivedDog]);
      setLoading(false);
    })
    .catch(e => { ... });
}

const removeDogHandler = dogId => {
  setLoading(true);
  apiFetch(...)
    .then(() => {
      setDogs(prevDogs => [...prevDogs].filter(dog => dog.id !== dogId));
      setLoading(false);
    })
    .catch(e => { ... });
}
```

Effect

À ce stade, je n'ai pas encore remis le chargement initial de la liste des chiens qu'on avait dans *componentDidMount*. En reprenant la logique utilisée jusqu'ici, cela devrait être assez rapide.

```
const Dogs = props => {

  const [dogs, setDogs] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState();

  // init
  setLoading(true);
  apiFetch('https://some-website.com/api/dogs', {
    method: 'GET'
  })
    .then(allDogs => {
      setDogs(allDogs);
      setLoading(false);
    })
    .catch(e => {
      setError(e);
      setLoading(false);
    });

  ...
```

Malheureusement, ce n'est pas aussi simple. Ce bloc de code fait exploser l'app. React explique qu'il provoque trop de *re-renders(4)*. Je vais donc utiliser un autre *hook*, nommé *useEffect*, qui s'exécute après les cycles de rendu des autres composants.

```
const Dogs = props => {

  const [dogs, setDogs] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState();

  // init
  useEffect(() => {
    setLoading(true);
    apiFetch('https://some-website.com/api/dogs', {
      method: 'GET'
    })
      .then(allDogs => {
        setDogs(allDogs);
        setLoading(false);
      })
      .catch(e => {
        setError(e);
        setLoading(false);
      });
  }, []);

  ...
```

Le tableau vide, passé en second argument de *useEffect*, indique que le code n'a aucune dépendance avec ce qui se passe en dehors. En réalité, il y a bien une dépendance vers *setDogs*, *setLoading* et *setError* mais ce sont des dépendances spéciales déjà prises en charge par *useState*.
Utilisé de cette façon, le *hook useEffect* se comporte de la même manière que *componentDidMount* et ne sera exécuté qu'une seule fois(5).

Reducer

Le code commence à être complexe. Et encore, je me suis limité sur le nombre de fonctionnalités. C'est là qu'intervient le *hook* nommé *useReducer*, qui n'a rien à voir avec Redux. Pour autant, si vous avez l'habitude d'utiliser Redux, vous ne serez pas perdu.
Pour commencer, j'écris un *reducer* pour mes chiens, idéalement en dehors de mon composant, en reproduisant une logique similaire à celle de Redux.

```
const dogReducer = (currentDogs, action) => {
  switch (action.type) {
    case 'GET':
      return action.dogs;

    case 'ADD':
      return [...currentDogs, action.dog];

    case 'REMOVE':
      return currentDogs.filter(dog => dog.id !== action.dogId);

    default:
      throw new Error('The specified action type is not correct!');
  }
};
```

Puis je remplace *useState* de *dogs* par *useReducer* auquel je passe le *dogReducer*, que je viens de créer, ainsi qu'un état initial.

Note

Contrairement à useState, qui renvoie la fonction de mise à jour du state en second élément, useReducer renvoie la fonction de dispatch de l'action spécifique. Je laisse l'ancienne version en commentaire dans le code pour mettre en évidence la différence.

```
const Dogs = props => {

  // const [dogs, setDogs] = useState([]);
  const [dogs, dispatchDogAction] = useReducer(dogReducer, []);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState();

  // init
  useEffect(() => {
    setLoading(true);
    apiFetch( ... )
      .then(allDogs => {
        // setDogs(allDogs);
        dispatchDogAction({
          type: 'GET',
          dogs: allDogs
        });
        setLoading(false);
      })
      .catch( ... );
  }, []);


  const addDogHandler = dog => {
    setLoading(true);
    apiFetch( ... )
      .then(receivedDog => {
        // setDogs(prevDogs => [...prevDogs, receivedDog]);
        dispatchDogAction({
          type: 'ADD',
          dog: receivedDog
        });
        setLoading(false);
      })
      .catch( ... );
  }

  const removeDogHandler = dogId => {
    setLoading(true);
    apiFetch( ... )
      .then(() => {
        // setDogs(prevDogs => [...prevDogs].filter(dog => dog.id !== dogId));
        dispatchDogAction({
          type: 'REMOVE',
          dogId: dogId
        });
        setLoading(false);
      })
      .catch(e => {
        setError(e);
        setLoading(false);
      });
  }
```

Cette façon de faire prend tout son sens dans une app plus conséquente où on voudrait sortir la logique des composants.

De plus, je manipule plusieurs *states* qui, même s'ils ne sont pas directement liés, fonctionnent toujours ensemble. Par exemple je change toujours la valeur de *loading* quand je veux mettre à jour *dogs*. J'aimerais que ce soit plus simple.
Je crée donc un nouveau *reducer* pour gérer le sablier et les erreurs, qui font de l'affichage, et qui fonctionnent clairement de pair.

```
const apiFetchStateReducer = (apiFetchState, action) => {
  switch (action.type) {
    case 'CALL':
      return { loading: true, error: null };

    case 'RESULT':
        return { ...apiFetchState, loading: false };

    case 'ERROR':
        return { loading: false, error: action.error };

    default:
      throw new Error('The specified action type is not correct!');
  };
};
```

Il suffit ensuite de l'utiliser comme j'ai fait pour le premier *reducer*. Ici, je change aussi les références dans le JSX puisque *loading* et *error* n'existent plus.

```
const Dogs = () => {

  const [dogs, dispatchDogAction] = useReducer(dogReducer, []);
  // const [loading, setLoading] = useState(false);
  // const [error, setError] = useState();
  const [apiFetchState, dispatchApiFetchStateAction]
        = useReducer(apiFetchStateReducer, { loading: false, error: null });


  // init
  useEffect(() => {
    // setLoading(true);
    dispatchApiFetchStateAction({ type: 'CALL' });
    apiFetch(...)
      .then(allDogs => {
        dispatchDogAction( ... );
        // setLoading(false);
        dispatchApiFetchStateAction({ type: 'RESULT' });
      })
      .catch(e => {
        // setError(e);
        // setLoading(false);
        dispatchApiFetchStateAction({ type: 'ERROR', error: e });
      });
  }, []);


  const addDogHandler = dog => {
    // setLoading(true);
    dispatchApiFetchStateAction({ type: 'CALL' });
    apiFetch( ... )
      .then(receivedDog => {
        dispatchDogAction( ... );
        // setLoading(false);
        dispatchApiFetchStateAction({ type: 'RESULT' });
      })
      .catch(e => {
        // setError(e);
        // setLoading(false);
        dispatchApiFetchStateAction({ type: 'ERROR', error: e });
      });
  }

  const removeDogHandler = dogId => {
    // setLoading(true);
    dispatchApiFetchStateAction({ type: 'CALL' });
    apiFetch( ... )
      .then(() => {
        dispatchDogAction( ... );
        // setLoading(false);
        dispatchApiFetchStateAction({ type: 'RESULT' });
      })
      .catch(e => {
        // setError(e);
        // setLoading(false);
        dispatchApiFetchStateAction({ type: 'ERROR', error: e });
      });
  }

  return (
    <div>
      {apiFetchState.error && <ErrorBox errorMessage={apiFetchState.error} />}
      <h1>Ma mini app de chiens</h1>
      <hr />
      <DogForm onAdd={addDogHandler} />
      <hr />
      <DogList
        dogs={dogs}
        onRemove={removeDogHandler}
        loading={apiFetchState.loading} />
    </div>
  );
}
```

Note

React garantit que l'identité de la fonction de dispatch est stable et ne changera pas d'un rendu à l'autre. C'est pourquoi on peut l'omettre de la liste des dépendances de useEffect et useCallback (cf. plus bas) en toute sécurité.

Callback - Memo

L'air de rien, ce code peut largement être encore travaillé, notamment si on s'intéresse aux rendus. Pour le mettre en évidence, j'ajoute un simple *log* dans *DogForm*.

```
const DogForm = props => {
  console.log('DogsForm');
  ...
```

On voit dans ces logs(6) que DogForm est appelé 4 fois au lancement de l'app, soit 3 fois de trop…

```
DogForm.jsx:5 DogsForm
DogForm.jsx:5 DogsForm
Dogs.jsx:30 Simulating fetch...
Dogs.jsx:40 URL: https://some-website.com/api/dogs
Dogs.jsx:41 Method: GET
Dogs.jsx:42 Body: undefined
Dogs.jsx:43 Headers: undefined
Dogs.jsx:47 Getting all dogs...
DogForm.jsx:5 DogsForm
DogForm.jsx:5 DogsForm
```

Je vais commencer par mettre la fonction *addDogHandler* en *callback*. Ça c'est gratuit. La fonction *useCallback* renvoie une version *mémoïsée* de la fonction de rappel qui changera uniquement si une des entrées a changé. Et, ici, elle n'a pas de raison de changer.
Puis, c'est tout le formulaire que je mémorise avec la fonction *useMemo*, qui recalcule la valeur *mémoïsée* seulement si une des entrées a changé. Cette optimisation permet d'éviter des calculs coûteux à chaque rendu. Ici je précise une dépendance à *addDogHandler*.

```
const addDogHandler = useCallback(dog => {
  ...
}, []);


const dogForm = useMemo(() => {
  return (
    <DogForm onAdd={addDogHandler} />
  )
}, [addDogHandler]);


return (
  <div>
    {apiFetchState.error && <ErrorBox errorMessage={apiFetchState.error} />}
    <h1>Ma mini app de chiens</h1>
    <hr />
    {dogForm}
    <hr />
    <DogList ... />
  </div>
);
```

Note

la fonction fournie à useMemo s'exécute pendant le rendu. N'y faites donc rien que vous ne feriez pas normalement pendant un rendu. Pour les effets de bord, utilisez useEffect de préférence.

Comme par magie, le nombre de rendu de DogForm devient celui que j'attends.

```
DogForm.jsx:5 DogsForm
Dogs.jsx:30 Simulating fetch...
Dogs.jsx:40 URL: https://some-website.com/api/dogs
Dogs.jsx:41 Method: GET
Dogs.jsx:42 Body: undefined
Dogs.jsx:43 Headers: undefined
Dogs.jsx:47 Getting all dogs...
```

Il ne vous reste plus qu'à faire pareil avec *DogList*, en précisant une dépendance vers *apiFetchState.loading* et *dogs*.

Façon maison

Ah ? Vous trouvez qu'il reste encore beaucoup de désordre et en particulier que la logique d'appel pourrait être externalisée : vous avez bien raison !... Cela va nous donner un prétexte pour écrire un *hook maison*.

Pour commencer, je déplace *apiFetchStateReducer* dans un fichier à part. Puis je crée la fonction *useApiFetch*, avec le préfixe *use*, qui va bien entendu utiliser le *dispatch* qu'on vient de déplacer. Au passage, j'ajoute *data*, *extra* et *method* à l'initialisation.

J'y place également la fonction *call* avec son *useCallback(7)* que je récupère du code précédemment écrit, avec une signature élargie. Quand je réceptionne des données, je les ajoute à l'action. Et j'en profite pour ajouter l'opération utilisée et un petit extra, qui me serviront plus tard. La fonction *useApiFetch* et *call* restent relativement génériques et ne dépendent en particulier pas des chiens.

```
const apiFetchStateReducer = (apiFetchState, action) => {
  switch (action.type) {
    case 'CALL':
      return {
        loading: true,
        error: null,
        data: null,
        extra: null,
        method: action.method
      };

    case 'RESULT':
      return {
        ...apiFetchState,
        loading: false,
        data: action.data,
        extra: action.extra,
        method: action.method
      };

    case 'ERROR':
      return { loading: false, error: action.error };

    default:
      throw new Error('The specified action type is not correct!');
  };
};


const useApiFetch = () => {
  const [apiFetchState, dispatchApiFetchStateAction]
    = useReducer(apiFetchStateReducer, {
      loading: false, error: null, data: null, extra: null, method: null
    });

  const call = useCallback((url, method, body, headers, extra) => {
    dispatchApiFetchStateAction({ type: 'CALL', method });
    apiFetch(url, { method, body, headers })
      .then(data => {
        dispatchApiFetchStateAction({ type: 'RESULT', data, extra, method });
      })
      .catch(e => {
        dispatchApiFetchStateAction({ type: 'ERROR', error: e });
      });
  }, []);

  const called = {
    loading: apiFetchState.loading,
    error: apiFetchState.error,
    data: apiFetchState.data,
    extra: apiFetchState.extra,
    method: apiFetchState.method,
  };

  return [called, call];
};
```

De retour dans *Dogs*, je remplace l'ancien *hook* par *useApiFetch*. J'en profite pour déstructurer l'état correspondant, afin que ce soit plus lisible, et j'adapte le code en conséquence.

```
const Dogs = () => {

  const [dogs, dispatchDogAction] = useReducer(dogReducer, []);
  // const [apiFetchState, dispatchApiFetchStateAction]
  //     = useReducer(apiFetchStateReducer, { loading: false, error: null });
  const [{ loading, error, data, extra, method }, call] = useApiFetch();
```

Du coup, le chargement initial ainsi que les fonctions d'ajout et de suppression deviennent radicalement plus simples.

```
const Dogs = () => {
  ...

  // init
  useEffect(() => {
    call('https://some-website.com/api/dogs', 'GET');
  }, [call]);


  const addDogHandler = useCallback(dog => {
    call('https://some-website.com/api/dogs', 'POST',
         JSON.stringify(dog), { 'Content-Type': 'application/json' });
  }, [call]);

  const removeDogHandler = useCallback(dogId => {
    call('https://some-website.com/api/dogs/' + dogId, 'DELETE',
         {}, {}, dogId);
  }, [call]);


```

Et pour que cela fonctionne automatiquement, je déclare un nouvel effet auquel je passe une belle liste de dépendances. N'ayez pas peur d'en oublier car React saura vous rappeler à l'ordre le cas échéant. C'est ici que *data* (qui contient le chien en cas d'ajout ou la liste des chiens en cas d'initialisation) et *extra* (qui contient l'id à supprimer car celui-ci n'est pas renvoyé par l'API) sont utilisés.

```
const Dogs = () => {
  ...


  useEffect(() => {
    if (loading || error) {
      return;
    }

    if (method === 'DELETE') {
      dispatchDogAction({ type: 'REMOVE', dogId: extra });
    } else if (method === 'POST') {
      dispatchDogAction({ type: 'ADD', dog: data });
    } else if (method === 'GET') {
      dispatchDogAction({ type: 'GET', dogs: data });
    }
  }, [data, extra, method, loading, error]);

```

Conclusion

L'utilisation des *hooks* est relativement simple. Il y a juste quelques pièges facilement contournables. Je trouve que *useReducer* est plus pratique et plus parlant que *useState*. Mais ce dernier fait déjà très bien le travail, en particulier dans une petite app.

Je vois deux raisons pour décider d'utiliser *useReducer*. La première est de vouloir réorganiser son code pour que la logique soit bien isolée. La seconde est qu'on a plusieurs states qui fonctionnent ensemble car il est plus cohérent de les manipuler de concert ou quand le nouvel état dépend du précédent.

La taille de l'app est aussi un bon indicateur pour faire ce choix. Et dans la foulée, il sera légitime de se poser la question pour Redux également dans une petite app, car il sera facile de s'en passer…

---

(1) Vous admirerez cette magnifique IHM :-)

(2) ie. promesse résolue.

(3) L'id du chien a été choisie coté serveur et non coté client.

(4) Boucle infinie

(5) Ce qui est bien ce que je voulais

(6) Les logs supplémentaires viennent de la fonction apiFetch, non présentée ici.

() Qui ne doit pas changer

Olivier LOURME
Enseignant en Génie Électrique & Informatique à l'Université de Lille, doctorant au laboratoire CRIStAL UMR 9189
Twitter : @OlivierLourme
Medium : medium.com/@o.lourme

ESP32, Mongoose OS et GCP Cloud IoT Core : un trio efficace et sûr pour l'IoT

Partie 2

Dans un contexte de développement autour des objets connectés, nous présentons ici une solution « Objet / OS / plateforme cloud » efficace en termes de temps de développement tout en étant « secure by design ». L'objet est un **ESP32** *du chinois Espressif, son OS est* **Mongoose OS** *et la plateforme retenue est* **Google Cloud Platform** *(GCP) avec sa brique « Cloud IoT Core ».* **Firebase** *sera également utilisé. Suite du dossier.*

Hello, World! pour ESP32 + Mongoose OS

Avant de présenter la plateforme GCP, voyons à quoi peut ressembler un fichier **init.js** permettant de faire une mesure de la température et du taux d'humidité toutes les 5 secondes, suivie d'une publication MQTT de cette information sur un *topic* à destination de **MQTT Bridge**, le point d'entrée de Cloud IoT Core.

Remarque : pour l'instant, cette publication MQTT n'aboutira pas puisque le projet n'a pas été configuré du côté de la plateforme GCP. Néanmoins, il y aura un affichage en console de la mesure faite.

Câblage de l'objet

L'association d'un ESP32 avec un DHT22 est très simple à réaliser. **Figure 3**.

On voit que nous avons choisi de connecter la broche de données/commande du capteur DHT22 à la **GPIO0** de l'ESP32.

On supposera par la suite que Mongoose OS a été installé sur l'ordinateur de développement et qu'un des ports USB de ce dernier est connecté à l'objet *via* son connecteur micro USB. Un répertoire pour le développement du projet a été créé sur l'ordinateur hôte.

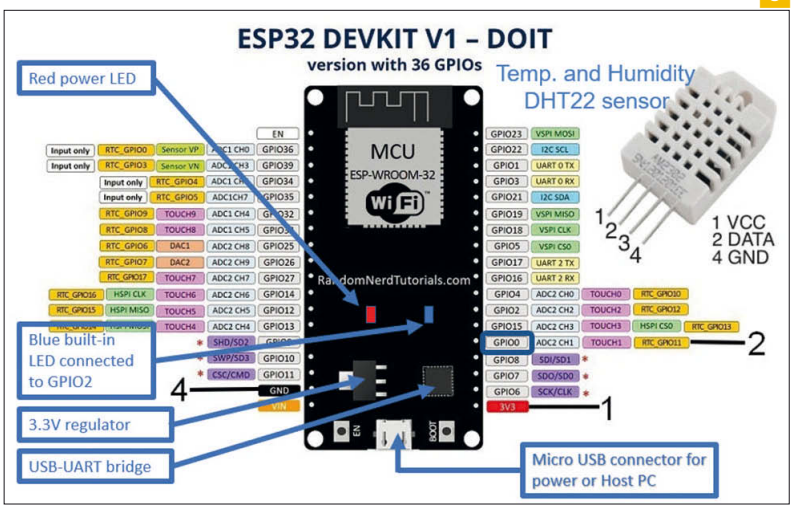
La communication objet-ordinateur peut se faire avec :

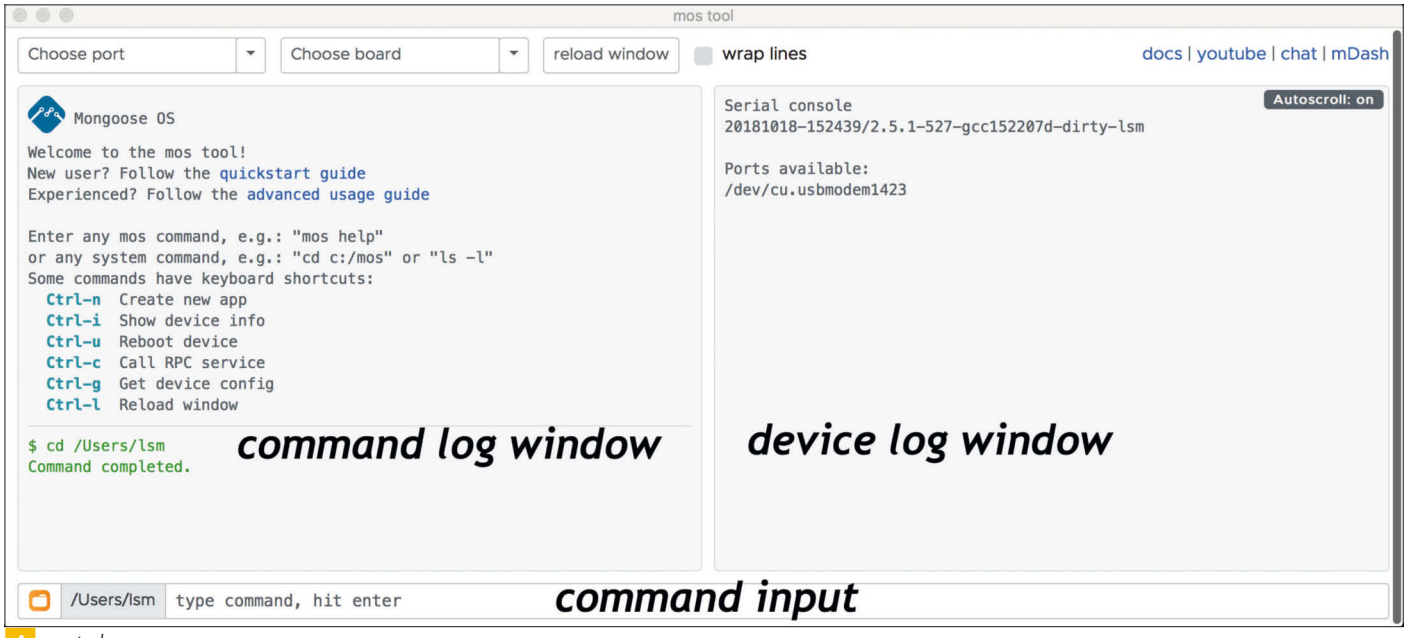
- Visual Studio Code (onglets « OUTPUT » et « TERMINAL »),
- une console, *i.e.* un terminal série,
- *mos tool*, assez pratique, notamment parce qu'il détecte bien les ports série (cf. **Figure 4**). Il est installé avec Mongoose OS.

Fichier init.js

Voici le code source du fichier **init.js** à éditer pour l'*uploader* plus tard dans l'objet. On peut difficilement faire plus court :

3

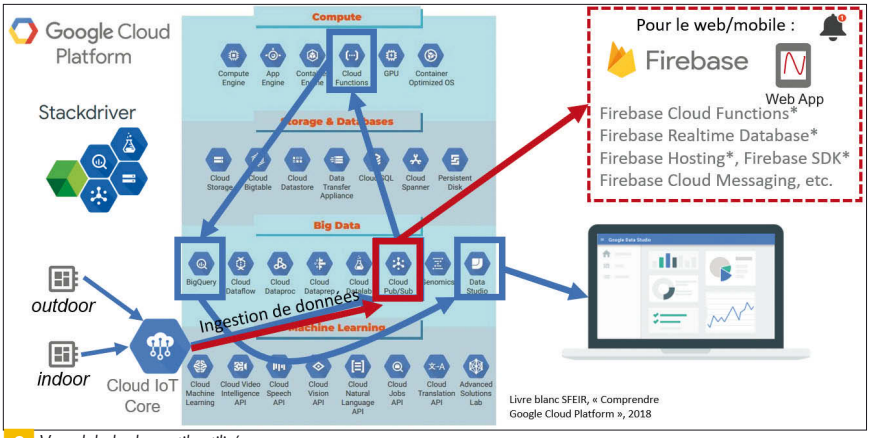




4 *mos tool*

```
[Feb 22 08:35:52.109] 0 {"humidity":43.400002,"temperature":24.900000}
[Feb 22 08:35:57.107] 0 {"humidity":43.299999,"temperature":24.799999}
[Feb 22 08:36:02.108] 0 {"humidity":43.299999,"temperature":24.799999}
```

5 *Trois mesures de la température et de l'humidité par l'objet*



6 *Vue globale des outils utilisés*

```
load('api_config.js'); // to get device id
load('api_dht.js');    // DHT 11 & DHT22 library
load('api_timer.js');  // timer library
load('api_mqtt.js');   // MQTT library

// With GCP, telemetry topic must have this name:
let topic = '/devices/' + Cfg.get('device.id') + '/events';

// Create an instance of DHT object on GPIO0:
let dht = DHT.create(0, DHT.DHT22);

// Timer period is in ms:
Timer.set(5000, true, function() {
 let msg = JSON.stringify({temperature: dht.getTemp(),
                           humidity: dht.getHumidity()});
 // Let's publish telemetry message with a Quality Of Service equal to 1.
 // Note: MQTT.pub() returns 1 in case of success, 0 otherwise.
 let ok = MQTT.pub(topic, msg, 1);
 print(ok, msg);
}, null);
```

Cfg.get('device.id') est une chaine représentant l'*ID* de l'objet, obtenue en se basant sur son adresse MAC à 6 octets. Par exemple elle vaut : **esp32_ABB3B4** pour l'ESP32 ayant ABB3B4 comme adresse MAC Wi-Fi. De ce fait la publication de la température et de l'humidité au format JSON sera faite sur le topic de nom **/devices/esp32_ABB3B4/events**.

MQTT propose trois ***Quality Of Service*** (QOS) :

- QOS = 0 : le message est délivré au plus une fois. Le message peut donc ne pas être délivré.
- QOS = 1 : le message est délivré au moins une fois.
- QOS = 2 : le message est délivré exactement une fois.

Cloud IoT Core ne sait traiter que les deux premiers QOS. Le deuxième (QOS = 1) étant plus sûr, c'est celui que nous avons retenu. L'API Mongoose OS relative à la publication MQTT sait gérer des messages avec un tel QOS.

Préparation du firmware

Abordons trois commandes **mos** à exécuter depuis le répertoire de développement. Si le parc ne comprend que des objets à base d'ESP32, la première ligne n'est à effectuer qu'une fois sur l'ordinateur hôte. La deuxième, concernant le *flash* du *firmware*, est à faire pour chaque objet et prend un certain temps (une à deux minutes). Notons que l'empreinte mémoire de Mongoose OS est entre 1 et 2 Mo. La troisième est également à faire pour chaque objet du parc afin de lui donner les crédences Wi-Fi. **Ainsi l'identifiant du réseau Wi-Fi et le mot de passe ne figurent pas dans le code source (on évite de coder en dur ces éléments, NDLR)**. Cette précaution basique est donc respectée ici.

```
# build app firmware (only once)
mos build --arch esp32
# flash firmware to device
mos flash
# give WiFi credentials to device
mos wifi WIFI_SSID WIFI_PASSWORD
```

Exécution du Hello, World!

On peut *uploader* le fichier **init.js** dans le système de fichiers de l'objet et de procéder au *reboot*. **Ces deux lignes seront les seules à effectuer à chaque mise à jour du projet**. C'est incroyablement plus court que de faire un *build* suivi d'un *flash* de la mémoire.

```
# upload JS file to device file system
mos put fs/init.js
# reboot device (by a RPC)
mos call Sys.Reboot
```

Le *reboot* est ici traité par un *Remote Procedure Call* (RPC) préexistant. Avec Mongoose OS, on crée facilement des RPC que l'on peut solliciter de n'importe où (grâce à cURL par exemple) mais cela sort du cadre de cet article.

Sur la Figure 5, on voit le résultat dans la console série connectée à l'ESP32 de trois exécutions successives de la ligne de code **print(ok, msg)**. Les 0 indiquent l'état de la variable **ok**, montrant sans surprise que la publication MQTT n'a pas pu se faire encore à ce stade d'avancement du projet. Les valeurs ont trop de décimales mais cela sera corrigé en aval. On voit aussi que Mongoose OS gère automatiquement l'heure sans que nous ayons eu à écrire quoi que ce soit ; en effet, il interroge des serveurs de temps lors de son *boot*.

PLATEFORME GCP, CLOUD IOT CORE, FIREBASE

Présentation

Sur la **Figure 6**, on peut observer nos deux objets *indoor* et *outdoor*, la plateforme GCP et le service Cloud IoT Core. Il y a plein d'autres briques pour faire de l'IA, du calcul, du stockage, etc. Quant à **Stackdriver**, il n'est pas abordé ici mais sachez qu'il permet de monitorer toutes les activités de la plateforme.

On a représenté deux chemins (parmi d'autres possibles) pour les données de **télémétrie** :

- Celui en bleu utilise des briques GCP, notamment **Big Query** pour la persistance et **Data Studio** pour la visualisation.

- Celui en rouge est identique au chemin précédent jusqu'à **Cloud Pub/Sub** mais s'oriente ensuite vers l'écosystème **Firebase**. C'est le chemin qu'on a choisi.

Firebase a des relations étroites avec GCP (les projets GCP et Firebase ne font qu'un) mais il propose les briques orientées web (et mobile) qui nous seront utiles pour la persistance des mesures en base de données et pour leur affichage web sous forme de **graphes temps réels**. Sur la **Figure 6**, les briques Firebase marquées d'une * sont celles que nous avons utilisées. Comme prolongement de ce projet, on peut imaginer la génération de notifications *via* Firebase Cloud Messaging, par exemple en cas de dépassement d'une certaine valeur de température pour un objet donné…

Architecture complète

La **Figure 7** présente l'architecture complète du projet. Outre trois points de sécurité (cadenas 1, 2 et 3) qui seront traités un peu plus loin, on remarque :

- les objets,
- les briques de GCP utilisées (Cloud IoT Core et Cloud Pub/Sub (*1*)),
- les briques de Firebase utilisées.

Nous commentons ici les aspects hors télémétrie, celle-ci étant traitée en fin d'article.

On voit que Cloud IoT Core est bien plus qu'un point d'entrée des données *via* son MQTT Bridge. En effet :

- Le gestionnaire d'objets (***Device Manager***) répertorie les objets correctement enregistrés auprès de GCP. Il permet d'autoriser individuellement chaque objet à communiquer avec la plateforme. Il se charge également de la gestion des registres (***Registry Management***).
- Un registre (***Registry***) permet de **regrouper des objets** qui partagent des points communs :
  - **Propos commun** : par exemple il semble cohérent de mettre dans un même registre les objets chargés de la télémétrie d'un même bâtiment,
  - **Protocole de connexion à Cloud IoT Core commun** : MQTT (à préférer, c'est d'ailleurs notre choix) ou HTTP (on ne l'a pas évoqué car il est moins performant mais parfois il existe des objets qui n'ont pas d'APIs MQTT),
  - ***Topic* Cloud Pub/Sub commun** : les messages des objets d'un même registre seront republiés par **Cloud Pub/Sub** sur un *topic* commun. Une **Cloud Function for Firebase** sera abonnée à ce *topic* et sera donc déclenchée à chaque publication d'un des objets du registre. Elle se chargera de la persistance en base de données de la mesure contenue dans un message.
- Le ***Monitoring*** des objets permet par exemple de savoir quand un objet a été vu la dernière fois. En outre il permet, grâce à des *topics* spéciaux nommés **config** et **state** (2) :
  - à Cloud IoT Core de publier sur le *topic* **config** dédié à l'objet un message de **configuration**, à destination de cet objet (abonné à ce *topic*),
  - à un objet de publier sur son *topic* dédié **state** un message décrivant son **état**, à destination de Cloud IoT Core (abonné à ce *topic*).

Ces mécanismes permettent de recréer un équivalent de *Shadow Device* d'AWS.

(*1*) https://youtu.be/0_XtBmjS0xI

(*2*) http://bit.ly/2VbRLCj *pour une gestion de ces topics.*

- Le bloc authentification (***Authentication***) vérifie qu'un objet voulant se connecter à Cloud IoT Core peut le faire. Ce point de sécurité est détaillé un peu plus loin parmi d'autres.

Mise en place du projet GCP

Il est temps de créer le projet GCP. Nous appelons celui-ci **hello-cloud-iot-core**. Par ailleurs, nous appelons le registre qui comportera les deux objets *indoor* et *outdoor* **weather-devices-registry** et le *topic* Cloud Pub/Sub associé **weather-telemetry-topic**.

Pour configurer / gérer un projet GCP et ses briques on a le choix entre :

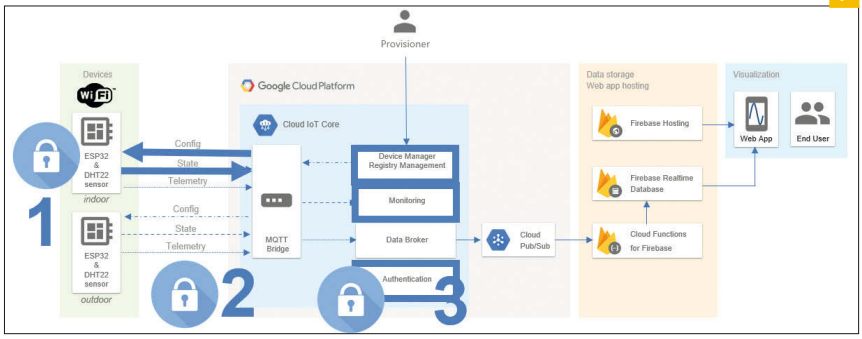
- une interface web que Google appelle la « console GCP »,
- un programme (Java, C++, Node JS, Python, etc.) utilisant des APIs GCP,
- des commandes **gcloud** à taper dans un terminal (à scripter éventuellement).

C'est cette dernière possibilité que nous avons choisi. Le SDK Google Cloud étant préalablement installé, la séquence de commandes est la suivante :

```
# Get authenticated with Google Cloud
gcloud auth login
# Create cloud project
gcloud projects create hello-cloud-iot-core
# Allow Cloud IoT Core to publish to Pub/Sub topics
gcloud projects add-iam-policy-binding hello-cloud-iot-core
 --member=serviceAccount:cloud-iot@system.gserviceaccount.com
 --role=roles/pubsub.publisher

# Create Pub/Sub topic for devices telemetry
gcloud beta pubsub topics create weather-telemetry-topic

# Create a devices registry - Associate it with the Pub/Sub topic -
# For safety, disallow HTTP device connections to the bridge
gcloud beta iot registries create weather-devices-registry
 --region europe-west1
 --no-enable-http-config
 --event-notification-config=topic=weather-telemetry-topic
```

Le projet GCP configuré, il reste à enregistrer les objets auprès de Cloud IoT Core, ce qui terminera le provisionnement. Cet aspect fait figurer entre autres la création de clés asymétriques qui sont un point de sécurité. C'est donc dans la partie suivante traitant de sécurité que la fin du provisionnement sera évoquée.

7



SÉCURITÉ (ET FIN DU PROVISIONNEMENT)

Sur la **Figure** 7, nous avons fait figurer les trois points de sécurité sur lesquels nous allons nous focaliser maintenant. Par la couverture de ces points, nous ne pouvons prétendre à une sécurité absolue (qui le peut ?) mais nous pensons que nous sommes sur un chemin vertueux. Deux points (2 et 3) sont imposés par la plateforme GCP ; c'est un des intérêts d'utiliser une plateforme sérieuse. Le (1) est de notre propre initiative, il concerne les mécanismes de protection à prévoir sur l'objet même. La mise en œuvre de ces différents points est coûteuse en termes de ressources. Heureusement, elle exploite intelligemment les possibilités de l'ESP32 (bloc d'accélération matérielle pour le chiffrement, mémoire flash chiffrable) ainsi que l'outillage et les APIs Mongoose OS.

La sécurité IoT est un énorme sujet. La consultation de l'**OWASP pour l'IoT** (3), du **guide sur la sécurité des objets connectés** édité par Captronic (4) et la lecture de **quelques ressources de l'ANSSI** (5) peuvent être un bon point de départ pour recenser les principales vulnérabilités et les réduire.

Limiter les vulnérabilités au niveau de l'objet lui-même

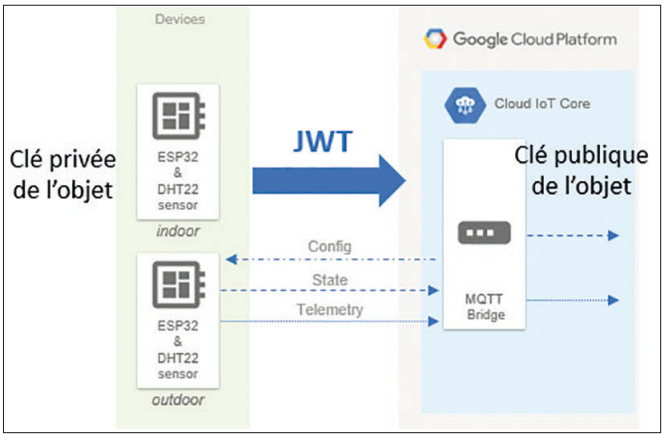
Pour cela, il convient :

- De chiffrer la mémoire flash des IoT. Ainsi la lecture du code source JS (*reverse-engineering*, etc.), l'obtention des clés, des SSID et mot de passe Wi-Fi seront impossibles. Une commande **mos** effectue ce chiffrement, nous l'avons testée dans (2).
- De vérifier l'intégrité du code après chaque *reset* (*secure boot*), *a fortiori* après une mise à jour du code, souvent faite *Over The Air*. Mongoose OS ne traite pas cela mais ESP-IDF oui. Nous avons reçu des garanties des éditeurs de Mongoose OS comme quoi cette cohabitation était possible.
- D'empêcher en production un ordinateur de se connecter à l'objet par JTAG ou UART (cette vulnérabilité matérielle est vicieuse et particulièrement efficace, NDLR). Cela passe par des mécanismes purement physiques : suppression des pattes concernées (ou absence d'accès possible à celles-ci), noyage dans la résine, etc.

(3) http://bit.ly/2Nhl2c1

(4) http://bit.ly/2PVigfr

(5) https://www.ssi.gouv.fr/guide/la-cybersecurite-des-systemes-industriels/



8 *Clés privée et publique pour l'authentification de l'objet par JWT*

En bref, il faut compliquer le « travail » de l'attaquant, comme expliqué dans cette vidéo de DEVOXX FR 2019 (6).

Authentifier la plateforme auprès de l'objet / Sécuriser les échanges objet-plateforme, grâce à TLS

GCP impose que les échanges objet-plateforme se fassent par le protocole MQTT (ou HTTP) *over* TLS 1.2 *over* TCP. L'objet est un **client** MQTT et le MQTT Bridge de Cloud IoT Core est un **serveur** MQTT.

TLS (*Transport Layer Security*) est bien adapté à ce mode client-serveur car avec lui :

- L'objet, ayant son « magasin de certificats », est sûr d'avoir affaire au serveur, c'est-à-dire qu'il y a **authentification** de la plateforme auprès de l'objet. Ce magasin de certificats prend la forme d'un fichier **ca.pem** dans le système de fichiers de Mongoose OS.
- La **confidentialité** est respectée car les données échangées sont chiffrées.
- L'**intégrité** des données est vérifiée.

Authentifier l'objet auprès de la plateforme (et finir le provisionnement)

Il est nécessaire qu'à son tour l'objet s'authentifie auprès de la plateforme. Chez AWS, cela prend la forme d'un *mutual TLS*. Google a choisi une autre solution : L'objet, une fois connecté à Cloud IoT Core, doit s'authentifier en lui envoyant un ***JSON Web Token*** (JWT) que Cloud IoT Core valide.

Pour cela, il est nécessaire qu'objet et plateforme disposent d'une paire de clés asymétriques, par exemple à **Courbe Elliptique ES256**. La clé privée est logée sur l'objet, la clé publique sur la plateforme, comme indiqué sur la **Figure** 8. Ce process est assuré lors de la dernière partie du provisionnement de l'objet que nous abordons maintenant.

Fin du provisionnement

Voyons comment les clés sont générées par une commande **mos** et comment cette commande termine le provisionnement de l'objet. La dernière commande **mos** à taper au niveau de l'objet est :

```
# register device with GCP & handle keys for JWT authentication
mos gcp-iot-setup --gcp-project hello-cloud-iot-core
        --gcp-region europe-west1
        --gcp-registry weather-devices-registry
```

Cette commande est riche : elle enregistre l'objet dans le projet GCP, elle lui fait faire partie d'un registre, elle génère une paire de clés et elle dépose chaque clé au bon endroit. Plusieurs commandes **gcloud** et **openssl** sont utilisées dans ce but et nous sommes bien contents de ne pas avoir à les gérer nous-mêmes !

Remarque : la clé privée déposée dans la mémoire flash de l'objet ne sera pas lisible si l'on chiffre cette dernière.

Dans le répertoire de développement on peut voir que les clés ont été créées : **gcp-esp32_ABB3B4.key.pem** pour la clé privée et **gcp-esp32_ABB3B4.pub.pem** pour la clé publique.

Pour la sécurité, il faut bien sûr **une paire unique de clés par objet**.

(6) https://youtu.be/TZony98Q7ZA

De toute façon l'usage de la commande **mos** fait que ce sera obligatoirement le cas.

Remarque : la génération, le stockage sur l'ordinateur hôte et le logement final à l'endroit approprié des clés est une problématique de sécurité très actuelle dans l'IoT.

Sur la **Figure 9**, on peut voir une des vues de la console GCP une fois que deux objets ont été provisionnés.

Mécanisme de l'authentification de l'objet par JWT

Après que la connexion sécurisée soit établie ou que le JWT précédent ait expiré, l'objet va calculer un nouveau JWT (7).
Le JWT comprend trois parties :

- Un ***header*** précisant l'algorithme **alg** utilisé pour générer la partie **signature** du JWT ainsi que le type **typ** de jeton utilisé : **{"alg": "ES256", "typ": "JWT"}**. Cette partie est ensuite encodée en base64url pour retirer les guillemets, accolades, etc.
- Un ***payload*** comprenant les ***claims*** nom du projet GCP (*aud*), date d'émission du token (*iat*) et date d'expiration (*exp*), par exemple : **{"aud": "hello-cloud-iot-core", "iat": 1509650801, "exp": 1509654401}**. Cette partie est ensuite encodée en base64url.
- Une ***signature*** sur les deux parties précédentes utilisant la clé privée de l'objet est calculée selon l'algorithme précisé dans le *header*. Cette signature est ensuite encodée en base64url.

Les différentes parties (*header* encodé, *payload* encodé, *signature* encodé) sont concaténées en les séparant par des points. Le résultat constitue enfin le JWT. Le JWT est alors envoyé à la plateforme. Avec Mongoose OS, cette gestion (calcul et envoi) se fait automatiquement, sans avoir à taper la moindre ligne de code. C'est appréciable.
Remarque : l'envoi du JWT se fait en tant que *password* dans un message MQTT de type *connect*.
Quand la plateforme reçoit le JWT, elle le valide (et par là elle valide aussi l'objet) en vérifiant la signature contenue grâce à sa clé publique.

DATA-FLOW DE LA TÉLÉMÉTRIE

Maintenant que tout est en place, voyons sur la **Figure 10**, la chronologie des évènements à partir du moment où un objet publie un message de mesure.

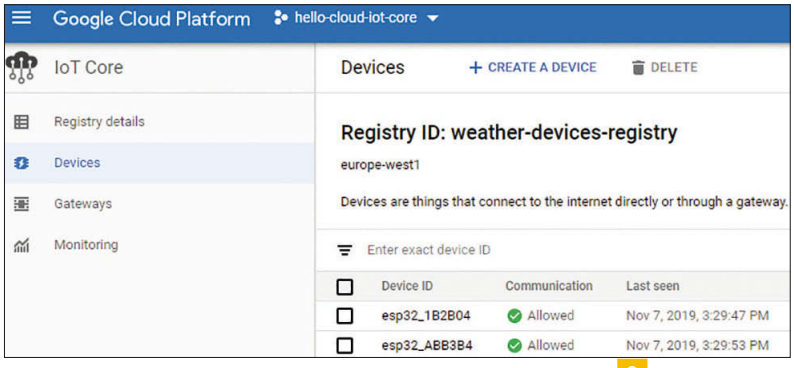
1 L'objet *indoor*, ayant pour *ID* **esp32_ABB3B4** et faisant partie du registre **weather-devices-registry**, publie avec un QOS de 1 sur son *topic* de télémétrie **/devices/esp_32_ABB3B4/events** le message **{"temperature": 22.0, "humidity": 41}**.

2 MQTT Bridge, qui reçoit les messages de tous les *topics* de télémétrie des objets enregistrés dans le projet GCP, reçoit donc le message de *indoor*. Il le transfère au Cloud Pub/Sub du projet.
3 Au niveau de Cloud Pub/Sub, un *topic* de télémétrie, de nom **weather-telemetry-topic**, a été préalablement créé pour tous les objets du registre **weather-devices-registry**. Cloud Pub/Sub republie sur ce *topic* le message venant du MQTT Bridge, qui est le message initial de *indoor*. Un horodatage est effectué.
4 Les éléments abonnés à ce *topic* reçoivent donc le message initial de *indoor*. Nous avons choisi d'abonner une **Cloud Function for Firebase** à ce *topic*. C'est en fait une fonction hébergée dans le *cloud* qui va se déclencher à chaque fois que Cloud Pub/Sub publie sur le *topic* de télémétrie (8).
Le code de la *Cloud Function* est à écrire, par exemple, en NodeJS dans un fichier de nom **index.js**. Sa trame est donnée ci-dessous. Dans les paramètres **message** et **contexte**, on retrouve toutes les informations liées à la mesure : *timestamp*, *id* de l'objet, mesure de température, mesure d'humidité. On en profite pour formater mathématiquement ces grandeurs, en phase avec la précision du DHT22.

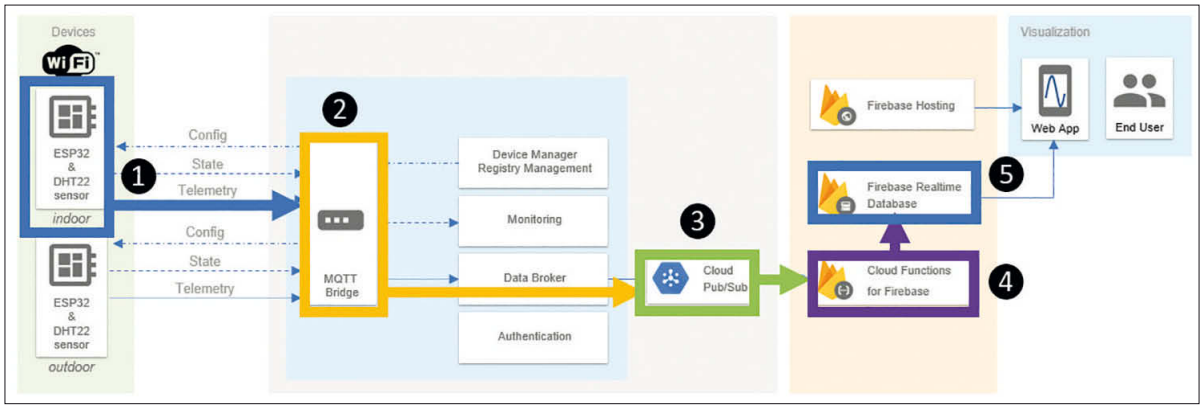
```
exports.detectTelemetryEvents = functions.pubsub.topic('weather-telemetry-topic').onPublish(
  (message, context) => {
    const timestamp = context.timestamp;
    const deviceId= message.attributes.deviceId;
    const temperature = message.json.temperature.toFixed(1);
    const humidity = Math.round(message.json.humidity);
    // TODO: Using Firebase SDK, let's populate
    // the Firebase Realtime Database with the new received measure...
  }
);
```

(7) https://cloud.google.com/iot/docs/how-tos/credentials/jwts

(8) https://firebase.google.com/docs/functions/pubsub-events



9 *Console GCP : deux objets enregistrés*



10 *Data flow de la télémétrie*

5 De ce fait, comme pressenti en lisant le code source ci-dessus, on peut faire persister la mesure obtenue dans la base NoSQL *Firebase Realtime Database*. Un aperçu de l'arborescence de cette base est donné **Figure 11**.

L'**écriture d'un site web utilisant lui aussi le Firebase SDK** permet qu'à chaque ajout dans la base de données, les courbes de la température et de l'humidité soient retracées, conférant ainsi **un aspect temps réel au projet**. Les courbes sont tracées avec la bibliothèque **plotly.js**, facile d'emploi. Le site web est hébergé par **Firebase Hosting**.

Les codes sources complet de la *Cloud Function* et du site web sont sur GitHub : https://github.com/olivierlourme/iot-store-display

Le site web est accessible ici : https://hello-cloud-iot-core.firebaseapp.com/. Il fonctionne sans interruption depuis mars 2019 (**Figure 12**).

Le coût d'utilisation (*pricing*) de la plateforme est à considérer : MQTT Bridge, Cloud Pub/Sub et Firebase. Pour des projets à petite échelle, il y a peu de chance que vous commenciez à atteindre le seuil de facturation ; il faudra quand même créer un compte pour le *billing*. Par exemple, le coût du MQTT Bridge est de 0 $ jusque 250 Mo / mois, un message étant facturé au minimum 1024 octets même si sa taille est moindre. Au-delà des 250 Mo / mois, il faut compter 0.0045 $ / Mo.

CONCLUSION

Le modèle de l'IoT est **un modèle de service**. Une plateforme *cloud* comme GCP permet de se conformer rapidement et efficacement à ce modèle :

- Elle impose un cadre pour le déploiement, la sécurité et l'exploitation.
- Elle propose des briques de services diversifiées s'interfaçant facilement entre elles. Dans un prochain projet nous aimerions par exemple tester l'association Big Query / Data Studio.

Mongoose OS participe activement à ce modèle :

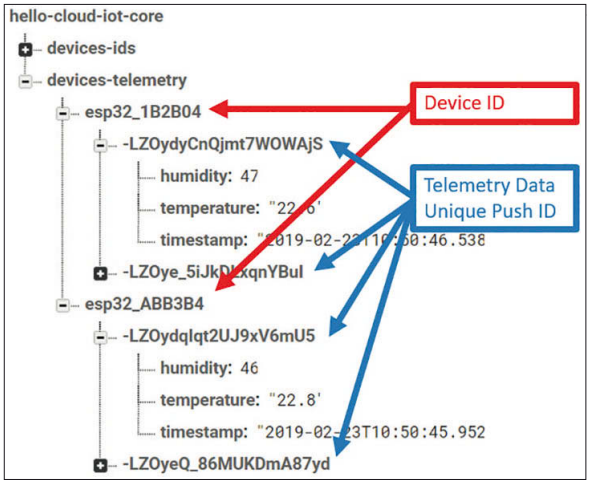
- Par ses APIs et ses outils haut niveau, il réduit la conception et la mise sur le marché. Et il fiabilise le développement du projet.
- Il est bien adapté à l'enseignement, au prototypage et à la production de milliers d'IoT. Il est livré prêt à travailler avec les principales plateforme *cloud* pour l'IoT. Par la suite, nous aimerions tester la mise à jour *Over The Air*.

Enfin, l'ESP32 est un composant très séduisant avec des ressources conséquentes pour un prix modique. Il forme un bon tandem avec Mongoose OS. Les qualités des ESP ne sont plus à démontrer !

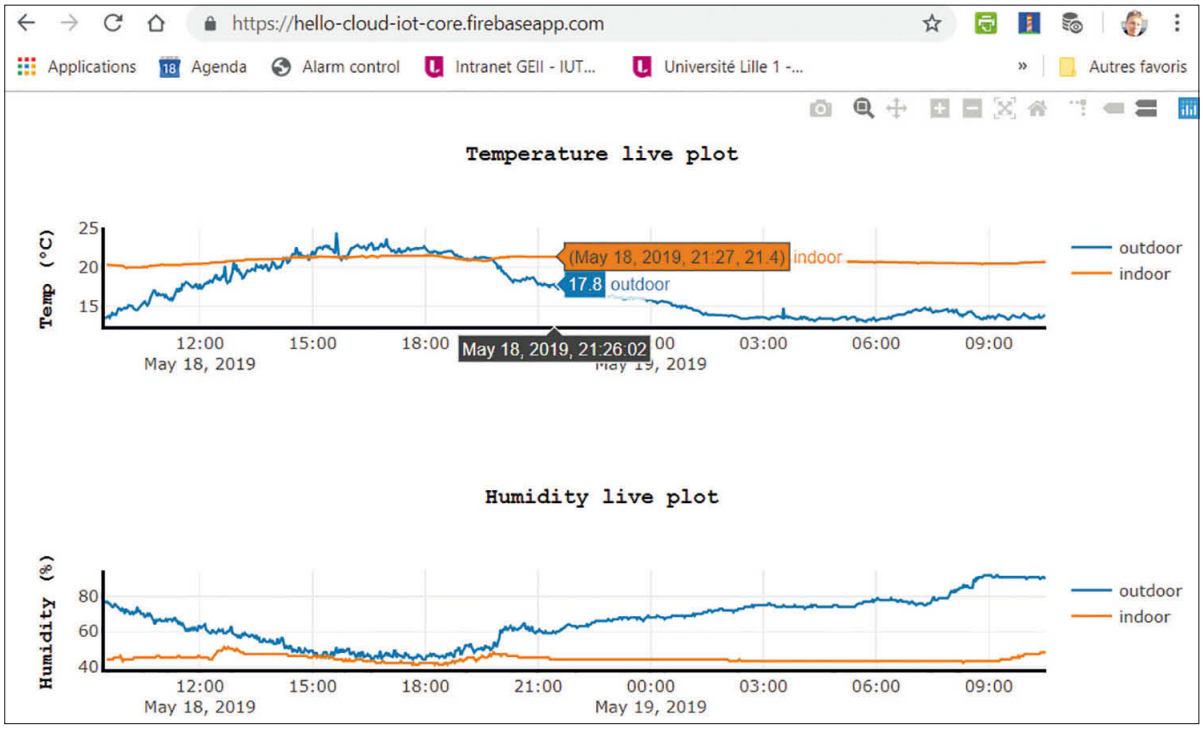
Nous espérons que la lecture de cet article vous a incités à faire passer vos projets IoT dans une autre dimension (si ce n'était pas encore le cas !). Si vous souhaitez reproduire toutes les manipulations présentées dans cet article, l'intégralité du projet est présentée dans deux *posts* Medium en anglais (9). N'hésitez pas à les consulter, à nous poser des questions et à nous faire un retour !

Bon développement IoT !

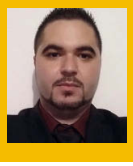
(9) http://bit.ly/2MrT9Pe *et* http://bit.ly/2VbRLCj



11 *Persistance des mesures dans Firebase Realtime Database*



12 *Allure du site final (courbes tracées avec plotly.js)*



Anthony Giretti
MVP, MCSD
Développer, Blogger, Speaker
anthony.giretti@gmail.com
http://anthonygiretti.com

Apprendre GIT en lignes de commandes dans Visual Studio 2019 et GitHub

Partie 3

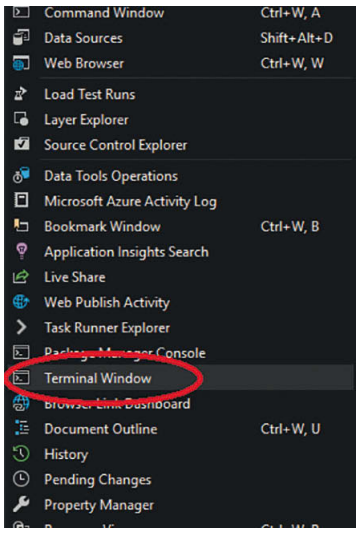
Dans le précédent article, nous avons vu comment débuter avec GIT. La partie 1 évoquait l'utilisation des bons outils pour travailler en lignes de commandes, comment créer des dépôts, effectuer des changements sur les fichiers, ainsi que la synchronisation des changements. La partie 2 quant à elle expliquait comment gérer l'historique du projet GIT, comparer les versions et annuler des commits si cela s'avère nécessaire. Dans cette partie, nous abordons, comment créer des branches de travail ; gérer les branches de travail ; résoudre des conflits ; utiliser et gérer les tags (balisage).

À noter que ce tutoriel (comme le précédent) est destiné aux débutants / intermédiaires, je n'aborderai pas toutes les options possibles de chaque commande GIT utilisée.

Rappel des outils pour utiliser GIT en lignes de commandes

Je vous ai proposé un terminal intégré à Visual Studio 2019 (et 2017) assez sympathique, il est nommé « Whack Whack » et il est disponible en téléchargement ici : https://marketplace.visualstudio.com/items?itemName=DanielGriffen.WhackWhackTerminal

Une fois installé vous devez activer le Windows Terminal dans le menu *View -> Other Windows*



J'ai recommandé également une autre extension nommée **posh-git**, permettant d'indiquer à tout moment, à côté du nom de la branche sur laquelle vous travaillez, les modifications courantes (fichiers modifiés, conflits, etc.). Il est disponible en téléchargement ici : https://git-scm.com/book/en/v2/Appendix-A%3A-Git-in-Other-Environments-Git-in-PowerShell

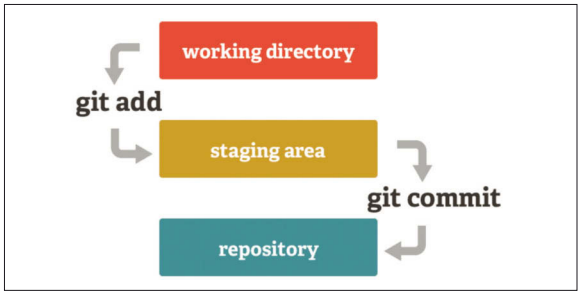
Si vous obtenez un message d'erreur indiquant que l'installation de **posh-git** est bloquée, car le script PowerShell n'est pas signé, veuillez exécuter cette commande qui lèvera les restrictions d'exécution de scripts PowerShell :

```
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
```

Si vous êtes déjà dans un repository GIT vous devriez obtenir quelque chose comme ceci :

```
E:\Codes sources\commonfeatures-webapi-aspnetcore [master ≡]>
```

Rappel du fonctionnement général de GIT



Avant d'aller plus loin dans cet article, je tiens à rappeler comment fonctionne GIT et quelles sont les différentes zones de travail. Pour la suite, je vais utiliser le terme « répertoire de travail » qui correspond au « working directory » en rouge sur l'image. Une fois les modifications ajoutées à « l'index » qui est nommé « staging area », un commit va pousser les modifications dans ce que je nomme le « dépôt local », « repository » sur l'image.

Utiliser et gérer des branches dans GIT

Lorsque l'on travaille avec GIT, on va être amené à utiliser des branches. Ces branches permettent de travailler sur différentes versions du code source de votre application qui diverge de la branche principale créée par défaut à l'initialisation d'un dépôt GIT (repository). En général on travaille avec des branches lorsque :

- Dans une équipe de développement, chaque développeur travaille sur une fonctionnalité spécifique ;
- Vous souhaitez faire des expérimentations sur votre projet, comme par exemple l'upgrade du runtime de votre application.

Créer une branche

```
git branch nomdevotrebranche
```

Ensuite, taper la commande suivante pour vous positionner dessus :

```
git checkout nomdevotrebranche
```

Exemple :

```
Terminal Window
~\source\repos\GitDemoCommandLine [master ≡]> git branch nouvelle-branche1
~\source\repos\GitDemoCommandLine [master ≡]> git checkout nouvelle-branche1
Switched to branch 'nouvelle-branche1'
~\source\repos\GitDemoCommandLine [nouvelle-branche1]>
```

Ici nous avons créé une branche à partir de la branche **master**
Il est possible de faire la même chose en une seule commande comme ceci :

git checkout -b nomdevotrebranche

```
~\source\repos\GitDemoCommandLine [nouvelle-branche1]> git checkout -b nouvelle-branche2
Switched to a new branch 'nouvelle-branche2'
~\source\repos\GitDemoCommandLine [nouvelle-branche2]>
```

Ici nous avons créé une branche à partir de la branche **nouvelle-branche1**

Fusionner deux branches

Une fois notre nouvelle fonctionnalité développée, approuvée et éprouvée nous allons la fusionner dans la branche **master** afin de conclure le développement de cette fonctionnalité.
Pour ce faire, il faut se positionner sur la branche **master** et taper la commande :

git merge nomdelabrancheafusionner

Exemple :

```
~\source\repos\GitDemoCommandLine [master ≡]> git merge nouvelle-branche1
Updating 0d5a4d4..f80f4ab
Fast-forward
 "GitDemoCommandLine/manouvellefonctionnalit\303\251.cs" | 11 ++++++++++
 1 file changed, 11 insertions(+)
 create mode 100644 "GitDemoCommandLine/manouvellefonctionnalit\303\251.cs"
~\source\repos\GitDemoCommandLine [master ↑1]>
```

Il est possible de s'en assurer une seconde fois en regardant l'historique (vu dans l'article précédent) des commits :

```
~\source\repos\GitDemoCommandLine [master ↑1]> git log
commit f80f4ab938db926d62e7382d66dc86cb6c885aa5 (HEAD -> master, nouvelle-branche1)
Author: Anthony Giretti <anthony.giretti@gmail.com>
Date:   Wed Feb 12 23:02:36 2020 -0500

    Ajout nouvelle fonctionnalit<C3><A9>
```

Supprimer une branche

Une fois notre développement terminé et notre « feature branch » fusionnée, nous n'avons plus besoin de cette dernière. Nous pouvons donc la supprimer, comme suit :

La branche locale :

git branch -D nomdelabrancheasupprimer

si la branche n'a jamais été poussée sur le serveur, sinon :

git branch -d nomdelabrancheasupprimer

Exemple :

```
~\source\repos\GitDemoCommandLine [master ↑1]> git branch -D nouvelle-branche1
Deleted branch nouvelle-branche1 (was f80f4ab).
```

La branche poussée sur le serveur :

git push nomdudepotdistant --delete nomdelabrancheasupprimer

Exemple :

```
~\source\repos\GitDemoCommandLine [master ↑1]> git push https://github.com/AnthonyGiretti/GitDemoCommandLine.git --delete nouvelle-branche2
To https://github.com/AnthonyGiretti/GitDemoCommandLine.git
 - [deleted]         nouvelle-branche2
~\source\repos\GitDemoCommandLine [master ↑1]>
```

Faire un inventaire des branches locales et remote

Parfois on peut oublier de supprimer une branche inutile, on peut à tout moment procéder à un inventaire des branches locales, et sur le serveur (remote).

git branch -a

pour les lister les branches locales et remote

git branch -r

pour les lister les branches remotes uniquement
Example :

```
~\source\repos\GitDemoCommandLine [master ↑1]> git branch -a
* master
  nouvelle-branche
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/nouvelle-branche2
~\source\repos\GitDemoCommandLine [master ↑1]> git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/nouvelle-branche2
~\source\repos\GitDemoCommandLine [master ↑1]>
```

Gérer les conflits

Il arrive parfois qu'un même fichier ait été modifié sur deux branches. Lorsque cela arrive, la fusion d'une branche dans l'autre échoue, car un conflit a été détecté :

```
~\source\repos\GitDemoCommandLine [master ↑1]> git merge nouvelle-branche2
Auto-merging GitDemoCommandLine/Controllers/DemoController.cs
CONFLICT (content): Merge conflict in GitDemoCommandLine/Controllers/DemoController.cs
Automatic merge failed; fix conflicts and then commit the result.
```

Pour régler le conflit, il vaut mieux dans ce cas-ci utiliser Visual Studio plutôt que Vim :

Avant :

```
namespace GitDemoCommandLine.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    0 references | Anthony Giretti, 122 days ago | 1 author, 2 changes
    public class DemoController : ControllerBase
    {
<<<<<<< HEAD
        0 references | Anthony Giretti, Less than 5 minutes ago | 1 author, 1 change
        public bool P1 { get; set; }
=======
        public DemoController()
        {

        }
>>>>>>> nouvelle-branche2
    }
}
```

Après :

```
namespace GitDemoCommandLine.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference | Anthony Giretti, 122 days ago | 1 author, 2 changes
    public class DemoController : ControllerBase
    {
        0 references | Anthony Giretti, Less than 5 minutes ago | 1 author, 1 change
        public bool P1 { get; set; }
        0 references | 0 changes | 0 authors, 0 changes
        public DemoController()
        {

        }
    }
}
```

Une fois le conflit résolu, il suffit d'ajouter le / les fichiers modifiés à l'index, commiter et pousser vers le serveur.
Noter (en entouré rouge) que **posh-git** si des conflits sont présents :

```
~\source\repos\GitDemoCommandLine [master ↑1 +0 ~0 -0 ~1 | +0 ~0 -0 ~1 ]> git add --all
~\source\repos\GitDemoCommandLine [master ↑1 +0 ~1 -0 ~]> git commit -m "résolution de conflit"
[master 17edfee] résolution de conflit
~\source\repos\GitDemoCommandLine [master ↑3]> git push
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (15/15), 1.37 KiB | 703.00 KiB/s, done.
Total 15 (delta 9), reused 0 (delta 0)
remote: Resolving deltas: 100% (9/9), completed with 3 local objects.
To https://github.com/AnthonyGiretti/GitDemoCommandLine.git
   8946aa8..17edfee  master -> master
~\source\repos\GitDemoCommandLine [master ≡]>
```

Utiliser et gérer les tags

Le balisage permet aux développeurs de marquer des points de contrôle importants au cours du développement de leurs projets. Par exemple, les versions des versions logicielles peuvent être étiquetées. (Ex: v1.1.0) Il vous permet essentiellement de donner à un commit un nom spécial (tag).
Pour créer un tag, tapez la commande suite :

git tag -a nomdutag -m « message du tag »

Exemple :

```
~\source\repos\GitDemoCommandLine [master ≡]> git tag v1 -m "Version 1.0 du projet"
```

Pour lister les tags existants taper la commande suivante : *git tag*

Exemple :

```
~\source\repos\GitDemoCommandLine [master ≡]> git tag
v1
```

Il est possible de créer des tags avec moins de métadonnées, on les appelle les « lightweight tags »
Pour en créer un taper la commande suivante : *git tag v2-lw* puis vérifier sa création :

```
~\source\repos\GitDemoCommandLine [master ≡]> git tag v2-lw
~\source\repos\GitDemoCommandLine [master ≡]> git tag
v1
v2-lw
```

La commande *git show* permet de voir les métadonnées du tag :

```
~\source\repos\GitDemoCommandLine [master ≡]> git show v1
tag v1
Tagger: Anthony Giretti <anthony.giretti@gmail.com>
Date:   Thu Feb 13 01:29:57 2020 -0500

Version 1.0 du projet

commit 17edfeeff5544dcea4eb2756d1fe67d5add56813 (HEAD -> master, tag: v2-lw, tag: v1, origin/master, origin/HEAD)
Merge: ec4830c 5643665
Author: Anthony Giretti <anthony.giretti@gmail.com>
Date:   Thu Feb 13 00:52:38 2020 -0500

    r<C3><A9>solution de conflit

diff --cc GitDemoCommandLine/Controllers/DemoController.cs
index 89504a5,88359ab..4814e62
```

Il est également possible de créer un tag à partir d'un numéro de commit avec la commande :

git tag -a nomdutag -m « message du tag » numerodecommit

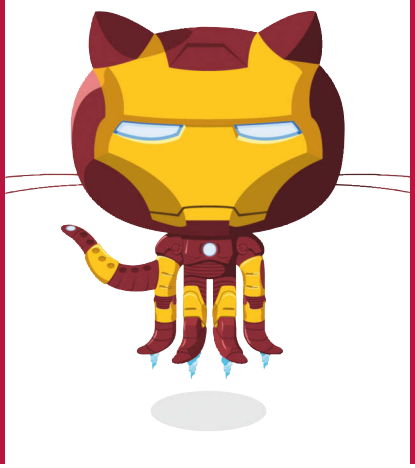
```
~\source\repos\GitDemoCommandLine [master ≡]> git tag -a -m "Préversion" v0.5 5643665ab2dda6e9724b5e8435585830916d30347
~\source\repos\GitDemoCommandLine [master ≡]> git tag
v0.5
v1
v2-lw
~\source\repos\GitDemoCommandLine [master ≡]>
```

Pour terminer la commande *git push nomdutag* (comme le commit) permet de pousser le tag vers le serveur distant. La commande *git checkout nomdutag* (comme le commit) permet de se déplacer dans les tags. Enfin *git tag –delete nomdutag* permet de supprimer un tag.
Exemples :

```
~\source\repos\GitDemoCommandLine [(v1)]> git push origin v2-lw
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/AnthonyGiretti/GitDemoCommandLine.git
 * [new tag]         v2-lw -> v2-lw
~\source\repos\GitDemoCommandLine [(v1)]> git tag --delete v0.5
Deleted tag 'v0.5' (was d7e2d59)
~\source\repos\GitDemoCommandLine [(v1)]> git checkout v2-lw
HEAD is now at 17edfee résolution de conflit
~\source\repos\GitDemoCommandLine [(v1)]>
```

Conclusion

Nous venons de voir comment gérer les branches d'un projet, comment également gérer les conflits et comment gérer les tags de vos versions applicatives pour obtenir un travail en collaboration efficace. Nous avons dans cette suite de trois articles vu les commandes les plus courantes pour travailler avec GIT.
Ce dernier étant très puissant, nous n'avons pas exploré toutes les possibilités qui sont offertes aux développeurs d'applications. Cependant vous saurez très bien gérer le cycle de vie de votre projet si vous maîtrisez tout ce que l'on a appris ensemble.
Si vous voulez en savoir davantage, vous trouverez des tutoriels forts intéressants à cette adresse : https://www.atlassian.com/fr/git.
Bon apprentissage !



Lætitia Avrot *est experte PostgreSQL chez EnterpriseDB. Elle est co-fondatrice du mouvement Postgres Women, membre du Postgres Advocacy Group et membre du Postgres Funds Group. Elle a aussi été pendant un an membre du comité du Code de Conduite de la communauté PostgreSQL.Elle a écrit plusieurs patchs pour le projet PostgreSQL et donne régulièrement des conférences dans des événements communautaires.*

Le SQL avancé pour de vraies performances

Partie 3

Le SQL est souvent sous-estimé par les développeurs car malheureusement peu enseigné dans le supérieur, que ce soit en France ou à l'étranger... Dans cet article, nous allons découvrir trois fonctionnalités très avancées du SQL : les jointures latérales, les window fonctions et les CTE récursives. Les performances de ces 3 fonctionnalités seront comparées aux performances d'un code équivalent en python. Suite et fin.

COMMON TABLE EXPRESSION OU CTE

Comme je vous l'ai annoncé en début d'article, le sql est Turing complete. cela n'est possible que depuis l'ajout de la récursivité au sql. Cet ajout a eu lieu dans la norme SQL:1999.

Une CTE est ce qu'on appelle aussi "la clause with" d'une requête SQL. Elle permet de déporter une sous-requête en début de requête. Cette sous-requête est ensuite utilisable dans toutes les autres clauses, pour peu qu'elle ait été référencée dans la clause from. L'intérêt de mettre la sous-requête en début, c'est que c'est exactement dans cet ordre qu'un humain a besoin de le lire.

Voici la syntaxe d'une CTE:

```
with ctename as
 (
  [any sql query]
 )
[any sql query using that cte]
```

Il est même possible d'utiliser plusieurs CTE dans une requête en les séparant par des virgules (il ne faut pas répéter le mot-clé with).

Premier exemple : la recherche de doublons logiques

Prenons un exemple assez connu de sous-requête: la recherche de doublons. Notre table des membres du club comporte une clé primaire artificielle (memid). La modélisation de cette table est mauvaise parce qu'aucune contrainte d'unicité n'a été ajoutée sur la clé primaire naturelle. Cela permet donc la création de doublons logiques (id différent mais même personne).

Pour lister les doublons, il suffit de compter le nombre de personnes identiques (on va considérer, pour simplifier, que deux personnes sont identiques si elles ont le même prénom, le même nom et la même date d'adhésion au club) et de ne ressortir que les lignes pour lesquelles ce comptage est supérieur à 1. Avant de commencer à s'amuser, ajoutons des doublons logiques à notre table :

```
insert into
 cd.members
 (
  select
   memid+100,
   surname,
   firstname,
   address,
   zipcode,
   telephone,
   recommendedby,
   joindate
  from cd.members
  where firstname in ('David', 'Darren', 'Tim')
 );
```

Les prénoms choisis sont les plus courants de notre table, ce qui permet d'avoir 7 doublons logiques.

Voici la requête permettant d'obtenir les doublons:

```
select
 members.memid,
 members.surname,
 members.firstname,
 members.address,
 members.zipcode,
 members.telephone,
 members.recommendedby

from
 cd.members as members
inner join
 /* chaque membre avec un count(*) > 1 est un doublon /*
 (
  select
   surname,
   firstname,
   joindate
  from cd.members
  group by
   surname,
   firstname,
   joindate
  having count(*) > 1
 ) as duplicates
on members.surname = duplicates.surname
 and members.firstname = duplicates.firstname
 and members.joindate = duplicates.joindate;

order by
 members.surname,
 members.firstname,
 members.memid
```

Voyons maintenant le rapport avec les CTE. Chaque sous-requête peut être réécrite comme une CTE.

```
with duplicates as
 /* chaque membre avec un count(*) > 1 est un doublon /*
 (
  select
   surname,
   firstname,
   joindate
  from cd.members
  group by
   surname,
   firstname,
   joindate
  having count(*) > 1
 )

select
 members.memid,
 members.surname,
 members.firstname,
 members.address,
 members.zipcode,
 members.telephone,
 members.recommendedby

from
 cd.members as members
inner join
 duplicates
on members.surname = duplicates.surname
 and members.firstname = duplicates.firstname
 and members.joindate = duplicates.joindate

order by
 members.surname,
 members.firstname,
 members.memid
```

C'était vraiment simple! Voyez comme la requête devient plus facile à lire maintenant que la sous-requête a été extraite!
Il y a cependant un inconvénient aux CTE. Certains optimiseurs ne les comprennent pas bien, ce qui peut entraîner des pertes de performance. Postgres avant la version 12 pouvait présenter ce problème. Voici donc le conseil que je donnais aux développeurs : "écrivez dans une premier temps avec des CTE, comparez les performances avec des sous-requêtes, si les performances sont moins bonnes, passez en sous-requête et documentez ce changement et sa raison." Maintenant que la version 12 de Postgres est sortie, ce n'est plus un problème.

La récursivité : syntaxe

Pour l'instant, nous n'avons pas encore été voir la récursivité, nous avons juste fait une petite modification d'écriture. Commençons par le commencement : dans quel cas pourrait-on avoir besoin de récursivité en SQL ?
Nous allons prendre un exemple très simple : dans notre table des membres du club, nous avons l'id du membre qui a recommandé une personne. Cette personne a certainement été elle même recommandée par quelqu'un et ainsi de suite... Nous voudrions pouvoir récupérer toute cette chaîne de recommandation avec une seule requête SQL. Et c'est possible!
Voici la syntaxe d'une CTE récursive :

```
with recursive [ctename] as
 (
  [first sql query]
  union all
  [recursive sql query using the first sql query]
 )
[any query using the cte]
```

Exemple de récursivité : parrainages directs et indirects

Appliquons maintenant cette syntaxe à notre recherche des parrains directs ou indirects d'un membre particulier de notre club (nous allons faire cette recherche pour l'id 27).

```
\set myid 27

with recursive recommenders(recommender) as
 (
  /* Get the first member's mentor */
  select recommendedby
  from cd.members
  where memid = :myid

  union all

  /* Get all the others mentors */
  select mems.recommendedby
  from recommenders recs
  inner join cd.members mems
   on mems.memid = recs.recommender
 )

/* Display the result */
select
 recs.recommender,
 mems.firstname,
 mems.surname

from recommenders recs
inner join cd.members mems
 on recs.recommender = mems.memid

order by memid desc;
```

J'utilise ici une variable. Sous psql (le seul client officiel de PostgreSQL), les variables sont préfixées avec deux points (:). Pour valoriser cette variable, on utilise la méta-commande \set variable value.
Voyons maintenant comment nous pourrions écrire cela en python.

```
#! /usr/bin/env python3

import sys
```

```
import psycopg2

CONNSTRING = "service=pgexercises"

def get_member_names_and_recommender(id):
 #Fetch the recommender's name and last name from his id
 sql = """
  select
   firstname,
   surname,
   recommendedby
  from cd.members
  where memid = %s
 """
 pgconn = psycopg2.connect(CONNSTRING)
 cursor = pgconn.cursor()
 cursor.execute(sql, (id,))

 # Id is a primary key -> only one result expected
 (firstname, surname, recommendedby) = cursor.fetchone()

 print('{0}, {1}, {2}'.format(recommendedby, firstname, surname))

 if recommendedby:
  # If recommendedby is not null, we haven't reached the end of
  # the recommendation chain
  get_member_names_and_recommender(recommendedby)
 return result

if __name__ == '__main__':
 get_member_names_and_recommender(27)
```

Ma requête avec cte récursive s'exécutait en 1 ms alors que mon programme python met plus de 500 ms à renvoyer le même résultat. Ma requête SQL est donc de l'ordre de 500 fois plus rapide que mon programme python.

Conclusion

Le SQL est un langage de programmation avancé qui permet de faire beaucoup de choses. C'est aussi un langage déclaratif, c'est-à-dire qu'il n'est pas nécessaire de décomposer notre problème en différentes étapes pour pouvoir le résoudre, mais qu'il est nécessaire d'arriver à exprimer son problème pour le résoudre.

Je tenais à vous montrer en détail ces fonctionnalités super puissantes du SQL, mais je tiens à vous rappeler de bien les utiliser avec modération, tant on a vu qu'une utilisation à mauvais escient de ces fonctionnalités pouvait aussi nuire aux performances.

Une bonne règle pour savoir si ces fonctionnalités peuvent être utile est de toujours penser KISS. Si vous n'êtes pas sûrs, n'hésitez pas à tester et à bencher pour savoir si cette fonctionnalité est vraiment intéressante dans votre cas.

Dans tous les cas, sachez qu'effectuer une seule requête sera plus rapide que d'en effectuer plusieurs. Si vos benchs vous montrent le contraire, regardez de près vos requêtes SQL et leurs plans d'exécution, il y a de fortes chances qu'il y ait matière à optimisation.

[3] https://www.postgresql.org/docs/current/functions-window.html

Aurélie Vache
Cloud Dev(Ops) chez Continental
Duchess France & DevFest Pitchouns Leader
Toulouse Data Science core-team & Marraine Elles bougent
@aurelievache

Tester un service gRPC en Go avec les Table Driven Tests

Partie 1

Tout le monde sait qu'une couverture de code à 100% n'existe pas et n'apporte aucune valeur ajoutée. En fait, au quotidien, ce que nous voulons vraiment c'est tester notre business logic, l'intelligence de notre application.
Dans cet article nous allons partir d'une petite application CLI codée en Go qui ne possède pas encore de tests unitaires, puis nous allons réaliser quelques tests unitaires de services gRPC.

Commençons par jeter un œil à notre application

Concevoir une CLI en Go est un jeu d'enfant et si vous avez lu un de mes précédents articles publiés dans ce magazine vous vous êtes déjà aperçu que c'est vraiment le cas, on peut créer une application de CLI en quelques minutes.
Je suis donc partie sur la création d'une CLI pour mon application, mais j'ai un peu corsé l'exercice en créant, cette fois-ci, une CLI permettant de lancer un serveur gRPC et de l'interroger avec un client gRPC.
Pour information, le gRPC est un protocole open source permettant de créer des clients et serveurs RPC via HTTP/2. Les données sont sérialisées et désérialisées grâce à Protocol Buffers.
Je vous invite à jeter un œil au code source et nous allons, étape par étape, explorer l'application et nous apercevoir qu'il n'y a aucun test unitaire qui a été créé !
Code source : https://github.com/scraly/hello-world
Concernant la gestion des dépendances, je suis directement partie sur l'utilisation des Go modules, grâce à cela vous n'avez plus à vous occuper du GOPATH.
Commencez donc par cloner le repository hébergé sur github (à l'extérieur de votre GOPATH, si vous en avez un) :

```
$ git clone https://github.com/scraly/hello-world.git
```

La première chose à savoir concernant notre application, est que pour nous faciliter la vie, pour builder, gérer les dépendances, tester, générer des fichiers/des mocks, formater notre code, exécuter les tests statiques … nous utilisons un magefile (https://magefile.org/). Il s'agit d'un Make file like codé en Go très pratique.
Grâce à ce magefile nous n'avons pas à exécuter toutes les commandes à rallonge qui nous permettront de générer les mocks, tester les fichiers statiques, exécuter les tests unitaires … et builder notre application pour générer le binaire.
Après avoir cloné le repository git, je vous invite à exécuter la commande suivante qui vous permettra de télécharger et d'installer les outils nécessaires :

```
$ go run mage.go -d tools
```

Et c'est tout ! Pas besoin d'aller récupérer des outils se trouvant dans 50 repositories sur GitHub, de faire des go get, des curls… en une ligne de commande vous récupérez tous les binaires utiles qui vous permettront de builder, d'exécuter un linter, vos tests unitaires, vos check de licences, de générer vos mocks …
La seule chose à faire est à présent de faire un build complet de notre application :

```
$ go run mage.go
 _ _      _ _       __    __          _    _
|||| ___ |||| ___   \\   // ___  _ __ || __||
||_|| / _ \|||| / _ \  \\ /\ // / _ \ |'__||| / _` |
| _ || __/|||||(_) |  \V  V/ |(_)||| |||(_||
|_||_| \___||_||_| \___/    \_/\_/  \___/ |_|  |_| \__,_|

# Build Info -----------------------------------------------------------------
Go version : go1.12
Git revision : f7ee9e3
Git branch : master
Tag : 0.0.1

# Core packages --------------------------------------------------------------
## Vendoring dependencies
## Generate code
### Protobuf
#### Lint protobuf
## Format everything
## Lint go code
## Running unit tests
∅ cli/hello-world
∅ cli/hello-world/cmd
∅ cli/hello-world/config
∅ cli/hello-world/dispatchers/grpc
∅ internal/services/pkg/v1
√ internal/services/pkg/v1/greeter (1.125s)
∅ internal/version
∅ pkg/protocol/helloworld/v1

DONE 0 tests in 3.425s

# Artifacts ------------------------------------------------------------------
> Building hello-world [github.com/scraly/hello-world/cli/hello-world]
```

Super ! Toutes les étapes se sont déroulées correctement, et on obtient à la fin notre petit binaire prêt à l'emploi !

```
$ ll bin
total 39232
-rwxr-xr-x  1 uidn3817  CW01\Domain Users   19M  7 août 18:55 hello-world
```

Jetons un œil à notre application, que permet-elle de faire au juste ? Commençons par lancer notre serveur gRPC :

```
$ bin/hello-world server grpc
```

Puis dans un autre onglet de votre terminal, lancez le client gRPC afin d'appeler notre méthode sayHello :

```
$ bin/hello-world client greeter sayHello -s 127.0.0.1:5555 <<< '{"name": "me"}'
{
 "message": "hello me"
}%
```

Notre application fonctionne correctement, elle fait ce que nous voulions qu'elle fasse, c'est à dire, nous répondre *hello + un champ string* lorsque l'on appelle sayHello.

Mais, suis-je obligé de builder et de tout regénérer afin d'exécuter les tests unitaires ?

Non, pas du tout, avec la commande suivante on peut facilement exécuter nos TU :

```
$ go run mage.go go:test
## Running unit tests
∅ cli/hello-world (1ms)
∅ cli/hello-world/cmd
∅ cli/hello-world/config
∅ cli/hello-world/dispatchers/grpc
∅ internal/services/pkg/v1
√ internal/services/pkg/v1/greeter (1.092s)
∅ internal/version
∅ pkg/protocol/helloworld/v1

DONE 0 tests in 3.239s
```

Comme vous pouvez le voir, 0 tests unitaires ont été exécutés avec succès, nous allons nous en occuper dans la suite, mais avant cela, il est nécessaire de savoir ce qu'il se cache derrière cette target go:test (dans notre fichier magefile.go):

```
// Test run go test
func (Go) Test() error {
  color.Cyan("## Running unit tests")
  sh.Run("mkdir", "-p", "test-results/junit")
  return sh.RunV("gotestsum", "--junitfile", "test-results/junit/unit-tests.xml", "--", "-short",
"-race", "-cover", "-coverprofile", "test-results/cover.out", "./...")
}
```

Le code ci-dessus montre que l'on utilise l'outil **gotestsum** pour exécuter nos tests unitaires et que les résultats des tests sont exportés au format JUnit dans un fichier nommé *test-results/junit/unit-tests.xml*.
Vous pouvez donc exécuter les tests à travers le magefile ou bien en utilisant directement gotestsum (si au préalable vous installez l'utilitaire sur votre machine) :

```
$ gotestsum --junitfile test-results/junit/unit-tests.xml -- -short -race -cover -coverprofile
 test-results/cover.out ./…
```

Gotestsum

Gotestsum, c'est quoi encore ce nouvel outil ? Go test ne suffit pas ? Répondons à cette question.
L'un des avantages du langage Go est son écosystème d'outils qui nous permet de nous faciliter la vie. Pour tester votre code il vous suffit de faire :

```
$ go test ./…
?       github.com/scraly/hello-world/cli/hello-world  [no test files]
?       github.com/scraly/hello-world/cli/hello-world/cmd     [no test files]
?       github.com/scraly/hello-world/cli/hello-world/config  [no test files]
?       github.com/scraly/hello-world/cli/hello-world/dispatchers/grpc  [no test files]
?       github.com/scraly/hello-world/internal/services/pkg/v1   [no test files]
ok      github.com/scraly/hello-world/internal/services/pkg/v1/greeter    0.037s [no tests
 to run]
?       github.com/scraly/hello-world/internal/version[no test files]
?       github.com/scraly/hello-world/pkg/protocol/helloworld/v1   [no test files]
```

L'outil de test est intégré à Go, pratique, mais pas très user friendly et intégrable dans toutes les solutions de CI/CD par exemple.
C'est pour cela qu'a été conçu **gotestsum**, un petit utilitaire écrit en Go qui exécute les tests avec *go test* en améliorant l'affichage des résultats, plus human friendly, avec de la couleur, un rapport pratique et avec un output possible directement au format JUnit.

On fait comment pour tester du gRPC ?

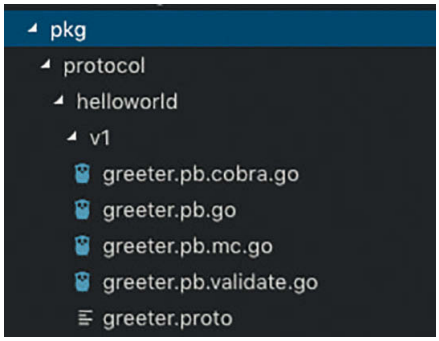
Notre appli est un client/serveur gRPC donc cela signifie que lorsque nous appelons la méthode say hello, une communication client/serveur est déclenchée, mais hors de question de tester les appels gRPC à travers nos tests unitaires, nous allons uniquement tester l'intelligence de notre application.
Notre serveur gRPC repose sur un fichier en protobuf nommé pkg/protocol/helloworld/v1/greeter.proto:

```
syntax = "proto3";

package helloworld.v1;

option csharp_namespace = "Helloworld.V1";
option go_package = "helloworldv1";
option java_multiple_files = true;
option java_outer_classname = "GreeterProto";
option java_package = "com.scraly.helloworld.v1";
option objc_class_prefix = "HXX";
option php_namespace = "Helloworld\\V1";

// The greeting service definition.
```

```
service Greeter {
 // Sends a greeting.
 rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
 string name = 1;
}

// The response message containing the greetings
message HelloReply {
 string message = 1;
}
```

À partir de ce proto nous avons généré les fichiers .go grâce à la commande gen protobuf :

```
$ go run mage.go gen:protobuf
### Protobuf
#### Lint protobuf
```



La bibliothèque standard de Go nous fournit de base un package nous permettant de tester notre programme Go, c'est le package *testing*. Un fichier de test en Go doit obligatoirement être placé dans le même dossier que le fichier que nous voulons tester et terminé par **_test.go**. Il faut suivre ce formalisme afin que l'exécutable Go reconnaisse nos fichiers de test.
La première étape consiste donc à créer un fichier **service_test.go** que l'on place à côté de service.go.
Nous allons nommer le package de ce fichier de test **greeter_test** et nous allons commencer par importer le package **testing** et créer la fonction que nous allons tester, ce qui donne :

```
package greeter_test

import (
  "testing"
)

func TestSayHello(t *testing.T) {

}
```

Attention : chaque fonction de test doit être écrite sous la forme func **Test***(t *testing.T)**, où *** représente le nom de la fonction que nous voulons tester.

Occupons-nous des tests avec les Table Driven Tests

Dans notre application, nous n'allons pas tout tester, mais nous allons commencer par tester notre *business logic*, l'intelligence de notre application. Dans notre appli ce qui nous intéresse est ce qui se trouve dans *service.go* :

```
hello-world
 internal
  services
   pkg
    v1
     greeter
      service.go
```

```
package greeter

import (
  "context"
  ...
)

type service struct {
}

// New services instance
func New() apiv1.Greeter {
  return &service{}
}

// ------------------------------------------------------------------------

func (s *service) SayHello(ctx context.Context, req *helloworldv1.HelloRequest) (*helloworl
dv1.HelloReply, error) {
  res := &helloworldv1.HelloReply{}

  // Check request
  if req == nil {
    log.Bg().Error("request must not be nil")
    return res, xerrors.Errorf("request must not be nil")
  }

  if req.Name == "" {
    log.Bg().Error("name but not be empty in the request")
    return res, xerrors.Errorf("name but not be empty in the request")
  }

  res.Message = "hello " + req.Name

  return res, nil
}
```

Comme vous pouvez le voir, afin de couvrir un maximum de notre code, nous devrons tester au moins trois cas :
- la request est nil
- la request est empty (le champ name n'est pas renseigné)
- le champ name est renseigné dans la request

Table Driven Tests

Au lieu de créer une méthode de test par cas de test, et de faire moult copier-coller, nous allons partir sur des **Table Driven Tests**, ce qui va nous faciliter grandement la vie.

Écrire de bons tests n'est pas une mince affaire, mais dans de nombreuses situations, on peut couvrir beaucoup de choses avec des tests pilotés par des tables (table driven tests) : chaque entrée du tableau est un scénario de test complet avec les entrées et les résultats attendus, et parfois avec des informations complémentaires et la sortie de test est facilement lisible. Si, d'habitude, vous vous retrouvez à utiliser le copier-coller lors de l'écriture d'un test, demandez-vous si la refactorisation dans un test piloté par une table peut être une meilleure option.

Étant donné une table de cas de test, le test réel parcourt simplement toutes les entrées de la table et effectue pour chaque entrée les tests nécessaires. Le code de test est écrit une fois et est amorti sur toutes les entrées de table. Il est donc plus facile d'écrire un test minutieux avec de bons messages d'erreur.

Exemple :

Je commence par définir mes cas de test :

```
testCases := []struct {
    name     string
    req      *helloworldv1.HelloRequest
    message  string
    expectedErr bool
  }{
    {
      name:     "req ok",
      req:      &helloworldv1.HelloRequest{Name: "me"},
      message:  "hello me",
      expectedErr: false,
    },
    {
      name:     "req with empty name",
      req:      &helloworldv1.HelloRequest{},
      expectedErr: true,
    },
    {
      name:     "nil request",
      req:      nil,
      expectedErr: true,
    },
  }
```

La bonne pratique est de fournir un nom (un name) à notre cas de test, ainsi si une erreur survient lors de son exécution le nom du cas de test sera écrit et nous saurons en un coup d'œil où se situe notre erreur.

Puis je boucle dans tous les cas de test, j'appelle mon service et selon si j'attends ou non une erreur je teste son existence, sinon je teste si le résultat est celui attendu :

```
for _, tc := range testCases {
  testCase := tc
  t.Run(testCase.name, func(t *testing.T) {
    t.Parallel()
    g := NewGomegaWithT(t)

    ctrl := gomock.NewController(t)
    defer ctrl.Finish()
    ctx := context.Background()

    // call
    greeterSvc := greeter.New()
    response, err := greeterSvc.SayHello(ctx, testCase.req)

    // assert results expectations
    if testCase.expectedErr {
      g.Expect(response).ToNot(BeNil(), "Result should be nil")
      g.Expect(err).ToNot(BeNil(), "Result should be nil")
    } else {
      g.Expect(response.Message).To(Equal(testCase.message))
    }
  })
}
```

La suite dans le numéro 239

Philippe BOULANGER
Manager des expertises C/C++ et Python
www.invivoo.com

INVIVOO BEYOND TECH

Appel de fonctions

Les processeurs ont une structure interne basée sur un nombre de registres limité et un accès à la mémoire. Le rôle du compilateur est de transformer le code C et/ou C++ en langage machine. Nos codes sont structurés avec des fonctions et des classes. Nous allons nous concentrer sur la façon dont le compilateur transforme un appel de fonction en langage machine et quelles sont les conséquences pour le programme.

Contexte

Cela s'appelle une convention d'appel qui peut varier selon le compilateur et parfois l'architecture cible (OS et/ou processeur). La convention d'appel va influer sur les points suivants :

- L'ordre d'empilage des arguments (à partir du premier ou à partir du dernier) ;
- Passage ou non des paramètres par des registres du processeur ;
- Appel de la fonction avec une adresse absolue ou relative par rapport à la position actuelle du « instruction pointer » ;
- Comment sont désempilés les arguments ? (dans la fonction appelée ou par l'appelant) ;
- La méthode de retour du résultat de la fonction.

Le but de cet article est de vous aider à comprendre ce que fait le compilateur et éventuellement comment rendre plus performant votre code.

La convention par défaut du C

La convention définie par le langage C consiste à empiler les arguments de droite à gauche qui seront dépilés par l'appelant. Le résultat est retourné via un registre du processeur (sur Intel : AL, AX, EAX ou RAX).

Nous allons vérifier cela en utilisant un petit exemple :

```
#include <stdio.h>


int test_cdecl(int x, const char *text)
{
  const char *ptr = text;
  while (0 != *ptr)
  {
    ptr++;
  }
  return x + int( ptr - text );
}


int main( int argc, char* argv[] )
{
  // cdecl
  int x = test_cdecl( 0, "toto" );
  printf( "test_cdecl( 0, \"toto\" ) = %d\n", x );

  return 0;
}
```

Nous avons deux fonctions main et test_cdecl. Le code généré par le compilateur (en debug pour garder le lien C++ vers langage machine) est le suivant :

```
int test_cdecl(int x, const char *text)
{
   009E1720 push      ebp
   009E1721 mov       ebp,esp
   009E1723 sub       esp,0CCh
   009E1729 push      ebx
   009E172A push      esi
   009E172B push      edi
   009E172C lea       edi,[ebp-0CCh]
   009E1732 mov       ecx,33h
   009E1737 mov       eax,0CCCCCCCCh
   009E173C rep stos  dword ptr es:[edi]
 const char *ptr = text;
   009E173E mov       eax,dword ptr [text]
   009E1741 mov       dword ptr [ptr],eax
 while (0 != *ptr)
   009E1744 mov       eax,dword ptr [ptr]
   009E1747 movsx     ecx,byte ptr [eax]
   009E174A test      ecx,ecx
   009E174C je        test_cdecl+39h (09E1759h)
 {
   ptr++;
   009E174E mov       eax,dword ptr [ptr]
   009E1751 add       eax,1
   009E1754 mov       dword ptr [ptr],eax
 }
   009E1757 jmp       test_cdecl+24h (09E1744h)
 return x + int( ptr - text );
   009E1759 mov       eax,dword ptr [ptr]
   009E175C sub       eax,dword ptr [text]
   009E175F add       eax,dword ptr [x]
}
   009E1762 pop       edi
   009E1763 pop       esi
   009E1764 pop       ebx
}
   009E1765 mov       esp,ebp
   009E1767 pop       ebp
   009E1768 ret
```

```
int main( int argc, char* argv[] )
{
    009E1840 push     ebp
    009E1841 mov      ebp,esp
    009E1843 sub      esp,0CCh
    009E1849 push     ebx
    009E184A push     esi
    009E184B push     edi
    009E184C lea      edi,[ebp-0CCh]
    009E1852 mov      ecx,33h
    009E1857 mov      eax,0CCCCCCCCh
    009E185C rep stos   dword ptr es:[edi]
  // cdecl
  int x = test_cdecl( 0, "toto" );
    009E185E push     offset string "toto" (09E7B30h)
    009E1863 push     0
    009E1865 call     test_cdecl (09E1154h)
    009E186A add      esp,8
    009E186D mov      dword ptr [x],eax
  printf( "test_cdecl( 0, \"toto\" ) = %d\n", x );
    009E1870 mov      eax,dword ptr [x]
    009E1873 push     eax
    009E1874 push     offset string "test_cdecl( 0, "toto" ) = %d\n" (09E7B38h)
    009E1879 call     _printf (09E1339h)
    009E187E add      esp,8

  return 0;
    009E1881 xor      eax,eax
}
    009E1883 pop      edi
    009E1884 pop      esi
    009E1885 pop      ebx
    009E1886 add      esp,0CCh
    009E188C cmp      ebp,esp
    009E188E call     __RTC_CheckEsp (09E1122h)
    009E1893 mov      esp,ebp
    009E1895 pop      ebp
    009E1896 ret
```

Concentrons-nous sur l'appel à test_cdecl :

```
009E185E push     offset string "toto" (09E7B30h)
009E1863 push     0
009E1865 call     test_cdecl (09E1154h)
009E186A add      esp,8
```

En bleu, « push » est l'instruction processeur pour empiler les arguments : on empile bien de droite à gauche les arguments… Puis on appelle la routine 'test_cdecl' via l'instruction « call » (en vert). Pour dépiler les arguments, on change la valeur du registre de pile (en rouge).

Le convention standard

Cette convention est la convention utilisée par défaut dans le langage Pascal et celle de toutes les API du Win32 SDK. Elle a souvent été utilisée pour faire communiquer des écrits dans des langages différents. Dans cette convention les arguments sont empilés de droite à gauche. La fonction appelée dépile elle-même ses arguments.

Que ce soit sous gcc ou Visual C++, il faut employer le mot-clé __sdtcall entre le type de retour et le nom de la fonction.
Nous allons vérifier cela en utilisant le même petit exemple que pour cdecl mais en y changeant la convention :

```
int __stdcall test_stdcall(int x, const char *text)
{
  const char *ptr = text;
  while (0 != *ptr)
  {
    ptr++;
  }
  return x + int(ptr - text);
}



int main( int argc, char* argv[] )
{
  // stdcall
  int x = test_stdcall(0, "toto");
  printf("test_stdcall( 0, \"toto\" ) = %d\n", x);

  return 0;
}
```

Le code généré par le compilateur (en debug) est le suivant :

```
int __stdcall test_stdcall(int x, const char *text)
{
    001A1770 push     ebp
    001A1771 mov      ebp,esp
    001A1773 sub      esp,0CCh
    001A1779 push     ebx
    001A177A push     esi
    001A177B push     edi
    001A177C lea      edi,[ebp-0CCh]
    001A1782 mov      ecx,33h
    001A1787 mov      eax,0CCCCCCCCh
    001A178C rep stos   dword ptr es:[edi]
  const char *ptr = text;
    001A178E mov      eax,dword ptr [text]
    001A1791 mov      dword ptr [ptr],eax
  while (0 != *ptr)
    001A1794 mov      eax,dword ptr [ptr]
    001A1797 movsx    ecx,byte ptr [eax]
    001A179A test     ecx,ecx
    001A179C je       test_stdcall+39h (01A17A9h)
  {
    ptr++;
    001A179E mov      eax,dword ptr [ptr]
    001A17A1 add      eax,1
    001A17A4 mov      dword ptr [ptr],eax
  }
    001A17A7 jmp      test_stdcall+24h (01A1794h)
  return x + int(ptr - text);
    001A17A9 mov      eax,dword ptr [ptr]
    001A17AC sub      eax,dword ptr [text]
    001A17AF add      eax,dword ptr [x]
```

```
}
   001A17B2 pop      edi
   001A17B3 pop      esi
   001A17B4 pop      ebx
   001A17B5 mov      esp,ebp
   001A17B7 pop      ebp
   001A17B8 ret      8

int main( int argc, char* argv[] )
{
   001A18B0 push     ebp
   001A18B1 mov      ebp,esp
   001A18B3 sub      esp,0CCh
   001A18B9 push     ebx
   001A18BA push     esi
   001A18BB push     edi
   001A18BC lea      edi,[ebp-0CCh]
   001A18C2 mov      ecx,33h
   001A18C7 mov      eax,0CCCCCCCCh
   001A18CC rep stos   dword ptr es:[edi]

 // stdcall
 int x = test_stdcall(0, "toto");
   001A18CE push     offset string "toto" (01A7B30h)
   001A18D3 push     0
   001A18D5 call     test_stdcall (01A137Ah)
   001A18DA mov      dword ptr [x],eax
 printf("test_stdcall( 0, \"toto\" ) = %d\n", x);
   001A18DD mov      eax,dword ptr [x]
   001A18E0 push     eax
   001A18E1 push     offset string "test_stdcall( 0, "toto" ) = %d\n" (01A7B38h)
   001A18E6 call     _printf (01A1339h)
   001A18EB add      esp,8

 return 0;
   001A18EE xor      eax,eax
}
   001A18F0 pop      edi
   001A18F1 pop      esi
   001A18F2 pop      ebx
   001A18F3 add      esp,0CCh
   001A18F9 cmp      ebp,esp
   001A18FB call     __RTC_CheckEsp (01A1122h)
   001A1900 mov      esp,ebp
   001A1902 pop      ebp
   001A1903 ret
```

Il n'y a que peu de différences entre l'appel cdecl ou stdcall en termes de code généré. Seul l'endroit où le pointeur de pile est mis à jour.

Le cas des fonctions 'inline'

Les fonctions '***inline***' sont une variante typée des macros ***#define***… Elles ont d'abord été normées dans le C++ avant d'être ajoutées à la norme du C (C90) bien que supportées par certains compilateurs avant la norme.

Lorsque le compilateur rencontre une fonction inline, il stocke cette fonction dans une table style clé/valeur avec pour clé le prototype de la fonction (sa signature) et comme valeur le code à substituer. Cette table a une taille limitée (et la taille dépend du compilateur utilisé) : il m'est déjà arrivé d'avoir des alertes de compilation sous un vieux Visual Studio m'indiquant qu'une fonction ne pouvait pas être « inlinée » faute d'espace… Lorsque le compilateur rencontre un appel à une fonction présente dans la table des fonctions inline, il va remplacer l'appel par le code de la fonction : on économisera le passage des arguments par la pile d'appel et le compilateur pourra appliquer plus de règles d'optimisations sur le code obtenu. Cette forme d'optimisation n'est disponible qu'en compilation release…

Voici comment on déclare une fonction inline :

```
inline int  test_inline(int x, const char *text)
{
  const char *ptr = text;
  while (0 != *ptr)
  {
    ptr++;
  }
  return x + int(ptr - text);
}
```

La convention rapide ou fastcall

Cette convention d'appel a été mise en place sous Windows pour les processeurs x86 et elle est supportée par le compilateur de Visual C++ et gcc. Le principe de la convention d'appel « __fastcall » est que les arguments des fonctions doivent être passés dans les registres, lorsque cela est possible. Cette technique permet d'éviter d'empiler les 2 premiers arguments et donc cela permet souvent des gains de performances pour les fonctions ayant un ou deux arguments. Les deux premiers DWORD ou arguments plus petits qui figurent dans la liste d'arguments de gauche à droite sont transmis dans les registres ; tous les autres arguments sont transmis sur la pile de droite à gauche. La fonction appelée enlève les arguments de la pile. Le mot clé « __fastcall » est accepté et ignoré par les compilateurs qui ciblent ARM et x64 architectures ; sur un processeur x64, par convention, les quatre premiers arguments sont passés dans les registres quand cela est possible, et les arguments supplémentaires sont passés sur la pile.

Voici comment on déclare la fonction :

```
int  __fastcall test_fastcall(int x, int y)
{
  return x + y;
}
```

Voici le code généré par le compilateur :

```
int  __fastcall test_fastcall(int x, int y)
{
   007317A0 push     ebp
   007317A1 mov      ebp,esp
   007317A3 sub      esp,0D8h
   007317A9 push     ebx
   007317AA push     esi
   007317AB push     edi
   007317AC push     ecx
```

```
   007317AD lea      edi,[ebp-0D8h]
   007317B3 mov      ecx,36h
   007317B8 mov      eax,0CCCCCCCCh
   007317BD rep stos   dword ptr es:[edi]
   007317BF pop      ecx
   007317C0 mov      dword ptr [y],edx
   007317C3 mov      dword ptr [x],ecx
 return x + y;
   007317C6 mov      eax,dword ptr [x]
   007317C9 add      eax,dword ptr [y]
}
   007317CC pop      edi
   007317CD pop      esi
   007317CE pop      ebx
   007317CF mov      esp,ebp
   007317D1 pop      ebp
   007317D2 ret


int main( int argc, char* argv[] )
{
   00731900 push     ebp
   00731901 mov      ebp,esp
   00731903 sub      esp,0CCh
   00731909 push     ebx
   0073190A push     esi
   0073190B push     edi
   0073190C lea      edi,[ebp-0CCh]
   00731912 mov      ecx,33h
   00731917 mov      eax,0CCCCCCCCh
   0073191C rep stos   dword ptr es:[edi]
// fastcall
 int x = test_fastcall(5, 3);
   0073191E mov      edx,3
   00731923 mov      ecx,5
   00731928 call     test_fastcall (073113Bh)
   0073192D mov      dword ptr [x],eax
 printf("test_fastcall( 5, 3 ) = %d\n", x);
   00731930 mov      eax,dword ptr [x]
   00731933 push     eax
   00731934 push     offset string "test_fastcall( 5, 3 ) = %d\n"... (0737B30h)
   00731939 call     _printf (0731343h)
   0073193E add      esp,8

 return 0;
   00731941 xor      eax,eax
}
   00731943 pop      edi
   00731944 pop      esi
   00731945 pop      ebx
   00731946 add      esp,0CCh
   0073194C cmp      ebp,esp
   0073194E call     __RTC_CheckEsp (0731122h)
   00731953 mov      esp,ebp
   00731955 pop      ebp
   00731956 ret
```

Lors de l'appel à la fonction on remarque bien que les paramètres sont passés dans les registres :

```
   0073191E mov      edx,3
   00731923 mov      ecx,5
   00731928 call     test_fastcall (073113Bh)
   0073192D mov      dword ptr [x],eax
```

En bleu nous pouvons voir le passage des arguments ; en vert nous avons l'appel de la fonction. En rouge, le retour de la fonction stockée dans le registre est transféré dans l'emplacement mémoire de la variable « x »…

Appel à une fonction membre de classe ou thiscall

Cette convention d'appel s'applique aux fonctions membres non-statiques. Il y a deux variantes du thiscall selon le compilateur et si la fonction a un nombre d'arguments variable :

- pour GCC, par exemple, thiscall est équivalent à cdecl : l'appelant empile les arguments de droite à gauche et ajoute le pointeur this (l'instance de l'objet) comme si c'était le premier argument de la fonction.
- pour Visual C++, le pointeur sur l'instance de l'objet est passé dans le registre ECX/RCX (suivant que l'on est en 32 ou 64 bits). Pour une fonction ayant un nombre fixe d'arguments, on se base sur la convention stdcall. Dans le cas d'un nombre variable d'arguments on se base sur la convention cdecl.

Prenons comme exemple :

```
struct A
{
  int m_a;

  A(int value) : m_a(value)
  {
  }

  int add(int x)
  {
    m_a += x;
    return m_a;
  }
};


int main( int argc, char* argv[] )
{
  // thiscall
  A a(5);
  int x = a.add(3);
  printf("test_thiscall = %d\n", x);

  return 0;
}
```

Le code généré pour la fonction A::add est le suivant sous Visual C++ 2019 :

Code complet sur le site

Et celui de la fonction main :
Code complet sur le site

On remarque que l'appel au constructeur se fait via les instructions suivantes :

```
005919D8 push     5           # on empile le paramètre « value »
005919DA lea      ecx,[a]     # on stocke dans le registre ECX l'adresse de l'instance 'a'
005919DD call     A::A (0591073h) # on appelle la fonction
```

De même pour l'appel à la fonction à a.add(3) :

```
005919E2 push     3           # on empile le paramètre « x » (de valeur 3)
005919E4 lea      ecx,[a]     # on stocke dans le registre ECX l'adresse de l'instance 'a'
005919E7 call     A::add (059123Ah)  # on appelle la fonction
005919EC mov      dword ptr [x],eax  # on stocke dans la variable 'x' le résultat de la fonction
```

Ce qu'il faut retenir de thiscall est que l'on passe systématiquement un argument supplémentaire : le pointeur de l'instance. Que ce soit en l'empilant ou en le passant dans un registre du processeur cela va rajouter des instructions qui auront un impact sur les performances si la fonction est appelée un grand nombre de fois. Si une fonction de l'objet n'a pas besoin d'accéder aux variables membres de celui-ci, il est plus performant de la déclarer comme fonction statique de la classe !

Le cas des fonctions virtuelles

La programmation orientée objet nous a apporté un moyen de spécialiser les objets en diminuant le coût d'écriture grâce aux fonctions virtuelles… Cependant la gestion de ces fonctions virtuelles induit un coût supplémentaire à l'appel tout en respectant la convention d'appel thiscall. Associé à chaque classe contenant des fonctions virtuelles, le compilateur crée une table (la VTABLE) qui va contenir les adresses des fonctions virtuelles (pour le compilateur chaque fonction est un index dans le tableau). Donc, lors d'un appel à une fonction virtuelle d'un objet, il y a :

- accès au type de l'objet (ce sont des données statiques associées à la classe) ;
- récupérer la VTABLE ;
- récupérer l'adresse de la fonction virtuelle ;
- appeler la fonction.

Prenons un exemple :

```
struct VA
{
  int m_x;

  VA( int x ) : m_x( x ) {}

  virtual ~VA() {}

  virtual int v_op(int y)
  {
    m_x += y;
    return m_x;
  }
};

struct VB : public VA
{
  VB() : VA(0) {}

  virtual ~VB() {}

  virtual int v_op(int y)
  {
    m_x -= y;
    return m_x;
  }
};

int main( int argc, char* argv[] )
{
  // thiscall
  VA *pVA = new VB();
  int x = pVA->v_op(3);
  printf("pVA->v_op(3) = %d\n", x);

  return 0;
}
```

Voici le code généré par pVA->v_op(3) :

```
  int x = pVA->v_op(3);
00851E38 push     3
00851E3A mov      eax,dword ptr [pVA]
00851E3D mov      edx,dword ptr [eax]
00851E3F mov      ecx,dword ptr [pVA]
00851E42 mov      eax,dword ptr [edx+4]
00851E45 call     eax
00851E4E mov      dword ptr [x],eax
```

En bleu, on prépare l'appel de la fonction en empilant le paramètre. En vert, on recherche l'adresse de la fonction virtuelle. En rouge on déclenche l'appel et en brun on récupère le résultat.

L'accès à une fonction virtuelle coûte donc 4 actions de MOV. C'est une raison pour laquelle certains experts de l'optimisation transforment l'héritage grâce à des templates et de la métaprogrammation mais ce sera l'occasion d'un autre article.

Conclusion

J'espère avoir réussi à démystifier les principales conventions d'appels de fonctions et ce que génèrent les compilateurs.
Donc :

- les fonctions inline permettent d'optimiser la génération du code dans le cas de petites fonctions mais cela peut augmenter la taille du binaire ;
- les fonctions fastcall permettent d'accélérer les appels de fonctions ayant un ou deux arguments ;
- stdcall est à préférer dès lors que le code doit être accessible depuis un autre langage ;
- cdecl est le mode par défaut du C… ;
- les fonctions virtuelles sont certes pratiques mais abaissent les performances à l'exécution.

Duy Anh PHAM
Ingénieur logiciel Java/JEE chez Clevermind, je développe des applications Web d'entreprises depuis plus de 10 ans et je me passionne à tout ce qui permet d'obtenir une application maintenable et performante du code source jusqu'au déploiement en particulier la concurrence/programmation concurrente.

L'essentiel des threads pour comprendre les phénomènes de concurrence

Partie 2

Avec la généralisation des processeurs multicœurs dans les appareils tels que les ordinateurs, smartphones, tablettes et autres dispositifs avec système d'exploitation, développer une application multithreadée devient un enjeu majeur pour répondre aux besoins toujours croissants et nombreux des utilisateurs. Suite de notre dossier sur les threads.

Emploi des classes dites thread-safe

Ecrire du code thread-safe peut sembler plus simple lorsque la plateforme Java met à disposition des collections synchronisées/ concurrentes sans besoin de synchroniser. Cependant il peut s'avérer nécessaire d'y avoir recours.

```
public class NonAtomicOperation {

    private List<String> names = Collections.synchronizedList(new LinkedList<String>());

    private void add(String name) {
        names.add(name);
    }

    private String removeFirst() {
        if (!names.isEmpty()) {
            return names.remove(0);
        } else {
            return null;
        }
    }

    public static void main(String[] args) {

        final NonAtomicOperation n1 = new NonAtomicOperation();
        n1.add("Foobar");

        Runnable r = () -> {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                // nothing to do
            }
            String name = n1.removeFirst();
            System.out.println(name);
        };

        Thread t1 = new Thread(r, "Thread-1");
        Thread t2 = new Thread(r, "Thread-2");
        t1.start();
        t2.start();
}
```

Résultat possible d'exécution du programme

```
Exception in thread "Thread-1" Foobar
java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
```

Les 2 threads accèdent à une liste chaînée dite thread-safe. Et pourtant une exception est levée. Nombre de développeurs seraient surpris par l'erreur en pensant que remplacer une collection par une collection synchronisée voire une collection concurrente de Java 5 empêcherait l'erreur de survenir.
Comme dans l'exemple d'Alice et Bob, la bonne approche pour résoudre est de toujours se poser la question suivante : quelles sont les opérations qui ont besoin d'être vues comme une opération atomique ? Dans notre exemple c'est **names.isEmpty()** et **names.remove(0)** qui doivent être indivisibles. Il faut donc synchroniser toute la méthode **removeFirst()**. Il est pertinent dans ce cas de commenter l'emploi du mot synchronized pour faciliter la compréhension du code par à un autre développeur.

Interaction entre threads

Les threads exécutent idéalement des tâches indépendantes mais il se peut par exemple qu'un thread attende la fin d'un ou plusieurs autres threads pour continuer. C'est là que la communication inter threads entre en jeu.

Les méthodes wait/notify de Object

Les threads au sein d'un même processus ont la possibilité de communiquer entre eux au travers des méthodes de la class Object présentées dans le tableau ci-dessous sur un verrou.

Nom de méthode	Description
wait	Met le thread en cours dans une liste d'attente jusqu'à ce qu'un autre thread invoque notify() ou notifyAll() sur le même objet
wait(timeout)	Met le thread en cours dans une liste d'attente jusqu'à ce qu'un autre thread invoque notify() ou notifyAll() sur le même objet, ou que le délai est expiré
notify	Signale un des threads qui attend le verrou
notifyAll	Signale tous les threads qui sont en attente du verrou

Ces méthodes doivent toujours être invoquées dans un contexte synchronisé.
Exemple de communication entre threads avec un modèle producteur consommateur :

- Le producteur produit continuellement jusqu'à ce que la file soit pleine,
- Le consommateur attend au moins la présence d'un élément dans la file pour lire.

```
public class ProducerConsumerThreadSafe {

    private static final int MAX_ELEMENTS = 20;

    private static final int BUFFER_SIZE = 5;

    private Queue<Integer> queue = new LinkedList<>();
```

```
    private final Object lock = new Object();

    public static void main(String[] args) {

        ProducerConsumerThreadSafe main = new ProducerConsumerThreadSafe();
        main.runDemo();
    }

    private void runDemo() {

        Runnable producer = () -> {

            for (int i = 0; i < MAX_ELEMENTS; i++) {

                synchronized (lock) {
                    while (queue.size() == BUFFER_SIZE) {
    System.out.println(Thread.currentThread().getName() + " : Queue is full. Waiting for produce ...");
                        try {
                            lock.wait();
                        } catch (InterruptedException e) {
                        }
                    }
                }

                synchronized (lock) {
                    System.out.println(Thread.currentThread().getName() + " put : " + i);
                    queue.offer(i);
                    lock.notify();
                }

            }
        };

        Runnable consumer = () -> {
            int counter = 0;

            while (counter < MAX_ELEMENTS) {

                synchronized (lock) {
                    while (queue.isEmpty()) {
    System.out.println(Thread.currentThread().getName() + " : Queue is empty. Waiting for data
to consume ...");
                        try {
                            lock.wait();
                        } catch (InterruptedException e) {
                        }
                    }
                }

                synchronized (lock) {
    System.out.println(Thread.currentThread().getName() + " get : " + queue.poll());
                    lock.notify();
                    counter++;
                }

            }
        };

        Thread threadProducer = new Thread(producer, "Producer");
        Thread threadConsumer = new Thread(consumer, "Consumer");

        threadProducer.start();
        threadConsumer.start();
    }

}
```

Un résultat d'exécution possible (une exécution répétée peut faire varier le résultat).

```
Producer put : 0
Producer put : 1
Producer put : 2
Consumer get : 0
Consumer get : 1
Consumer get : 2
Consumer : Queue is empty. Waiting for data to consume ...
Producer put : 3
Producer put : 4
Producer put : 5
Producer put : 6
Producer put : 7
Producer : Queue is full. Waiting for produce ...
Consumer get : 3
Consumer get : 4
Consumer get : 5
Consumer get : 6
Consumer get : 7
Consumer : Queue is empty. Waiting for data to consume ...
Producer put : 8
Producer put : 9
Producer put : 10
Producer put : 11
Producer put : 12
Producer : Queue is full. Waiting for produce ...
Consumer get : 8
Consumer get : 9
Consumer get : 10
Consumer get : 11
Consumer get : 12
Consumer : Queue is empty. Waiting for data to consume ...
Producer put : 13
Producer put : 14
Producer put : 15
Producer put : 16
Producer put : 17
Producer : Queue is full. Waiting for produce ...
Consumer get : 13
Consumer get : 14
Consumer get : 15
Consumer get : 16
Consumer get : 17
Consumer : Queue is empty. Waiting for data to consume ...
Producer put : 18
Producer put : 19
```

```
Consumer get : 18
Consumer get : 19
```

La synchronisation est effective puisque le producteur ne peut produire lorsque le buffer est plein et que le consommateur ne peut pas lire avec un buffer vide. Le système de wait()/notify() est utilisé dans les versions jusqu'à Java 1.4.

Les outils de synchronisation depuis Java 5

Depuis Java 5 avec l'API Concurrency, l'interface Lock peut remplacer l'utilisation du mot clé synchronized. Elle fournit également des extensions utiles sur les opérations de verrou. De manière générale, la synchronisation sur un bloc de code est remplacée par un bloc try/finally avec un objet de type ReentrantLock implémentant l'interface Lock.
Prise de verrou avec libération dans un bloc finally. Cette écriture est l'équivalent de l'utilisation du mot clé synchronized.

```
private final Lock lock = new ReentrantLock();

public void doAction() {

    // do something by multiple threads

    try {
        lock.lock();
        // critical session executed by one thread only
    } finally {
        lock.unlock();
    }

    // continue do something by multiple threads
}
```

Tentative de prise de verrou avec retour immédiat en fonction du résultat. Cela permet d'obtenir une application plus réactive.
Code complet sur le site

Tentative de prise de verrou jusqu'à 2 secondes maximum avec retour en fonction du résultat. Avec le mot clé synchronized, le thread peut attendre indéfiniment avant que le verrou soit libéré. Cela permet d'obtenir une application plus réactive tout en cherchant à obtenir le verrou.

```
private final Lock lock = new ReentrantLock();

public void doAction() {

    // do something by multiple threads

    if (lock.tryLock(2, TimeUnit.SECONDS)) {
        try {
            // critical session executed by one thread only
        } finally {
            lock.unlock();
        }
    } else {
        // do something else by multiple threads
    }
    // continue do something by multiple threads
}
```

Ci-dessous la version de la classe ProducerConsumerThreadSafe avec la nouvelle API :

```
public class ProducerConsumerThreadSafeJava5 {

    private static final int MAX_ELEMENTS = 20;

    private static final int BUFFER_SIZE = 5;

    private Queue<Integer> queue = new LinkedList<>();

    private final Lock lock = new ReentrantLock();

    public static void main(String[] args) {

        ProducerConsumerThreadSafeJava5 main = new ProducerConsumerThreadSafeJava5();
        main.runDemo();
    }

    private void runDemo() {

        final Condition conditionBufferEmpty = lock.newCondition();
        final Condition conditionBufferFull = lock.newCondition();

        Runnable producer = () -> {

            Thread.currentThread().setName("Producer");

            for (int i = 0; i < MAX_ELEMENTS; i++) {

                try {
                    lock.lock();
                    while (queue.size() == BUFFER_SIZE) {
System.out.println(Thread.currentThread().getName() + " : Queue is full. Waiting for produce ...");
                        try {
                            conditionBufferFull.await();
                        } catch (InterruptedException e) {
                        }
                    }
                } finally {
                    lock.unlock();
                }

                try {
                    lock.lock();
System.out.println(Thread.currentThread().getName() + " put : " + i);
                    queue.offer(i);
                    conditionBufferEmpty.signal();
                } finally {
                    lock.unlock();
                }

            }
        };

        Runnable consumer = () -> {
            int counter = 0;
            Thread.currentThread().setName("Consumer");
```

```
        while (counter < MAX_ELEMENTS) {

            try {
                lock.lock();
                while (queue.isEmpty()) {
    System.out.println(Thread.currentThread().getName() + " : Queue is empty. Waiting for data
to consume ...");
                    try {
                        conditionBufferEmpty.await();
                    } catch (InterruptedException e) {
                    }
                }
            } finally {
                lock.unlock();
            }

            try {
                lock.lock();
    System.out.println(Thread.currentThread().getName() + " get : " + queue.poll());
                conditionBufferFull.signal();
                counter++;
            } finally {
                lock.unlock();
            }

        }
    };

    ExecutorService executor = Executors.newCachedThreadPool();
    executor.execute(producer);
    executor.execute(consumer);

    executor.shutdown();
  }

}
```

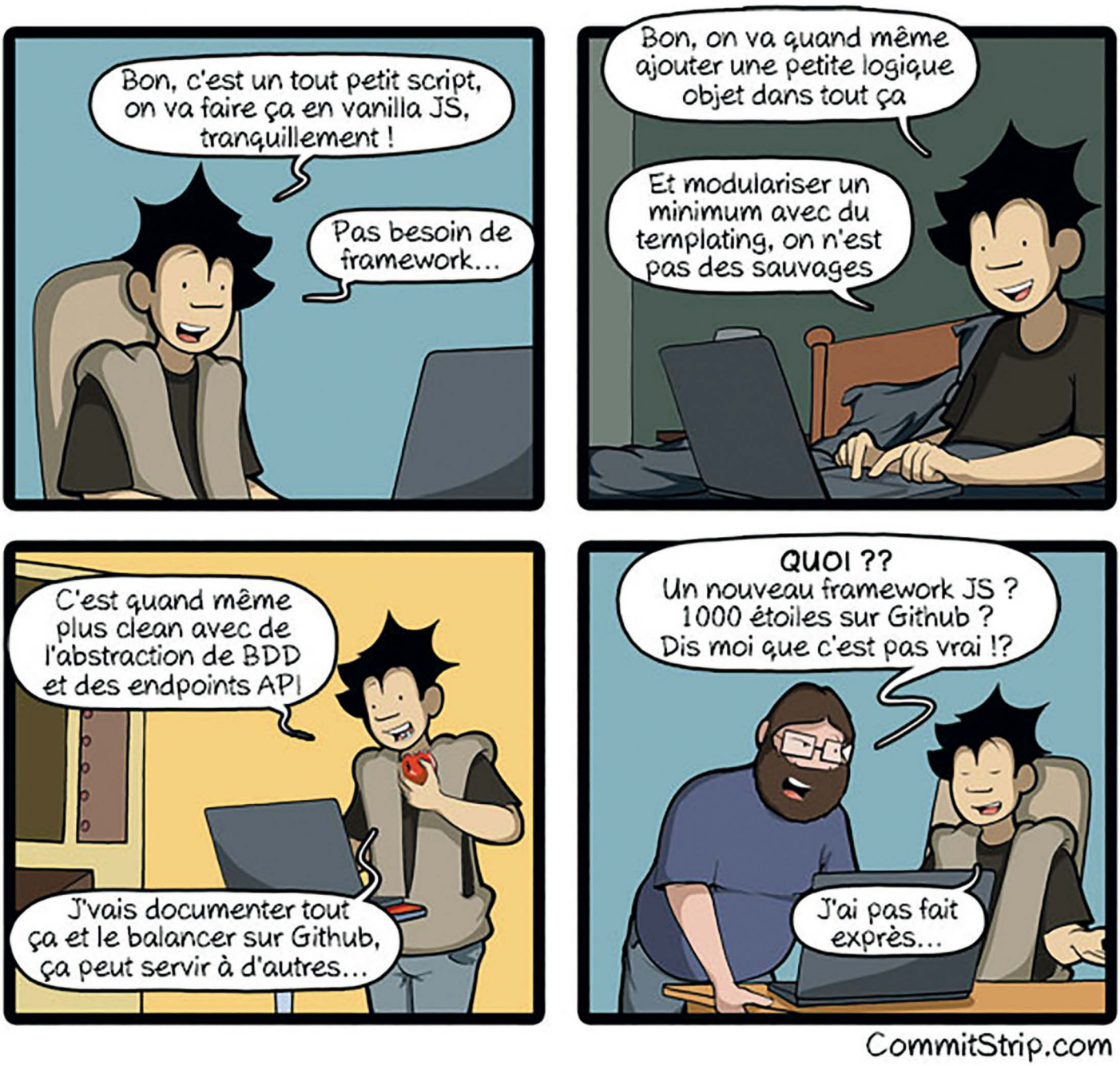
Un objet de type Condition remplace l'utilisation des méthodes wait/notify de la classe Object pour suspendre un thread et être réveillé par un autre. Une Condition doit donc être protégée par un verrou de type Lock. L'instance d'une Condition est intimement liée au verrou en invoquant la méthode **newCondition()**. Dans le cas du producteur-consommateur, il y a 2 conditions pour lesquelles les 2 threads s'attendent mutuellement : le buffer plein et vide.
Pour aller encore plus loin, Java 5 fournit des collections concurrentes bloquantes qui répondent au problème de producteur consommateur de manière élégante notamment avec l'API **BlockingQueue**.

```
public class ProducerConsumerUsingBlockingQueue {

  private static final int MAX_ELEMENTS = 20;

  private static final int BUFFER_SIZE = 5;

  private BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(BUFFER_SIZE);

  public static void main(String[] args) {
    ProducerConsumerUsingBlockingQueue main = new ProducerConsumerUsingBlocking
Queue();
    main.runDemo();
  }

  private void runDemo() {

    Runnable producer = () -> {

      Thread.currentThread().setName("Producer");

      for (int i = 0; i < MAX_ELEMENTS; i++) {

        try {
  System.out.println(Thread.currentThread().getName() + " put : " + i);
          queue.put(i);
        } catch (InterruptedException e) {
          // nothing to do
        }
      }

    };

    Runnable consumer = () -> {
      int counter = 0;
      Thread.currentThread().setName("Consumer");

      while (counter < MAX_ELEMENTS) {

        try {
  System.out.println(Thread.currentThread().getName() + " get : " + queue.take());
          counter++;
        } catch (InterruptedException e) {
          // nothing to do
        }
      }
    };

    ExecutorService executor = Executors.newCachedThreadPool();
    executor.execute(producer);
    executor.execute(consumer);

    executor.shutdown();
  }
}
```

La méthode **put()** est bloquante lorsque la file est pleine tandis que **take()** est bloquante lorsqu'elle est vide. Ainsi, les blocs de synchronisations et les boucles d'attente vérifiant qu'un thread est bien réveillé sont totalement éliminés. Le code est considérablement simplifié et lisible.

Suite et fin dans le numéro 239

Un framework malgré moi



Une publication Nefer-IT, 57 rue de Gisors, 95300 Pontoise - redaction@programmez.com
Tél. : 09 86 73 61 08 - Directeur de la publication : François Tonic
Rédacteurs en chef : François Tonic
Secrétaire de rédaction : Olivier Pavie
Ont collaboré à ce numéro : ZDNet

Nos experts techniques : M. Guillot, H. Blin Dray, M. Pichaud, O. Bouzereau, P. Charrière, G. Guillou, J. Lépine, T. Bollet, D. Danse, V. Foyang Keuko, M. Pashkina, D. Ortega, S. Michalak, T. Leriche, O. Lourme, A. Giretti, L. Avrot, A. Vache, P. Boulanger, D. Anh Pham, Commitstrip
Couverture :gopher : © D.R. - ville : JIMMY UTHE - **Maquette** : Pierre Sandré.
Publicité : François Tonic / Nefer-IT - Tél. : 09 86 73 61 08 - ftonic@programmez.com.
Imprimeur : SIB Imprimerie
Marketing et promotion des ventes : Agence BOCONSEIL - Analyse Media Etude - Directeur : Otto BORSCHA oborscha@boconseilame.fr
Responsable titre : Terry MATTARD Téléphone : 09 67 32 09 34

Contacts : Rédacteur en chef : ftonic@programmez.com - Rédaction : redaction@programmez.com - Webmaster : webmaster@programmez.com
Evenements / agenda : redaction@programmez.com

Dépôt légal : à parution - Commission paritaire : 1220K78366 - ISSN : 1627-0908 - © NEFER-IT / Programmez, février 2020

Abonnement :
Service Abonnements PROGRAMMEZ, 4 Rue de Mouchy, 60438 Noailles Cedex. - Tél. : 01 55 56 70 55 - *abonnements.programmez @groupe-gli.com*
Fax : 01 55 56 70 91 - du lundi au jeudi de 9h30 à 12h30 et de 13h30 à 17h00, le vendredi de 9h00 à 12h00 et de 14h00 à 16h30.

Tarifs
Abonnement (magazine seul) : 1 an - 11 numéros France métropolitaine : 49 € - Etudiant : 39 € CEE et Suisse : 55,82 € - Algérie, Maroc, Tunisie : 59,89 € Canada : 68,36 € - Tom : 83,65 € - Dom : 66,82 € - Autres pays : nous consulter.

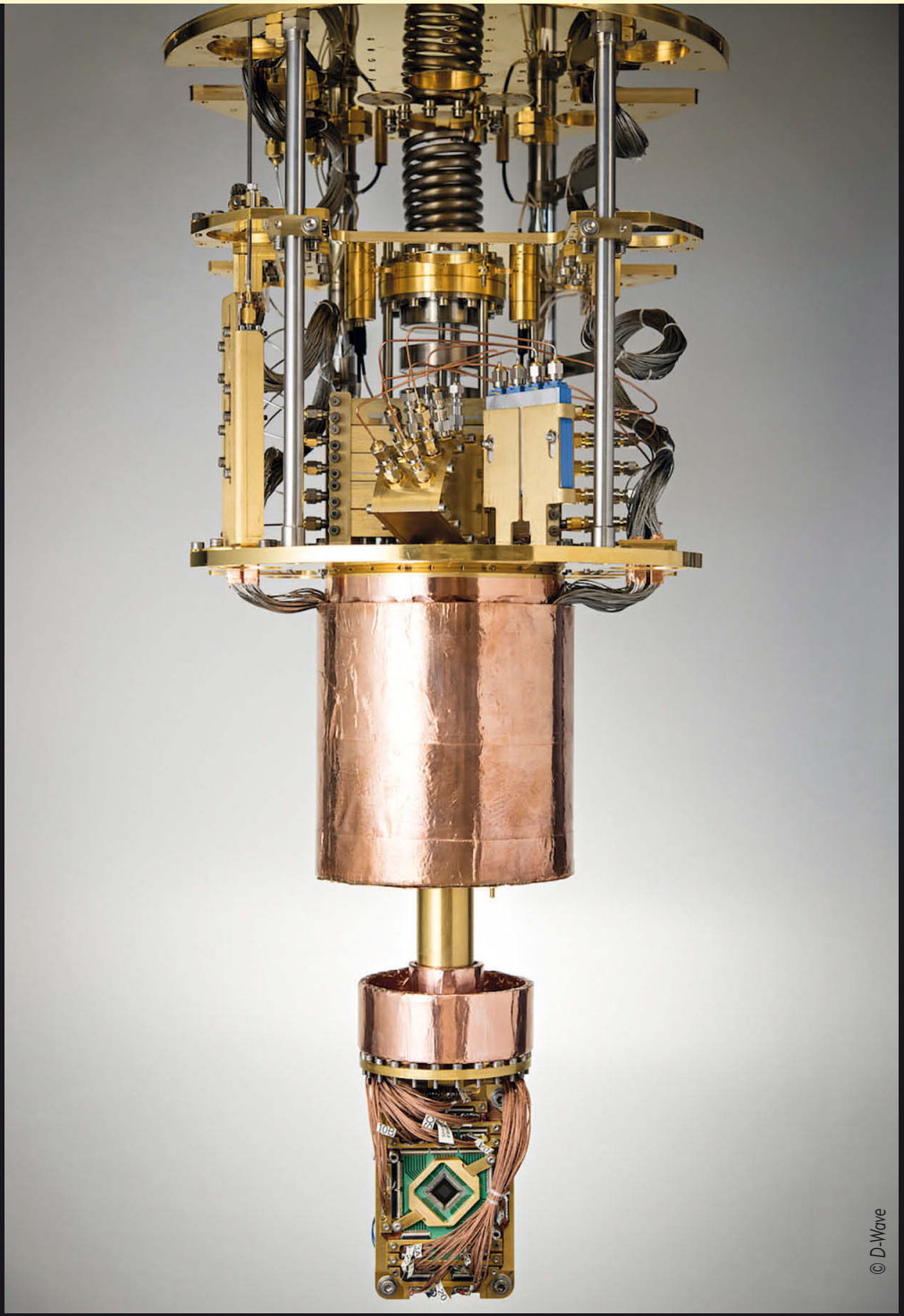
PDF
35 € (monde entier) souscription sur www.programmez.com

DevCon#9

INFORMATIQUE QUANTIQUE

1 journée
+10 sessions & ateliers

28 mai à partir de 9h30



Informations & inscriptions : www.programmez.com

G·ECHO
cybersécurité

Solutions pour la cybersécurité

MICRO FOCUS
Gold Partner
Security

Sécurité des développements : Fortify
Protection des données : Voltage
Evènements de sécurité : ArcSight

beyond SECURITY

Trouvez vos 0-days : bestorm
Scan de vulnérabilités : besecure

Conseil, services, développement

- Audits de code, test d'intrusion,
- Conseil, analyse de risque, PSSI,
- R&D, développement de solutions. …

Formations et recrutement en SSI

www.g-echo.fr
contact@g-echo.fr
Toulouse - Paris