PROGRAMMEZ!

Le magazine des développeurs

11/12 2020 N°243 23e année

Smalltalk C'est l'avenir du code !

Le { c'est la vie

Low Code / No Code : ce n'est pas la mort du développeur...

NotePad & Emacs pour tou s et tous !

Comprendre et coder des apps PWA

TypeScript 4.0

La librairie standard Python

IoT : c'est quoi exactement ?

® RichVintage

M 04319 - 243 - 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

EDITO

Quel langage est-il le plus efficace au niveau énergétique ?

Préserver la planète est une évidence. On pointe souvent du doigt (ah le fameux doigt inquisiteur) : l'informatique, les technologies… et le développeur ! Eh oui, ami(e) développeuse/eur, toi aussi tu es responsable du gâchis énergétique. Ben quoi ? Tu te croyais à l'abri ?

Développer du code exige de l'énergie. Installer une app depuis un App Store exige encore de l'énergie. Envoyer des mails, regarder YouTube, utiliser des services cloud, tout utilise de l'énergie… Finalement, toute action consomme de l'énergie. Même s'il est impossible de réduire à 0 la consommation ou l'émission de pollution, on peut optimiser les usages.

Un article universitaire, Energy Efficiency across Programming Languages, tente de savoir : quel est donc le langage magique qui consomme le moins d'énergie, tout en étant efficace en exécution, en consommation de mémoire et en temps de compilation ? Le langage C est considéré comme le plus efficace. Rust, Pascal, C++, avec Go, ne sont pas loin. Java ne s'en sort pas si mal, sauf sur les besoins mémoires. Les langages compilés sont logiquement en tête. Les langages interprétés sont souvent à la traîne, notamment sur les besoins en mémoire et sur les temps d'exécution.

Cependant, gardons à l'esprit que la qualité du code, sa structuration, le respect des bonnes pratiques sont des éléments cruciaux. Pour être efficace (minimisation du processeur, de l'énergie, des réseaux, etc.), l'app doit être très propre. Le langage n'est qu'un élément parmi les multiples couches technologiques. Mais le développeur, en codant le plus proprement possible, est un des maillons de l'écoresponsabilité dans les technologies.

Nous aurons l'occasion de revenir dessus dans les prochains numéros.

Dans ce numéro, nous vous proposons beaucoup de thèmes : retour sur Smalltalk et sa nouvelle implémentation, Pharo, la suite du dossier Tanzu et la fin sur les algos génétiques. On parlera Python, PWA, Kotlin et Typescript !

On reviendra sur un sujet qui cristallise parfois les tensions dans les équipes : les outils low code et no code. Est-ce la mort des développeurs ? Une démocratisation de la technologie auprès du tout public ? Une volonté de redonner à tout le monde le contrôle de la technologie ? Et donc en finir avec cette vision qu'une minorité sait parler aux technologies… Ces outils ne sont pas nouveaux, mais le confinement du printemps et la crise du Covid ont contribué à booster ces outils. Soyons clairs : non ce n'est pas la mort du développeur. Le développeur restera au cœur des applications et de la technologie. Mais les outils low code/no code peuvent répondre à des besoins immédiats et aider des personnes souhaitant produire un site, une app mobile à y parvenir rapidement.

De nombreux projets métiers n'ont pas besoin de sortir tout l'arsenal technologique (IDE, debug, compilateur, Java ou C#, x semaines de développement). Ces outils peuvent démocratiser la technologie avec une approche moins hermétique et sans trop parler de codes. Rendez-vous en page 41 !

Bon code !

François Tonic - ftonic@programmez.com
0 errors
0 warnings

SOMMAIRE #243

Prochain numéro :

PROGRAMMEZ! 244

Disponible le 4 janvier 2021

RUST de A à Z
Utiliser Tensorflow
Cloud Computing & les développeurs

Ubisoft et Crytek dans la panade

Le groupe Egregor affirme avoir piraté les éditeurs Crytek et Ubisoft, ce qui leur aurait permis de chiffrer les machines de Crytek et de voler des données sensibles chez Ubisoft et Crytek. Le groupe est spécialisé dans les attaques de type ransomware et le vol de données. Le groupe cybercriminel menace notamment Ubisoft de diffuser le code source du jeu Watch Dogs Legion, dont la sortie est imminente. Pour l'instant, ni l'un ni l'autre n'ont accepté de négocier avec les attaquants, ni d'ailleurs de communiquer au sujet de l'attaque.

Ciao, Djingo

Djingo, l'enceinte connectée d'Orange, tire sa révérence. Commercialisé depuis quelques mois et faisant les frais d'un lancement chaotique, le projet sera donc mis au rebut. L'opérateur a décidé de se concentrer sur ses priorités plutôt que d'aller chercher Amazon ou Google sur les assistants. Et au vu des chiffres de vente apparemment décevants de Djingo, on comprend qu'Orange préfère arrêter les frais plutôt que d'investir sur un secteur largement dominé par la concurrence américaine.

iPhone 12 : virage 5G

Réglé comme une horloge, Apple a profité du mois d'octobre pour présenter sa gamme d'iPhone 12. Au total, 4 modèles seront proposés sur cette appellation, et les prix s'échelonneront entre 809 euros pour l'iPhone 12 mini avec 64 Go de stockage et 1609 euros pour l'iPhone 12 Pro Max avec 512 Go de stockage. Cette nouvelle gamme emporte comme chaque fois son lot de nouveautés matérielles et logicielles, ainsi que le support des réseaux 5G qui commenceront à être massivement déployés dans le courant de l'année prochaine. Les modèles iPhone 12 et iPhone 12 pro sont disponibles, et le 13 novembre pour les versions iPhone 12 pro Max et iPhone 12 mini. Les délais se sont rapidement allongés dès le premier jour de précommande… À noter qu'Apple a présenté un nouveau format d'image : ProRaw.
NDLR : la version mini est très sympa.

Health Data hub : le feuilleton de la rentrée continue

Health Data Hub est une plateforme gouvernementale lancée fin 2019. Elle vise à centraliser sur un même dispositif l'ensemble des données de santé générées par les établissements français afin de faciliter le travail des chercheurs. Pour ne pas perdre de temps avec le lancement du projet, le gouvernement s'est tourné vers le cloud Azure de Microsoft pour l'héberger. Quelques voix soucieuses des questions de souveraineté avaient trouvé à y redire, mais c'est finalement l'invalidation par la justice européenne du Privacy Shield qui a fini par faire bouger les lignes. La CNIL a demandé un rapatriement des données sur le territoire européen tandis que le Conseil d'État évoque de son côté « un risque théorique » qui plaiderait en ce sens. Le gouvernement a donc annoncé qu'un appel d'offres serait relancé afin d'ouvrir la compétition aux fournisseurs français et européens. Mieux vaut tard que jamais.

2,8 milliards d'euros pour les fréquences 5G

Les enchères pour les fréquences 5G se sont achevées et les quatre opérateurs français ont déboursé au total la coquette somme de 2,8 milliards d'euros pour s'offrir les bandes de fréquences réservées à la nouvelle évolution du protocole. Orange s'est arrogé la part du lion en se réservant quatre blocs de fréquences, suivi de trois blocs pour SFR et deux pour Free et Bouygues. Et le gouvernement, qui espérait obtenir 2,17 milliards d'euros sur cette vente aux enchères, se réjouit de la bonne affaire.

GPT3 cède sa licence à Microsoft

GPT3, c'est le moteur de génération de texte par machine learning si puissante que ses auteurs avaient dans un premier temps indiqué qu'ils préféraient le garder pour eux afin d'éviter qu'il ne tombe entre de mauvaises mains. Mais le groupe OpenAI, qui coordonne le développement de GPT3, a finalement adouci sa position et a annoncé avoir accordé à Microsoft la licence exclusive, qui pourra donc être intégrée au sein des produits Microsoft. Il faut dire que la firme de Redmond avait eu le bon goût de mettre sur la table un chèque d'un milliard d'euros pour financer le projet et avait annoncé la construction d'un datacenter dédié à l'entraînement des modèles d'apprentissage automatique de GPT3. Un engagement qui a fini par payer.

Dernière minute

SpaceX va-t-il devenir un transporteur d'armes dans l'espace ?

Les Etats-Unis cherchent-ils un transport rapide pour pouvoir envoyer dans l'espace des armes et munitions si nécessaires ? SpaceX ne serait pas opposé à devenir un tel transporteur si besoin… Mais le sujet est sensible et pour le moment le constructeur reste très discret. SpaceX travaille déjà pour l'armée américaine.

SALTO : les services de streaming vont trembler… ou pas

Depuis le 20 octobre, SALTO, le service de streaming de TF1 / M6 / France Télévision, est disponible. Vous pouvez voir des séries quotidiennes en avance, voir le replay des chaînes et des chaines en direct… + un catalogue de séries et de films français et des exclusivités et du contenu international. Mais le service est avant tout franco-français. SALTO veut trouver sa place en France mais la concurrence est déjà très vive et les géants comme Netflix, Disney+ et Amazon Prime Video sont déjà bien installés, avec un contenu largement supérieur.
SALTO nécessite un abonnement : 6,99 € pour un seul écran. Disney+ est beaucoup plus intéressant. Les tarifs du service s'alignent plus sur Netflix. SALTO n'est pas disponible sur les box mais uniquement sur les TV connectées et les apps mobiles ainsi que le site web. C'est une erreur stratégique de ne pas être présent dès le démarrage sur les box et les boîtiers de type FireStick.
Le service veut miser sur les propositions, les playlists, etc. Mais quid du budget ? Quid des investissements ? Netflix c'est presque 20 milliards d'investissement dans la production et le catalogue, avec une infrastructure mondiale. Aujourd'hui, ces services, pour vivre, ont une dimension mondiale, avec de nombreuses productions. Amazon réalise en ce moment le Seigneur des Anneaux, Apple mise beaucoup sur le cycle Fondation d'Asimov. SALTO ne joue dans la même catégorie quoi qu'on puisse en dire ! F.T.

MEETUPS PROGRAMMEZ!

10 novembre : parlons serverless. Avec Aymeric Weinbach

22 décembre : C++20, avec Christophe Pichaud

19 janvier 2021 : être développeur en 2021, en partenariat avec Arolla

23 février : Java, en partenariat avec Arolla

DEVCON#9 : SPÉCIALE .NET 5 + AZURE + WINDOWS

+10 sessions & ateliers

19 novembre, à partir de 13h30

TOUS NOS ÉVÉNEMENTS SONT, POUR LE MOMENT, VIRTUELS.

Informations & inscription : programmez.com

NOVEMBRE

1	2	3	4	5	6	7
8	9	**10** Meetup Programmez!/	11	12	13	14
15	16	**17** DevDay (Belgique) / en ligne	**18** Sophi.A	**19** Sophi.A DevCon spéciale .Net 5 / en ligne Cloudnord 2020 / en ligne	**20** Sophi.A	21
22	**23** Voice Tech Paris	**24** Voice Tech Paris	25	26	27	28
29	30					

DÉCEMBRE

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22 Meetup Programmez!	23	24	25	26	27	28
29	30	31				

JANVIER 2021

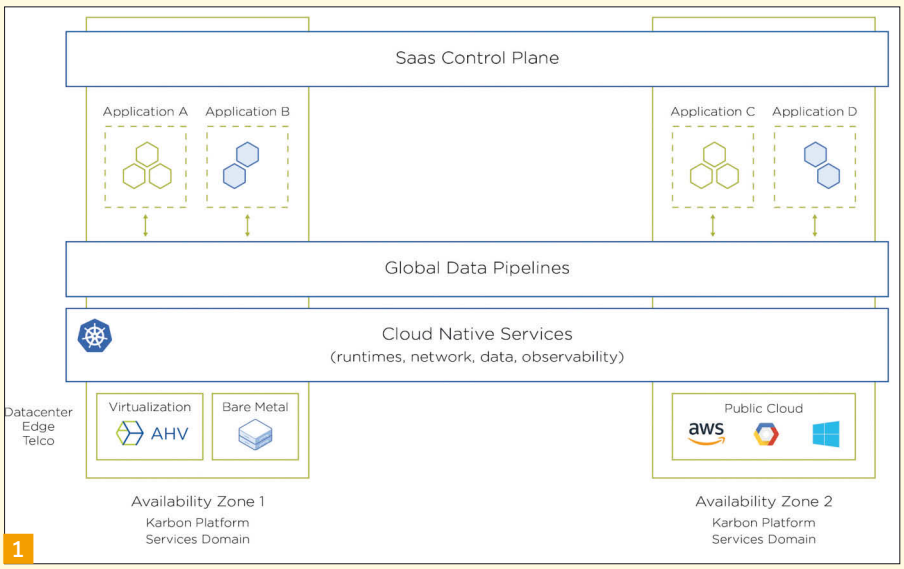
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	**19** Meetup Programmez!	20	21
22	23	24	25	26	27	28
29	30	31				

Merci à Aurélie Vache pour la liste 2020/2021, consultable sur son GitHub :
https://github.com/scraly/developers-conferences-agenda/blob/master/README.md

Nutanix : de l'infrastructure invisible à Kubernetes invisible

Depuis quelques mois, et notamment lors de son évènement .Next Virtual qui s'est tenu en septembre, Nutanix n'a cessé de multiplier les services cloud et PaaS venant s'ajouter à sa solution d'hyperconvergence. À travers ces services qui englobent aussi bien la gestion des Kubernetes, des bases de données, du cycle de vie des applications cloud native et legacy, la microsegmentation, ou encore l'IoT, la firme de San José, souhaite proposer aux entreprises et aux développeurs une expérience qui se rapproche toujours plus de ce que proposent les géants du cloud, aussi bien sur site que via un mode hybride ou multicloud.

Tout le monde connaît aujourd'hui Nutanix pour ses solutions d'infrastructure hyperconvergée. Pourtant, lors de votre dernière conférence, vous avez beaucoup parlé de services orientés PaaS, de Kubernetes, de support d'AWS, Google Cloud Platform, d'Azure, etc. Finalement, quelle est aujourd'hui votre offre ? Devenir fournisseur de services cloud ?



1

Karbon Platform Services regroupe une large variété de services PaaS pour les déploiements Kubernetes, sur site, dans le cloud et sur le Edge.

Christophe Bardy : Clairement, avec l'hyperconvergence nous avons réussi à créer un socle d'architecture offrant des attributs similaires à ceux du cloud public tout en offrant l'ensemble des services de résilience, de protection de données et de disponibilité requis par les applications d'entreprises. Notre plateforme AOS est distribuée, largement automatisée et pilotable par le biais d'une console unique.

Nous avons progressivement enrichi notre offre en posant au-dessus de ce socle de nouveaux services, permettant à nos clients de bâtir un cloud privé offrant tous les services du cloud public. Notre plateforme s'est ainsi enrichie de services d'automatisation applicative, de services de stockage sophistiqués (stockage objet et stockage NAS), d'une offre de Database as a Service unique en son genre, d'une plateforme Kubernetes managée et plus récemment d'une solution PaaS multicloud et d'une offre de Disaster Recovery as a Service. Notre ambition n'est pas d'être un fournisseur de cloud public, mais de proposer aux clients une plateforme qui s'opère comme telle, quelles que soient leurs applications. Mieux, pour répondre aux besoins de nos clients de déployer des services de cloud hybride, nous cherchons à gommer les frontières entre cloud privé et cloud public. C'est ce que nous avons annoncé lors de notre récente conférence .Next en dévoilant Nutanix Clusters, une offre qui permet de déployer des infrastructures Nutanix dans les clouds publics AWS et Azure.

Justement, Nutanix évoque de plus en plus la possibilité de déplacer les applications legacy vers le cloud. Qu'est-ce que vous entendez par applications legacy et qu'est-ce que cela implique du point de vue de l'infrastructure ?

C.B : On qualifie généralement d'applications legacy des applications dont la disponibilité est dépendante de l'infrastructure sous-jacente, par opposition aux applications modernes ou cloud natives qui sont en général conteneurisées, distribuées et qui organisent elles-mêmes leur protection. Concrètement, si un composant d'une application legacy vient à défaillir, l'application tombe alors que dans une application moderne la défaillance d'un composant n'affecte pas l'application.

Pour assurer la disponibilité des applications legacy, les entreprises ont massivement investi dans des mécanismes de redondance au niveau infrastructure (réplication synchrone, métro cluster, stretched cluster). Le problème est que ces mécanismes n'existent pas dans le cloud public, ce qui rend très compliqué le lift and shift des applications legacy.

Les entreprises doivent en effet recréer dans le cloud public les mécanismes de protection qu'elles ont mis en place en interne, ce qui suppose la mise en œuvre de composants logiciels additionnels, souvent différents de ceux utilisés en interne. Cela entraîne inévitablement une augmentation des coûts et de la complexité, ce qui est à l'opposé de ce que les organisations recherchent quand elles vont dans le cloud. En offrant notre plateforme sur les principaux clouds du marché, nous voulons gommer les différences entre privé et public, tout en permettant une mobilité transparente des applications entre les deux mondes : des applications legacy nées dans le cloud privé peuvent migrer de façon transparente vers le cloud public et à l'inverse des applications cloud natives peuvent être rapatriées de façon transparente en interne.

Quel est l'intérêt d'avoir une couche « standard » sur les principaux cloud publics ? Aujourd'hui, nous constatons que les développeurs et les entreprises déploient souvent leurs applications cloud natives sur une seule plateforme ?

C.B : La réalité est plus complexe. Aujourd'hui, la plupart des clients n'ont pas de socle d'infrastructure ou de PaaS unique et ont aussi fait des choix multiples de cloud. Ils déploient certaines de leurs applications dans Azure, d'autre sur GCP ou AWS. L'intérêt d'offrir des briques standard est de simplifier le déploiement des applications. Les développeurs et les entreprises peuvent ainsi déployer leurs applications cloud natives sur l'infrastructure la plus adaptée à leurs besoins.

Dans l'approche cloud native, vous n'êtes pas les seuls à vouloir proposer ce type d'architecture. Quels sont vos avantages par rapport à d'autres fournisseurs ? Comment assurer une interopérabilité/réversibilité de Kubernetes ?

C.B : Nutanix est surtout connu pour son offre d'infrastructure. Pourtant, nous avons été parmi les premiers à embarquer en standard une solution Kubernetes managée dans notre plateforme et nous avons récemment conforté notre offre avec un PaaS complet. Notre solution Kubernetes, Karbon, est certifiée par la Cloud Native Computing Fondation (CNCF), la fondation open source qui gère Kubernetes. Karbon est totalement intégré à la plateforme d'infrastructure de Nutanix, ce qui lui permet de profiter de tous les services qu'elle propose déjà, comme la persistance du stockage. Les entreprises n'ont plus à se préoccuper du patching, de la maintenance et du déploiement de Kubernetes. L'objectif, comme ça l'est déjà pour l'infrastructure, est de rendre invisible Kubernetes. 1 Aujourd'hui, les clients Nutanix peuvent consommer des services Karbon sans coût additionnel. Nous estimons en effet que nos clients doivent pouvoir déployer aussi bien des VM que des applications conteneurisées sur notre plateforme. Karbon étant certifiée CNCF, la réversibilité est garantie avec toutes les autres distributions Kubernetes certifiées.

Pour les entreprises ayant besoin de composants additionnels, Nutanix propose un PaaS complet pour les applications conteneurisées, Karbon Platform Services, qui apporte des services comme Istio (Service Mesh), Traefik (Ingress Controler), Kafka (data streaming) ainsi que des services sophistiqués de « data pipeline ».

Depuis 24 mois, on voit se multiplier les couches Kubernetes, comme Karbon. Pourquoi un tel engouement ?

C.B : L'engouement actuel est le signal de l'arrivée à maturité dans les entreprises des architectures cloud native. Après 5 à 6 ans d'évangélisations et d'apprentissage, les entreprises commencent aujourd'hui à déployer de façon industrielle des applications conteneurisées. Et c'est pour cela que nos clients nous demandent des infrastructures capables de les supporter. Le succès de Karbon auprès de nos clients n'est que le reflet de cette nouvelle réalité.

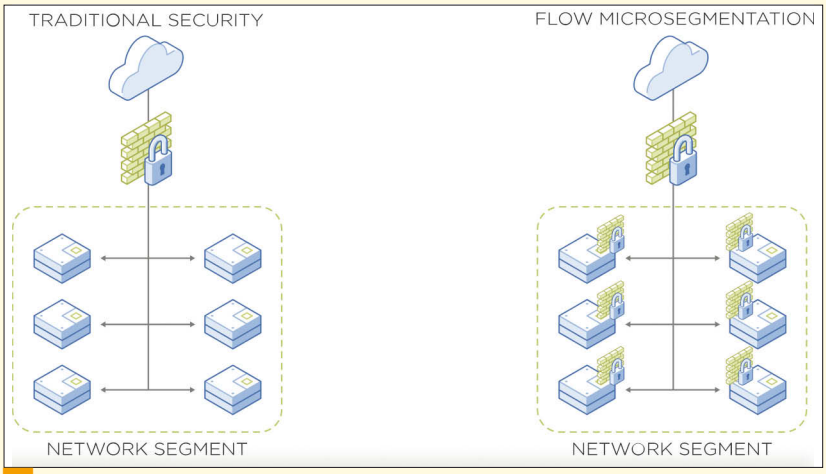
Séduire les développeurs est un enjeu pour de nombreux fournisseurs de Cloud et de PaaS. Comment adressez-vous cette population technique ? Est-ce tous les développeurs ou seulement une population précise ?

C.B : Notre offre de PaaS a été conçue pour séduire une large population de développeurs. Mais plus largement, ce sont les approches DevOps que nous souhaitons adresser. La complexité et le faible niveau d'automatisation des infrastructures traditionnelles font qu'elles ne peuvent satisfaire le rythme d'itération de plus en plus rapide des applications. La plateforme de Nutanix est largement automatisée et est aussi l'une des rares plateformes on-premises dont l'ensemble des fonctions sont pilotables par API. Cette approche API first est bien adaptée aux approches DevOps.

Un autre impératif pour nous est de permettre aux développeurs de déployer leurs applications où ils le souhaitent. Notre plateforme d'automatisation Calm et notre plateforme Paas Karbon Platform Services permettent de choisir librement où seront déployées les applications, que ce soit sur notre propre plateforme, sur des clouds publics et même sur des serveurs bare-metal. Enfin, nous avons un focus particulier sur le développement et le déploiement d'applications IA et IoT sur les edge avec des services d'inférence avancés.

Puisque vous parlez de l'IoT (découverte, compute local, connexion et récolte des données) et de l'architecture Edge. Comment adressez-vous ces modèles ?

C.B : Avec notre solution Xi IoT, nous avons été l'un des premiers à offrir une solution de déploiement à grande échelle d'application à l'edge. Cette solution est aujourd'hui utilisée dans l'industrie, le retail ou la logistique. Désormais les fonctionnalités de Xi IoT sont intégrées à Karbon Platform Services. Cette solution permet de déployer massivement des applications IoT et IA à l'edge, mais aussi dans le cloud et de les gérer sur l'ensemble de leur cycle de vie. Ce que l'on constate est que de nombreuses entreprises ont développé des prototypes viables d'application IoT ou d'applications conteneurisées pour l'edge. Mais elles butent sur le déploiement à grande échelle et sur la gestion du cycle de vie de ces applications. C'est l'un des problèmes que résout Karbon Platform Services, par exemple en gérant la mise à jour régulière des modèles de machine learning.



2 *La microsegmentation permet d'instaurer des règles et des pare-feux au niveau même des VM.*

Pour Nutanix, qu'est-ce que la microsegmentation ?

C.B : Historiquement, les entreprises ont sécurisé leur SI avec une approche périmétrique en bordant et ceinturant leur SI avec des pare-feux. Cette approche de type ligne Maginot n'est plus suffisante. Les architectures applicatives ont évolué et les applications sont de plus en plus distribuées, éclatées en de multiples composants. 2 Notre offre de microsegmentation Flow permet de déployer des services de pare-feu distribués au cœur même de l'infrastructure pour mettre en œuvre des mécanismes d'étanchéité entre bulles applicatives. Flow permet de cartographier les trafics entre VM et de définir des politiques de filtrage sophistiquées entre VM. Selon les règles définies, on peut activer ou désactiver certains flux, placer certaines VM en quarantaine, etc. Dans la dernière mouture de Flow, il est même possible de mettre en place des politiques de filtrage en fonction de l'identité des utilisateurs.

Depuis quelques semaines, vous annoncez que vos briques techniques arrivent sur les datacenters OVH. Qu'est-ce que cela signifie concrètement ?

C.B : Concrètement, OVH a annoncé lors de notre dernière conférence utilisateurs mondiale qu'ils allaient enrichir leurs services de cloud privés hébergés avec une offre basée sur notre plateforme. Cette offre va permettre à nos clients de mettre en place des clusters Nutanix dédiés dans l'infrastructure du leader français du cloud. Pour nos clients, il s'agit d'une option de déploiement additionnelle, qui vient s'ajouter aux options existantes (déploiement on premise, consommation as a service chez des ISP, déploiement bare-metal dans un cloud public). L'option de déploiement chez OVH est importante pour nos clients européens les plus soucieux de souveraineté. OVH est en effet un acteur européen qui n'est pas soumis aux dispositions du Cloud Act et du Patriot Act américains et qui dispose de datacenters dans la plupart des pays de l'Union.

Roadmap des langages

Déjà disponible

Java 15

Notons l'arrivée de l'algorithme de signature numérique Edwards-Curve, et les classes cachées (hidden classes). Les classes cachées sont des classes qui ne peuvent pas être utilisées directement par le bytecode d'autres classes. Les classes cachées sont destinées à être utilisées par les frameworks qui génèrent des classes au moment de l'exécution et les utilisent indirectement, via la réflexion. Une classe cachée peut être définie comme membre d'une imbrication de contrôle d'accès et peut être déchargée indépendamment des autres classes. (JEP 371). Plusieurs fonctionnalités avaient été introduites dans les versions antérieures, notamment le Z Garbage Collector et les blocs de texte et sont maintenant finalisées : les blocs de texte sont des littéraux chaînes à plusieurs lignes. Comme toujours, le nouveau Java est disponible en 2 versions : OpenJDK 15 et JDK 15 d'Oracle.

Kotlin 1.4

La principale nouveauté est un nouvel algo d'inférence de type (en préversion dans la 1.3). Le nouvel algorithme déduira des types dans de nombreux cas où l'ancienne inférence vous obligeait à les spécifier explicitement. Par exemple, dans l'exemple suivant, le type du paramètre lambda it est-il correctement déduit comme étant String ?

```
val rulesMap : Map<String, (String ?) -> Boolean> = mapOf(
  "weak" to { it != null },
  "medium" to { !it.isNullOrBlank() },
  "strong" to { it != null && "^[a-zA-Z0-9]+$".toRegex().matches(it) }
)
```

Le nouvel allocateur d'objets fonctionnera jusqu'à deux fois plus rapidement sur certains benchmarks, selon JetBrains. Cette version inclura aussi un nouveau compilateur IR (intermediate representation) côté backend Kotlin / JS. Mais l'éditeur prévient qu'il y aura un problème de compatibilité binaire

Python 3.9.0

La 3.9 inaugure une nouvelle cadence de mise à jour du langage : 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.

Swift 5.3

Cette nouvelle version de Swift propose beaucoup de choses : performances en hausses, Float16, disponibilité sur Windows, nouvelles distributions Linux supportées, possibilité d'appeler une fonction contenant plusieurs closures, arrivés de comparable sur les enums, plus de nécessité de multiplier les self, nouvel attribut @main, nouveaux usages de where dans les génériques, le gestionnaire de packages a été largement amélioré. Au-delà, la version 6 apportera pas mal de nouveautés et d'améliorations :

- Possibilité de porter le langage sur de nouvelles plateformes ;
- Des améliorations sur le gestionnaire de paquets ;
- Temps de compilation réduits ;
- Meilleure autocomplétion du code.

Pas de visibilité sur la roadmap.

Fin 2020 & au-delà

.Net 5

.Net 5 doit apporter une unification des différentes éditions de .Net et servir de fondation sur toutes les plateformes supportées. Parmi les nombreuses nouveautés et évolutions annoncées, notons :

- Expérience du SDK .NET unifié :
  - BCL (bibliothèque de classes de base) unique dans toutes les applications .NET 5. Aujourd'hui, les applications Xamarin utilisent le Mono BCL, mais viendront à utiliser le .NET Core BCL, améliorant ainsi la compatibilité entre les modèles d'applications.
  - Le développement mobile (Xamarin) est intégré dans .NET 5. Cela signifie que le SDK .NET prend en charge les mobiles. Par exemple, vous pouvez utiliser «dotnet new XamarinForms» pour créer une application mobile.
- Applications natives prenant en charge plusieurs plateformes : projet Single Device qui prend en charge une application pouvant fonctionner sur plusieurs appareils. Par exemple Windows Desktop, Microsoft Duo (Android) et iOS à l'aide des contrôles natifs seront pris en charge.
- Applications Web prenant en charge plusieurs plateformes : projet Single Blazor qui prend en charge une application pouvant fonctionner dans les navigateurs, sur les appareils mobiles et en tant qu'application de bureau native (Windows 10x par exemple).
- Cloud Native Applications : hautes performances, fichier unique (.exe) <50 Mo de microservices et prise en charge de la création de plusieurs projets (API, frontaux Web, conteneurs) à la fois localement et dans le cloud.
- Améliorations continues, telles que : algorithmes plus rapides dans la BCL, meilleure prise en charge des conteneurs lors de l'exécution, prise en charge de HTTP3.

Avec .Net 5, Microsoft inaugure une nouvelle cadence d'évolution annuelle de .Net pour ne plus attendre plusieurs années pour introduire une évolution majeure. En cela, l'éditeur suit Oracle sur Java.

PHP 8.0

La prochaine grande version de PHP doit arriver fin 2020. Cette version annonce beaucoup de choses, dont :

- Casse de compatibilité avec les versions 7.x sur certaines évolutions ;
- 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.

Angular vNext

Les prochaines versions apporteront pas mal de choses :

- Support de TypeScript 4 ;
- Généralisation d'Ivy dans Angular ;
- Changement et évolution du support RxJS ;
- Support natif de Trusted Types :
- Intégration de MDC Web dans Angular Material (GUI).

React 17

Les équipes React ont annoncé cet été que cette version n'aura pas de nouvelles fonctionnalités. Le focus est mis sur la stabilité et le nettoyage des fondations. Cette version n'est qu'une étape intermédiaire vers les futures versions notamment sur les mécanismes de mise à jour des modules internes.

Retour sur GDevelop avec son créateur

Franck Dubois revient pour nous sur GDevelop avec son créateur, Florian Rival. Pour en savoir plus sur ce superbe environnement de développement, Franck avait publié un gros dossier dans le hors série été 2020.

Peux-tu nous en dire plus sur la genèse de GDevelop ?

J'ai un peu galéré avec le C puis le C++ : pour le langage en lui-même, tu prends un bouquin, tu vois comment ça marche, mais tu te rends compte aussi que faire un vrai programme ça peut vite devenir compliqué. J'ai donc eu envie de rendre ça plus accessible en l'appliquant aux jeux vidéo, c'est un domaine assez incroyable où l'imagination n'a pas de limite. Je me suis dit que j'allais créer un outil plus abordable qu'un langage de programmation, j'ai donc commencé avec des jeux d'aventure « point and click » que j'ai étendu à d'autres types de jeux, c'est là qu'est né GDevelop.

L'approche que tu as adoptée pour la création de jeux avec GDevelop, l'idée t'est venue comment ?

Les 1ères lignes de code datent de 2008, avant cela j'ai bien évidemment exploré des alternatives sans que ça ne réponde à mon besoin qui était d'avoir un programme qui permette à l'utilisateur de faire des jeux vidéo avec un éditeur bien construit dans lequel on puisse placer ses objets un peu comme dans des slides. Faire un slide ça reste très accessible. C'est l'idée qui m'a un peu guidé, créer des jeux comme on crée des slides, une prise en main de l'outil rapide et intuitive. La question de la logique du jeu a été un peu plus épineuse, comment la rendre accessible ? C'est de la programmation en réalité. Je voulais que les possibilités soient illimitées, j'ai donc créé le système d'évènements basés sur des conditions et des actions. Il permet de programmer sans syntaxe à apprendre, ça prend la forme de phrases en anglais ou en français lisibles et compréhensibles et c'est un modèle qui fonctionne très bien, les conditions/actions sont pour les connaisseurs des « if/then », mais lisibles et compréhensibles pour le débutant. Au-delà, il est possible de rajouter des boucles, on reste sur du codage avec une syntaxe proche du langage parlé. J'ai poussé un peu les choses en donnant des comportements aux objets, tu crées des objets, tu leur affectes des comportements. Par exemple avec le comportement « Platformer object », mon objet se déplace sur la plateforme sans avoir à coder quoi que ce soit. Je voulais aussi que les créateurs puissent créer leurs propres comportements et j'ai donc basé le système sur de simples règles créées avec le système d'évènements. N'importe qui peut aujourd'hui créer ses propres comportements. Quand tu crées tes comportements, tu es dans un usage avancé de GDevelop. C'est un truc que j'ai ajouté récemment et je vois que ça marche très bien.

Et le « else » dans tout ça ?

Oui, pas de « else », ça oblige à des contorsions si on veut implémenter un else, j'y pense toujours, ce n'est pas facile à faire, car le moteur qui retranscrit les évènements ne traduit pas en pur « if/then » les conditions d'actions. C'est un système de filtre qui s'applique à une liste d'objets qui permet de lancer des actions sur chacun d'entre eux. Le « else » ne peut pas entrer dans ce système. Je pense le faire, mais c'est un peu compliqué.

Tu as parlé du C++ à l'origine pour le projet…

Le développement de GDevelop a commencé en C++. En 2008, le web n'était pas encore assez mature pour ce type d'application, on faisait du site web, mais pas plus. J'ai donc créé GDevelop en C++, aussi bien l'éditeur que le moteur avec OpenGL, la liste d'évènements était interprétée et traduite en C++.

Il y a un moteur JavaScript avec un build Cordova pour la partie mobile, pourquoi ce choix JavaScript/html5 ? Compte tenu des particularités du langage, j'imagine que derrière, il doit y avoir un gros boulot ?

Il y a environ 6/7 ans, j'ai commencé à voir émerger des jeux en ligne écrits en JavaScript, j'étais un peu sceptique à l'époque : pourquoi refaire en JavaScript ce que l'on sait déjà faire dans d'autres langages. A l'époque, ce n'était pas encore la panacée, mais les navigateurs modernes en ont fait un vrai langage avec des moteurs JavaScript qui font de la compilation à la volée. J'ai donc sauté le pas sans tout remettre en question. GDevelop est construit sur un modèle flexible, j'ai donc gardé l'éditeur en C++, par contre le moteur a été réécrit en JavaScript avec WebGL, tout aussi puissant qu'OpenGL. Au final, le choix était bon et il s'avère plus souple que C++. Les jeux n'ont pas été moins performants en utilisant le bon moteur de rendu.

En passant à JavaScript, tu n'as pas réécrit le moteur de rendu ?

Pas entièrement. Mon approche se veut pragmatique, j'utilise « Pixi.js » qui sert de moteur de rendu, une lib open source utilisée pour les jeux. Je veux avant tout me concentrer sur l'aspect « user friendly », et montrer que le dev de jeux n'est pas un domaine réservé. Donc si je peux utiliser un truc éprouvé, performant et open source, je prends.

L'évolution ES6 a-t-elle eu un impact sur les développements ? Avec notamment la syntaxe des fonctions et le « this » qui n'a pas la même signification où on n'est plus lié au contexte d'appel, mais plus lié à l'objet ? Cela a-t-il changé ou facilité des choses dans ton approche ?

J'ai commencé avec ES5, pas très agréable, mais en se limitant à une approche orientée objet, et une bonne connaissance du fonctionnement de « this », cela reste relativement simple, pas de « mixin » compliqué, pas de multi héritage. Alors ça n'était pas forcément très beau, mais depuis ES6, je suis devenu assez fan de JavaScript. C'est d'ailleurs assez marrant pour moi qui ait fait beaucoup de C++. Avec les « fat arrow » pour les fonctions, le fait de pouvoir faire du « TypeScript », ça me rend plus confiant dans le code que j'écris, et même par rapport à du C++. Le code est plus concis en JavaScript alors qu'en C++, on crache vite le programme avec un pointeur qui n'est pas bon. La dernière nouveauté en date est l'éditeur réécrit en technos web. Les frameworks « front-end » modernes permettent de créer une interface performante rapidement avec un code qui se lit très bien avec la nouvelle version de JavaScript. Le résultat est très modulaire et lisible.

Pourquoi Cordova pour encapsuler le build mobile ?

C'est un problème d'agilité, je ne pouvais pas me permettre de créer un moteur de jeu par plateforme. J'ai donc cherché une approche « cross platform ». Mon moteur de jeu étant en JavaScript, ça m'a paru une bonne façon de packager l'application. Avec les versions récentes de Chrome/Safari pour lesquelles JavaScript est compilé à la volée, et

grâce à WebGL qui utilise le GPU, les résultats sont très bons.

Le C++ finira-t-il par disparaître du projet ?
Non, puisque j'ai gardé la logique écrite en C++ pour traduire les évènements en JavaScript. Tout ça c'est du code brut que j'ai écrit au fil des années et je n'exclus pas à l'avenir de refaire un moteur de jeu en C++. L'éditeur est fait en « React » pour le reste c'est « Javascript », « Pixi.js », et « webGL ». L'éditeur en ligne fonctionne bien sur les téléphones récents, même s'il n'est pas fait pour être utilisé sur un mobile. En C++, ça n'aurait pas été possible.

Des mises à jour très régulières, donc très dynamiques, mais toujours en bêta ? Tu n'es pas encore satisfait du résultat alors que ça marche très bien ?
Entièrement d'accord avec toi, si ça ne tenait qu'à moi, je ne serais plus en bêta depuis longtemps. C'est plus un problème de gestion de la communauté, chacun a son idée très précise de ce que devrait être GDevelop en version finale. Par exemple, un truc qui manque et qui est très demandé par la communauté, c'est de pouvoir faire des cartes. Un contributeur y travaille actuellement. Les « tilemap », on peut le faire avec d'autres objets, mais ça manque pour faire un rpg de manière plus rapide. Et donc, pour certains, sans cet objet, on ne peut pas sortir de la version bêta. Pour moi c'est un logiciel qui permet de produire de vrais jeux, de les exporter partout, et les performances sont bonnes. Alors bien sûr tout peut être amélioré, mais je pense probablement passer en version 5.0 puis ensuite nous sortirons des versions 5.1, 5.2, etc.

On a aussi vu des fonctionnalités de GDevelop disparaître comme par exemple la découpe des « sprite sheet », sans être bloquant, ça oblige d'ailleurs à passer par des outils tiers…
En effet, c'est une fonctionnalité que j'avais sous-estimée, ainsi que l'intérêt de ces petits outils complémentaires. L'idée de départ était de rendre accessible la programmation de jeu, mais pour tout ce qui est de la partie artistique, je pensais que les gens les feraient par eux-mêmes, ce qui est d'ailleurs le cas sans quoi il n'y aurait pas de jeu produit. Mais j'ai sous-estimé ces outils intégrés, car il est très pratique de les avoir. Je trouvais celui de la version 4 (à l'époque de l'éditeur en C++) un peu basique, et me disais que les gens ne l'utilisaient pas vraiment. Ce qui, en réalité, n'était pas vrai. On est revenu un peu en arrière sur l'idée.

Il y a des outils de ce genre en préparation ?
Il y a un éditeur de Sprite intégré, « Piskel », très utile pour faire des prototypes. Les utilisateurs adorent ça parce que tout est intégré au même endroit. Il y a également un générateur d'effets sonores intégré.

Combien de contributeurs actifs aujourd'hui ?
Je suis très content parce que le projet a été choisi pour participer au « Google Summer of Code », ce qui a amené beaucoup de nouveaux contributeurs qui m'ont envoyé pas mal de choses. De façon régulière, on est un noyau dur de 5 contributeurs, avec d'autres contributeurs satellisés. J'ai confiance, ça prend de l'ampleur progressivement. Le confinement avec le covid a été aussi une période où le nombre d'utilisateurs a doublé.

L'avenir de GDevelop, j'imagine que c'est un produit dans lequel tu crois ? C'est quoi le devenir ? Tu veux en faire quoi ? Un vrai concurrent à ce qui existe ?
L'ambition est bien là, je pense qu'il y a un potentiel énorme, car ce sont des outils pas suffisamment explorés à mon sens. On commence à voir émerger des outils « nocode » pour produire du site web, ça n'a pas encore véritablement percé, mais je pense qu'il y a un vrai potentiel, voire une opportunité. Notamment pour créer des jeux au sein de studios plus importants, et qui permettrait d'imaginer des équipes avec des personnes qui programment, mais aussi d'autres qui peuvent expérimenter le gameplay directement sans savoir programmer. Les choses en amenant d'autres, ça permet de l'émulation, la stimulation des imaginations.

As-tu une idée du modèle économique en tête ?
Un truc important à mes yeux est de garder le projet en open source pour assurer une certaine pérennité pour des jeux qui auraient des durées de vie de 3, 4 ou 5 ans. Il me paraît important de donner confiance aux créateurs des jeux en assurant cette certaine pérennité et c'est la raison pour laquelle toutes les dépendances sont aussi open source. A voir après s'il est possible de faire quelque chose autour de ça pour supporter GDevelop ou créer une plateforme propulsée par GDevelop, créer une communauté qui allierait joueur et développeur, relier les 2 mondes, c'est juste une réflexion que j'ai.

Quels conseils donnerais-tu à celui qui voudrait se lancer dans l'aventure du jeu vidéo alors qu'il n'a aucune connaissance en programmation ?
D'abord je lui conseillerais d'utiliser GDevelop. Blague à part, la motivation est le moteur de la création, et pour garder cette motivation, il faut des résultats rapides. C'est d'ailleurs un des principes fondateurs de GDevelop. Ne pas s'enfermer dans l'apprentissage long et complexe d'un langage de programmation, mais essayer d'avoir des résultats concrets rapidement - ce que j'essaie de faire avec GDevelop. Essayer des moteurs de jeux comme Unreal ? Oui pourquoi pas, mais cela nécessite une sacrée motivation et ça peut vite être décourageant, mon conseil serait de commencer par un petit projet qui grandirait peu à peu.

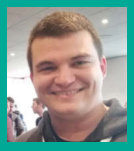
A quand la 3D avec GDevelop ?
Il est vrai qu'avec GDevelop, il n'est pas encore possible de créer des jeux 3D rapidement. Aujourd'hui, la seule façon de faire un GTA6, est de prendre un modèle de jeu GTA et de le modifier pour en changer des choses et de pouvoir dire c'est moi qui l'aie créé. Ça peut créer des vocations, mais ça ressemble plus à un mode éditeur de niveaux. Compte tenu de la charge de travail, c'est actuellement hors de portée, peut-être dans le futur, mais ça nécessitera de passer à la vitesse supérieure avec un business model adapté. La 2D a encore de beaux jours devant elle, on a de très bons jeux en « pixel art », 50% des jeux qui rapportent le plus sont en 2D.

Il y a un truc qui me dérange un peu, ce sont les « prefab » qu'on a chez les concurrents qu'on retrouve sous forme de « templates » dans GDevelop, ça donne des projets paresseux et sans originalité…
Oui tout à fait, il y a plein de gens qui font des vidéos de leur jeu, qui partent du « prefab » ou du « template », ce n'est pas quelque chose qui me gêne, ils se font plaisir, c'est un début qui peut ouvrir une porte à plus. Et puis ça a aussi un côté éducatif, il y a possibilité de voir ce qui se passe dans le « template » et de comprendre comment ça marche.

Le build GDevelop par tranche de 24h, c'est quoi l'idée ?
Les builds en ligne coûtent un peu d'argent, donc ça permet de les financer et d'en amortir les coûts. Je donne déjà de mon temps, et je n'ai pas les moyens de les offrir. La limitation à 2 builds par jour est gratuite donc ça n'empêche pas la création. Le payant au-delà évite la surcharge des serveurs et permet de supporter les coûts et d'investir dans la communauté.

Comment te trouve-t-on sur internet ?
On peut me contacter sur Twitter (twitter.com/florianrival), sur mon linkedin (https://www.linkedin.com/in/florianrival) et, bien sûr, sur le GitHub et les forums/chat de GDevelop!

Alexandre CAUSSIGNAC
Solution Engineer Manager chez VMware
Ma mission est de conseiller et d'accompagner les entreprises dans leur transformation digitale. Au-delà des technologies, j'adore échanger avec les gens sur leurs besoins finaux, car je pense que les seules barrières à ce jour dans notre secteur d'activité sont notre imagination et non notre savoir-faire technique.



Alexandre ROMAN
Solution Engineer chez VMware
Alexandre surfe sur la vague Java depuis presque 20 ans. Utilisateur de longue date de Spring Framework et Kotlin addict, il partage sa passion du développement cloud-native dans des conférences publiques comme Devoxx ou SpringOne et des webinars en ligne.

Déployer une application en mode GitOps avec Tanzu

Partie 2

Dans le n°242, nous vous avons montré comment déployer des clusters Kubernetes (K8s) prêts à l'emploi et ce, sur n'importe quel type de plateforme Cloud (On-Premise ou publique). A présent que ces clusters sont déployés et utilisables, nous allons détailler un cas pratique de déploiement d'une application sur K8s depuis le code source avec une approche GitOps. Pour rappel, l'approche GitOps repose en grande partie sur la formalisation déclarative de notre infrastructure. Cela se traduit notamment par l'usage d'outils tels qu'Ansible, Terraform et bien sûr K8s. Au même titre que l'on centralise les codes de l'application, on applique ce même modèle à l'infrastructure. Cela prend tout son sens avec un orchestrateur comme K8s.

Schéma de principe

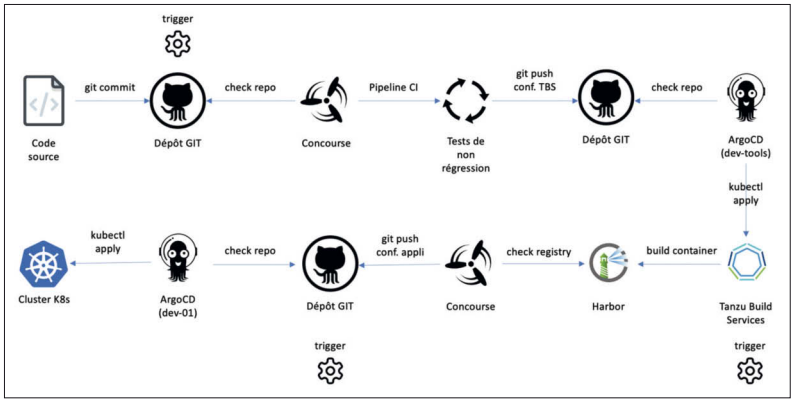
- Le développeur valide son code source et l'envoie sur un dépôt Git.
- L'outil Concourse vérifie le dépôt Git, récupère le dernier commit et déclenche un pipeline de tests de non-régression.
- Si les tests sont concluants, Concourse dépose sur le dépôt Git la configuration au format YAML pour Tanzu Build Service (TBS) qui permettra de packager le code au format container.
- L'outil ArgoCD vérifie le dépôt Git et met à jour la configuration de TBS sur le cluster K8s dev-tools.
- TBS package le code au format container et le dépose dans une registry (Harbor dans cet exemple).
- Concourse vérifie la registry et met à jour les fichiers YAML de l'application sur un dépôt Git (en particulier la version de l'image à déployer).
- ArgoCD vérifie le dépôt Git et applique la nouvelle version de l'application sur le cluster K8s.

Mise en œuvre d'un pipeline GitOps avec Tanzu

Dans cet article, nous vous proposons de mettre en place un pipeline GitOps pour vous permettre de déployer une application sur un cluster Kubernetes, sans qu'aucune intervention ne soit nécessaire. En tant que développeur, la seule chose qui vous importe c'est d'écrire du code, pour implémenter de nouvelles fonctionnalités et améliorer l'expérience utilisateur : c'est l'objectif de ce pipeline. Un commit + push sur Git, et votre application finit par atterrir sur votre cluster Kubernetes.

En tant qu'opérateur, vous voulez par-dessus tout de la stabilité sur votre infrastructure : en cas d'incident, vous voulez être en mesure de réagir rapidement. S'agissant de la sécurité des infrastructures et des applications, même combat : pas question de laisser des failles de sécurité qui pourraient nuire à la stabilité du service rendu, voire nuire à la réputation de votre marque. L'enjeu est donc d'allier stabilité de la plateforme avec une sécurisation permanente.

Notre pipeline GitOps est un moyen de répondre à ces deux problématiques : aller en production le plus rapidement possible, sans sacrifier la stabilité et la sécurité de l'ensemble.



Retrouvez l'ensemble des fichiers de configuration nécessaires au déploiement de ce pipeline sur le dépôt GitHub suivant : github.com/alexandreroman/cicd-with-tanzu.

Prérequis

Nous avons besoin des éléments suivants pour implémenter notre pipeline GitOps :

- Un cluster dédié à l'outillage : devtools ;
- Un cluster pour déployer notre application : dev01.

Avec TKG, créer le cluster devtools avec la commande suivante :

```
$ tkg create cluster devtools -p dev -w 4
```

Idem pour le cluster dev01 :

```
$ tkg create cluster dev01 -p dev -w 3
```

Assurez-vous que ces deux clusters disposent d'une configuration de stockage Kubernetes (StorageClass). Sous vSphere, vous pouvez connecter un cluster avec un DataStore existant avec la configuration suivante :

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
 name: vsphere-storageclass
```

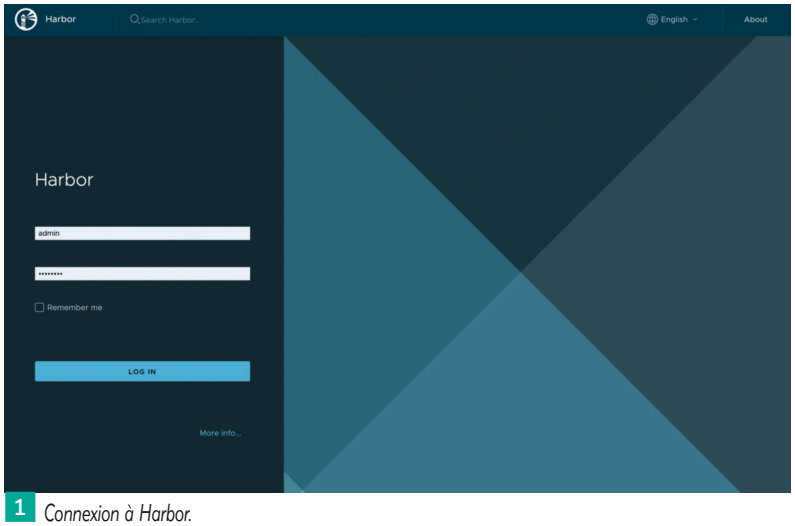
```
 annotations:
  storageclass.kubernetes.io/is-default-class: "true"
provisioner: csi.vsphere.vmware.com
parameters:
 DatastoreURL: "ds:///vmfs/volumes/5e67f6ae-7260ff3a-0d7a-000e1e617a30/"
```

Ajustez la variable DatastoreURL en fonction de votre datastore vSphere.
Afin d'exposer les services Kubernetes de type LoadBalancer, vous aurez besoin de déployer le composant open source MetalLB sur chaque cluster. En lisant la documentation du projet, vous vous apercevrez que le déploiement de MetalLB est très simple :

```
$ METALLB_VERSION=0.9.3
$ kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v$METALLB_VERSION
/manifests/namespace.yaml
$ kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v$METALLB_VERSION
/manifests/metallb.yaml
$ kubectl create secret generic -n metallb-system memberlist --from-literal=secretkey="$
(openssl rand -base64 128)"
```

Vient ensuite la configuration de MetalLB, dans laquelle vous spécifiez l'intervalle d'adresses IP à distribuer pour les services exposés Kubernetes. Utilisez deux intervalles distincts pour chaque cluster Kubernetes :

```
$ cat << EOF | kubectl apply -f -
apiVersion: v1
kind: ConfigMap
metadata:
 namespace: metallb-system
 name: config
data:
 config: |
  address-pools:
  - name: default
    protocol: layer2
    addresses:
    - 10.197.28.10-10.197.28.15
EOF
```



1 *Connexion à Harbor.*

Pour limiter le nombre d'adresses IP publiques exposées par nos clusters, nous mettons également en œuvre un IngressController. Cette étape est optionnelle, mais le déploiement d'un tel service sur une plateforme Kubernetes est courante.
Il existe différentes implémentations disponibles : nous utilisons ici Contour, un projet open source par VMware Tanzu.
Déployez Contour avec une configuration standard :

```
$ kubectl apply -f https://projectcontour.io/quickstart/contour.yaml
```

Enfin, la majorité des composants logiciels déployés sur Kubernetes s'appuient sur Helm. Ce logiciel open source est intégré avec votre gestionnaire de paquets favori (apt-get, brew, etc.). Nous utilisons ici la version 3 de Helm
Vous aurez besoin d'un Docker daemon actif pour certaines des composants au moment de l'installation.
Vous êtes prêt !

Où stocker vos images Docker ?

L'image Docker est l'unité de déploiement basique avec Kubernetes. Le registre de conteneurs (*container registry*) est un élément essentiel d'une plateforme propulsée par Kubernetes.
Notre choix se porte sur Harbor, le registre de conteneurs open source de VMware Tanzu.
Créez le fichier de configuration Helm pour Harbor :

```
expose:
 type: ingress
 ingress:
  hosts:
   core: harbor.withtanzu.com
   notary: notary.harbor.withtanzu.com
 tls:
  secretName: harbor-cert-tls
  notarySecretName: harbor-cert-tls
externalURL: https://harbor.withtanzu.com
persistence:
 persistentVolumeClaim:
  registry:
   size: 30Gi
harborAdminPassword: changeme
```

Vous l'aurez compris : vous devez adapter l'URL de connexion dans la configuration Harbor avec votre contexte de déploiement.
Cette configuration s'appuie sur des certificats TLS reconnus par notre installation. Le cas échéant, des certificats auto-signés seront générés au moment du déploiement via Helm.
Vous êtes prêt à déployer Harbor sur devtools :

```
$ kubectl create ns harbor
$ kubectl -n harbor create secret tls harbor-cert-tls \
   --cert=moncert.pem --key=moncert.key
$ helm repo add harbor https://helm.goharbor.io
$ helm install -n harbor -f harbor.conf.yml \
   harbor harbor/harbor --version 1.4.2
```

1

Dockerfile : « sa place est dans un musée ! »

Passons au vif du sujet : le déploiement de Tanzu Build Service (TBS). Ce produit de VMware Tanzu est la déclinaison commerciale des projets open source Cloud Native Buildpacks, Paketo et kpack. En combinant ces différents projets, Tanzu Build Service vous permet de construire des conteneurs sécurisés et optimisés pour déployer vos applications Kubernetes. Pas besoin d'écrire un Dockerfile : TBS se charge de détecter votre projet, et choisit les buildpacks nécessaires à la conteneurisation. Il existe des buildpacks pour différents langages et frameworks (Java, .Net, PHP, NodeJS, etc.). TBS se charge également de configurer votre application pour qu'elle fonctionne parfaitement dans un environnement conteneurisé. Fini les OutOfMemoryError pour une application Java qui ne tient pas compte du paramétrage mémoire du conteneur par exemple !

Puisque TBS s'appuie sur des technologies standard (les Cloud Native Buildpacks font partie de la fondation CNCF), vous avez la liberté de mettre TBS sur n'importe quelle distribution Kubernetes. Dans le cas présent, c'est un cluster géré par TKG qui fera tourner TBS, mais libre à vous de profiter de la puissance des buildpacks sur votre propre environnement.

Avant toute chose, téléchargez les outils kapp, ytt, et **kbld** de la suite open source Carvel.

Téléchargez TBS depuis le site Tanzu Network.

Décompressez l'archive sur votre poste de travail :

```
$ tar xvf build-service-<version>.tar -C /tmp
```

Téléchargez l'outil kp depuis Tanzu Network. C'est la commande qui sera nécessaire pour interagir avec Tanzu Build Service.

Enfin, **téléchargez le descripteur des dépendances depuis Tanzu Network** (fichier **descriptor-version.yaml**). Ce fichier référence les buildpacks disponibles dans TBS.

La prochaine étape consiste à charger TBS sur l'instance Harbor.

Connectez-vous à l'instance Harbor :

```
$ docker login harbor.withtanzu.com
```

Puis, connectez-vous à l'instance Tanzu Network Registry, où sont stockées les images de TBS :

```
$ docker login registry.pivotal.io
```

Depuis Harbor, créez un projet public **tanzu** pour stocker les images TBS : 2

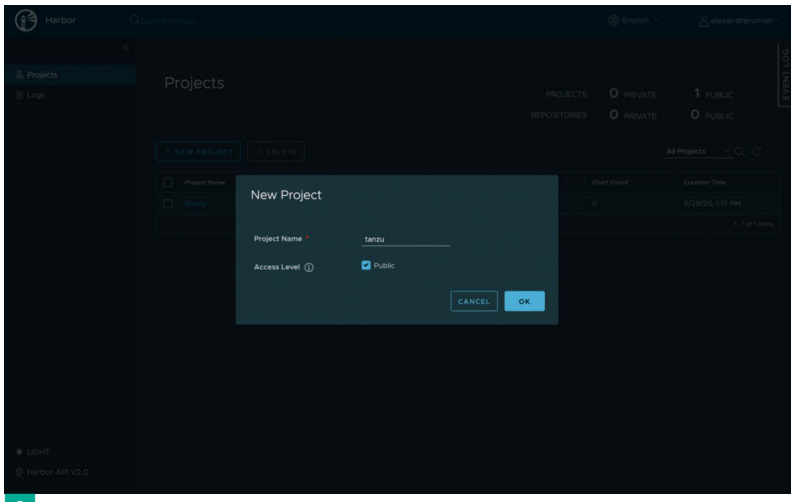
Avec la commande suivante, nous allons maintenant réaliser le chargement des images TBS :

```
$ kbld relocate -f /tmp/images.lock \
  --lock-output /tmp/images-relocated.lock \
  --repository harbor.withtanzu.com/tanzu/build-service
```

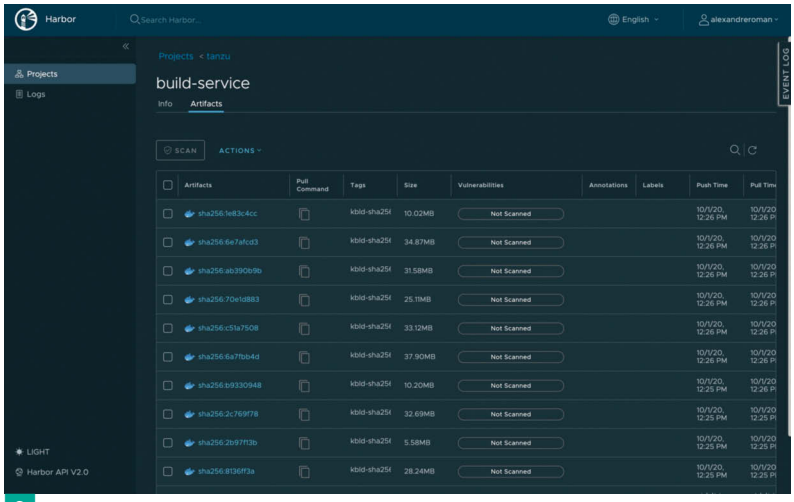
Si tout se passe bien, vous vous retrouvez avec les images TBS déployées sur Harbor : 3

Passons au déploiement de TBS sur le cluster **devtools** :

```
$ ytt -f /tmp/values.yaml \
  -f /tmp/manifests/ \
  -v docker_repository="harbor.withtanzu.com/tanzu/build-service" \
  -v docker_username="admin" \
```



2 *Création d'un projet Tanzu dans Harbor pour stocker les images.*



3 *Les images de Tanzu Build Service après chargement dans Harbor.*

```
  -v docker_password="changeme" \
  | kbld -f /tmp/images-relocated.lock -f- \
  | kapp deploy -a tanzu-build-service -f- -y
```

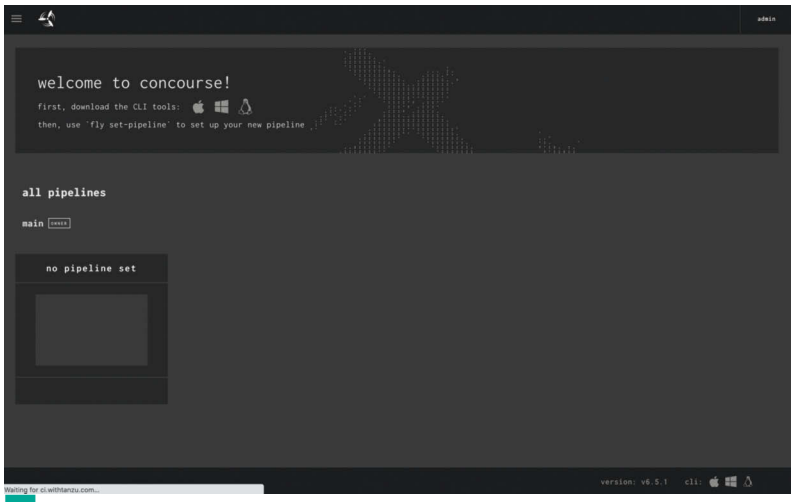
Il ne reste plus qu'à charger les buildpacks dans TBS. Cette étape peut prendre du temps, mais elle permet surtout d'en gagner par la suite ! En effet, à chaque conteneurisation d'une application, les dépendances externes (un JRE pour Java, un runtime NodeJS, etc.) seront chargées à partir d'un cache, ce qui pourra accélérer le processus. De plus, cela permet à TBS de fonctionner dans un environnement non connecté à Internet.

Lancez la commande suivante pour charger les buildpacks :

```
$ kp import -f /tmp/descriptor-<version>.yaml
```

Il est temps de vérifier si l'installation de TBS est opérationnelle :

```
$ kp clusterbuilder list
NAME    READY  STACK                IMAGE
base    true   io.buildpacks.stacks.bionic  harbor.withtanzu.com/tanzu/build-service/
base@sha256:<sha256>
default true   io.buildpacks.stacks.bionic  harbor.withtanzu.com/tanzu/build-service/
```



4 *Concourse est prêt !*

```
default@sha256:<sha256>
full true io.buildpacks.stacks.bionic harbor.withtanzu.com/tanzu/build-service
/full@sha256:<sha256>
tiny true io.paketo.stacks.tiny harbor.withtanzu.com/tanzu/build-service
/tiny@sha256:<sha256>
```

Il ne reste plus qu'à donner accès à l'instance Harbor à Tanzu Build Service :

```
$ kp secret create harbor --registry harbor.withtanzu.com --registry-user admin
```

Félicitations : désormais, vous n'aurez plus besoin d'écrire de Dockerfiles pour vos futurs conteneurs !

Un peu de CI…

Créer une image pour un conteneur ne fait pas tout : il faut tout de même s'assurer que notre projet compile, que les tests unitaires n'introduisent pas de régression, etc. Ce rôle est dévolu à un outil d'intégration continue (*Continuous Integration*). L'offre en matière d'outils d'intégration continue est riche : Jenkins ou GitLab sont parmi les plus connus.

Pour ce pipeline GitOps, notre choix se porte sur un outil open source méconnu, mais non moins efficace : Concourse. Déployé dans le cluster devtools, le rôle de Concourse est de faire le passage de relai entre les différents outils et étapes du pipeline. Avec son approche stateless (rien n'est stocké dans Concourse), c'est un outil qui se marie très bien avec les approches de développement moderne (d'aucuns diront *cloud native*).

Passons au déploiement de Concourse sur le cluster devtools :

```
concourse:
  web:
    externalUrl: http://ci.withtanzu.com
    localAuth:
      enabled: true
    auth:
      mainTeam:
        config: |
          roles:
          - name: owner
            local:
              users: ["admin"]
            localUser: "admin"
      kubernetes:
        enabled: true
      configRBAC: |
        owner:
        - SetTeam
        member:
        - CreateBuild
web:
  ingress:
    enabled: true
    hosts:
    - ci.withtanzu.com
secrets:
  localUsers: "admin:changeme"
```

Déployons Concourse sur le cluster :

```
$ kubectl create ns concourse
$ helm repo add concourse https://concourse-charts.storage.googleapis.com
$ helm upgrade concourse concourse/concourse -n concourse \
  -f concourse.conf.yml --version 13.0.0 --install
```

Nous y sommes presque ! 4

… et de CD !

Il nous reste un dernier outil à déployer : ArgoCD. Comme son nom le laisse penser, il est responsable de déployer des éléments sur nos clusters Kubernetes. Cet outil open source est la pierre angulaire de l'approche GitOps : tous les états sont stockés sur un dépôt Git. Parmi ces informations, nous retrouvons :

- Le commit Git correspondant à la version de notre application à déployer, celle qui a passé la batterie de tests de non-régression ;
- Le tag de l'image Docker contenant notre application (stockée dans Harbor) ;
- La configuration Kubernetes nécessaire pour le déploiement de l'application.

L'intérêt de l'approche GitOps, c'est de pouvoir rejouer / annuler tout changement d'état.

Revenir à la version précédente d'une application ? un simple git revert suffit.

Restaurer l'application telle qu'elle a été déployée il y a 3 mois ? Idem.

Par-dessus tout, il est plus facile d'interchanger les outils avec une approche GitOps.

Nous allons nous appuyer sur ArgoCD pour deux raisons :

- Configurer TBS dans devtools pour ordonner la construction d'une image en référençant un commit Git de notre application, en indiquant les coordonnées de l'image à produire ;
- Déployer l'application sur le cluster dev01 une fois l'image prête.

A chaque fois, le rôle d'ArgoCD est de synchroniser la configuration stockée sur un dépôt Git dédié, et d'appliquer cette configuration là où c'est nécessaire. Dans le cas présent, TBS n'a pas besoin de connaître l'existence de Concourse, mais seulement le commit Git de l'application à conteneuriser. Idem pour le déploiement de l'ap-

plication sur un cluster Kubernetes : l'origine de l'image importe peu (« s'agit-il d'une nouvelle version ou d'une ancienne ? »).
Pour chacun des clusters devtools et dev01, déployons ArgoCD :

```
$ kubectl create ns argocd
$ kubectl apply -n argocd -f \
  https://raw.githubusercontent.com/argoproj/argo-cd/v1.7.4/manifests/install.yaml
```

Une fois déployées, voyons comment nous connecter à ces deux instances ArgoCD. Il est recommandé de ne pas exposer ArgoCD à l'extérieur, via une adresse IP publique ou un LoadBalancer, pour des raisons de sécurité. En effet, ArgoCD dispose d'informations de connexion sensibles, et sa manipulation par un tiers donne un accès direct à un cluster. C'est pourquoi nous allons ouvrir un tunnel (connexion éphémère) vers les instances ArgoCD lorsque nous en aurons besoin, que nous fermerons à la fin des opérations.
Pour cela, nous allons utiliser la fonction de kubectl pour ouvrir une connexion vers une application Kubernetes.
Ouvrons la connexion vers ArgoCD/devtools :

```
$ kubectl --context devtools -n argocd port-forward \
  svc/argocd-server 9000:443
```

Idem avec ArgoCD/dev01 :

```
$ kubectl --context dev01 -n argocd port-forward \
  svc/argocd-server 9001:443
```

Il est temps de se connecter. Le mot de passe par défaut d'une instance ArgoCD correspond au nom du pod argocd-server. Vu que Kubernetes alloue un nom aléatoire à chaque pod, c'est un mécanisme plutôt malin qui garantit que chaque installation est unique.
Récupérons le mot de passe pour chaque instance ArgoCD :

```
$ kubectl -n argocd get po -l app.kubernetes.io/name=argocd-server \
  --no-headers -o custom-columns=":metadata.name"
```

Il est désormais possible de se connecter.
Connectons-nous sur ArgoCD/devtools :

```
$ argocd login localhost:9000
```

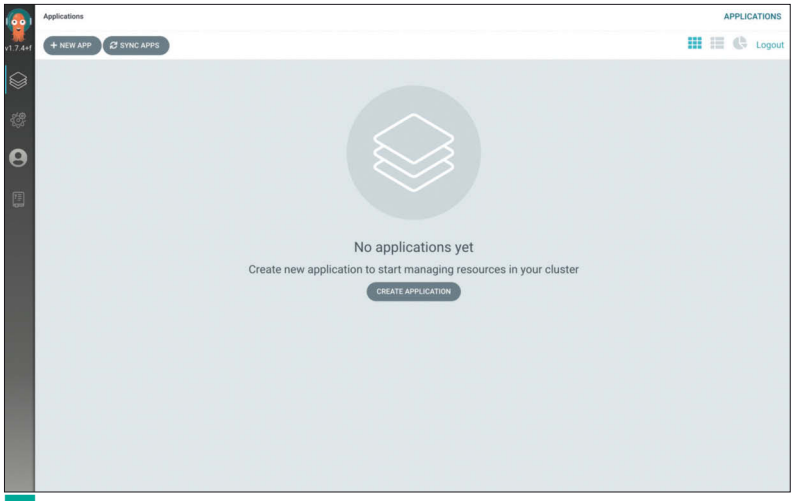
Si besoin, il est possible de changer le mot de passe :

```
$ argocd account update-password
```

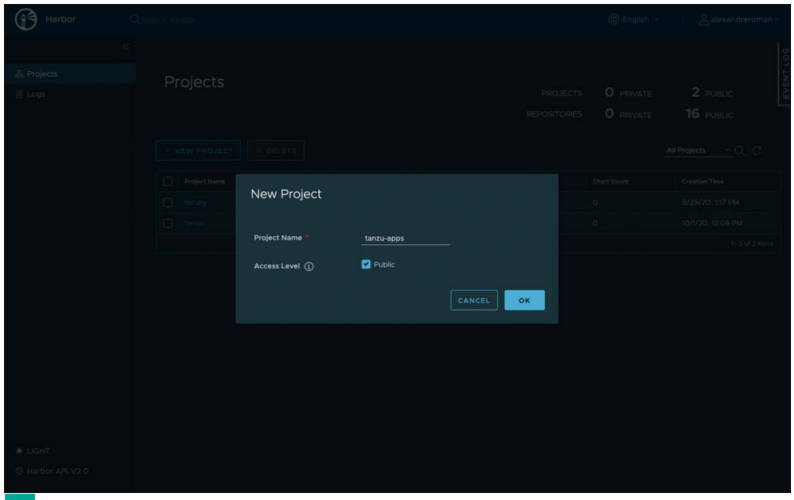
Il est temps de configurer ArgoCD. Connectons ArgoCD au dépôt Git du pipeline GitOps :

```
$ argocd repo add \
  https://github.com/alexandreroman/cicd-with-tanzu-gitops.git \
  --username mygithubhandle --name gitops
```

Vous noterez l'utilisation du dépôt cicd-with-tanzu-gitops. Il contient les différents états dont nous parlions précédemment. Vous aurez besoin d'un accès en écriture à ce dépôt : n'hésitez pas à le cloner pour vos propres besoins.
Répétez les opérations précédentes pour le cluster dev01.
Notez qu'ArgoCD propose également une IHM accessible depuis votre navigateur. 5



5 *ArgoCD est prêt à déployer.*



6 *Création d'un projet Harbor pour stocker les images de nos applications.*

Ouvrons les vannes !

Vous êtes prêt à créer un premier pipeline pour une application. Pour cela, nous utiliserons une simple application NodeJS, disponible à cette adresse : github.com/alexandreroman/cnb-nodejs.
Récupérez https://github.com/alexandreroman/cicd-with-tanzu.
Différents exemples sont présents dans le dossier samples.
Éditez l'exemple pour cnb-nodejs dans le fichier samples/app-cnb-nodejs.yml, et ajustez les paramètres en fonction de votre instance Harbor, le dépôt GitOps, etc.
Vous aurez également besoin de créer le fichier samples/git.yml en renseignant vos identifiant / mot de passe GitHub :

```
git-username: mygithubhandle
git-password: github-access-token
```

Créez un projet public dans Harbor, dans lequel les images seront stockées : 6
Déployez le pipeline dans Concourse :

```
$ fly -t concourse set-pipeline -p cnb-nodejs \
  -c concourse/nodejs-pipeline.yml \
```

```
-l samples/git.yml \
-l samples/app-cnb-nodejs.yml
```

Le pipeline est disponible dans Concourse : activez-le. 7
A partir de là, le code source disponible dans GitHub va être téléchargé. Concourse se charge d'exécuter les tests unitaires, pour vérifier si des régressions ont été introduites. En cas de succès, un fichier de configuration pour Tanzu Build Service est créé :

```
apiVersion: kpack.io/v1alpha1
kind: Image
metadata:
 name: cnb-nodejs
 namespace: default
spec:
 tag: harbor.withtanzu.com/tanzu-apps/cnb-nodejs
 serviceAccount: default
 builder:
  name: default
  kind: ClusterBuilder
 cacheSize: "2Gi"
 source:
  git:
   url: https://github.com/alexandreroman/cnb-nodejs
   revision: 483b9f0e41667232f7762fa178d46589bf71cb0c
```

Ce fichier référence toutes les informations importantes pour créer une image depuis TBS :

- Où trouver le code source de l'application ? (commit Git) ;
- Comment construire l'application ? (référence au ClusterBuilder, utile pour personnaliser le contenu de l'image produite) ;
- Où stocker l'image produite ? (emplacement de l'image sur Harbor).

A partir de là, TBS est autonome pour assurer la conteneurisation de votre application. Dans le cas où des buildpacks seraient mis à jour (par exemple : nouvelle version de NodeJS), toutes les images connues de TBS sont reconstruites à partir des informations disponibles. TBS est donc capable de mettre à jour des conteneurs automatiquement sans nécessiter une action de l'équipe de développement. Dans le cas d'une mise à jour de buildpacks faisant suite à la correction d'une faille de sécurité, TBS est donc capable de patcher vos images à l'échelle sans nécessiter l'intervention de votre part.
Le fichier de configuration de TBS est stocké dans le dépôt GitOps, dans le dossier kpack. Un nouveau commit a été produit avec cette nouvelle image.
Le rôle d'ArgoCD sur le cluster devtools est de surveiller ce dossier.
Ajoutons la configuration nécessaire pour appliquer la configuration de l'image dans TBS depuis le dépôt GitOps :

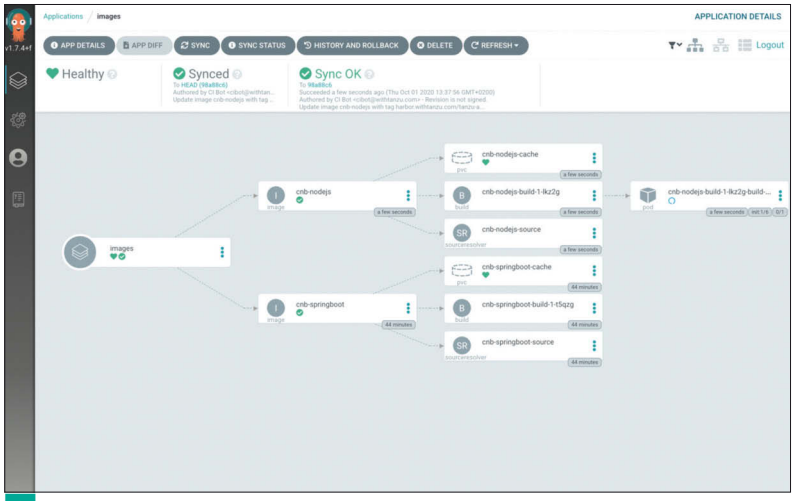
```
$ argocd login localhost:9000
$ argocd app create images --file samples/argocd-images.yml
```

Le fichier de configuration ArgoCD définit où trouver la configuration GitOps, et sa destination :

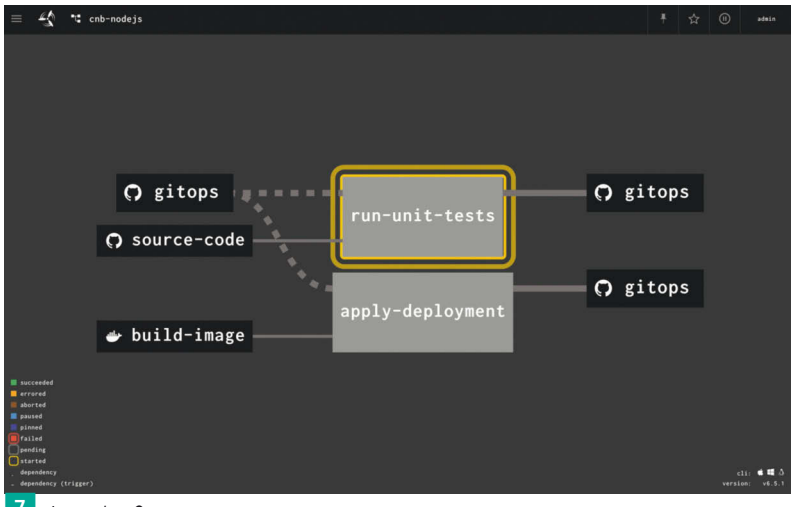
```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
 name: images
spec:
 project: default
 source:
  repoURL: 'https://github.com/alexandreroman/cicd-with-tanzu-gitops.git'
  path: kpack
  targetRevision: HEAD
 destination:
  server: 'https://kubernetes.default.svc'
  namespace: default
 syncPolicy:
  automated:
   selfHeal: true
```

En se connectant au dépôt GitOps, ArgoCD se rend compte que la configuration de TBS n'est pas synchronisée : la nouvelle configuration est donc appliquée, ce qui déclenche la conteneurisation par TBS. 8
Visualisons l'état de l'image cnb-nodejs :

```
$ kp build status cnb-nodejs
Image:      harbor.withtanzu.com/tanzu-apps/cnb-nodejs@sha256:<sha256>
```



8 *Déclenchement automatique de la conteneurisation via Tanzu Build Service à partir d'ArgoCD.*



7 *Le pipeline Concourse en action.*

```
Status:        SUCCESS
Build Reasons:  CONFIG

Pod Name:   cnb-nodejs-build-1-jplkz-build-pod

Builder:     harbor.withtanzu.com/tanzu/build-service/default@sha256:<sha256>
Run Image:   harbor.withtanzu.com/tanzu/build-service/run@sha256:<sha256>

Source:     GitUrl
Url:        https://github.com/alexandreroman/cnb-nodejs
Revision:   483b9f0e41667232f7762fa178d46589bf71cb0c

BUILDPACK ID                BUILDPACK VERSION
tanzu-buildpacks/node-engine   0.0.52
tanzu-buildpacks/npm           0.0.48
```

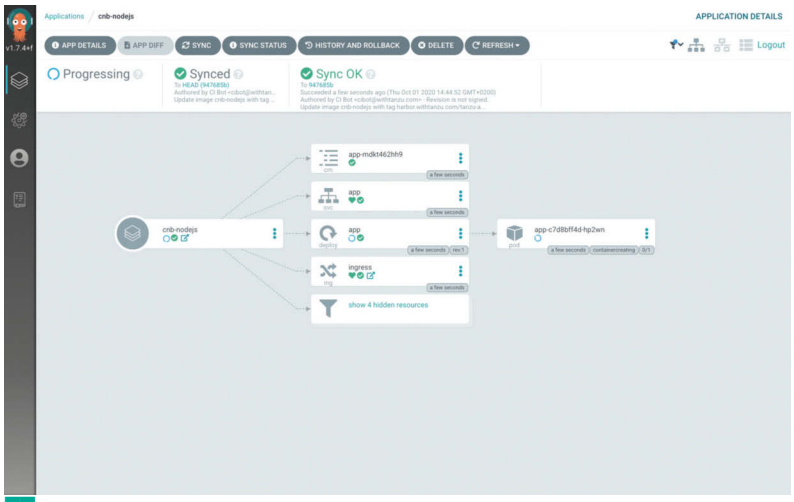
Une image a été produite pour l'application `cnb-nodejs` : elle est désormais disponible dans Harbor. Notez que vous pouvez récupérer les logs de construction de l'image via la commande suivante :

```
$ kp build logs cnb-nodejs
…
Adding label 'io.buildpacks.lifecycle.metadata'
Adding label 'io.buildpacks.build.metadata'
Adding label 'io.buildpacks.project.metadata'
*** Images (sha256:<sha256>):
    harbor.withtanzu.com/tanzu-apps/cnb-nodejs
    harbor.withtanzu.com/tanzu-apps/cnb-nodejs:b1.20200916.185854
Adding cache layer 'tanzu-buildpacks/node-engine:node'
===> COMPLETION
Build successful
```

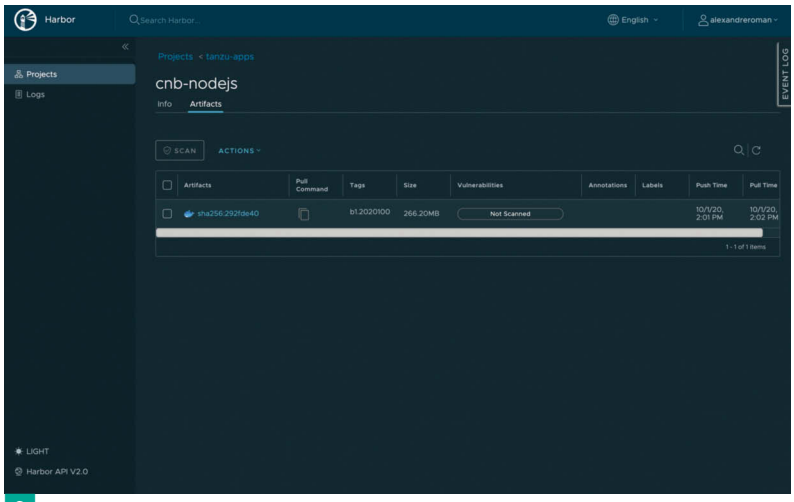
Jetons un œil justement à cette image dans Harbor : 9
Voilà donc une image Docker correspondante à cette application NodeJS, pour laquelle il n'a pas été nécessaire d'écrire un Dockerfile ! C'est là que la seconde étape du pipeline intervient : lorsqu'une nouvelle image est disponible dans Harbor, le tag correspondant est identifié pour être inscrit dans la configuration de déploiement Kubernetes. C'est Concourse qui se charge de stocker cette information en ajoutant un commit dans le dépôt GitOps. 10
La configuration de déploiement Kubernetes est mise à jour en référençant l'image produite :

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../base
images:
- digest: sha256:<sha256>
  name: cnb-nodejs
  newName: harbor.withtanzu.com/tanzu-apps/cnb-nodejs
commonAnnotations:
  timestamp: 20200916T202249
```

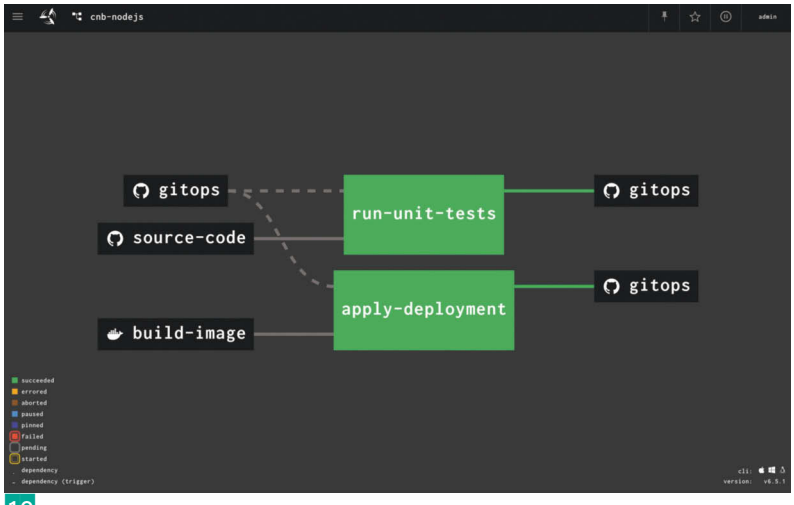
Seule la référence au tag de l'image est mise à jour. En effet, la configuration de déploiement Kubernetes de cette application s'appuie sur Kustomize, un outil bien pratique pour gérer les nombreuses lignes de fichiers YAML nécessaires pour toute application



11 *ArgoCD se charge de déployer notre application dans le cluster Kubernetes.*



9 *Tanzu Build Service a construit une image pour nous, désormais stockée dans Harbor.*



10 *Mise à jour de la configuration GitOps de déploiement via Concourse.*

12 *La nouvelle version de l'application est déployée à la suite d'un commit Git par un développeur.*

Kubernetes. Venons-en à la dernière étape : déployons l'application sur le cluster dev01.
Une fois encore, c'est le rôle d'ArgoCD de synchroniser la configuration du dépôt GitOps (contenant la configuration de déploiement Kubernetes) avec le cluster cible.
Déployons la configuration ArgoCD pour dev01 :

```
$ argocd login localhost:9001
$ argocd app create cnb-nodejs --file samples/argocd-cnb-nodejs.yml
```

En quelques instants, ArgoCD se rend compte que la configuration doit être synchronisée, et le déploiement de l'application commence. En moins d'une minute, l'application cnb-nodejs est déployée sur dev01. 11
Pour saisir tout l'intérêt de ce pipeline GitOps, répétez les opérations suivantes en modifiant le code source de l'application. Vous vous apercevrez que tout le processus s'exécute de nouveau, sans qu'aucune intervention ne soit nécessaire. 12
Mieux encore, annulez cette modification en modifiant manuellement le dépôt GitOps. Puisque tout est stocké en gestion de configuration, vous avez l'opportunité de reproduire le déploiement de l'application comme bon vous semble. Un exemple mettant en œuvre une application Java avec Spring Boot est également disponible : cnb-springboot.

Harbor

Harbor est un produit open source à l'initiative de VMware. C'est un serveur permettant de stocker On-Premise ou dans le cloud des images Docker de manière fiable et sécurisée. Parmi les grandes fonctionnalités, on retrouve :

- Gestion de la sécurité, des comptes, rôles, habilitation (RBAC) ;
- Réplication des images entre plusieurs instances de Harbor ;
- Portail web ;
- Logging de toutes les opérations à des fins d'audit ;
- API RESTful ;
- Scan de vulnérabilité intégré ;
- Nettoyage automatique des images inutiles (garbage collection).

Tanzu Build Services (TBS)

Créer et gérer manuellement le packaging d'artefact au format container est tout à fait faisable mais dans certaines conditions (nombre important de tests, usine logicielle, …) il peut s'avérer utile de déléguer cette tâche à une plateforme (PaaS). Les options ne manquent pas pour créer des containers à partir de code source. Toutefois, nombre d'entre elles sont laborieuses et nécessitent une maintenance constante. Avec Tanzu Build Service, nul besoin de maîtriser parfaitement les formats de packaging des containers, ou encore la rédaction d'un script de création de containers pour un langage de programmation donné.

Concourse

Concourse CI est un outil ultra léger et simple d'utilisation qui permet d'automatiser la construction d'un projet, l'exécution des tests de ce projet, mais aussi de lancer des analyses sur le code source… bref des tâches souvent rébarbatives et chronophages pour un développeur.

ArgoCD

Un des composants majeurs du projet Argo s'appelle ArgoCD qui facilite les déploiements sur le modèle GitOps. La Cloud Native Computing Foundation (CNCF) prend sous son aile ce projet, qui regroupe des outils natifs à K8s pour le déploiement et la gestion de workflows en conteneurs. Nous l'utilisons beaucoup pour mettre à jour nos composants d'infrastructure via un dépôt Git mais cette fois-ci nous l'utiliserons également pour déployer notre application.

CONCLUSION

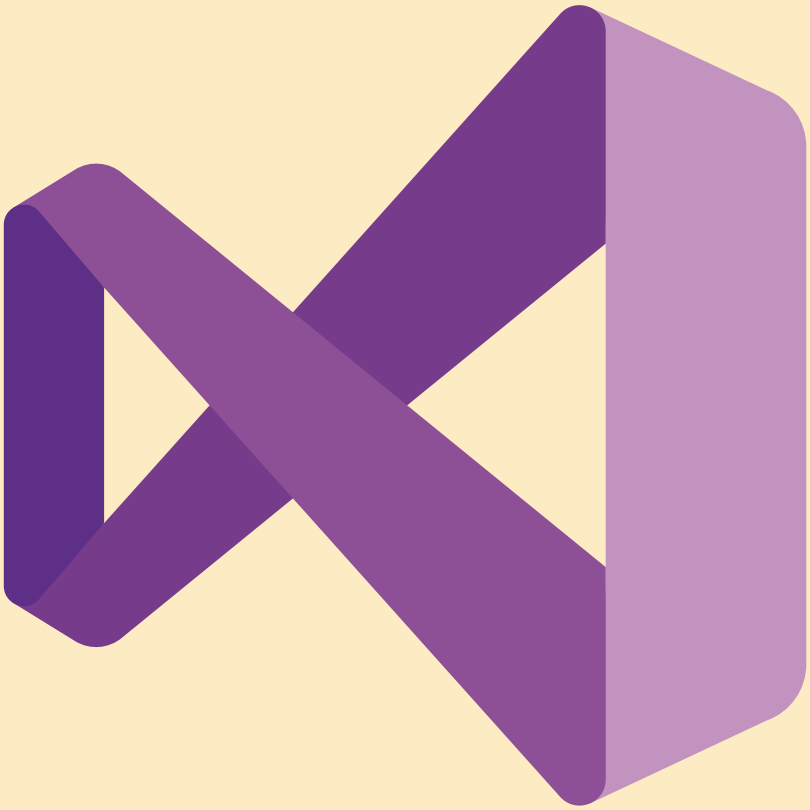
A travers ce dossier, vous avez pu manipuler des éléments d'infrastructure (clusters K8s, load balancer, persistent volume, …) ainsi que des applications via du code source à travers une approche GitOps permettant de tirer la quintessence des architectures dites déclaratives. Nous avons tenu à détailler l'implémentation de chaque composant en vue de vous expliquer les briques techniques utilisées. Cependant, la mise en œuvre de ces produits nécessite beaucoup de compétences et requiert la maintenance de beaucoup de fichiers de configuration (propres à l'approche K8s), ce qui peut être un frein à la mise en production de vos applications. Pour y remédier, VMware propose également une plateforme prête à l'emploi qui simplifie drastiquement le path to production grâce à Tanzu Application Service. Ce produit, construit sur la base open source Cloud Foundry, est un accélérateur pour vos équipes de développement et d'opération, puisqu'il vous permet de déployer des applications sans avoir à vous préoccuper d'autre chose que l'écriture du code source. Tout le reste (load balancer, conteneurisation, ingress, connexion à des services, etc.) est pris en charge par la plateforme, qu'elle soit on-premise ou sur une offre de cloud public.

Pour en savoir plus

VMware Tanzu aborde ces sujets de simplification du path to production et la mise en œuvre de Kubernetes en entreprise dans une série de webcasts gratuits. Rendez-vous sur https://tanzu.tv .

DevCon#9

Conférence développeur 100 % .Net+Azure+Windows



19 NOVEMBRE 2020

A PARTIR DE 13H30

10 sessions & quickies

.Net 5 - C++20 - Grafana+Azure - Projet Orleans
Blazor - .Net Core - etc.

Agenda complet et inscription : **https://www.programmez.com**

Evènement virtuel

Conférence du magazine Programmez!

En partenariat avec SoftFluent, ZDNet et CACD2

Smalltak :
le retour d'un langage mythique

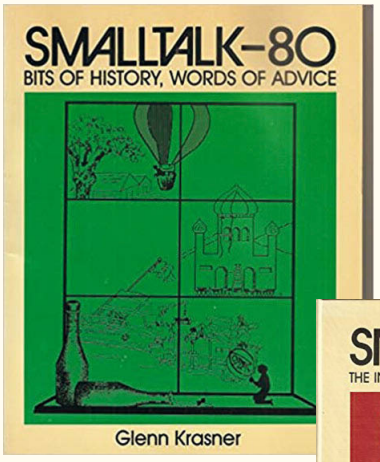
PARTIE 1

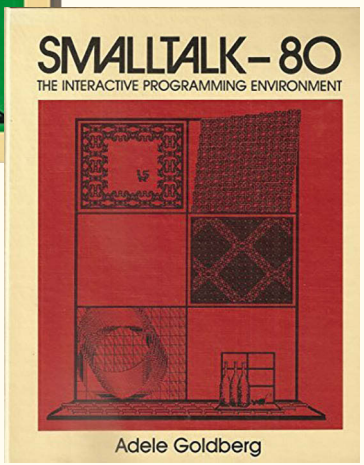
Smalltalk n'est pas le langage de programmation le plus connu. Et pourtant, il joua un immense rôle dans la micro-informatique et les grandes révolutions technologiques des années 70 et 80.

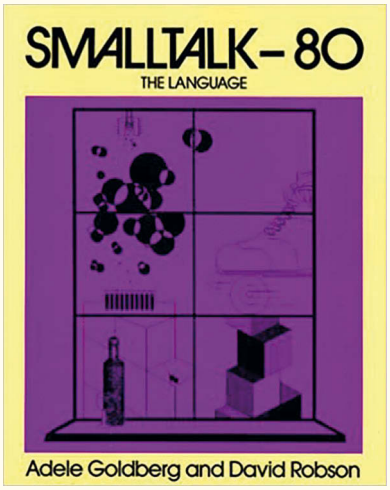
Un des créateurs du langage fut Alan Kay. Il oeuvra au Xerox PARC (voir notre Histoire de l'informatique volumes 1 et 2, Technosaures n°4) avant d'intégrer Apple. SmallTalk fut influencé par plusieurs langages tels que FLEX, Planner, LOGO, META II. La première véritable version apparaît fin 1972. Il fallut attendre la version dite Smalltalk-80 pour voir un langage mature et exploitable. Il aurait pu devenir le langage de référence d'Apple pour les projets Lisa et Macintosh. Pascal lui fut préféré.

Programmez! vous propose de découvrir, de redécouvrir ce fantastique langage avant de longuement aborder Pharo, implémentation moderne de Smalltalk.

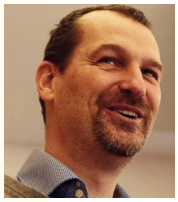
Enjoy !



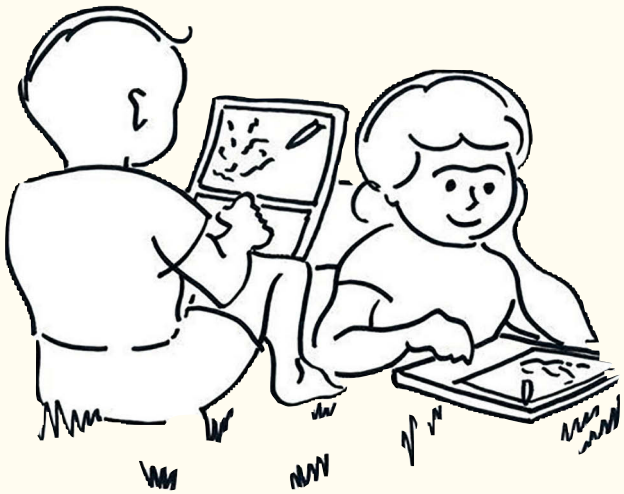




Remerciements
Merci à Laurent Julliard pour avoir proposé ce grand dossier spécial.
Merci x2 à Stéphane Ducasse (INRIA) pour son travail d'écriture, de relecture et de traduction !
Merci à tous les auteurs et aux communautés Smalltalk et Pharo.



Dossier relu et assemblé par Stéphane Ducasse
Directeur de recherche Inria, dirige l'equipe RMOD. Expert en conception objet, conception de langages à objets, programmation réflexive ainsi que maintenance et l'évolution de large applications (visualisation, métriques, meta modelisation). Ses travaux sur les traits ont été introduits dans AmbientTalk, Slate, Pharo, Perl-6, PHP 5.4 et Squeak et ont influencées les langages Scala et Fortress. Président du consortium de Pharo http://consortium.pharo.org. *Auteurs de nombreux ouvrages dont* http://books.pharo.org



La Saga Smalltalk

Début 1980, PARC, le célèbre centre de recherche de Xerox situé à Palo-Alto publie Smalltalk-80. C'est le point d'orgue de près de 10 ans de recherche. Smalltalk-80 et son indissociable compagnon, l'ordinateur personnel Alto, ont dix ans d'avance sur le marché. Voici leur histoire.

Laurent Julliard
Laurent a une longue carrière dans la R&D et l'ingénierie logicielle chez HP, Xerox, Schneider Electric et deux startups. Fondateur du premier Linux User Group en France en 1995 et pionnier Ruby et Rails dans les années 2000, c'est un Open Sourcer convaincu. Il travaille actuellement chez Google Cloud.

1980, je viens d'entrer au lycée. Un lycée de province comme les autres ou presque… Celui-ci a en effet le privilège de faire partie du Plan Informatique mis en place par l'Etat pour favoriser l'enseignement de l'informatique. Plantons le décor : un Mitra 15, mini-ordinateur de fabrication française de la taille d'une armoire normande, permet de s'initier à la programmation avec le langage LSE. Les sauvegardes des programmes se font sur bandes perforées en papier. Pas d'internet, pas encore de micro-ordinateurs mais tout cela me paraît déjà magique. Je suis loin de me douter qu'au même moment, à l'autre bout de la planète, Xerox PARC vient tout juste de publier Smalltalk-80, un environnement de programmation et une interface utilisateur pour ordinateur personnel totalement révolutionnaires.

La vision Smalltalk

Lorsque Xerox PARC est officiellement constitué le 1er Juillet 1970, le mandat qui lui est donné par l'équipe dirigeante de Xerox est d'inventer le futur avec une ligne d'horizon de 5 à 10 ans. Les plus brillants chercheurs de l'époque y trouvent une ambiance de créativité débridée et des moyens financiers considérables. 1 2

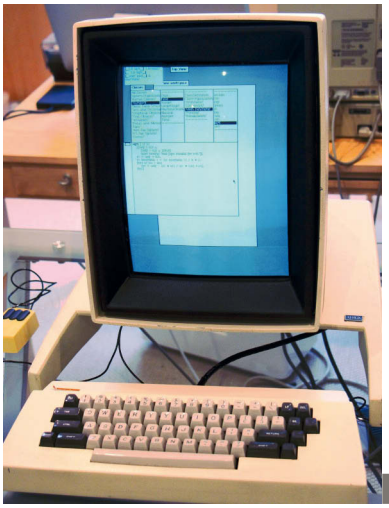
L'équipe du Computer Science Laboratory (CSL) de Xerox PARC commence à se former. Arrivent d'abord Charles P. Thacker et Butler W. Lampson, deux génies du hardware qui sont convaincus que le futur de l'informatique réside dans des ordinateurs personnels puissants, compacts, communicants qui interagissent directement avec les utilisateurs. Ils commencent à travailler d'arrache-pied sur un ordinateur révolutionnaire qui matérialise leur vision : l'Alto. Pour la première fois on retrouve sur une même machine : un écran bitmap noir et blanc 606 x 808, 72 dpi orientable en mode horizontal ou portrait, 128 à 256 Ko de RAM, un disque dur amovible de 2,5 Mo (les disquettes n'existent pas encore), un clavier et surtout, pour la première fois, une souris à 3 boutons. Bob Metcalfe rejoint bientôt l'équipe, invente Ethernet et met les Altos en réseau. Initialement Xerox PARC avait prévu de fabriquer une trentaine d'Alto uniquement pour ses besoins de recherche. Au final c'est plus de 1000 unités qui seront produites mais jamais à des fins commerciales. 3

Côté logiciel, dès 1970, Alan Kay rejoint PARC et forme le Learning Research Group (LRG) qui affiche clairement la couleur : l'environnement Smalltalk doit avoir des vertus pédagogiques et permettre aux enfants d'interagir avec un ordinateur et de le programmer (d'où le choix du nom Smalltalk qui pourrait se traduire par "bavardage des petits" en français). La première version de Smalltalk sort en 1971 mais ne reçoit pas de nom de baptême. Très inspirée des langages académiques de l'époque comme FLEX, PLANNER, LOGO et META II, cette première version propose un langage de type symbolique. Rapidement, Alan Kay conclura qu'il est trop difficile à apprendre par des enfants.

Entre temps, d'autres brillants esprits comme Dan Ingalls (inventeur de l'algorithme BitBlit qu'on retrouvera dans l'Alto et plus tard sous forme d'ASIC dans les Atari ST et les Amiga), Ted Kaeler (inventeur du programme de musique Twang pour l'Alto), Adele Goldberg et Dave Robson rejoignent le groupe LRG. Un matin, une discussion de couloir s'engage entre Alan Kay, Dan

1

2



3

Ingalls et Ted Kaeler sur la question du niveau de complexité que doit atteindre un langage de programmation pour être qualifié de "puissant". Alan Kay, dans un excès de vantardise, leur répond : "je vous parie que je peux spécifier un langage de programmation puissant en moins d'une page". Ses deux compères le mettent au défi d'y parvenir.

Pendant les 2 semaines qui suivent Alan Kay va travailler 16 heures par jour à la spécification de ce nouveau langage et le tester sur le papier. La présentation qu'il en fait à ses collègues 2 semaines plus tard emporte l'adhésion. Smalltalk-72 vient de naître et avec lui la programmation orientée objet (OOP) dont Alan Kay vient d'inventer les 6 principes fondateurs :

- Tout est objet.
- Les objets communiquent entre eux en envoyant et en recevant des messages.
- Chaque objet a sa propre mémoire (comprendre : l'état d'un objet est stocké dans un espace mémoire qui lui est propre).
- Chaque objet est l'instance d'une classe.
- La classe contient la spécification du comportement commun à toutes ses instances.
- Pour évaluer un programme le contrôle est passé au premier objet, et le reste est considéré comme des messages qui lui sont destinés.

A l'époque, une des grandes forces du LRG réside dans sa capacité à expérimenter rapidement toutes les nouvelles idées sur l'Alto. En quelques jours Dan Ingalls crée un premier interpréteur et évalue avec succès l'expression 3 + 4, qui en Smalltalk signifie : envoie à l'objet 3 appelé receveur le message + avec comme argument l'objet 4. Le résultat s'affiche : 7. Victoire ! L'interpréteur fonctionne quoique lentement, ce qui amènera plus tard à la définition d'un bytecode et d'une machine virtuelle dès la version Smalltalk-76.

Un visiteur à PARC : Steve Jobs

Avance rapide: nous sommes en 1979. Smalltalk-78 est sorti l'année précédente. Le langage reste fidèle à ses origines avec quelques évolutions syntaxiques mais c'est surtout l'environnement Smalltalk qui a énormément évolué. Outre le bytecode et la machine virtuelle apparus dans la version précédente, la panoplie de classes standards s'est considérablement étoffée et de nombreuses applications ont été développées avec Smalltalk par plusieurs équipes de PARC : le traitement de texte WYSIWYG (acronyme inventé à PARC) appelé Bravo, logiciel de dessin Draw, le logiciel de musique Twang, un environnement de développement intégré (IDE) totalement dynamique capable de conserver son état d'exécution d'une session de travail à l'autre, une caractéristique qui encore aujourd'hui n'a pas d'équivalent sur le marché. Et le tout, bien sûr, en multi-fenêtres et totalement piloté à la souris. Du jamais vu qui laisse bouche bée les rares visiteurs qui ont la chance d'assister à une démo à Xerox PARC.

Parmi ces visiteurs privilégiés, Steve Jobs. A cette époque, fin 1979, Apple rencontre le succès avec l'Apple II et commence tout juste à imaginer son prochain ordinateur : le Lisa. De son côté, la Xerox Development Corporation, la division de Xerox dont l'objectif est d'investir dans des sociétés encore jeunes mais prometteuses, cherche à entrer au capital de Apple. Des négociations commerciales s'engagent entre les deux sociétés. Steve Jobs, qui n'est pas encore CEO d'Apple à l'époque, tente un coup de poker : il propose à Xerox d'entrer au capital mais il y met comme condition de pouvoir accéder aux recherches de Xerox PARC. Il s'ensuit des discussions houleuses en interne à PARC : certains sont pour, d'autres y sont farouchement opposés. Finalement la demande est acceptée mais, sans le savoir, Apple ne verra en réalité qu'un dixième des innovations de PARC. Cela suffit néanmoins à déclencher un enthousiasme débordant chez Steve Jobs et ses équipes techniques. De l'avis de toutes les personnes présentes ce jour là et bien que Steve Jobs s'en soit toujours défendu en interview, il sort de cette visite totalement hypnotisé par l'environnement graphique qu'il vient de voir. Le Lisa sortira sur le marché en 1983 avec une interface graphique, une souris et des logiciels très inspirés (pour ne pas dire copiés) des travaux de Xerox PARC.

L'année qui suit la visite de Steve Jobs, 1980, est celle de la maturité pour Smalltalk. Apple, HP, Tektronix et l'Université de Berkeley portent avec succès la machine virtuelle ST-80 sur leurs propres machines. Et surtout, le très célèbre magazine de micro-informatique américain BYTE qui tire à 500 000 exemplaires consacre son numéro d'Août 1981 à Smalltalk-80 (si vous trouvez ce numéro, conservez-le précieusement car c'est devenu un collector absolu). 4

Ce numéro spécial est une véritable révélation pour le monde de la micro- informatique, tout comme le sera 18 mois plus tard l'ouvrage ultime sur Smalltalk: "Smalltalk-80, the language and its implementation" co-écrit par Adele Goldberg et David Robson de Xerox PARC. Ce volume, devenu à ce point mythique qu'il est maintenant tout simplement appelé "The Blue Book" en référence à sa couleur, présente en 700 pages le langage Smalltalk, sa bibliothèque de classes standards, son environnement de développement et surtout une implémentation complète de la machine virtuelle elle-même écrite en Smalltalk. Aujourd'hui encore, la lecture de cet ouvrage est un incontournable si vous voulez vraiment comprendre comment créer un langage de programmation, son environnement et comment fonctionne une machine virtuelle.

L'aventure commerciale

Si la plateforme Smalltalk-80 est une réussite totale sur le plan de la recherche et de ses retombées, sa destinée commerciale sera beaucoup plus chaotique. Une série de décisions incompréhensibles ou de circonstances contraires entraveront considérablement son essor. 5

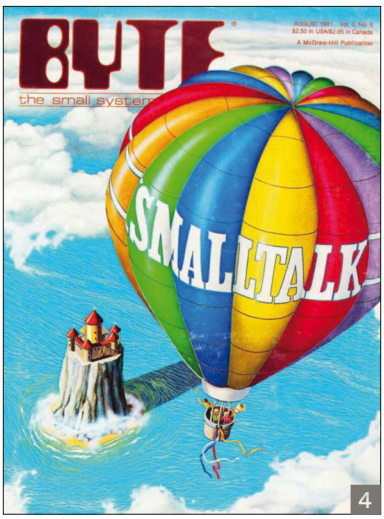
Tout d'abord jusqu'en 1985, pour Smalltalk, la barrière est d'ordre matériel. Aucune machine du marché n'a les capacités de l'Alto. Seule exception : le poste bureautique Xerox 8010 (plus connu sous le nom de Xerox Star) qui sort en Avril 1981. Conçu par la division Systèmes de Xerox (SDD), c'est un clone de l'Alto au niveau matériel. On aurait donc pu s'attendre à trouver Smalltalk sous le capot. Il n'en est rien. SDD a pris la décision saugrenue de réécrire complètement les applications bureautiques développées à PARC en langage MESA, un dialecte de Pascal ! C'est une machine superbe et très coûteuse : vendue 16 000 $ l'unité (environ 40 000 $ actuels), peu d'entreprises peuvent se l'offrir. De plus la Xerox 8010 est une machine totalement fermée et dépourvue d'environnement de programmation ! Pas de Smalltalk à l'horizon : un comble !! En 1985, Xerox corrige le tir avec la Xerox 6085 mais il est déjà trop tard : le premier Macintosh est sorti un an plus tôt et, dans l'entreprise les compatibles PC sous MSDOS/Windows 1.0 sont en passe de s'imposer.

Durant cette même période, des personnes

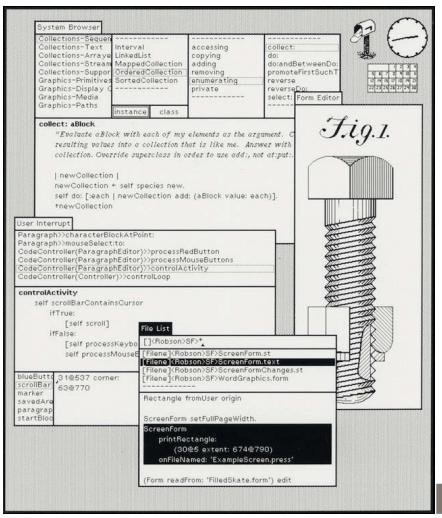
clés de Xerox PARC, lassées de l'incapacité du management de Xerox à exploiter la mine d'or que représentent Smalltalk-80 et ses applications, quittent la société. Larry Tesler, un des concepteurs de l'interface graphique de Smalltalk, rejoint Apple dès 1981 après que Xerox ait refusé de licencier Smalltalk à Apple. Il jouera un rôle décisif dans la conception de Lisa et du Macintosh 512. Alan Kay le rejoint en 1984. Charles Simonyi, le concepteur du traitement de texte WYSIWYG Bravo, rejoint Microsoft et créera MS Word. Bob Metcalfe, inventeur du réseau Ethernet de l'Alto part lui aussi pour fonder 3Com. John Warnock et Charles Geschke, créateur des langages de description de pages Interpress et Postscript, fondent Adobe Systems. Et beaucoup d'autres suivront. Il n'est pas exagéré de dire que Xerox, en laissant partir ses talents, va largement contribuer à l'essor de la Silicon Valley et de l'informatique personnelle.

Quant à Adele Goldberg, dernier pilier de Smalltalk après le départ de Alan Kay, elle quitte PARC à son tour en 1986 et crée la société ParcPlace dans le but de commercialiser l'environnement Smalltalk hors de Xerox. ParcPlace fusionne bientôt avec Digitalk pour devenir ObjectShare et lance VisualWorks, un environnement Smalltalk de qualité professionnelle disponible sur Unix, Mac et PC. La société affiche rapidement une belle santé avec un chiffre d'affaire de 55 M$ fin 1994.

C'est alors que Sun Microsystems (un mastodonte de l'informatique d'entreprise à l'époque) approche ParcPlace pour acquérir les droits de licence de Smalltalk et en faire sa plateforme de développement. ParcPlace refuse l'offre de Sun jugée trop basse. Ce refus va s'avérer fatal ! En réaction, Sun va pousser un petit projet interne du nom de OAK qui sera présenté au public en Mai 1995 sous son nouveau nom commercial Java. Peu après avoir décliné l'offre de Sun, les financiers qui ont pris le pouvoir chez ObjectShare en font une société de consulting spécialisée en programmation orientée objets. Dans la foulée, ils publient une lettre ouverte totalement surréaliste à destination de leurs clients VisualWorks (le propre environnement Smalltalk de ObjectShare) recommandant d'adopter Java qui, pourtant, n'en est qu'à ses balbutiements ! Cette lettre ouverte achève de désorienter les clients et le déclin commence peu après.



4



5

VisualWorks sera finalement racheté par Cincom qui le commercialise encore aujourd'hui sur Linux, Mac et Windows. Sun de son côté récupèrera l'équipe qui a développé la machine virtuelle de Smalltalk, puis la VM hyper rapide Strongtalk dont les avancées technologiques seront finalement intégrées dans la VM Hotspot de Java.

Côté Open Source, les choses bougent aussi. Dès la publication du Blue Book, plusieurs implémentations libres de Smalltalk apparaissent : GNU Smalltalk, Little Smalltalk, ... En 1995, Alan Kay et Dan Ingalls, désormais tous deux chez Apple "remettent le couvert" en créant Squeak, une implémentation ouverte, hautement portable de Smalltalk dont la machine virtuelle est elle-même écrite en Smalltalk. De 1996 à 2010 Squeak est un projet Open Source actif mais son orientation très marquée vers les applications ludiques et multimédia va limiter son adoption dans l'entreprise.

En 2008, pour répondre aux besoins des développeurs professionnels, Stéphane Ducasse et Marcus Denker, responsables de la version 3.9 de Squeak, créent Pharo, un fork de Squeak disponible sous licence MIT. Plusieurs dizaines d'entreprises utilisent aujourd'hui Pharo pour leurs développements logiciels dans de nombreux domaines applicatifs. Comme vous le verrez dans ce numéro, la communauté Pharo est très active et Smalltalk reste plus que jamais un environnement de développement innovant et inspirant.

Moralité de l'histoire

Sur les 40 années écoulées, Smalltalk aura donc réussi le tour de force d'être à l'origine d'une grande partie des technologies matérielles et logicielles actuelles sans jamais apparaître dans le classement des langages les plus populaires. Smalltalk a lourdement influencé Java (VM), Objective-C, Ruby, PHP, Perl, Python, Dart, Groovy, Scala et beaucoup d'autres. C'est à ce jour le seul langage écrit en lui-même et totalement réflexif. On lui doit le premier IDE moderne avec éditeur interactif, navigateur de classe, inspection d'objet et debugger intégré pour une interaction immersive avec le code. C'est aussi Smalltalk qui a permis l'invention du Test Driven Development, de l'Extreme Programming et du refactoring dès 1995. C'est Smalltalk toujours qui a popularisé le modèle de programmation MVC. C'est Smalltalk encore qui a véritablement assis la crédibilité des machines virtuelles et qui le premier intègre la compilation à la volée (Just In Time, JIT).

Des décisions stratégiques désastreuses, des batailles politico-juridiques et, sans doute, une avance trop importante sur son temps ont malheureusement empêché Smalltalk de prendre son envol sur le plan industriel. Cela n'enlève rien à ses qualités intrinsèques. Smalltalk reste un langage orienté objet des plus remarquables, et sa syntaxe n'a pas varié d'un iota depuis 40 ans, preuve de sa puissance.

Tout ceci devrait vous inciter à lever le voile pour comprendre ce qui rend cette plateforme si unique. Ce numéro spécial est là pour vous y aider. Bonne lecture !

Références

The Early History of Smalltalk, Alan C. Kay, 1993, Apple Computer

Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age de Michael A. Hiltzik, Avril 2000, Harper Business

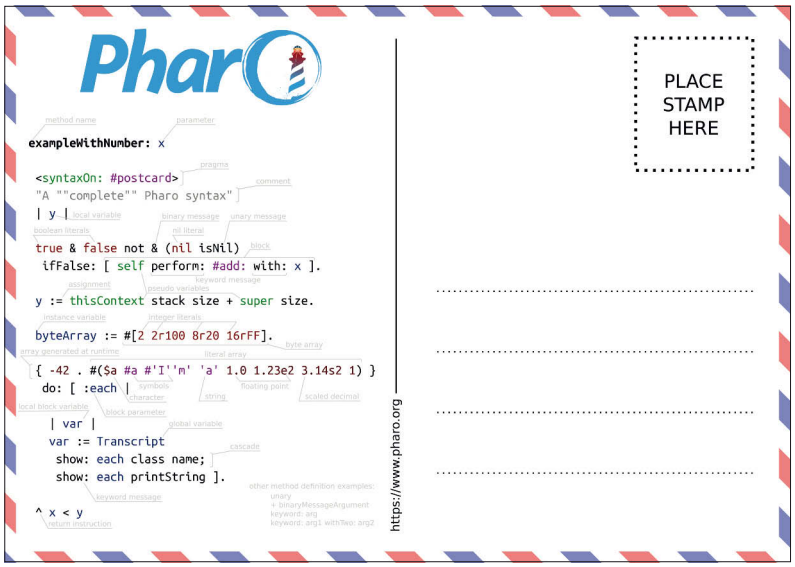
Laurent Julliard
Google Cloud

Introduction à Smalltalk

Smalltalk est né d'un projet de recherche initié à Xerox PARC au début des années 1970 qui atteindra sa pleine maturité en 1980. 40 ans plus tard sa syntaxe, remarquable de simplicité, n'a pas subi de modifications. Attention ! En 3 pages, vous allez apprendre tout Smalltalk.

Des objets et des messages

Si vous avez déjà entendu parler de Smalltalk, vous avez peut-être aussi entendu dire que sa syntaxe tient sur une simple carte postale. Il ne s'agit ni d'une légende, ni d'une exagération : une carte postale suffit bel et bien pour définir toute la syntaxe de Smalltalk. Voyez plutôt :



Avant de s'appuyer sur cette carte postale pour vous initier à Smalltalk, passons d'abord en revue quelques grands principes fondateurs du langage.

En premier lieu, Smalltalk est un langage orienté objet, et, pour être plus précis, c'est historiquement le premier langage à avoir embrassé l'approche objet dans son intégralité. En Smalltalk tout est objet ; sans exception aucune et jusqu'aux types les plus simples comme les entiers et les booléens. Tout objet est l'instance d'une classe. D'ailleurs les classes elles-mêmes sont des objets. Ceci fait de Smalltalk un des langages les plus réflexifs qui soit avec des capacités d'introspection et de méta-programmation rarement égalées.

Deuxième principe fondateur : vous agissez sur les objets en leur envoyant des messages, lesquels messages sont implémentés dans des méthodes. La syntaxe générale d'un message suit le patron 'receveur message', où receveur est l'objet auquel on envoie le message. Le message lui-même peut être de 3 types :

Les messages unaires : comme par exemple 5 sqrt qui envoie le message sqrt (racine carrée) à l'objet 5 (une instance de la classe Integer). Ou bien encore Date today qui envoie le message today à l'objet Date (qui ici représente la classe Date) ou pour finir true not Qui envoie à l'objet true, unique instance de la classe True, le message not pour obtenir bien sûr en retour false.

Les messages binaires : comme par exemple 3 + 4 qui littéralement envoie le message + à l'objet 3 avec comme unique argument 4 donnant comme résultat 7. Ou bien 4 <= 3 qui envoie à l'objet 4 le message <= avec comme argument l'objet 3 retournant false.

Les messages à mots-clés : ce sont des messages formés d'une série de mots clés se terminant par le signe deux-points suivi chacun d'un argument. Ainsi :
1 to: 10 crée l'intervalle 1 à 10
Color r: 1 g: 0 b: 0 crée un objet instance de la classe Color en passant 3 arguments pour chaque canal rouge, vert et bleu.
Transcript show: 'Hello World !' demande à la classe Transcript qui représente en Smalltalk l'équivalent d'une fenêtre textuelle d'afficher la chaîne de caractères Hello World !

Dans une expression Smalltalk comprenant plusieurs messages l'évaluation se fait par ordre de précédence d'abord sur les messages unaires, puis les binaires et enfin les messages à mots-clés. Pour les messages de même type qui se succèdent, l'évaluation est effectuée de gauche à droite. Prenons un exemple pour que vous compreniez bien cette mécanique essentielle :

```
2 raisedTo: 1 + 3 factorial
```

Dans cet exemple nous trouvons le message à mots-clés raisedTo: (qui fait une élévation à la puissance), le message binaire + et le message unaire factorial. Si vous appliquez la règle de priorité c'est donc '3 factorial' qui est d'abord évalué soit 6, puis le message binaire + soit 1 + 6 donnant 7, et enfin 2 raisedTo: 7 qui donne 128. Est-ce que vous aviez trouvé le bon résultat ?
Prenons un second exemple. Là encore essayez de trouver le résultat par vous-même :

```
3 + 4 * 2
```

Vous avez trouvé 11 ? C'est effectivement le résultat que donneraient à peu près tous les langages de programmation. Mais pas Smalltalk. Pourquoi ? Appliquons la règle de priorité apprise plus haut. Avec deux messages de même type qui se suivent, ici les messages binaires + et *, l'évaluation se fait de gauche à droite. Donc 3 + 4 égale 7, puis 7 * 2 donne 14.
Cet exemple a pour but d'introduire un autre élément de syntaxe de Smalltalk que sont les parenthèses. Elles permettent assez logi-

quement de modifier l'ordre d'évaluation des messages. Ainsi pour obtenir un résultat qui respecte la précédence habituelle des opérateurs arithmétiques l'expression précédente aurait dû être écrite de la façon suivante :

```
3 + (4 * 2)
```

Maintenant que vous savez comment envoyer des messages aux objets voyons deux autres éléments syntaxiques en rapport. Le premier est l'utilisation du point. En Smalltalk le point permet de séparer les expressions. On le place en général en fin de ligne avant d'exprimer une nouvelle expression sur la ligne suivante mais ce n'est pas une obligation. Ainsi :

```
x := 10.
y := x + 20
```

Sont 2 expressions distinctes qui assignent respectivement une valeur aux variables locales x et y. Vous noterez que le point peut être omis pour la dernière expression.
L'autre élément syntaxique important est le point-virgule. Il sert à envoyer une suite de messages au même objet receveur sans devoir le répéter à chaque fois. Ainsi dans l'exemple qui suit, 3 messages successifs sont envoyés à la même variable globale Transcript. D'abord le message 'cr' qui passe à la ligne, puis le message show: demandant l'affichage de la chaîne de caractère 'hello world' suivi à nouveau d'un retour chariot.

```
Transcript cr;
    show: 'hello world';
    cr
```

Fermetures lexicales

Un aspect très important est la définition et exécution de fermetures lexicales (appelées blocs dans le jargon Smalltalk). La définition de fermetures se fait à l'aide de [ ].
Ainsi [1 + 2] définit une fermeture lexicale qui lorsqu'elle sera exécutée retournera 3. Notez que [1 / 0] retourne aussi une fermeture (un objet bloc) et non pas une exception puisque le code à l'intérieur du bloc n'est pas exécuté au moment de sa définition. L'exécution d'une fermeture est déclenchée en lui envoyant le message value.

```
[ 1 + 2 ] value
>>> 3
[ 1 / 0 ] value
>>> lève une erreur de division par zéro.
```

Les fermetures couplées à l'envoi de messages rendent la syntaxe de Smalltalk extrêmement compacte. En gérant une séquence de code comme un objet autonome, les blocs permettent d'en déclencher l'exécution ou pas à la demande. C'est une fonctionnalité essentielle pour les structures de contrôle comme nous allons le voir.

Structures de contrôle

Si toute la syntaxe de Smalltalk tient sur une carte postale, c'est en grande partie parce qu'il n'existe pas de syntaxe particulière ou mots-clés réservés pour les structures de contrôle comme les branchements conditionnels ou les boucles. Comment est-ce possible ? Et bien en Smalltalk toutes ces structures de contrôle sont tout simplement exprimées sous forme de messages comme les autres avec des blocs de code utilisés en argument ou en receveur. Prenons en exemple la bien connue méthode factorielle :

```
factoriel
    "Renvoie le factoriel du receveur"
        ^ self = 0
         ifTrue: [ 1 ]
         ifFalse: [ self * (self - 1) factorial]
```

Avant d'en venir à la structure de contrôle présente dans cet exemple, voici quelques informations complémentaires qui vous permettront de mieux comprendre le code présenté ici :

- Toute séquence de caractères entre double apostrophe est un commentaire
- = est le message binaire qui permet de tester l'égalité entre deux objets avec pour résultat un booléen true ou false.
- Comme dit précédemment, toute expression entre crochets [….] est un bloc (aussi appelé savamment closure dans d'autres langages). En Smalltalk un bloc est aussi un objet qui contient des expressions dont l'exécution est différée et qu'on peut par exemple passer en paramètre à un message.
- Le caractère ^ interrompt l'exécution du contexte en cours et retourne la valeur de l'expression qui suit. C'est l'équivalent de return dans d'autres langages.

Avec ces nouveaux éléments vous avez surement déjà compris que la construction syntaxique "if (condition) then … else … end" qu'on trouve dans beaucoup de langages est tout simplement exprimée sous forme d'un message de type mots-clés en Smalltalk de la forme suivante :

```
ifTrue: blocSiVrai ifFalse: blocSiFaux
```

La méthode correspondante exécutera donc le bloc de code 'blocSiVrai' ou 'blocSiFaux' suivant que le receveur (ici le résultat du message binaire =) est vrai ou faux. C'est ce caractère universel des messages qui donne à Smalltalk à la fois concision et puissance. Là où la plupart des autres langages doivent enrichir leur syntaxe pour apporter de nouveaux concepts, Smalltalk, lui, a uniquement besoin de définir de nouvelles classes et de nouveaux messages.
Le même principe s'applique aux structures de boucle. Là non plus pas de syntaxe particulière du langage mais simplement des messages envoyés à des objets. En voici quelques exemples :

```
4 timesRepeat: [ … ]  # répète 4 fois le bloc passé en argument
[ n < 100 ] whileTrue: [ … ]  # exécute le bloc en argument tant que n < 100
(1 to: 10) collect: [ :i | i * i ]  # collecte le carré des 10 premiers entiers dans un tableau
```

Les variables

En Smalltalk il existe 3 types de variables : les variables locales, les variables partagées et les pseudo-variables.

Les variables locales sont déclarées au début d'une méthode entre deux barres verticales | .... | Leur nom commence par une lettre minuscule et suit la convention de nommage dite Camel Case comme par exemple :

```
|x y poids vitesseMaximale|
```

Les variables partagées : leur nom commence obligatoirement par une lettre majuscule et suit aussi la convention de nommage Camel Case. Il en existe de 4 types : les variables globales, les variables de classe, les variables de pool et les noms de classe. Ces variables se distinguent davantage par leur localité et leur visibilité que par leur nature, car, au final, elles désignent toutes un object Smalltalk.

Une variable globale comme son nom l'indique est une variable accessible depuis n'importe quel endroit dans votre code. Par exemple la variable Transcript qui désigne l'équivalent d'une console texte est une variable globale instance de la classe TranscriptStream. Les noms de classe sont elles aussi des variables partagées comme par exemple Integer, String, Array, OrderedCollection, ... Enfin une variable partagée peut être assignée à un "pool" de variables c'est à dire un dictionnaire de variables qu'on peut rendre accessible à différentes classes, de façon sélective, contrairement aux variables globales qui, elles, sont visibles de toutes les classes.

Terminons avec les pseudo-variables qui ne sont rien d'autre que des noms de variables réservés. Elles sont au nombre de 6 : self (qui désigne le receveur de la méthode courante), super (la superclasse de self), true et false (instances uniques des classes True et False), nil (objet indéfini, unique instance de la classe UndefinedObject), et, enfin, thisContext qui pointe sur le frame le plus récent dans la stack d'exécution).

Les littéraux

Nous venons de couvrir les objets, les messages et les variables qui à eux seuls représentent 90% de la syntaxe de Smalltalk. Il ne nous reste plus qu'à introduire les formes littérales utilisées pour créer les objets de base comme les entiers, les chaînes (String) et les tableaux (Array), et vous saurez ainsi tout ce qu'il y a savoir sur la syntaxe Smalltalk.

Les entiers et les flottants s'expriment de façon habituelle : -5, 10, 3.14, 2.4e7,... A noter que vous pouvez exprimer un entier dans n'importe quelle base de la façon suivante : 2r1001 (l'entier 9 en base 2), 8r70 (56 en octal) ou encore 16rFFFF (65535 en base 16).

On peut créer un objet caractère (instance de la classe Character) en utilisant le signe $. Ainsi '$a' exprime le caractère ASCII 'a'. Une chaîne de caractères (instance de la class String) s'exprime entre apostrophe comme dans 'Hello World'. Un tableau statique s'initialise en utilisant la syntaxe '#(....)' comme par exemple ici le tableau des 5 premiers entiers #(1 2 3 4 5). Les objets utilisés dans les tableaux statiques doivent tous pouvoir être compilés avant l'exécution. Vous pouvez aussi initialiser un tableau dynamique avec la notation { ... } dans lequel chaque élément est une expression Smalltalk comme dans { 1. 2. 1+2 } qui va générer un tableau dynamique contenant au final les entiers 1, 2 et 3 après évaluation.

Aller plus loin

Envie de mettre en pratique ce que vous venez d'apprendre ? Rien de plus simple. Téléchargez Pharo pour Linux, Mac ou Windows depuis http://pharo.org et suivez le tutoriel fourni. Vous trouverez aussi dans ce numéro spécial un article complet sur l'environnement de programmation Pharo.
Comme vous avez pu le constater, la syntaxe de Smalltalk est à la fois désarmante de simplicité et d'une incroyable puissance. En cela, Alan Kay, concepteur du langage dans les années 1970, a parfaitement répondu au défi qu'il s'était lui-même fixé (voir l'article sur l'Histoire de Smalltalk). 40 ans plus tard la syntaxe n'a pas changé d'un iota et Smalltalk, soutenu par une communauté toujours très active, continue à jouer son rôle d'aiguillon. Dans les articles qui suivent vous trouverez de nombreux autres exemples de code Smalltalk.

Tableau résumant la syntaxe

Syntaxe	Ce que ça représente
startPoint	une variable locale
Transcript	une variable partagée
self	pseudo-variable
1	Entier décimal
2r101	Entier binaire
1.5	Nombre flottant
2.4e7	Flottant en notation exponentielle
$a	Le caractère 'a'
'Hello'	La chaîne "Hello"
#Hello	Le symbole #Hello
#(1 2 3)	Un tableau de littéraux
{1. 2. 1+2}	Un tableau dynamique
"a comment"	un commentaire
| x y |	déclaration de variables locales x and y
x := 1	affecte 1 à x
[x + y]	Un bloc qui évalue x+y
<primitive: 1>	Primitive de la VM ou annotation
3 factorial	Message unaire
3+4	Message binaire
2 raisedTo: 6 modulo: 10	Message à mots-clés
^ true	Renvoie l'objet vrai
Transcript show: 'hello'. Transcript cr	Séparateur d'expression (.)
Transcript show: 'hello'; cr	Cascade de messages (;)

Laurent Julliard
Google Cloud

L'héritage de Smalltalk

Depuis sa création à Xerox Parc au début des années 1970 jusqu'à la publication de ses spécifications au début des années 80, l'histoire de Smalltalk est une véritable saga (ndlr : voir l'article sur l'Histoire de Smalltalk dans ce numéro). Mais ce qui fascine encore davantage ce sont les innombrables innovations que l'équipe CSL (Computer Systems Lab) de Xerox Parc nous a laissé en héritage durant cette même période. Aujourd'hui, dans la plupart des outils, langages et méthodologies utilisés nous retrouvons l'empreinte indélébile de cette folle décennie de recherche.

PROGRAMMATION ORIENTÉE OBJET

C'est bien sûr une contribution majeure de Smalltalk-80 au monde de l'informatique. Certes le langage Simula avait déjà ébauché quelques pistes auparavant mais c'est vraiment Alan Kay dans sa définition de Smalltalk-80 qui va définir les notions d'objet, de classes, de superclasses, d'héritage, de polymorphisme, de variables de classes et d'instance, etc. Et surtout systématiser leur usage, y compris pour la définition du système Smalltalk lui-même.

Ainsi la couche réflexive qui permet une introspection et modification des objets représentant les programmes permet de définir rapidement des IDE et en particulier un excellent débuggeur. Pour faire court, tout le bestiaire de la programmation objet nous vient en ligne directe de Smalltalk. A ce titre on peut considérer qu'il a influencé quasiment tous les langages orientés objet qui ont suivi.

MACHINE VIRTUELLE

C'est aussi Smalltalk-80 qui va donner ses lettres de noblesse aux machines virtuelles. Là encore CSL a été totalement visionnaire. Alors même qu'à l'époque l'environnement Smalltalk-80 ne pouvait s'exprimer pleinement que sur l'ordinateur Alto de Xerox Parc (voir entrée "Ordinateur personnel"), Alan Kay voulait déjà un système totalement portable sur tout autre ordinateur personnel à venir dont l'imaginaire Dynabook (voir plus bas). La machine virtuelle de Smalltalk a donc été créée pour isoler le système Smalltalk du matériel sous-jacent et, ainsi, faciliter son portage sur d'autres plateformes. Mais alors qu'aujourd'hui la plupart des langages utilisent des ramasse-miettes pour gérer automatiquement la mémoire, son utilisation précoce dans Smalltalk a été longtemps décriée et a joué le rôle de frein idéologique. On peut considérer que toutes les VM qui ont suivi sont des descendants directs de la VM de Smalltalk et de ses nombreuses techniques d'optimisation inventées au fil du temps. La VM de Java descend directement de la VM Smalltalk (sa VM Hotspot a été écrite par des anciens de ParcPlace, première société à commercialiser Smalltalk-80). Puis viendra la VM Strongtalk toujours sur la base de la VM Smalltalk avec des optimisations dites de "type feedback" hérité du langage Self de Sun. Dans l'équipe Strongtalk [1] on trouve un certain Lars Bak qui, plus tard, créera chez Google la VM Javascript dite V8. De la même façon les VM de Python, Ruby et la VM Graal, et bien d'autres, reprennent les concepts et techniques de la VM Smalltalk. Pour paraphraser Stephen Hawking, toutes ces VMs s'appuient sur les épaules des géants.

INTERFACE UTILISATEUR

Smalltalk n'a pas été conçu comme un langage mais comme un système complet avec pour objectif de rendre la programmation accessible au plus grand nombre y compris les enfants (d'où le nom Smalltalk qu'on pourrait traduire en bavardage ou babillage). L'équipe CSL de Xerox Parc a donc inventé tout ce qui était nécessaire à leurs yeux pour atteindre cet objectif. Un des livres mythiques de Smalltalk-80 (le livre Orange : The Interactive Computing Environment) est d'ailleurs consacré entièrement à la conception d'interface utilisateur. On y trouve en un volume tout ce qui constitue encore aujourd'hui nos interfaces utilisateurs : écran bitmap (voir plus bas), souris à 3 boutons, fenêtres superposées, menu déroulant, boîtes de dialogue, barre de défilement, etc. Le tout parfaitement conçu dans une hiérarchie de classes Smalltalk-80. Steve Jobs a 'volé' l'idée du fenêtrage et de l'utilisation de la souris à cette même équipe.

MODÈLE MVC

Pattern de design universellement utilisé, le modèle MVC (Modèle - Vue - Contrôleur) est aussi né à Xerox Parc sous l'impulsion de Trygve Reenskaug, Jim Althoff et d'autres. Le pattern MVC est présent dans les librairies de Smalltalk-80 et Il est utilisé dans le livre Orange "The Interactive Computing Environment" déjà cité précédemment. Le modèle MVC est toujours très largement utilisé aujourd'hui et a même fait un come-back remarqué dans le monde du Web dans les années 2005-2006 quand Ruby on Rails est revenu mettre de l'ordre dans le développement d'applications Web. Depuis, il n'est pas un framework Web qui ne suive ce modèle MVC.

ECRAN BITMAP

Chuck Thacker, un des principaux inventeurs de l'ordinateur Xerox Alto dès 1972 (voir l'entrée "Ordinateur personnel"), a tout de suite compris la nécessité d'un écran bitmap pour les futurs ordinateurs personnels. Jusqu'alors tous les écrans ne savaient afficher que des matrices de caractères déjà présentes en ROM. Avec l'écran bitmap chaque point de l'écran devient adressable en mémoire. L'Alto possédait un écran monochrome (1 point = 1 bit 0 ou 1 en mémoire) d'un format de 606 par 808 pixels, à peu près le même ratio que le format d'une feuille Letter en mode portrait.

TABLETTE

Comme dit plus haut Smalltalk-80 a été conçu comme un système complet destiné à permettre l'accès à l'informatique et à la programmation au plus grand nombre. Cette idée avait germé dans l'esprit d'Alan Kay dès sa thèse à l'Université d'Utah de 1966 à 69. En 1970, nouvellement arrivé à Xerox Parc, il imagine le Dynabook. Souvent décrit comme la conceptualisation du premier ordinateur portable, c'est bel et bien la préfiguration d'une tablette. Là encore Alan Kay devra attendre plus de 15 ans pour voir arriver sur le marché des équipements capables de matérialiser les concepts du Dynabook.

SOURIS

Ce n'est pas Xerox Parc qui a inventé la souris mais Doug Englebart qui dépose un brevet en 1967 alors qu'il travaillait au Stanford Research Institute à Menlo Park dans la Silicon Valley. Mais c'est véritablement l'équipe CSL qui va faire de la souris à 3 boutons l'outil d'interaction universel avec les interfaces graphiques (voir entrée "Interface graphique"). Elle sera popularisée par l'ordinateur personnel Alto de Xerox (voir "Ordinateur personnel").

ORDINATEUR PERSONNEL

Ce qui est remarquable dans la folle décennie de recherche de Xerox Parc (en gros de 1972 à 1983), c'est une incroyable capacité à innover tant sur le plan logiciel que matériel. Les deux domaines vont véritablement se faire la courte échelle pendant 10 ans. Chuck Thacker, un ingénieur hardware génial, imagine l'ordinateur personnel Alto en 1972. A l'époque rien de tel n'existe, les mini-ordinateurs de la taille d'un frigo (américain le frigo !) règnent en maître. Dans la foulée il le dote d'un écran bitmap (voir entrée "Ecran bitmap"), d'une souris à 3 boutons, d'un disque dur amovible. L'Alto est né. Au départ Xerox Parc devait en construire (à la main !!) seulement une vingtaine d'exemplaires pour ses propres besoins de recherche mais ce sont finalement plus de 1000 unités qui seront construites tant la demande interne mais aussi externe sera forte. Il faut dire que l'Alto avait plus de 10 ans d'avance sur le marché. C'est seulement à partir de 1982 qu'on commencera à trouver des micro-ordinateurs à peu près équivalent sur le marché (le Lisa de Apple, la station Star de Xerox notamment).

WYSIWYG

What You See is What You Get, acronyme créé à Xerox Parc en 1974. Avec un ordinateur comme l'Alto, doté d'un écran bitmap au format Letter, quoi de plus naturel que de créer un traitement de texte dont la représentation à l'écran serait fidèle au rendu final. C'est ce que feront Butler Lampson et Charles Simonyi, deux chercheurs de Xerox Parc, inventeurs de l'éditeur de texte Bravo pour la station Alto, le tout écrit en Smalltalk bien entendu. Avec Bravo ils créeront aussi le premier gestionnaire de fontes. Leur travail incitera à la création et au développement des langages de description de page (voir entrée "Postscript") et des premières imprimantes laser (voir entrée "Imprimante Laser"). Charles Simonyi, lassé de l'incapacité de Xerox à transformer ses innovations en succès commerciaux, quittera Xerox Parc en 1980 pour rejoindre une jeune société baptisée Microsoft où il développera en 1981 MS Word et Excel (excusez du peu…).

COUPER-COPIER-COLLER

Travaillant aux côtés de Charles Simonyi dans l'équipe interface utilisateur de Xerox Parc, Larry Tesler, ingénieur informaticien considéré comme l'un des pères de l'interface utilisateur moderne et, plus tard, une des figures clé d'Apple durant les premières années de l'entreprise, est l'inventeur du "couper-copier-coller".

HYPERTEXTE

On a souvent coutume de dire que la quasi totalité des grandes inventions de l'informatique modernes sont sorties de Xerox Parc sauf le World Wide Web. Eh bien ce n'est pas totalement vrai. En effet, dans le traitement de texte Bravo dont on a parlé plus haut se trouvait déjà la notion d'hyperlien, une portion de texte soulignée à l'écran associée à une action déclenchée par un clic de souris. Au sens strict du terme c'est Doug Englebart qui a inventé la notion d'hypertexte (voir entrée "souris") mais son usage dans une application utilisateur est apparu avec le traitement de texte Bravo (voir entrée "WYSIWYG"). Certes la notion d'hyperlien ne constitue qu'une partie du WWW mais son antériorité méritait d'être soulignée.

ETHERNET

Ethernet a été développé à Xerox PARC entre 1973 et 1974. Inspiré de ALOHAnet, que Robert Metcalfe avait étudié durant son PhD. Metcalfe, arrivé à Xerox Parc en 1972, publie un mémo interne le 22 Mai 1973 décrivant "un médium physique omniprésent et complètement passif pour la propagation des ondes électromagnétiques" qu'il baptise Ethernet. En 1976, après qu'Ethernet ait été déployé et utilisé quotidiennement par des centaines de personnes à Xerox Parc, la version 10 Mbit/s du protocole sera mise sur le marché en 1980.

Robert Metcalfe, lassé lui aussi des valses-hésitations de Xerox sur la commercialisation de tous ces produits innovants, quitte Xerox en 1979 pour fonder 3Com avec le succès qu'on sait. 3Com sera racheté par HP en 2009 pour 2,7 milliards de dollars.

POSTSCRIPT

En parallèle des efforts de Charles Simonyi sur le premier traitement de texte WYSIWYG Bravo (voir entrée "WYSIWYG"), une autre équipe s'attaque au problème du rendu de la superbe Interface Utilisateur de Smalltalk sur le papier. Deux innovations majeures sortiront de cet effort : en 1978, John Gaffney et Martin Newell définissent le langage de description de pages (PDL) Interpress qui évoluera ensuite vers un autre PDL conçu lui aussi à Xerox Parc par John Warnock, Charles Geschke, j'ai nommé Postscript.

Là encore, Xerox ayant fait le choix de Interpress en interne plutôt que de Postscript, Warnock et Geschke quitteront Xerox Parc pour fonder Adobe Systems en 1982.

IMPRIMANTE LASER

L'ordinateur Alto de Xerox Parc était pourvu dès l'origine d'un superbe écran bitmap noir et blanc. Avec des logiciels comme Bravo (traitement de texte), Draw (pour le dessin), et d'autres développés avec Smalltalk-80, la question de l'impression de ces

documents s'est posée très tôt. C'est ce qui a suscité les travaux sur l'imprimante laser que vous pouvez voir d'une certaine façon comme la prolongation sur le papier d'un écran bitmap. C'est Gary Starkweather qui inventera au début des années 1970 la première imprimante laser à Xerox Parc, la Xerox 9700, en "greffant" une source laser sur un copieur de Xerox.

EXTREME PROGRAMMING

Sortons maintenant des années Xerox Parc pour nous transporter en 1996. Kent Beck invente l'extreme programming comme une méthode d'amélioration de la qualité des logiciels. Déployé pour la première fois chez Chrysler pour redesigner complètement son système de paie vieillissant. Smalltalk est le langage retenu par Kent Beck pour mettre en valeur toute la flexibilité de l'Extreme Programming.

REFACTORINGS

C'est aussi en Smalltalk que le premier support pour refactoring professionnel a été développé par John Brant et Don Roberts (alors que les premiers travaux sur les refactorings n'étaient que conceptuels et non disponibles sous forme d'outils). Dès 1995, le Refactoring Browser offrait refactorings, analyses de qualités (SmallLint), et outils de réécriture de code. John Brant proposait et propose toujours aussi un méta compilateur (SMaCC) permettant de la transformation de programmes entre langages (tel Delphi en C#, ...).

TDD

Kent Beck a aussi inventé et popularisé le développement piloté par les tests (TDD) au début des années 2000. Sa première librairie de test unitaires SUnit a été écrite pour Smalltalk qu'il utilisait alors énormément chez Chrysler. Tous les autres langages se sont mis au diapason dans les années qui ont suivi en développant leur version de SUnit (JUnit en Java, Test::Unit en Ruby, etc.)

Il y aurait encore beaucoup à dire sur les innovations du Xerox Parc des années 70 et début 80. Pour ceux qui veulent en savoir plus, je recommande vivement la lecture de l'ouvrage "Dealers of Lightning" [2], une enquête journalistique très fouillée sur cette période clé de l'histoire de l'informatique où la réalité dépasse de loin la fiction.

On critique aussi très souvent la société Xerox pour ne pas avoir su transformer les innovations de Xerox Parc en succès commerciaux. La critique est fondée mais ce n'est pas non plus le gâchis qu'on pourrait croire. A la lecture de cet article vous vous êtes sans doute aperçu(e) que Xerox Parc a essaimé un grand nombre de ses inventeurs géniaux dans des entreprises de la Silicon Valley et au-delà. En cela, on peut dire que Xerox a grandement aidé à l'essor de l'industrie informatique.

Reste encore à comprendre comment autant d'inventions ont pu voir le jour en un même lieu, sur une période aussi courte, et influencer aussi durablement le monde de l'informatique. De multiples explications ont été avancées pour expliquer cette créativité débridée. Pour ma part je vous en propose une à méditer lorsque vous viendra l'idée de vous lancer sur un projet innovant : assurez-vous que votre activité définit le pourquoi de son existence et non pas le comment. Le premier vous emmènera loin alors que le second est sujet à toutes sortes d'aléas (techniques, commerciaux, réglementaires, effets de mode,...). En définissant dès sa création en 1970 un pourquoi puissant "créer l'informatique personnelle et la rendre accessible à tous" Xerox Parc a ouvert la voie à toutes sortes d'inventions qui sont autant de manières d'y répondre. Et encore aujourd'hui de nouveaux produits comme les smartphones ou les objets communicants continuent à réinterpréter cette grande vision.

Références

[1] VM Strongtalk: http://strongtalk.org

[2] Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age de Michael A. Hiltzik, Avril 2000, Harper Business

Noury BOURAQADI
Noury est un développeur-maker, expert en génie logiciel et en systèmes multi-robots mobiles et autonomes. Impliqué dans différents projets open-source, il utilise Pharo professionnellement pour le développement d'applications réparties, web et mobiles. Actuellement, il occupe les fonctions d'enseignant-chercheur en informatique à l'IMT Lille Douai.

Luc FABRESSE
Luc souhaite pouvoir utiliser sa force Jedi pour contrôler des robots mobiles ;-) En attendant, il est professeur en informatique à l'IMT Lille Douai où il enseigne et concrétise ses travaux de recherche en Pharo (cf. http://car.imt-lille-douai.fr/luc).

Pharo : pureté, interactivité, réflexivité

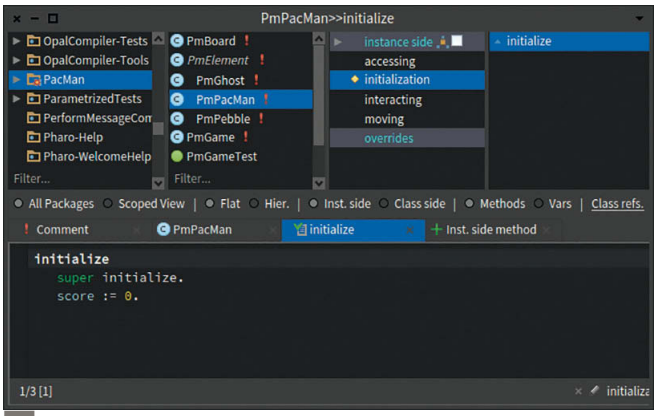
Faisons un tour d'horizon de Pharo. La syntaxe étant à quelques nuances près identique à celle de Smalltalk (voir l'article d'introduction à Smalltalk dans ce numéro), nous allons surtout nous concentrer sur son puissant environnement de développement immersif. Nous allons montrer comment il accélère le développement via une boucle de rétroaction immédiate avec le développeur qui s'appuie sur des outils extensibles d'introspection et de modification de code à chaud.

Avant de poursuivre nous vous conseillons d'installer Pharo sur Linux, Mac ou Windows en vous rendant sur pharo.org, rubrique Download/Téléchargement.

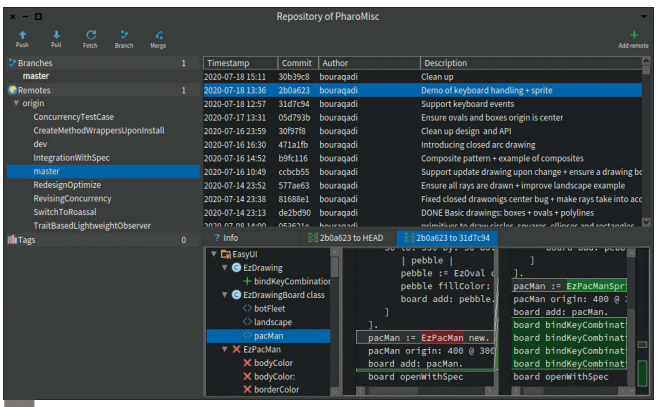
Pharo [1] est un descendant direct et libre de Smalltalk, lancé en 2008 sous licence MIT. Il allie stabilité et innovation, grâce à une communauté qui regroupe aussi bien des entreprises que des établissements académiques.

Oubliez les fichiers !

Si le concept de fichier est intéressant pour le stockage et l'échange de documents, il reste éloigné des concepts de la programmation par objets. En effet, lorsqu'on code, on raisonne en termes de paquetages (*packages*), de classes et de méthodes. Ce sont justement ces concepts qui sont mis en avant par l'outil ***System Browser*** de Pharo, illustré par la figure 1. Pour ouvrir le System Browser utiliser le menu Tools dans la barre de menu de l'environnement Pharo.



1 *L'outil Système Browser permet d'éditer les packages, les classes, les protocoles et les méthodes.*



2 *Vue d'un dépôt de code Git géré avec l'outil Iceberg.*

Le *Système Browser* permet de naviguer et d'éditer le code des programmes. Les quatre colonnes en haut permettent d'afficher respectivement de gauche à droite : les packages ou paquetages (groupes de classes), les classes, les protocoles (groupe de méthodes) et les méthodes. Le contenu d'une colonne dépend de ce qui est sélectionné dans la colonne à sa gauche. Ainsi sur la figure 1, on peut voir que le package *PacMan* (1ère colonne) contient plusieurs classes dont la classe *PmPacMan* (2ème colonne) actuellement sélectionnée. Le protocole *initialization* est sélectionné dans la 3ème colonne. La 4ème colonne n'affiche donc que les méthodes relatives à l'initialisation des instances de *PmPacMan*. Le cadre en bas du Browser est un éditeur. Il affiche et permet de modifier le code de l'élément actuellement sélectionné.

Ainsi, le *Système Browser* permet de naviguer dans le code, et d'opérer des ajouts et des modifications. Le développeur pense exclusivement en termes objets. Cela dit, pour l'échange et la diffusion du code, Pharo permet d'importer et d'exporter des fichiers de code. Un tel fichier peut contenir un paquetage, une classe ou même une seule méthode. Ce mécanisme est utilisé par l'outil de gestion de versions ***Iceberg*** (voir l'article sur Pharo Iceberg dans ce numéro). Il permet de sauvegarder les projets à l'aide du gestionnaire de version *git* en local et à distance, comme par exemple sur *GitHub* ou *Bitbucket*. Le code source de Pharo est ainsi publiquement disponible en ligne [2].

La figure 2 donne un exemple de dépôt *git* pour un projet Pharo. L'outil permet entre autres de visualiser les différents *commit* et leurs différences. Celles-ci sont exprimées en termes de paquetages, classes et méthodes.

2 fichiers pour les contrôler tous

En digne descendant de Smalltalk, Pharo s'appuie sur une machine virtuelle (VM, *Virtual Machine*) pour l'exécution des programmes. Cela signifie que le code source de Pharo est compilé sous la forme de code-octets (bytecodes) qui sont ensuite interprétés par la VM de façon analogue à la plupart des langages modernes comme Java, Python, Javascript ou encore Ruby (voir l'article sur la VM de Pharo dans ce numéro).

Une différence majeure subsiste entre l'exécution des programmes dans Pharo par rapport aux autres langages que nous venons de citer : il s'agit de la persistance de l'environnement de programmation et d'exécution. En effet, le développeur Pharo crée et initialise les objets qui constituent le programme. Puis, il sauvegarde le contenu de la mémoire de travail dans un fichier binaire dit *image*. A l'exécution, la machine virtuelle Pharo transfère le contenu du fichier *image* dans la mémoire RAM. On retrouve ainsi tous les objets sauvegardés, directement prêts à l'emploi. La majorité des opérations d'initialisation étant faites lors du développement, le chargement du programme se fait à la vitesse de l'éclair !

Ainsi, un programme Pharo se résume fondamentalement à 2 fichiers :

- La machine virtuelle, un fichier exécutable spécifique selon que vous êtes Linux, Mac ou Windows.
- Le fichier binaire *image* : appelé ainsi, car il représente une "photographie" du contenu de la mémoire. Ce fichier image peut aussi être transporté d'une machine virtuelle à une autre.

Boucle de rétroaction immédiate

Pharo est à la fois un langage de programmation à objets pur *et* un environnement de développement simple. Cette symbiose du langage de programmation avec son IDE(1) est extrêmement puissante. Elle permet aux développeurs de manipuler directement les objets et obtenir immédiatement un retour sur les conséquences des actions effectuées.

Par exemple, l'outil ***Playground*** (menu Tools) permet d'accéder à n'importe quelle variable globale, créer des instances de n'importe quelle classe, les stocker dans des variables locales et les manipuler de manière interactive. Cela est illustré par la figure 3, avec un script pour créer un dessin de PacMan interactif. Le script peut être exécuté dans sa totalité pour obtenir l'affichage représenté sur la Figure 3.

Mais, le plus intéressant est qu'on peut exécuter chaque expression séparément. Les variables locales au *Playground* - comme *board* ou *pacMan* - sont persistantes tant que le *Playground* est ouvert. On peut les utiliser pour envoyer des messages(2) aux objets qu'elles référencent, à des moments différents. Entre deux exécutions, le développeur peut utiliser d'autres outils comme le *System Browser* pour consulter ou modifier les définitions des classes utilisées.

Bien que d'apparence simple, le *Playground* offre plusieurs opérations dans ses menus. En plus d'exécuter l'expression choisie, on peut en afficher le résultat ou encore l'inspecter. Cette opération se traduit par l'ouverture d'un ***Inspector*** qui permet d'afficher et de manipuler la structure d'un l'objet. La figure 4 donne l'exemple d'un inspecteur sur une instance de la classe *EzPacManSprite*. Tous les champs (terme désignant une variable d'instance en Pharo) peuvent être lus ou modifiés. Il est également possible d'inspecter les objets référencés par les champs. La zone inférieure de l'inspecteur est un *mini-Playground* où la pseudo-variable *self* référence l'objet inspecté. Il est ainsi possible de lui envoyer des messages ou de le passer comme paramètre dans des messages destinés à d'autres objets.

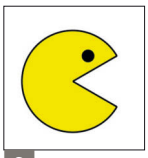
Développement interactif piloté par les tests

Une autre caractéristique de Pharo tient à sa capacité de remplacer du code *à chaud*, c'est-à-dire en cours d'exécution. Le développement d'applications peut ainsi être réalisé de manière incrémentale. Le développement débute avec un programme vide. On crée des classes sans champs ni méthodes, qu'on instancie pour obtenir les objets applicatifs. Les champs et les méthodes dont on a besoin sont ajoutés au fur et à mesure de l'introduction des fonctionnalités. Les objets sont automatiquement mis à jour par Pharo de manière à intégrer les changements sans jamais avoir à relancer un quelconque interpréteur ou à recompiler des fichiers. C'est un gain de temps considérable par rapport à d'autres langages y compris des langages dynamiques comme Python ou Ruby.

Cette interactivité se conjugue parfaitement avec la démarche *TDD (Test Driven Development)* de développement piloté par les tests [3], bien connue dans les méthodologies de gestion de projet agile. Chaque test est défini comme une méthode d'une sous-classe de ***TestCase***. Cette méthode vérifie le comportement attendu pour quelques objets applicatifs. Le test comporte des envois de messages à des objets impliqués dans une fonctionnalité. Puis, on vérifie que le résultat obtenu est bien celui escompté.

La figure 5 donne l'exemple d'un test d'incrémentation du score d'un jeu lorsque PacMan consomme une pac-gomme. Notez que le navigateur de code *System Browser*, reconnaît les classes et les méthodes de test en les dotant de boutons en forme de pastilles de couleur. Un clic sur la pastille d'une méthode de test exécute le test en question. Initialement grise, la pastille passe au vert si le test réussit et la fonctionnalité est remplie. Le jaune indique que la fonctionnalité n'est pas remplie et le rouge indique une exception.

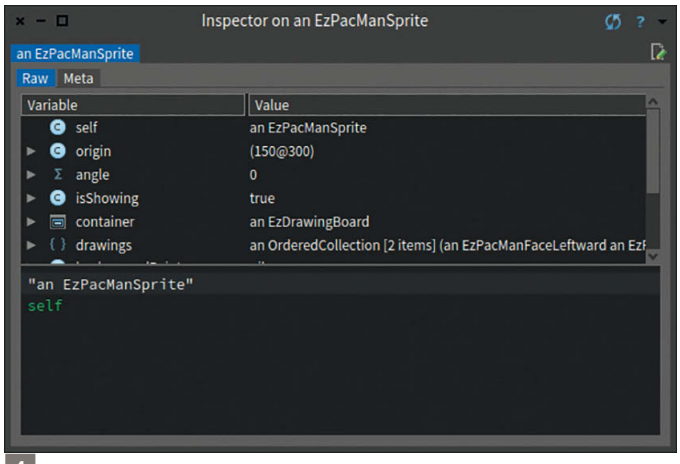
De la même manière, on peut lancer toutes les méthodes d'une classe de test, directement dans le ***System Browser***, en cliquant sur sa pastille. L'outil ***Test Runner*** permet de faire de même et davantage encore. Il est ainsi possible de lancer tous les tests d'un ensemble de classes de différents paquetages. Au-delà de la simple vérification de la conformité des tests, *Test Runner* permet d'analyser les performances des fonctionnalités testées. Il permet également de mesurer la *couverture* d'une batterie de tests. Il indique le pourcentage de code qui est exécuté pendant les tests. Un pourcentage élevé est un indicateur de la fiabilité du code. Là encore, vous noterez le niveau d'intégration atteint entre différents outils qui dans d'autres langages sont dispersés dans des multiples outils pas toujours simples à mettre en oeuvre et surtout qui interagissent peu (voir pas du tout) entre eux.



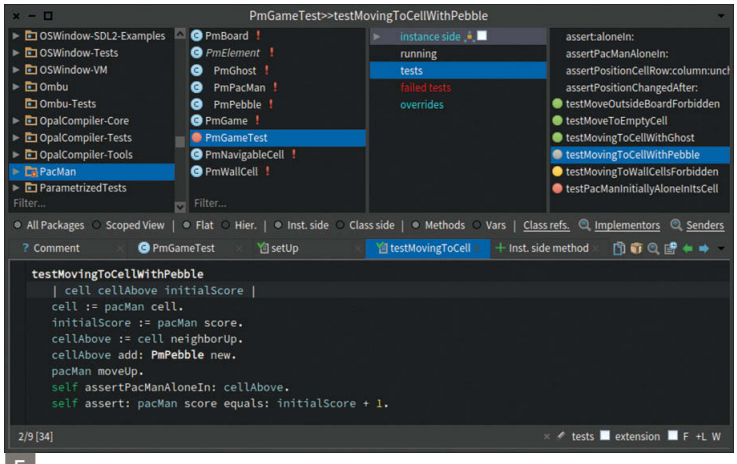
3 *Dessin produit par le script du Playground.*

(1) IDE acronyme anglais pour Environement de Développement Intégré.

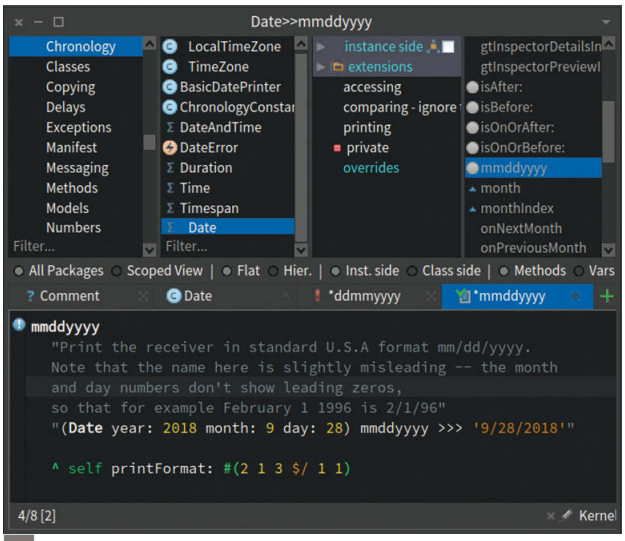
(2) La résolution d'un message se traduit par l'exécution d'une méthode



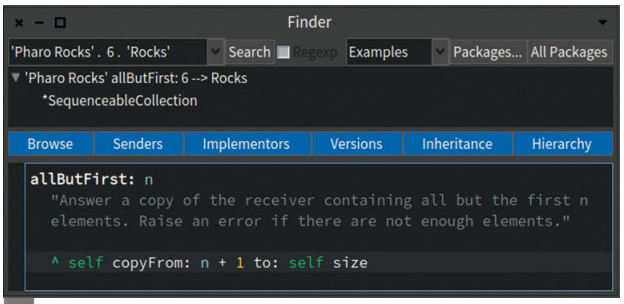
4 *L'inspecteur donne accès à la structure privée d'une instance.*



5 *Exemple de test d'un jeu PacMan.*



6 *Les commentaires peuvent contenir des exemples exécutables.*



7 *L'outil Finder permet de rechercher des méthodes à partir d'un exemple de code.*

Pharo : un environnement facile à découvrir

Outre les outils et leur intégration, Pharo aide également les développeurs à découvrir et naviguer dans les objets du système. Par exemple, les commentaires d'une classe ou d'une méthode peuvent contenir des exemples de code exécutables qui illustrent comment l'utiliser. La figure 6 montre le commentaire de la méthode *mmddyyyy* de la classe *Date* qui contient directement un exemple exécutable et le résultat attendu. L'outil **Senders** (accessible depuis le menu contextuel sur une méthode) montre le code de toutes les méthodes du système qui contiennent un envoi de message avec ce sélecteur. Cela permet de voir des exemples d'utilisation de cette méthode.

En termes de découverte, l'outil **Finder** (menu Tools) est certainement l'un des plus emblématiques. Qui n'a jamais rêvé d'un outil de navigation qui vous donnerait le nom de la méthode à partir d'un exemple de données en entrée et du résultat attendu en sortie ? La figure 7 montre justement cet outil en action dans Pharo. Le développeur cherche une méthode qui, étant donnés les arguments 'Pharo Rocks' et l'entier 6, aurait pour résultat la chaîne de caractères 'Rocks'. Sur cet exemple, **Finder** propose un seul résultat : la méthode *allButFirst:* implémentée dans la classe *SequenceableCollection*.

Tout se passe à l'exécution !

Lorsqu'un test, lancé seul, échoue, son exécution est suspendue. Le débogueur est alors automatiquement ouvert. Il permet d'analyser la pile d'appels depuis la méthode de test jusqu'à la méthode jusqu'à la méthode qui a provoqué l'erreur. Le milieu de la fenêtre donne accès à la méthode sélectionnée dans la pile d'appels. C'est un éditeur qui permet de corriger le *bug* et de poursuivre l'exécution sans avoir à relancer la totalité du test.

L'analyse du *bug* nécessite souvent d'inspecter les objets impliqués dans l'erreur. A cet effet, la partie inférieure du débogueur liste les différents objets accessibles par des champs, des variables, ou des paramètres. Le développeur peut les inspecter et les manipuler comme avec l'outil *Playground*. Il est en effet possible d'exécuter des expressions quelconques sur tout objet connu et ceci dans le contexte de la méthode déboguée.

Le débogueur Pharo est tellement puissant qu'il est possible de développer des pans entiers d'applications avec ce seul outil. Ce travail se fait alors "à chaud". Les modifications de code sont insérées directement dans la pile d'appels de méthodes momentanément suspendue. Le typage dynamique permet d'insérer des appels de méthodes qui ne sont pas encore réalisées. Elles sont ajoutées, à l'aide du bouton ***create*** du débogueur dans la classe appropriée. On peut également ajouter de nouveaux champs et même des classes, à l'aide du débogueur. Cette souplesse présente l'avantage de corriger le code en ayant directement accès au contexte qui a provoqué le *bug*. Le développeur constate directement les conséquences de ses changements en observant leur impact sur l'état des objets et le déroulement de l'exécution. Cette puissance découle directement des capacités réflexives de Pharo. Ce sujet dépasse largement le cadre de cet article, mais ayez présent à l'esprit que, 40 ans après la création de Smalltalk, c'est encore aujourd'hui une caractéristique unique. Nous nous contenterons ici de dire que tout est objet, y compris les éléments du langage lui-même. Les classes, les méthodes la pile d'exécution, le compilateur sont autant d'exemples d'objets de haut niveau. Ils sont notamment utilisés par les développeurs Pharo pour réaliser les outils et faire évoluer le langage lui-même sans jamais devoir modifier sa syntaxe primaire (Article d'Introduction à Smalltalk dans ce numéro).

L'introduction de l'héritage multiple avec la notion de ***Trait*** [4] est un exemple d'une telle extension.

Conclusion

Nous avons fait un tour d'horizon rapide des caractéristiques de Pharo et de son IDE. Vous trouverez dans la suite de ce numéro des articles qui abordent plus en détail certains éléments de ce langage à objets pur, de son infrastructure et de ses capacités réflexives. D'autres articles présenteront des bibliothèques Pharo pour différents domaines applicatifs et à leur tête, le développement d'applications web. Pour ceux qui souhaitent approfondir davantage et vraiment toucher du doigt (ou du clavier ;-) la puissance de Pharo, nous recommandons de commencer par ces 3 supports :

- Le cours en ligne libre : MOOC Pharo [5]
- La collection de livres sur Pharo avec téléchargement gratuit de PDF [6]
- La documentation en ligne de Pharo avec le Wiki pour accompagner les débutants et une sélection de projets qui reflètent la richesse et la versatilité de Pharo [7]

Références

[1] Site officiel de Pharo https://pharo.org/
[2] Code source de Pharo et autres projets associés https://github.com/pharo-project/
[3] Présentation du développement piloté par les tests, par l'*Agile Alliance* https://www.agilealliance.org/glossary/tdd/
[4] Le concept de Trait dans Pharo https://github.com/pharo-open-documentation/pharo-wiki/blob/master/General/Traits.md
[5] MOOC Pharo http://mooc.pharo.org/
[6] Livres Pharo à télécharger gratuitement http://books.pharo.org/
[7] Documentation en ligne de Pharo https://github.com/pharo-open-documentation

Pablo Tesone
Ingénieur pour le Consortium Pharo il se spécialise dans les aspects bas niveau tels que la génération d'assembleur, les communications entre processus et bien sûr la machine virtuelle. Avant cela Pablo a travaillé pendant plus de 10 ans dans les projets industriels tels que GemPlus. Il a récemment obtenu un doctorat sur la mise à jour logicielle dynamique appliquée aux environnements de programmation Live, aux systèmes distribués et aux applications robotiques. Il est un passionné de programmation orientée objet et de leurs outils.

La machine Virtuelle de Pharo

Dans cet article, nous présentons un bref aperçu de la machine virtuelle (VM) utilisée par Pharo. Cette VM présente des capacités intéressantes qui met en évidence la puissance de Smalltalk. De plus, c'est une excellente VM aussi bien pour faire de la recherche qu'à utiliser en production grâce à son histoire longue et riche et à son avenir très prometteur.

Qu'est-ce qu'une machine virtuelle ?

Pharo s'exécute au-dessus d'une machine virtuelle : la VM Pharo. Dès sa création dans les années 70 et 80 Smalltalk a été un pionnier de cette approche et de nombreuses versions de VMs ont enrichi l'histoire du langage et essaimé ensuite sur bien d'autres langages dont Java. La VM Pharo est une évolution de l'open-Smalltalk VM.
Une machine virtuelle fournit un environnement d'exécution commun pour différents systèmes d'exploitation et plateformes. Une VM abstrait et encapsule les différences qui existent entre les différentes machines, systèmes d'exploitation et plateformes. Cette fonction permet au même programme Pharo de s'exécuter sur différentes machines et systèmes d'exploitation.
Lorsqu'une méthode Pharo est compilée, elle est compilée en bytecode puis de ce bytecode vers l'assembleur de la machine cible lors de l'exécution. Le bytecode est une représentation binaire d'un programme. Nous pouvons le voir comme un code machine spécifique à une VM. Ce bytecode encapsule toutes les opérations du langage de manière compacte, la plupart des opérations sont encodées dans un seul octet d'où l'origine du nom bytecode. De plus, le bytecode prend en compte la présence d'objets et de leurs instances. Chaque instruction Pharo est codifiée en un ou plusieurs bytecodes. Par exemple, nous avons des opérations pour accéder à une variable d'instance, envoyer un message et accéder à des variables temporelles. Le bytecode produit s'exécute ensuite sur n'importe quelle machine virtuelle Pharo sans qu'il soit nécessaire de le modifier pour une plateforme donnée : le même programme s'exécute sur une machine Windows, Mac, Linux ou Raspberry PI.
En outre, la VM encapsule toutes les opérations de bas niveau, permettant aux programmeurs de se concentrer sur des concepts de niveau supérieur produisant de bons logiciels avec moins d'effort. Par exemple, la VM gère toutes les communications réseau en présentant une seule façon de les utiliser. Elle gère les différences d'utilisation des opérations réseau dans différents systèmes d'exploitation.

1 *Les niveaux d'interaction de la VM Pharo.*

Pharo utilise sa propre VM car cela nous permet de contrôler à la fois l'exécution et aussi l'évolution du langage. De plus, il est important de conserver une machine virtuelle complètement open-source et utilisable dans des applications commerciales, des projets open-source ou dans le milieu universitaire. Cette indépendance permet de définir des capacités spécifiques à Pharo telles que le support efficace pour become (qui échange des pointeurs d'objets de façon atomique) ou la définition d'éphémerons (structures de données élémentaires pour la définition des structures de données faibles (weak data structures) c'est à dire qui ne sont pas prises en compte par le ramasse miettes (garbage collector).
La machine virtuelle de Pharo est développée en Pharo lui-même, C et Assembleur. Le code Pharo de la machine virtuelle est aussi traduit en C. A l'exécution, le bytecode (ou autre représentation intermédiaire) est transformée en assembleur auto modifiant avec des tas de caches et optimisations. 1

Survol de la VM Pharo

La machine virtuelle est constituée de différents composants qui collaborent pour exécuter un programme. Tous les composants ont une forte interdépendance entre eux, et il est parfois difficile d'établir une limite claire entre eux. Mais nous pouvons les identifier par leurs responsabilités respectives :

L'interprète de bytecode.

L'interpréteur de bytecode est un composant clé. Il lit le bytecode d'une méthode et il exécute la tâche correspondante. C'est une machine à pile, donc toutes les opérations passent par une pile de valeurs. Par exemple, si le bytecode est «Read Instance Variable 0», il lira la première variable d'instance du récepteur du message et la poussera dans la pile. Les opérations suivantes prendront une telle valeur et effectueront d'autres opérations. Un autre exemple est le bytecode «Send Message» qui envoie un message spécifique à un objet. Le récepteur du nouveau message est pris de la pile avec tous les arguments de ce message (ils sont également pris sur la pile). L'interpréteur est un point clé de la VM, il «exécute» les opérations successives qui, ensemble, font notre programme.

Représentation Mémoire

Certaines entreprises qui utilisent Pharo gèrent plus de 30 millions de lignes de code Pharo. La représentation mémoire des objets est donc très importante puisqu'en Pharo tout est objet, les classes comme les méthodes. L'interpréteur manipule nos objets, leur envoie des messages, en crée de nouveaux et les stocke en mémoire. Un autre composant important de la machine virtuelle est la façon dont ces instances sont représentées dans la mémoire. Ce composant est vraiment important car un bon modèle de mémoire permet à l'interpréteur de s'exécuter plus rapidement et d'utiliser moins de mémoire. Mais ces deux facettes sont toujours antagonistes. Ainsi, le point clé d'une bonne représentation est qu'elle doit équilibrer la vitesse d'accès aux objets et l'espace mémoire occupé. La représentation mémoire inclut la façon dont les objets sont structurés, s'ils ont une en-tête, comment les variables d'instances sont stockées, etc. Dans Pharo, chaque objet doit connaître sa classe. Une solution possible est donc d'ajouter un pointeur vers la classe dans l'en-tête de l'objet. Cependant, dans les architectures modernes, cela nécessiterait 64 bits (ou 8 octets) par objet. C'est beaucoup de surcharge par objet, donc le modèle de mémoire Pharo utilise 22 bits pour encoder un

identifiant de classe. Cette décision minimise l'utilisation de la mémoire mais elle nécessite une recherche dans une table de classe, c'est un exemple clair de compromis permanent entre vitesse d'accès à l'information recherchée et l'espace mémoire utilisée pour son stockage.

Le ramasse-miette

Pharo est un langage qui a une gestion automatique de la mémoire. Le ramasse-miette doit enregistrer la mémoire utilisée par un programme (et le reste de l'environnement Pharo), à quel moment celle-ci peut être libérée et rendue au système d'exploitation ou quand allouer davantage de mémoire. Dans les systèmes d'exploitation modernes, un programme ne fonctionne pas seul, il doit interagir avec d'autres programmes s'exécutant en même temps. Il doit également être un bon «citoyen» et n'utiliser que les ressources nécessaires. Tout le monde a vu des programmes qui prennent plus de ressources que nécessaire, et qui finissent par planter le système d'exploitation. Ainsi, le garbage collector exécute périodiquement la vérification des objets accessibles et utilisés par le programme. Si un objet est inutilisé, il libérera cet objet permettant à d'autres objets d'être instanciés à sa place. Une autre responsabilité importante est de conserver une liste d'espace libre, de sorte que la VM sache à tout moment et de façon rapide quelle mémoire est (ré)utilisable et éviter ainsi d'en demander plus au système d'exploitation. Enfin, lors de l'exécution, la mémoire se fragmente (trop de petits morceaux rendent impossible d'allouer des gros). Ainsi, le ramasse-miettes est aussi en charge de compacter la mémoire. Pharo possède plusieurs algorithmes de ramasse-miettes à l'état de l'art : un garbage collector générationnel incrémental et un mark and sweep. Dans le futur nous allons introduire un mark and sweep trois couleurs encore plus efficace.

Primitives

Avoir des programmes du monde réel exige d'interagir avec des librairies en dehors de notre environnement objets fort sympathique. Par exemple, faire une requête réseau, ou appeler le système d'exploitation pour ouvrir une fenêtre, ou lire l'entrée de l'utilisateur dans la console. Toutes ces opérations sont implémentées en tant que primitives de la VM. Fondamentalement, une primitive est une routine qui communique avec le monde extérieur lors de son exécution. Dans Pharo, une primitive est essentiellement une méthode qui n'est pas implémentée en Smalltalk dans Pharo lui-même mais au niveau VM. Lorsqu'un message est envoyé et que la méthode correspondante est activée, la primitive est alors exécutée. Par exemple, lorsqu'un fichier est ouvert, une primitive est exécutée. Cette primitive appelle le système d'exploitation pour ouvrir le fichier et stocke le descripteur de fichier qui identifie le fichier ouvert. Chaque opération de base sur les fichiers nécessite de parler avec le système d'exploitation. Toutes ces opérations sont implémentées en tant que primitives de la machine virtuelle.

Optimisations

Enfin, la VM comprend un autre composant important, un ensemble énorme de techniques d'optimisation développées et enrichies au fil de quatre décennies d'existence de Smalltalk. Il est possible d'implémenter une VM sans ces techniques d'optimisation mais le résultat sera lent. Une machine virtuelle de production met en œuvre un nombre conséquent d'optimisations. Par exemple, une optimisation simple est celle qui est utilisée dans la recherche de méthodes. Lorsqu'un message est envoyé à un objet, la méthode à exécuter est recherchée dans toute la hiérarchie des classes de cet objet. Ainsi, un simple cache de méthodes améliore énormément la vitesse d'exécution. L'un des exemples les plus importants d'optimisation est l'utilisation de la compilation Just-in-Time. Cette technique transforme le bytecode d'une méthode en sa représentation de code machine. Une fois que la méthode est compilée en code machine, elle peut être exécutée directement par le processeur de la machine, plus rapidement que l'exécution sur l'interpréteur. D'autres optimisations peuvent également être effectuées dans cette transformation. Comme cette transformation prend du temps et dépend de la machine, nous ne voulons la faire que dans les méthodes qui sont exécutées le plus souvent. Comme nous l'avons vu précédemment, dans la VM, tout est question d'équilibre.

Un peu d'histoire et une vision de l'avenir

La Pharo VM est le produit d'une évolution continue qui s'étale sur une longue période de temps. Elle comprend des éléments qui ont été introduits dans les versions originales de Smalltalk et qui ont évolué depuis. Elle inclut des techniques modernes telles que la compilation Just In Time et des ramasse-miettes automatique avancés. LA VM Pharo a débuté par une branche de la machine virtuelle Squeak et elle inclut le développement effectué par la communauté OpenSmalltalk-VM. Ce projet est actif depuis 1995 et comprend les contributions de nombreux développeurs différents. Nous avons bifurqué de ce projet non pas pour des raisons techniques mais pour des raisons philosophiques. Notre branche est guidée par un souci de démocratisation de l'accès à la VM : avoir une vraie VM ouverte et productive pour tous et modifiable par n'importe qui. Nous sommes fiers du passé, mais ouverts à un avenir encore meilleur.

L'un des principaux avantages de la VM Pharo est qu'elle est écrite en Pharo. Elle peut être exécutée comme n'importe quel autre programme Pharo. Cela nous permet d'écrire des tests, de simuler l'exécution de la VM, de la modulariser, d'utiliser tous les outils Pharo existants et de la documenter comme n'importe quel autre programme. La VM est un très gros programme avec beaucoup de décisions de conception complexes et nécessite beaucoup de connaissances spécifiques, mais notre objectif est de l'ouvrir pour qu'il soit plus accessible à tous les utilisateurs de Pharo. Pour simplifier, nous voulons éliminer toute la complexité accidentelle afin de pouvoir remettre en question ce qui est vraiment important.

Pour atteindre cet objectif, nous avons lancé un processus pour extraire et publier toutes les connaissances qui ont été inscrites dans la VM au cours de sa longue vie. Nous écrivons des tests, de la documentation et nettoyons l'ancien code. Les tests facilitent la modification de la VM en réduisant le risque et en l'ouvrant à des modifications.

Nous avons amélioré le processus de construction et appliqué des techniques utilisées dans toutes les grandes applications logicielles. Nous sommes convaincus que la VM doit être ouverte à la communauté et elle doit permettre à la communauté et à Pharo de continuer à évoluer et à couvrir ses besoins. L'avenir ne se cantonne pas à de la documentation. Nous proposons une véritable version open-source pour ARM 64 bits ; versions mises à jour pour Windows, Linux et Mac. Nous voulons explorer de nouvelles manières de produire le code assembleur. Et de nouvelles façons d'interagir avec le système d'exploitation (FFI) et d'autres bibliothèques. Toute notre feuille de route est ouverte et elle est disponible sur le projet Github de la VM.

Ressources :

- Pharo VM : https://github.com/pharo-project/pharo-vm
- Two Decades of Smalltalk VM Development: Live VM development through Simulation Tools, Eliot Miranda, Clément Bera, Elisa Gonzalez Boix, Dan Ingalls
- Sista : a metacircular architecture for runtime optimization persistence, Clément Bera, These Université de Lille
- Open Documentation Effort : https://github.com/SquareBracketAssociates/Booklet-PharoVirtualMachine

Guillermo Polito
Guillermo Polito (Dr.) est ingénieur de recherche CNRS, affecté au laboratoire CRIStAL de l'Université de Lille. Il travaille en proche relation avec l'équipe de recherche RMoD. Depuis 2008, Guillermo travaille comme ingénieur en génie logiciel, et depuis 2012, il fait des recherches sur la modularité et les langages de programmation. Depuis 2010 il participe activement au projet open-source Pharo, et depuis 2018, il fait partie du comité qui le dirige.

Iceberg : une intégration Git pour Pharo

Depuis deux ans, Pharo offre une intégration très proche du gestionnaire de versions Git, qui a remplacé Monticello, le gestionnaire de versions que Pharo utilisait depuis 2008. Non seulement les projets écrits en Pharo peuvent être stockés sur des forges et dépôts Git, mais le projet Pharo lui-même est géré avec un dépôt Git. Cet article présente le module d'interfaçage à Git nommé Iceberg : sa raison d'être, ce qu'il offre et ses avantages par rapport à l'utilisation de Git depuis la ligne de commande.

Intégrer Git et Pharo ne consiste pas uniquement à sauver le code Smalltalk dans un dépôt Git. Iceberg gère en effet une double copie de travail (répertoire de travail) car le code Smalltalk de Pharo existe à la fois sous forme d'un code source mais aussi en tant qu'objets vivants dans la mémoire de Pharo. Iceberg offre le meilleur des deux mondes : de la programmation avec des objets vivants, un des concepts fondamentaux de Smalltalk, et une sauvegarde textuelle transparente comme le font classiquement les IDE du marché.

Un peu d'histoire

Depuis ses débuts, Pharo utilisait un gestionnaire de code nommé Monticello. C'est un système de gestion de versions distribué dont l'esprit est très proche de Git. Développer se fait sur un dépôt local sous forme de commits locaux, et puis ces commits sont envoyés à un dépôt distant. Le dépôt local n'est qu'une copie du dépôt distant. Ce type de systèmes distribués permet de travailler hors-ligne et de disposer de plusieurs dépôts distants simultanément.

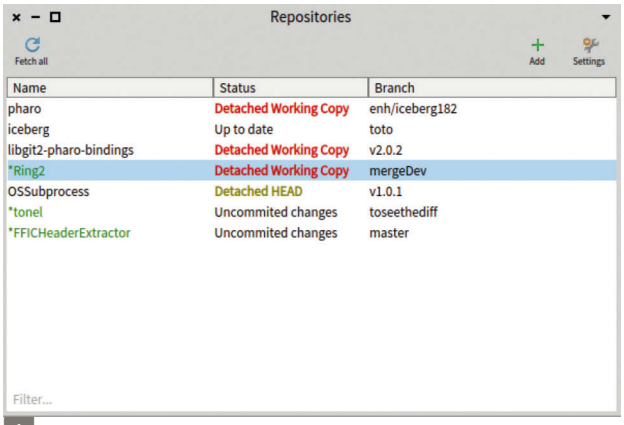
Le système Monticello est taillé sur mesure pour des systèmes Smalltalk. Pour l'utiliser, on a besoin d'une librairie cliente et d'un serveur ou forge Monticello. La librairie cliente est normalement fournie avec l'environnement Smalltalk utilisé comme c'est le cas pour Pharo. Le serveur Monticello gère le stockage des dépôts distants en plus d'autres responsabilités comme la communication ou l'authentification d'utilisateurs. Les forges Monticello, depuis leur début, ont été maintenus par la communauté de développeurs.

Pourquoi Git ?

Monticello a servi très correctement les besoins de la communauté Smalltalk depuis sa création en 2004, malgré quelques faiblesses avérées comme une gestion de branches peu ad-hoc et sous-dimensionnée. Pourquoi donc passer à Git ? Il y a plusieurs raisons très pratiques comme, par exemple, améliorer la maintenabilité du système, baisser la barrière d'entrée aux nouveaux utilisateurs, et augmenter la visibilité du langage et de ses librairies.

En premier, Monticello nécessite une librairie cliente et une forge Monticello, et que celles-ci doivent être maintenues par la communauté. Lorsqu'on la compare à d'autres communautés comme celles de Python ou Java, la communauté Pharo est une communauté très active mais petite. Maintenir des serveurs est une tâche qui requiert beaucoup d'attention et d'effort de la part des développeurs : les serveurs, les polices de sécurité, site web et autres services (recherches projets, wikis…) doivent être mis à jour pour satisfaire les nouveaux standards. Utiliser Git permet à la communauté Pharo d'utiliser des forges standard et déjà maintenues.

En second, Git comme gestionnaire des versions est devenu presque un standard de-facto. Une bonne partie de l'industrie du logiciel utilise aujourd'hui Git. Les universités enseignent Git. Choisir Git comme gestionnaire de versions pour Pharo permet donc de simplifier l'accès aux nouveaux utilisateurs. Jusqu'à présent un développeur désireux d'apprendre Pharo devait apprendre un nouveau langage, un nouvel environnement de développement (IDE) et en plus un nouveau système de gestion de versions. Désormais il ou elle peut se concentrer sur son code, et en plus se retrouver avec son système de gestion de versions préféré : Git. Finalement, une bonne quantité de développeurs open-source travaillent aujourd'hui avec des forges Git comme GitHub ou GitLab. Avoir une intégration avec Git permet aux développeurs Pharo d'utiliser ces forges et de bénéficier directement de leur visibilité mondiale.



1 *La liste de dépôts et leur état.*

Un tour d'Iceberg

Iceberg offre toutes les opérations basiques d'un dépôt Git :

- créer un dépôt (git init) ;
- cloner un dépot existant (git clone) ;
- créer et checkouter des branches (git checkout) ;
- committer des changements dans une branche (git commit) ;
- merger deux branches (git merge) ;
- observer les changements du dépôt.

Iceberg, comme un gestionnaire de dépôts Git, montre dans son écran principal la liste de dépôts chargés et leur états : y-a-t-il des changements à committer ou publier (push) ? Sur quel commit ou branche se trouve un projet ? 1

Depuis cet écran on peut créer un dépôt ou cloner un dépôt existant. Le dialogue de création de dépôt fournit des accès simplifiés pour des forges connues comme GitHub.com ou Gitlab.com, mais elle permet ultérieurement de cloner un dépôt à partir de n'importe quel url http ou ssh

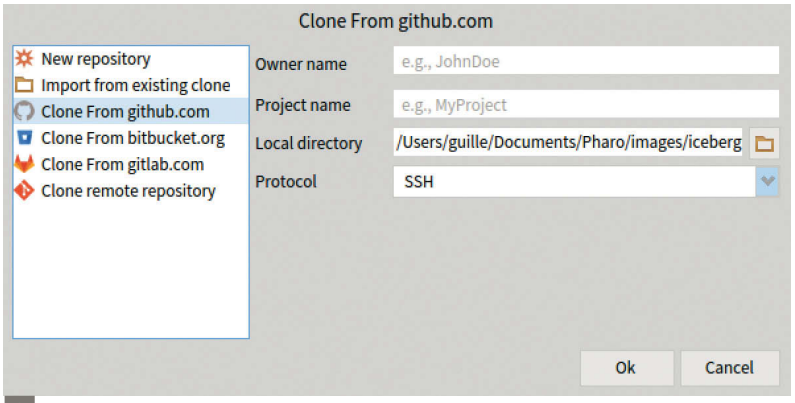
pointant sur un dépôt Git. 2
Si on rentre dans un dépôt, on voit la liste des packages qu'il définit, et on a des options pour regarder leur code, les charger dans notre image de développement, ou de voir l'historique. 3
À partir de la vue du dépôt, on a accès aussi aux options de création de branches et merge, pull et push. L'interface utilisateur d'Iceberg a été conçue pour montrer à l'utilisateur les effets avant qu'ils se produisent. Pour cela, toutes les opérations destructives de Git sont d'abord montrées avec une preview. Par exemple, le bouton pull montre les commits à puller et leur code, avant d'effectuer le pull. Ceci permet aux utilisateurs de s'assurer que les opérations qu'ils font sont les bonnes. Le même concept est utilisé pour les écrans de commit, push, checkout, et merge. 4

L'architecture d'Iceberg

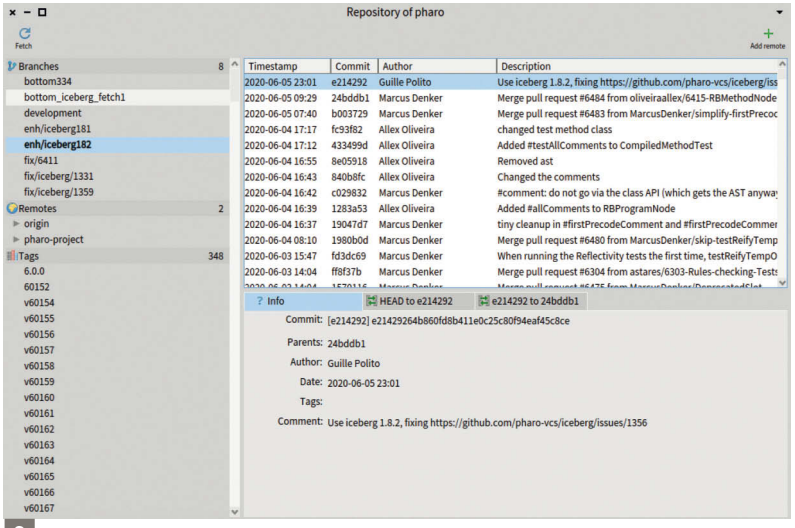
L'architecture d'Iceberg se fixe comme objectif de donner aux utilisateurs un accès à Git tout en gérant, de la façon la plus transparent possible, l'articulation entre le monde des fichiers source et leur représentation objet constamment présente dans les images Smalltalk. Avant de rentrer dans les détails de l'architecture d'Iceberg, rappelons brièvement la structure d'un dépôt Git.

La structure d'un dépôt Git

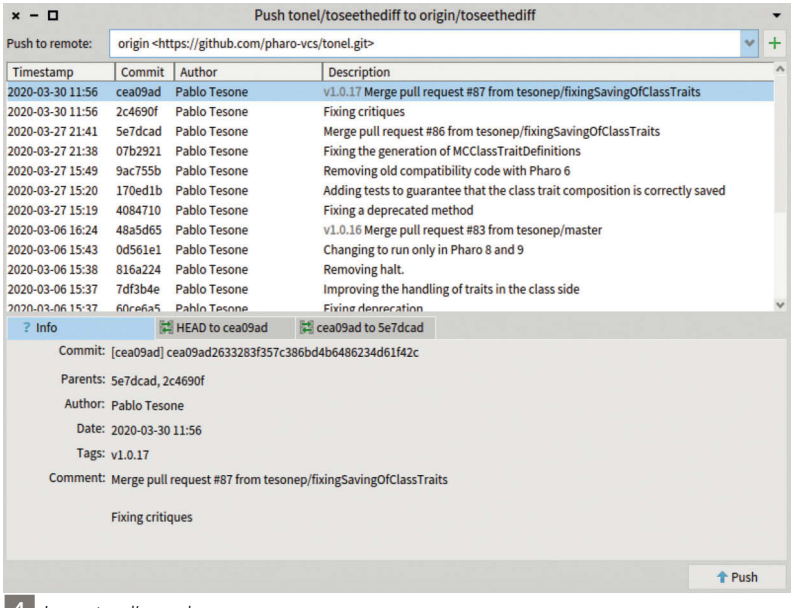
Le dépôt est formé du dépôt proprement dit et d'un répertoire de travail (working directory). Le dépôt proprement dit fonctionne comme une base de données où l'on stocke tous les commits avec leur contenu. Le répertoire de travail contient les fichiers sur lesquels on travaille à un instant donné. Lorsque nous modifions un fichier de notre répertoire de travail, Git détecte ce changement par rapport au commit actuel et il marque ce fichier comme étant modifié. La commande git commit prend les changements dans notre répertoire de travail et les applique dans notre dépôt, en créant un nouveau commit et en mettant à jour la branche actuelle. Push envoie les commits de notre dépôt local vers le dépôt distant. Fetch télécharge les commits du dépôt distant dans notre dépôt local. Merge mixe deux commits dans notre dépôt local puis applique ces changements dans notre répertoire de travail. Pull fait un fetch puis un merge. 5



2 *Le dialogue de création, clone et import de dépôt.*



3 *La vue de l'histoire du dépôt*



4 *Le preview d'un push.*

Le double répertoire de travail d'Iceberg

L'intégration Git dans Pharo doit s'insérer dans ce schéma avec deux différences : en premier, les éditions de code en Pharo ne se font pas dans le répertoire de travail mais dans l'image ; en second, une image Pharo est stateful (un espace mémoire dans lequel vivent les objets qui représentent les packages, classes,…) et donc contient le code dans sa dernière sauvegarde. Ces deux différences impactent l'architecture d'Iceberg de deux façons. Iceberg est donc conçu pour gérer deux répertoires de travail différents : le répertoire de travail pour Git, et le répertoire de travail représentant le code dans l'image Pharo. 6

Pour donner un exemple du fonctionnement de cet double répertoire de travail, quand un développeur Pharo travaille, il modifie le répertoire de travail dans l'image Pharo. Au moment de committer, il doit donc mettre à jour le répertoire de travail Git puis effectuer le commit Git proprement dit. Cela, Iceberg le fait de manière transparente pour le développeur. Aussi, si ce développeur ouvre une image utilisée il y a plusieurs jours et que son dépôt Git d'origine a été modifié entre temps, Iceberg se rappellera du commit et du code chargé originellement et proposera au développeur, par exemple, de charger le nouveau code.

Ces deux répertoires de travail peuvent ne pas être synchronisés. Iceberg doit donc s'assurer que les deux répertoires de travail sont synchronisés quand une opération Git le requiert. Pour minimiser l'impact de cette décision, les dépôts Iceberg détectent quand ils sont désynchronisés et proposent aux développeurs des options de réparation automatique.

Une installation simplifiée

Pour simplifier l'installation de Pharo et Git, on a décidé de ne pas demander par défaut aux développeurs de faire une installation Git, ce qui peut s'avérer assez fastidieux pour les développeurs Windows par exemple.

Pour pallier cette difficulté, Iceberg utilise la librairie libgit, qui implémente les fonctionnalités de base Git, et qui est écrite en C de manière portable. Iceberg utilise libgit via la bibliothèque UnifiedFFI.

UnifiedFFI permet d'interfacer Pharo avec des bibliothèques externes. Le FFI permet d'appeler des bibliothèques externes qui ont une convention standard d'appel, comme le code C, avec une liaison dynamique.

Algorithmes de diff et merge sur mesure

Un point à remarquer sur la librairie client d'Iceberg est qu'elle permet de faire des merge et diff de code Pharo avec une sémantique Pharo. Qu'entendons nous par là ? Git travaille au niveau des lignes de texte d'un fichier source. C'est à dire, que quand on committe un changement, on committe les lignes de texte modifiées. De la même façon les conflits lors de fusion de branches (merge) sont gérés au niveau des lignes de code. Les développeurs Pharo sont habitués à travailler sur des éléments plus proches de la sémantique du langage : des packages, des classes, des protocoles, des méthodes... Donc Iceberg fournit des algorithmes de diff et merge plus proches du langage Pharo que du modèle de ligne de Git.

L'exemple suivant montre qu'un diff entre deux commits met en lumière les méthodes modifiées et leurs classes et non pas uniquement une à une les lignes de code impactées. 7

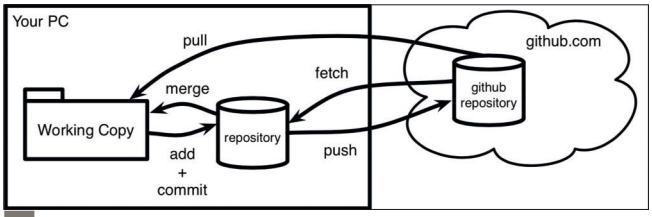
Pour le code, Iceberg le manipule au format Tonel. Tonel est un nouveau format pour l'encodage de code Pharo. Il est une amélioration par rapport au format "chunk" de Smalltalk car il permet de mieux lire le corps des méthodes.

Conclusion

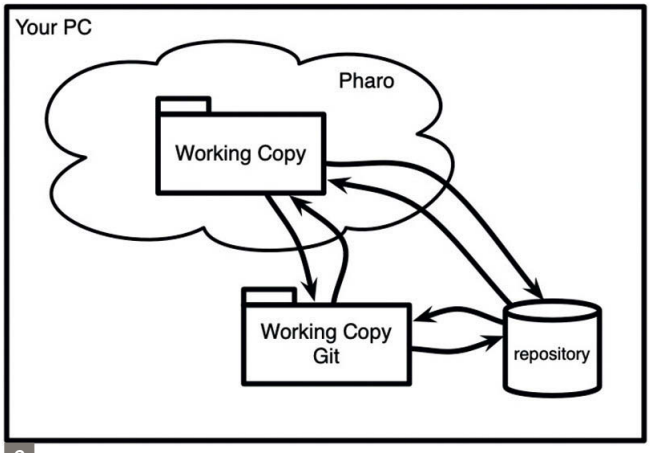
Cet article donne une vue d'ensemble d'Iceberg : l'outil qui intègre Git avec Pharo. Cet outil baisse la barrière d'entrée des nouveaux utilisateurs car il supporte entre autres les opérations plus courantes de Git. Mais Iceberg réconcilie le modèle de développement fichier avec celui centré objet promu par un environnement totalement objet tel que Pharo. En plus, Iceberg étend le modèle de Git avec un merge plus sémantique pour donner aux développeurs Pharo une vue plus proche de leurs habitudes.

Références

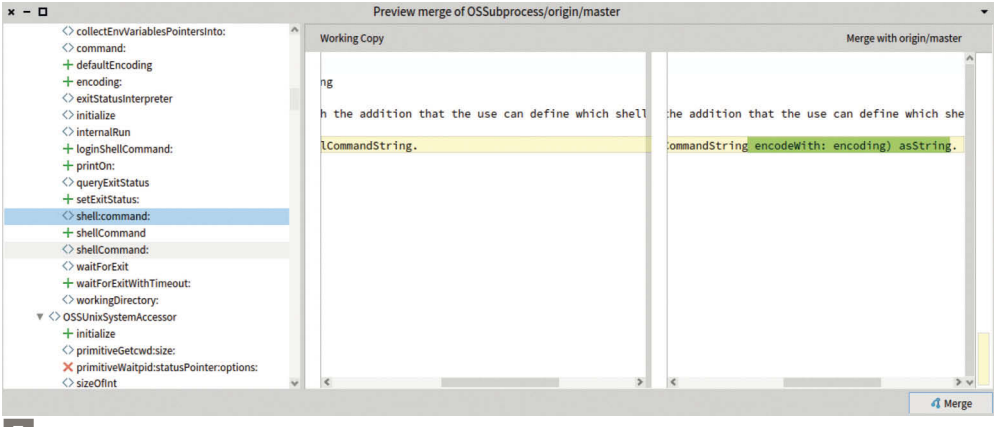
https://github.com/pharo-vcs/iceberg
https://github.com/pharo-vcs/libgit2-pharo-bindings
https://libgit2.org/
http://books.pharo.org/booklet-uffi/
https://pharo.org/



5 *Vue simplifiée d'un dépôt Git*



6 *Dans Pharo il y a un deuxième répertoire de travail.*



7 *Vue d'un merge par méthodes et des classes.*

Carolina Hernández Phillips
Étudiante en PhD IMT Lille Douai, France. Membre de l'équipe RMoD depuis 2018. Civil Ingénieur en Computer Science, Université du Chili.

MetaL : un framework pour bootstrapper de petites images Pharo

Si vous avez lu l'article de présentation de l'environnement Pharo vous savez déjà qu'un environnement d'exécution Pharo (et Smalltalk en général) se compose d'une machine virtuelle capable d'exécuter du bytecode, et d'un fichier image qui comporte tous les objets vivants exploités par la machine virtuelle. Comme en Smalltalk, absolument tout est objet, toutes les classes et méthodes de Pharo (y compris celles de vos programmes) sont dans cette image.

Pour certains projets, soit pour des raisons de sécurité ou d'empreinte mémoire limitée, il est intéressant de générer des images réduites comportant uniquement les classes et méthodes dont vous avez besoin. Le framework MetaL permet précisément de générer des images Pharo (aussi appelées noyaux) réduites qui peuvent descendre jusqu'à 40 Koctets tout en restant totalement compatibles avec la machine virtuelle Pharo. De quoi travailler sur des plateformes matérielles très réduites comme vous l'imaginez. Aujourd'hui, chaque image Pharo est générée en compilant le code source de l'image via une application dont le processus de génération s'appuie sur la technique dite de bootstrap [1]. L'application en question est spécifique à chaque noyau généré. Pour générer un noyau différent, l'application doit être modifiée, une tâche sujette à de nombreuses erreurs en particulier pour les développeurs qui manquent de pratique. MetaL est justement là pour simplifier ce processus de création en matérialisant toutes les exigences de la machine virtuelle, en fournissant des tests automatisés et un des méta-modèles du langage pour manipuler la définition du noyau, et ce que vous voulez y mettre.

Le bootstrap qu'est-ce que c'est ?

Bootstrapper signifie compiler un langage en utilisant un compilateur écrit dans ce même langage. Dans le cas de Pharo cela veut dire que l'image d'une nouvelle version de Pharo est générée par une application Pharo (appelée PharoBuilder) qui s'exécute dans une version antérieure et totalement fonctionnelle de Pharo. Le bootstrap doit mettre en place les objets qui représentent le futur noyau et son exécution. Il manipule ainsi un morceau de mémoire qui correspondra à la nouvelle image créée.
Le script qui suit illustre comment Pharo 9 est bootstrappé à partir de Pharo 8, et comment l'image ainsi générée est chargée par la machine virtuelle Pharo :

```
$ ./vm80/pharo Pharo8.image eval "PharoBuilder bootstrap: '/path/to/pharo9/sourcecode'"
Generated image saved as Pharo9.image
$ ./vm90/pharo Pharo9.image
```

Bootstrapper Pharo implique qu'une partie des instructions exécutées par PharoBuilder pour générer l'image sont définies dans le code source de l'image elle-même. Concrètement, dans l'exemple précédent, pour générer Pharo9.image, le programme PharoBuilder exécute à la fois le code de Pharo 8 et de Pharo 9.

Pourquoi bootstrapper est-il compliqué ?

Il y a deux raisons à cela. La première tient au fait, comme on l'a dit plus haut, que l'application de bootstrap est spécifique à chaque noyau et qu'elle est difficile à modifier. Ainsi pour Pharo, l'application PharoBuilder est littéralement différente pour chaque nouvelle version de Pharo. Générer une nouvelle image nécessite donc de modifier le code source de l'image et celui de PharoBuilder tout en préservant la compatibilité entre eux. Une chose délicate à faire.
La seconde raison vient de ce que la machine virtuelle impose implicitement des restrictions sur le noyau. En effet un noyau est un snapshot mémoire sauvegardé dans un fichier binaire qui contient tous les objets du système. La VM n'a qu'une connaissance limitée du programme qu'elle exécute. Par contre, elle impose des contraintes en dictant le format et l'emplacement des objets en mémoire. La VM restreint aussi les relations entre objets (par exemple chaque objet est forcément l'instance d'une classe).
Ainsi, la structure et le contenu d'un noyau sont partiellement dictés par la VM. Un noyau qui ne satisferait pas ces règles amène invariablement au plantage de la VM avant ou pendant la boucle d'interprétation du bytecode (et parfois très tard dans l'exécution). Pour trouver la cause du crash il faut tout d'abord debugger le code de la VM pour identifier l'objet manquant ou mal structuré à l'origine du problème, puis débugger le processus de bootstrap du kernel pour comprendre pourquoi cet objet a été mal, ou pas du tout, généré. Bref, le processus de débogage peut s'avérer fastidieux et requiert une très bonne connaissance des mécanismes internes de la VM.

MetaL à la rescousse

MetaL est un framework qui permet de définir et de générer toute une gamme de noyaux compatibles avec la VM de Pharo. Grâce à MetaL le développeur définit le nouveau noyau en créant le méta-modèle du langage et en l'utilisant pour construire son propre modèle. Le modèle sert ensuite d'entrée au processus de bootstrap du noyau. Avant de voir comment utiliser MetaL, introduisons 3 de ses concepts clés :
Bootstrapper: c'est le processus de génération de MetaL. Une application Pharo prend un modèle de langage en entrée et produit le fichier du noyau (Figure 1). Cette application est agnostique vis à vis du modèle de langage et peut donc générer différents noyaux. Le noyau est construit en mémoire dans un objet Pharo de la classe ByteArray puis sauvegardé sur disque.

Modèle de langage: un ensemble d'objets Pharo (appelé model-objects) définissant chacun un élément du langage : classes, méthodes, globals, variables et le langage lui-même (voir Figure 1).
Méta-modèle de langage: (voir Ring [2]) un ensemble de classes Pharo dont les instances sont des model-objects (voir Figure 2). Ces classes sont appelées classes de meta-modèles. Elles définissent des méthodes (hooks) contenant des instructions spécifiques pour l'installation d'un modèle d'objets dans le noyau. Les hooks doivent être définis par l'utilisateur.

Le flot MetaL

Une fois connus ces concepts de base, nous présentons maintenant le flot suivi par MetaL pour générer un nouveau noyau :
Construire la définition du noyau :

- Définir le méta-modèle du langage en sous-classant les classes du méta-modèle souhaitées pour le noyau.
- Utiliser les classes du méta-modèle pour instancier et construire le modèle de langage lui même.

Exécuter l'application *bootstrapper* de MetaL et à chaque fois que le bootstrapper stoppe son exécution en demandant un hook, fournir son implémentation. Vous trouverez un exemple concret du flot emprunté par MetaL dans les sections qui suivent.

MetaL par l'exemple

Pour illustrer l'utilisation de MetaL nous allons utiliser Pharo Candle. Pharo Candle (ou Candle pour faire court) est un mini Pharo qui contient seulement 51 classes (une image Pharo standard en comprend plus de 9000) et dont le noyau n'occupe que 179 Ko en mémoire. Candle retient les mêmes syntaxe et sémantique que Pharo à l'exception des Traits et des Slots.
Pour distinguer clairement le code de Candle de celui de Pharo, nous utiliserons le préfixe "PC" dans les noms de classe de Candle (comme PCArray, PCObject, etc.). Ici on voit que la classe Array de Candle n'est pas la même que celle de Pharo. C'est la même chose pour les autres classes qui constituent un noyau comme Candle. Le comportement de ces classes est spécifié par la définition de code équivalent. Comme nous souhaitons que Candle soit similaire à Pharo du point de vue du langage, MLanguage est la seule classe du méta-modèle à sous-classer. En créant cette sous classe que nous appelons MCandle nous tirons à nous toutes les caractéristiques du langage Pharo.

```
MLanguage subclass: #MCandle
```

Ensuite nous instancions la classe MCandle pour créer un modèle vierge :

```
candle := MCandle new
```

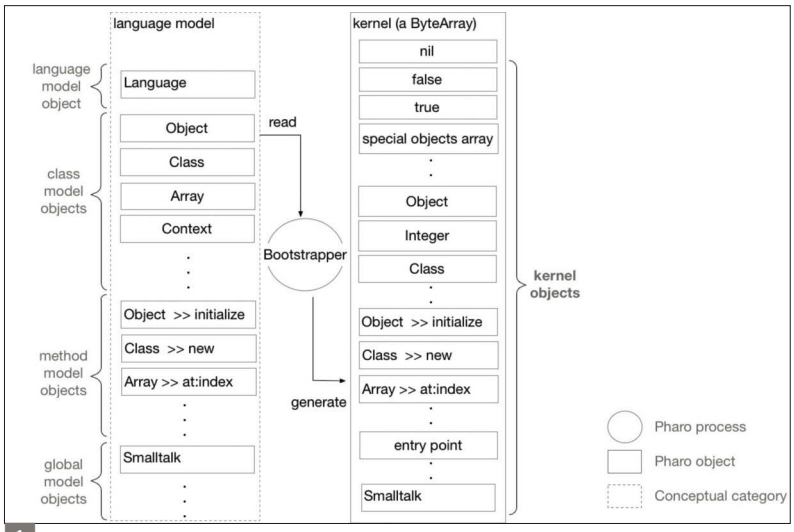
Puis nous y ajoutons des objets-modèles qui vont représenter les classes, méthodes et variables de Candle au modèle de langage. Cela peut être fait de 2 façons.
Soit en important des objets-modèles depuis un dépôt. Le code source de Pharo Candle [3] contient la définition de son modèle de langage, et vous pouvez le télécharger et importer son contenu directement de la façon suivante :

```
candle loadTonelRepository: '/path/to/repository/' asFileReference
```

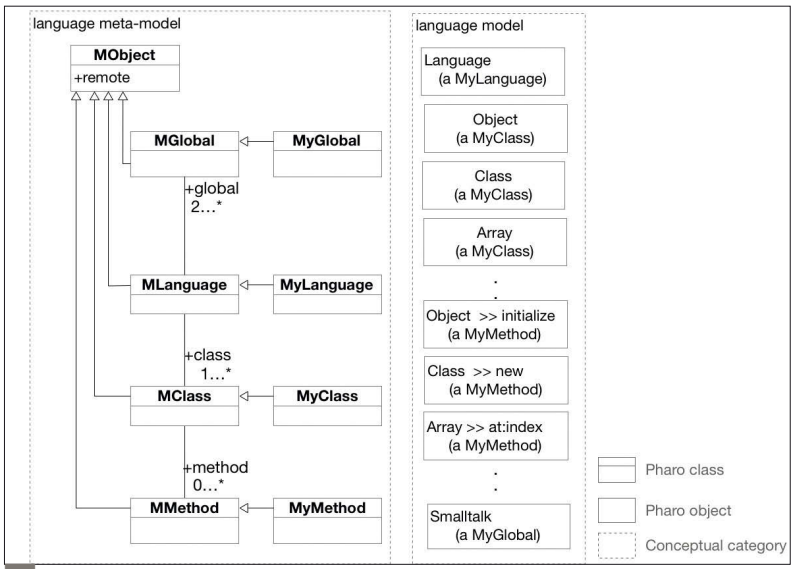
Soit vous définissez les objets-modèles un par un en utilisant directement l'API du méta-modèle ou bien l'éditeur graphique de MetaL disponible dans l'environnement Pharo. Dans le premier cas vous construisez les objets-modèles, puis vous les reliez entre eux. Et pour finir vous les ajoutez au langage. Dans l'exemple qui suit on crée la classe racine PCobject et on définit sa méthode '=' qui teste l'égalité entre deux instances de PCObject.

```
newClass := MClass named: #PCObject.
method := MMethod named: #= parent: newClass.
method sourceCode: '= anObject ^ self == anObject '.
newClass addLocalMethod: method.
candle addBehavior: newClass
```

En passant par l'éditeur graphique de MetaL, ces mêmes opérations se font encore plus simplement. On instancie un nouveau méta-modèle Candle comme vu précédemment puis on ouvre l'éditeur en exécutant candle browse. On peut alors définir nos classes et méthodes du modèle de langage comme nous le ferions pour du code Pharo mais nos définitions n'affectent que le périmètre de notre modèle de langage. Cet éditeur dédié permet aussi le refacto-



1 *Le bootstrappeur lit un modèle de langage et génère un noyau. Les objets du noyau sont générés en suivant les spécifications du modèle de langage.*



2 *Les objets dans le modèle de langage sont des instances des classes du meta model du langage.*

ring et la navigation intelligente dans le code comme le System Browser standard de Pharo.
Une fois notre modèle de langage défini, MetaL est suffisamment sophistiqué pour nous proposer des tests afin de vérifier l'existence et le format des objets-modèles tels qu'attendus par la machine virtuelle. En exécutant les tests de Candle, on teste ainsi le modèle de langage lui-même et tout son contenu. Tous les tests doivent être passés avec succès avant de passer à la phase de bootstrap. Dans le cas contraire, le processus de bootstrap provoque une exception pour cause de modèle de langage incohérent.

Générer le noyau

Une fois tous les tests passés avec succès, lancez le processus de bootstrap en exécutant :

```
candle bootstrap
```

Le noyau est alors alloué en mémoire puis on y installe les bouchons (stubs) des classes et instances spéciales. A ce stade les bouchons en question sont des versions temporaires et encore incomplètes des objets du futur noyau. A chaque installation dans le noyau d'un nouveau bouchon de classe, une référence vers ce bouchon est conservée dans la variable d'instance remote de l'objet-modèle contenant la définition de cette classe. Une fois les bouchons installés, on passe à l'exécution des hooks définis par l'utilisateur. Les hooks les plus importants dans Candle sont les suivants :

- **Définition des rôles des classes et variables globales** : les noms des classes et variables globales requises doivent être explicitement définis dans un dictionnaire (hash) Pharo comme montré ci-dessous. Ainsi par exemple, on dit au builder que la racine d'heritage est PCObject, ou que la classe pour les tableaux est la classe PCArray.

```
MCandle >> classRoles
  ^ {#ProtoObject -> #PCObject.
#Metaclass -> #PCMetaclass.
#Array -> #PCArray.
…
} asDictionary

MCandle >> globalVariableRoles
  ^ {#Smalltalk -> #SmalltalkCandle.
…
} asDictionary
```

Il n'est pas nécessaire d'activer toutes les classes dans un modèle de langage mais toutes celles que vous souhaitez utiliser doivent être impérativement définies de cette façon.

- **Initialiser les variables globales** : le concepteur du langage doit aussi initialiser les valeurs des variables globales et des variables de classe. Dans l'exemple ci-dessous le hook de Candle indique comment initialiser la variable globale SmalltalkCandle.

```
   MCandle >> initializeGlobals
| code |
code := 'SmalltalkCandle := PCCandleImage new'.
^ (self newInstruction code: code) evaluate
```

Notez que ce qui est stocké dans la variable code ci-dessus n'est pas du code Pharo mais bien du code Pharo Candle dont l'exécution va contribuer à la génération du nouveau noyau. Pour exécuter ce code, un objet Pharo spécial est créé en envoyant le message newInstruction à self qui représente ici le modèle de langage candle instancié au début. L'envoi du message evaluate à cet objet, provoque l'exécution du code stocké dans la variable code.

- **Installer les classes** : les bouchons de classe temporaires sont remplacés par leur version définitive. Les méthodes ne sont pas encore installées mais un dictionnaire (hash) vide est créé pour les accueillir dans le hook suivant. Pour implémenter ce hook nous mélangeons l'exécution de code Pharo et de code Pharo Candle.

```
MCandle >> installClass: aMClass
   | methodDict |
methodDict := (self newInstruction
  code: 'PCMethodDictionary new') evaluate.
aMClass remote
      methodDict: methodDict;
      superclass: aMClass superclass remote;
      format: aMClass format
```

L'argument aMClass de la méthode installClass: est un objet-modèle. Pour générer le dictionnaire de méthode aMClass nous évaluons le code Candle et sauvons le résultat dans la variable temporaire methodDict. Nous obtenons la référence à la super classe de aMClass dans le noyau en envoyant le message remote à l'objet-modèle qui représente la super classe de aMClass. Nous récupérons le format mémoire de aMClass directement de aMClass. Enfin aMClass remote est mis à jour avec les nouvelles valeurs.

- **Installer les méthodes** : à l'idée du compilateur Pharo le développeur compile les méthodes et les installe ensuite dans le noyau.

Tester et inspecter le noyau

Pour préserver la compatibilité avec la VM, MetaL teste le noyau après l'exécution de chaque hook. Si le noyau est incohérent, le processus s'interrompt et indique le dernier hook exécuté.
A noter que dès que les bouchons (stubs) sont créés (et dans toute la suite du processus de création), il est possible d'inspecter le noyau pour vérifier chacun des objets qui s'y trouvent. Pour ce faire il suffit d'évaluer l'expression :

```
(EP2SpurImage onFirstObject: candle remote nilObject) inspect.
```

Conclusion

Cet article est un survol rapide des capacités de MetaL. Grâce à ce framework vous pouvez générer des systèmes Pharo personnalisés au niveau du langage et des objets présents dans l'image. Pour en savoir plus sur MetaL nous vous invitons à consulter les documents mentionnés dans liens.

Références

[1] Pharo repository
https://github.com/pharo-project/pharo#bootstrapping-pharo-from-sources
[2] Ring: a Unifying Meta-Model and Infrastructure for Smalltalk Source Code Analysis Tools https://rmod.inria.fr/archives/papers/Uqui11a-RingJournalPaper-CSSJournal.pdf
[3] Pharo Candle repository https://github.com/carolahp/PharoCandleSrc
[4] Pharo Bootstrapper repository https://github.com/carolahp/PharoBootstrapGeneric

François Tonic
Programmez!

Low-code
va-t-il rendre le développeur inutile ?

Low-code, no-code, ce sont des outils qui font de plus en plus parler d'eux. On les oppose souvent aux développeurs. A tort ou à raison. Il y a beaucoup de fantasmes autour de ces environnements. Et aussi, beaucoup de questions. Mais il faut arrêter de vouloir opposer low-code – no-code et surtout d'opposer ces outils aux développeurs.

A en croire des études, rapports, analyses, avant, pendant ou après le Covid-19, le low code (et éventuellement le no code) va bouleverser le marché. Selon Gartner / Forrester, et cité par Simplicité Software), on aurait :

- 50 % des développeurs envisagent d'utiliser du low-code ;
- 75 % d'ici 2024 des entreprises utiliseront 4 plateformes low code pour des développements ;
- 2/3 des applications développeurs d'ici 2024 seraient des apps low-code ;
- 53 milliards : c'est le poids du marché low-code, d'ici 2024.

On croit, ou non, les analystes. Tout comme, on croit ou non au boost du low-code durant le confinement et le Covid. Est-ce une tendance lourde, donc dans la durée, ou de l'opportunisme ? Comme toujours, il y a un peu des deux.

Tendance ou simple opportunité ?

Le confinement, la crise Covid ont bouleversé de nombreuses entreprises et tout naturellement les services informatique, les équipes techniques. La désorganisation, une DSI moins réactive, contribuent à favoriser les utilisateurs avancées à créer leurs propres services / applications. De nombreuses usages métiers / marketing n'ont pas besoin des outils et plateformes de développement traditionnels. C'est là que du low-code peut être pertinent.

La question n'est pas de réinventer la programmation mais plutôt de répondre rapidement à des besoins immédiats. C'est une des motivations du shadow IT. Le low-code convient bien à des assemblages de services / données, automatisation de tâches. Le facteur temps est important, surtout pour projets / applications qui ne nécessitent pas de programmation objet ou d'instancier des ressources d'infrastructures nécessitant des autorisations et des procédures à suivre, pouvant prendre plusieurs semaines.

Sans doute que le Covid va conforter la place du low-code dans le paysage technologique.

Pour quels usages ?

Pas facile de répondre à cette question, car le low-code peut servir à tout et n'importe quoi. Tout va dépendre des capacités de la plateformes choisie. En réalité, les usages sont très nombreux : applications métiers, pour créer des dashboard, des outils de suivis, de gestion(supply chain, marketing, etc.). On peut aussi rencontrer ces outils dans l'automatisation de flux, de traitements.

Le low-code ne signifie pas : 0 (zéro) code. Il y a toujours de la glue à y mettre mais ces outils sont là pour la minimiser. Ces outils sont là pour « mâcher » le travail, réduire au maximum le code via du paramètrage et de multiples connecteurs.

Une des idées est de donner la possibilité à des non-développeurs, avec tout de même une certaine maîtrise de l'outil technologique, de concevoir des applications rapidement. Voire, de proposer aux équipes IT d'aller

Low code : quelques points à regarder

Roald Dupuis, informaticien, met en oeuvre du no-code. Son expérience l'a amené à soulever plusieurs points sensibles qu'il faut avoir en tête, surtout en contexte de production, déploiement :

- Stabilité de l'app générée : les ressources externes au low-code / no-code que l'on va utiliser peuvent changer ou disparaître. Cela aura forcément un impact. Quand la solution utilisée change de version, même chose : quel impact pour mes projets ?
- Evaluer la qualité de solution utilisée ;
- Pérennité de la solution utilisée : arrêt de la solution, rachat, etc. toujours à avoir l'esprit cette réalité ;
- Avoir conscience qu'une panne réseau, une dégradation du service aura un impact important sur l'application créée. Quel risque de perdre des données en cas de panne ?

Si le service ne fonctionne plus ou n'est plus disponible pour ma version quel plan de secours (ou ne pas oublier de la créer) ? En combien de temps puis-je basculer vers la solution de secours (éventuellement en mode dégradé) ? Il faut faire une analyse des risques (= impact business).

Boutique Programmez!

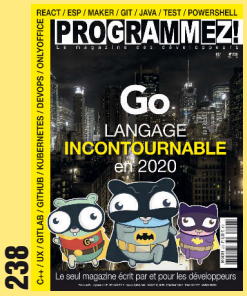
Les anciens numéros de PROGRAMMEZ!

Le magazine des développeurs

226

231

236



238

239

240

241

HS 01 été 2020

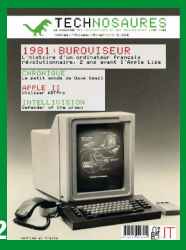
242

Tarif unitaire 6,5 € (frais postaux inclus)

Technosaures

Le magazine à remonter le temps !

N°1



N°2

N°3

N°4 Standard 10 €

N°4 Deluxe 15 €

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

Histoire de la micro-informatique 1973-2007

12,99 € (frais postaux inclus)

☐ 226	: __ ex	☐ 240	: __ ex
☐ 236	: __ ex	☐ 241	: __ ex
☐ 238	: __ ex	☐ HS 01	: __ ex
☐ 239	: __ ex	☐ 242	: __ ex

soit __ exemplaires x 6,50 € = ____ €

Technosaures ☐ N°1 ☐ N°2 ☐ N°3
soit __ exemplaires x 7,66 € = ____ €
☐ N°4 Deluxe ..... 15 €
☐ N°4 Standard ..... 10 €
☐ Histoire de la Micro-informatique ..... 12,99 €

soit au **TOTAL** = ____ €

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

☐ M. ☐ Mme ☐ Mlle Entreprise : ____ Fonction : ____

Prénom : ____ Nom : ____

Adresse : ____

Code postal : ____ Ville : ____

Règlement par chèque à l'ordre de Programmez ! I Disponible sur www.programmez.com

Offres pouvant s'arrêter à tout moment, sans préavis PROG 243

beaucoup plus vite dans la livraison. On parle de quelques semaines et non de plusieurs mois. Encore une fois, tous les besoins / projets ne nécessitent pas de sortir les IDE, les frameworks X, Y, Z.

Un marché dynamique !

Le marché est dynamique depuis plusieurs années. Plusieurs poids lourds dominent le marché tels que Appian, Pega System, Outsystem. Les grands fournisseurs tels que Salesforce, Microsoft, IBM n'hésitent pas à investir ce marché. Quelques éditeurs français ont su se faire une place, notamment Convertigo et Simplicité Software.

Côté Google, l'éditeur a annoncé la fermeture d'App Maker. Depuis avril, il n'est plus possible de créer des apps, et le 19 janvier 2021 marque la fin définitive du service. Google met en avant AppSheet. Ce dernier a été racheté par Google début 2020. Et depuis, l'outil fait partie de l'offre Google Cloud.

Plusieurs questions à se poser

Utiliser un outil low-code oblige aussi à se poser quelques questions, parfois sensibles :

- réversibilité : un point particulièrement sensible. Que se passe-t-il quand je veux changer de plateforme ou quand j'arrête d'utiliser/de payer la plateforme courante ? Qu'est-ce que je récupère ? Qu'est-ce qui est utilisable ?
- Documentation technique : quelle documentation technique est générée ? Sous quelle forme ?
- Qualité du code généré et finalement à qui appartient le code ?
- Coût de la plateforme : coûts réels, coûts cachés ;
- Support, réactivité sur les bugs.

Soyez attentives/tifs à ces éléments, car on oublie trop souvent le coup d'après. C'est le même débat que nous avions eu avec certains services Cloud il y a quelques années.

La notion de no-code

Il faut distinguer le low-code et no-code. Le no-code signifie : 0 code à écrire. Ces outils sont destinés à tout le monde et pas uniquement aux développeurs et utilisateurs avancés. « Démocratisation de la technologie numérique » explique Erwan Kezzar et Alexis Kovalenko (contournement.io). Un des premiers outils fut Microsoft FrontPage et Geocities.

Un outil no-code va permettre de paramétrer, d'assembler, de fédérer des fonctionnalités, des données, des process. Cela peut aider des personnes et des entreprises à construire un site web ou une app rapidement sans avoir à se former durant plusieurs semaines ni à payer une agence web ou un développeur. Comme pour le low code, il s'agit d'aller vite et de ne pas dépendre des équipes techniques. Les usages sont très larges et vont dépendre de votre capacité à maîtriser l'outil et selon les capacités de l'outil. Aujourd'hui, les outils no-code sont particulièrement intéressants pour les chatbots, créer des API, les app mobiles.

Pour les fondateurs, le no-code est là pour dépasser certaines contraintes (apprendre le code, manipuler des outils lourds, investissement non négligeable, etc.). Le no-code permet de rapidement éprouver vos idées, de réaliser soi-même des sites et apps, d'avoir une certaine indépendance vis-à-vis de prestataires ou des équipes techniques. Le no-code peut aussi permettre de recentrer le travail de développement sur des projets, des apps apportant de la valeur, se concentrer sur le code critique et ne pas s'occuper du codage sans valeur. 'Le low-code est parfois compliqué et on est (souvent) enfermé par la plateforme choisie. Est-ce opposé aux développeurs ? Ces outils peuvent aider le développeur à aller plus vite : moins de tâches, moins de code à écrire (le fameux pisseur de code, NDLR).' poursuit Erwan.

À qui s'adresse le no-code ? Cela peut être un particulier, une entreprise, des équipes d'une entreprise, etc. Et il y a un autre élément. Parfois, une entreprise n'a pas les moyens de se payer une agence, une ESN ou un développeur pour concevoir tel projet. Le no-code devient alors une alternative.

Erwan résume ainsi l'approche no-code et son intérêt :
1. démocratiser le numérique et amorcer un projet, une idée ;
2. simplifier l'investissement, miser sur l'automatisation ;
3. permettre de mieux comprendre la programmation, le rôle du code ;
4. revaloriser le développeur, les équipes techniques tout en se déchargeant des tâches courantes ou des codes à faible valeur.

Le marché low-code est important avec des acteurs installés. Le no-code apparaît, en France, beaucoup plus discret, même si l'intérêt est bel et bien là. Il y a peu de freelances no-code, mais cela bouge. Ainsi, Erwan travaille avec Octo autour du no-code/low-code même si la démarche reste timide.

Bref, le développeur ne va pas disparaître à cause du no-code. Cette approche peut par contre initier des projets, des idées plus difficiles, ou impossibles, à prototyper sans investissement.

Convertigo : la plateforme LowCode - NoCode mise sur l'ouverture

Convertigo est un des rares acteurs français du low-code ayant une dimension internationale. La particularité est de promouvoir le low-code open source et de s'appuyer sur des standards du marché développeur pour réduire au maximum l'adhérence technique. Convertigo est aussi un acteur du NoCode, sur un marché en pleine croissance. Nous avons posé quelques questions à Olivier Picciotto.

Convertigo est un acteur français du low-code. Comment se comporte le marché français ? Quelle est la demande ? Etre un éditeur français est-il un avantage ?

Olivier Picciotto : effectivement nous sommes un acteur Français, mais notre terrain de jeu est international. Nous effectuons environ 30% de notre CA à l'étranger (USA, Europe et Afrique). Traditionnellement, la France est plus réticente à l'adoption de nouvelles technologies surtout dans les grands comptes. Cela a été le cas pour le Low Code et le No Code comparé à d'autres pays. Cependant, nous avons remarqué une nette accélération en France depuis ces derniers mois. On pense que l'effet COVID 19 y est pour quelque chose, les entreprises se rendent compte que pour rattraper le temps perdu, il faut maintenant innover et appliquer de nouvelles technos et produits pour produire leurs apps métier. Il est certain que nous nous présentons comme éditeur français à nos prospects CAC 40, ça les rassure au niveau du support et pour la proximité.

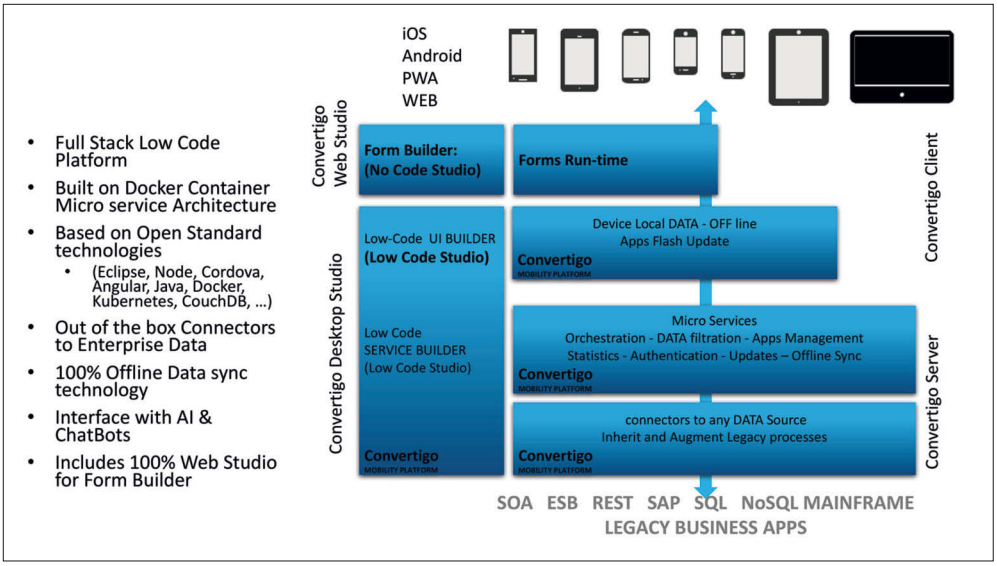
Low-code, no-code, parfois il est difficile de comprendre les deux notions. Quels sont les points communs et les points propres à chaque approche ?

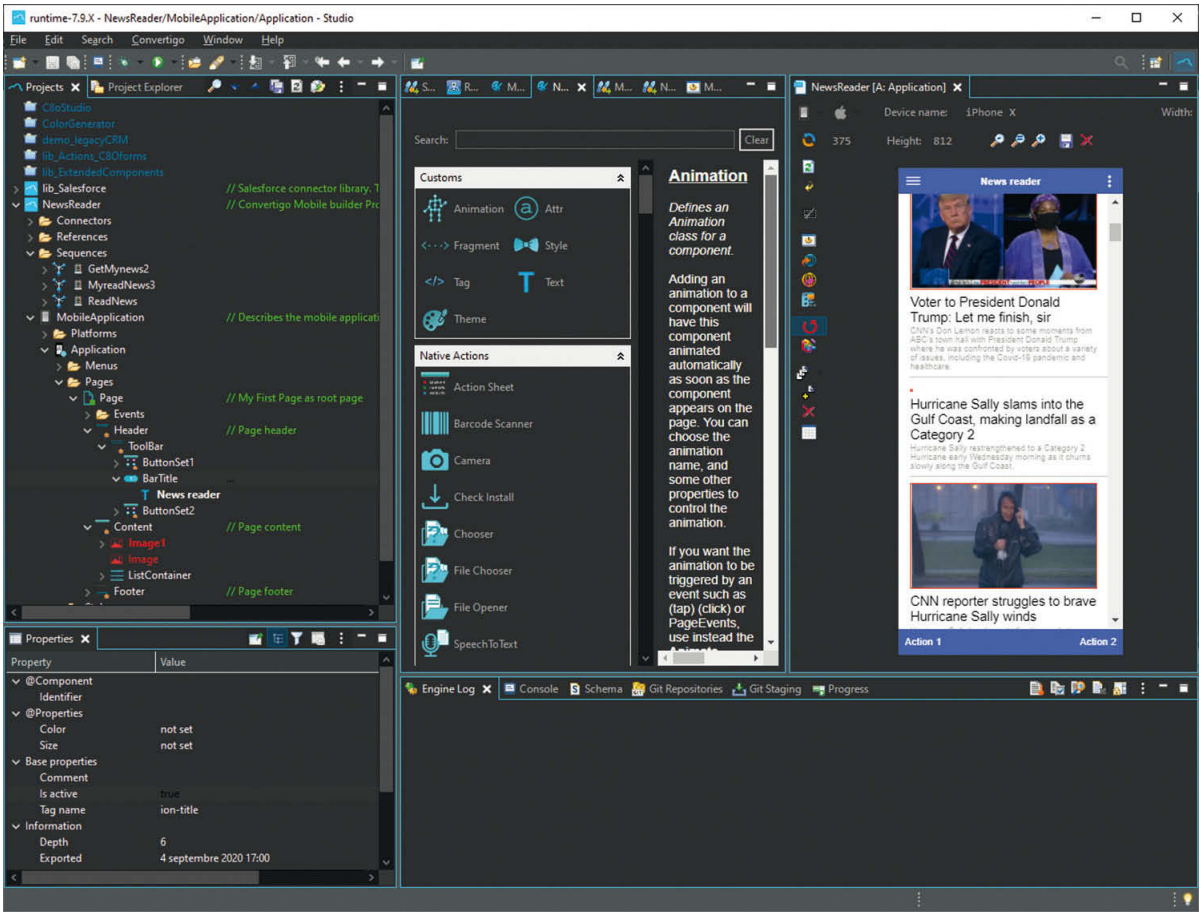
Olivier Picciotto : exact, il y a de trop de confusion entre les deux notions. Le No Code c'est zéro code, ni plus, ni moins ! On peut même considérer que Excel n'est pas du No Code car il faut programmer des formules dans les cellules... Le No Code doit rester à la portée d'utilisateurs n'ayant aucune connaissances techniques. Nous considérons qu'un produit comme Google Forms ou Convertigo Forms sont de véritables produits No Code. Le No Code reste toutefois, en général, limité dans ses possibilités.

En revanche, le Low Code n'a jamais voulu dire zéro Code ! On en écrit beaucoup moins comparé au développement classique... Chez Convertigo nous considérons un ratio de 1/10 à 1/50 selon les projets. Le Low Code permet en revanche - et nous le revendiquons chez Convertigo - de développer des applications complexes sans aucune limite de fonctionnalités. Les designers d'interfaces en Drag & Drop sont bien sur un des éléments essentiels d'une plateforme Low-Code mais cela ne suffit pas. Android Studio ou Visual Studio proposent aussi ces éditeurs mais ils restent des IDE pour les développeurs. Le point commun de chaque approche est de réduire le temps et le budget nécessaires pour développer une app. Là où on touche le jackpot c'est quand on peut combiner l'approche No Code et Low Code dans une même plateforme ! C'est ce que propose Convertigo avec son Studio No Code destiné aux développeurs "citoyens" et son Studio Low Code destiné aux développeurs "professionnels." Et ils interagissent entre eux.

Traditionnellement, on dit que le low code concerne surtout les applications métiers, logistiques, pour automatiser certaines actions, agglomérer des données / applications ? Ou développer rapidement des projets internes qui ne nécessitent pas d'utiliser Eclipse ou Visual Studio ? Qu'en est-il aujourd'hui ?

Olivier Picciotto : justement il faut sortir de cette logique et de cette idée reçue. Il est tout à fait possible de développer des projets très complexes en Low Code, C'est juste beaucoup plus rapide à faire et demande beaucoup moins de maintenance par la suite. Il y a sur le marché des produits Low - Code "pour Jouer" (prototypage, maquettes, nombre d'utilisateurs limité, etc.) et des produits Low Code professionnels qui respectent les critères de qualité de développement nécessaires aujourd'hui (DevOps, intégration continue, SDLC, full-stack, architecture Micro services, conteneurs et montée en charge). Nous nous situons dans la seconde catégorie.





Typiquement, comment fonctionne une plateforme de low code ? 80 % de configuration et 20 % de code ?

Olivier Picciotto : oui. En Low Code statistiquement, ce ratio est assez exact. Bien que certains projets puissent approcher le 98% / 2%

Parmi les critiques que l'on fait au low code, c'est l'aspect fermé / propriétaire de certaines plateformes. Impossible de changer de plateforme, réversibilité peu claire, qualité de code généré aléatoire, tarification, etc. Voit-on une évolution sur ces sujets sensibles ?

Olivier Picciotto : oui et heureusement. Ce sont des critiques récurrentes mais ce n'est pas une fatalité. C'est un point que nous pensons traiter correctement grâce à 3 critères :

- Notre approche Open Source : la plateforme est publique et disponible en open source sur GitHub. Nous sommes sans doute les plus à même de la maintenir et de de la faire évoluer mais rien n'empêche un client de le forker et de l'améliorer. Par exemple en cas du rachat de Convertigo par un éditeur qui multiplie les prix par 10, nos clients pourraient forker. Nous diminuons donc fortement la dépendance vendeur de nos clients.

- Le choix des technologies de base : dans la mesure du possible nous nous efforçons d'utiliser comme composants dans notre plateforme des technologies standards open source et reconnues par les communautés. Cela rend la plateforme pérenne et maintenable. Par exemple notre couche backend est basée sur Tomcat et Kubernetes, standards connus. Notre frontend repose sur Angular.

- La réversibilité du code généré : pour la partie front, le Low Code Convertigo génère une application Angular 9 standard qui pourrait être maintenue "à la main" en cas de "divorce" avec l'éditeur. Pour la partie back, il n'y a pas de code généré, il suffit simplement d'utiliser le moteur d'exécution Convertigo sur son serveur.

C'est cette approche assez unique sur le marché qui permet à des grands comptes de capitaliser sur une plateforme telle que la nôtre. Il faut noter que la taille d'un éditeur n'est pas non plus un gage de sécurité ! De nombreux outils sont abandonnés par les éditeurs.

Comment positionner le low code face aux développeurs ? Certains éditeurs n'hésitent pas à l'opposer. A tort ou à raison ?

Olivier Picciotto : à tort ! Pour le No Code cela peut se comprendre, mais pas pour le Low Code ! Au contraire, un développeur aura tout intérêt à se former au Low Code. Grâce à lui, il va pouvoir être plus efficace dans son développement et augmenter sa productivité. Il livrera plus rapidement les projets. Et potentiellement, il augmentera son TJM. Le tout en continuant à se faire plaisir en écrivant les 10 à 20% de codes complexes nécessaires ou en participant à la communauté open source Convertigo.

Sébastien Stormacq
Sébastien inspire les développeurs à tirer parti du cloud AWS, en utilisant son mélange secret de passion, d'enthousiasme, d'obsession des clients, de curiosité et de créativité et vous propose tous les mois le podcast AWS en français **https://stormacq.com/podcasts/index.html**.

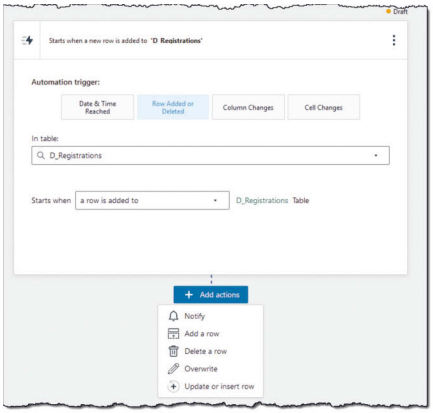
Créez des applications Web et mobiles sans écrire de code avec Amazon Honeycode

Quand j'ai découvert Lotus 1-2-3 au début des années 90, j'étais subjugué par la simplicité d'utilisation et la puissance d'analyse que ces outils donnaient à tout un chacun. J'ai rapidement compris que les tableurs allaient changer le monde de l'entreprise.

Aujourd'hui ces solutions sont à la portée de tous et se positionnent dans l'espace laissé libre entre les grosses applications vendues par des éditeurs de logiciels et des solutions faites sur mesure typiquement créées et maintenues par des équipes de développement dédiées. En effet, quoi de plus normal dans une entreprise, grande ou petite, que de créer un tableau pour analyser des chiffres, suivre les résultats de son équipe, gérer les stocks, les salaires, etc. ? Mais chaque outil a ses limites. Partager des fichiers avec son équipe, travailler avec de gros volumes de données, intégrer ces solutions avec les autres applications utilisées en entreprise, automatiser certaines tâches… sont autant de cas de figure où les tableurs montrent leurs limites. Dans beaucoup de cas, une application faite sur mesure serait plus appropriée, mais le manque de développeurs ou de ressources au sein du département IT, fait que ces applications sont rarement développées.

Amazon Honeycode

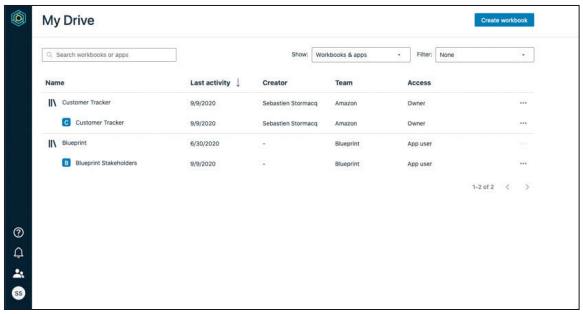
AWS propose Honeycode **www.honeycode.aws,** aujourd'hui disponible en version bêta dans la région de Virginie du Nord (Northern Virginia, us-east-1). Ce service entièrement hébergé et géré par AWS vous donne la possibilité de créer des applications web et mobiles pour votre organisation, sans devoir écrire une seule ligne de code.
La gestion des données et des relations entre elles se fait sous la forme de tableaux, selon les modèles qui vous sont déjà familiers. Tout ce que vous connaissez déjà au sujet des tableaux : les tables, les colonnes, les cellules, les formules sont toujours là. Vous créez vos écrans de visualisation ou d'entrée de données par simple glisser-déposer de composants à partir d'une palette de composants conçus pour organiser, visualiser ou éditer des données. Enfin, vous pouvez automatiser certaines tâches selon un modèle « évènement – action » : quand un évènement se produit, par exemple lorsqu'à une certaine date atteinte, une donnée est ajoutée, modifiée ou effacée, Honeycode déclenche une action, telle qu'envoyer une notification par email ou ajouter, modifier ou effacer d'autres données.



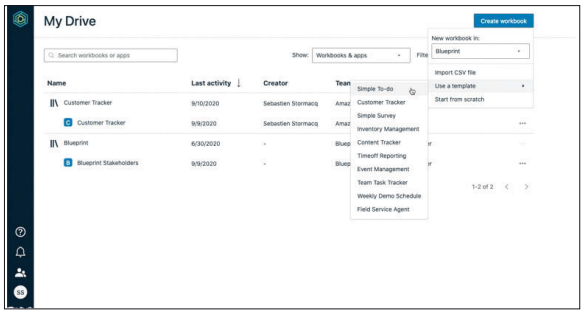
Les applications que vous assemblez sont disponibles via un navigateur ou dans l'application mobile Amazon Honeycode, disponible gratuitement sur l'App Store **https://apps.apple.com/us/app/amazon-honeycode/id1502619411** ou le Play Store **https://play.google.com/store/apps/details?id=com.amazon.aws.honeycode&hl=en**.

Un petit tuto : gérer ma liste de tâches

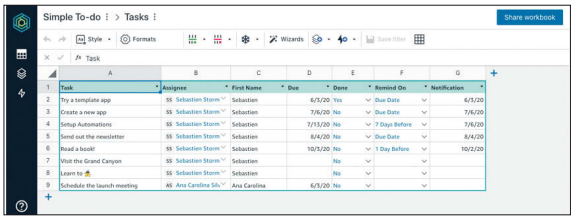
Pour aller plus dans les détails et apprendre comment créer une application avec Honeycode, j'ouvre Chrome, je démarre Honeycode builder **https://builder.honeycode.aws/** et je crée un compte. Il n'est pas nécessaire d'avoir un compte AWS pour utiliser Honeycode.
Une fois connecté, je vois **My Drive** qui contient mes projets et mes applications.



Je peux ouvrir mes projets existants ou je clicke sur **Create workbook** pour démarrer un nouveau projet. Pour ce tutoriel, je choisis le template **Simple To-Do**.



Le projet, les tables et l'application sont créés automatiquement et je peux les modifier ou déployer immédiatement. Je peux effacer les données d'exemple qui sont dans les tables et distribuer l'application à mon équipe sans plus attendre. Je peux aussi regarder comment cette application est construite et la modifier selon mes besoins. Je vais commencer par la disséquer, je la partagerai ensuite.
Après avoir créé mon projet, la table des tâches est affichée et je peux voir des données de démo.

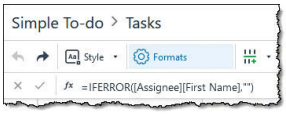


Honeycode utilise des tables, similaires aux tableaux que vous utilisez déjà, comme source de données pour vos applications. Cependant il y a quelques possibilités qui méritent d'être mentionnées, colonne par colonne :

A – Tasks : juste du texte

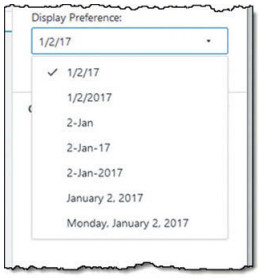
B – Assignee : du texte, formaté comme un contact.

C – First name : du texte, fourni par une formule :

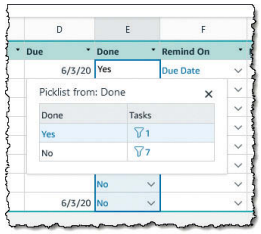


Dans cette formule, **Assignee** est une référence vers la colonne B et **First Name** indique le champ « prénom » du contact.

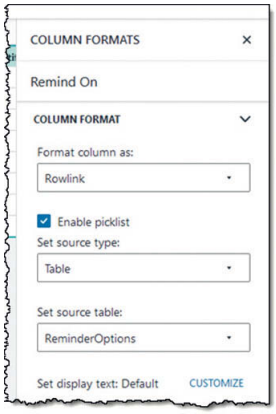
D – Due – une date, où vous pouvez choisir le format :



E – Done : une liste de sélection qui prend ses valeurs dans une autre table et qui est formatée comme un lien (**rowlink**). Ceci permet de restreindre les valeurs à une liste déclarée dans un autre table (**Done**, dans cet exemple, avec les valeurs **Yes** et **No**) et de voir la valeur importée de l'autre table.

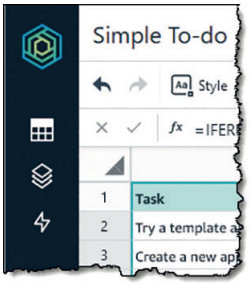


F – Remind On : une autre liste de sélection, celle-ci prend ses valeurs dans la table **ReminderOptions** :

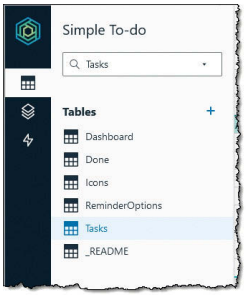


G – Notifications : une autre date

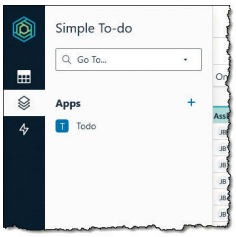
Cette table n'utilise que quelques-unes des possibilités offertes par HoneyCode. Quand je clique sur les icônes à gauche, je peux voir les autres composants de mon projet :



Je peux voir les tables :



Je peux aussi voir les applications. Un projet Honeycode peut contenir plusieurs applications qui partagent les mêmes tables.

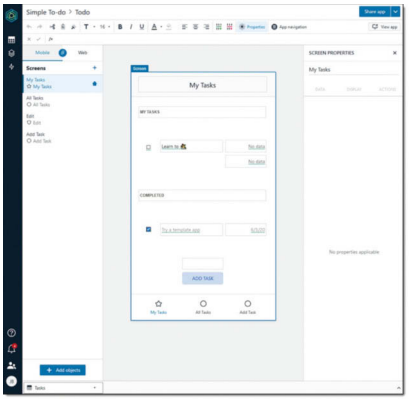


Je vais explorer les applications et le **Builder** dans quelques instants, mais avant de passer à ça, je clique sur l'icône des traitements automatiques (**Automations**)

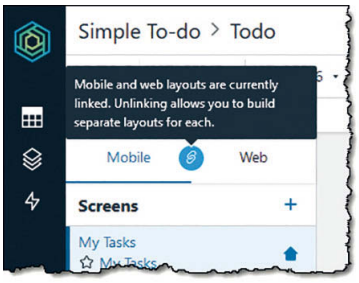
De la même manière, les traitements automatiques peuvent être utilisés par toutes les applications et les tables dans un même projet.

L'environnement de développement : Honeycode App Builder

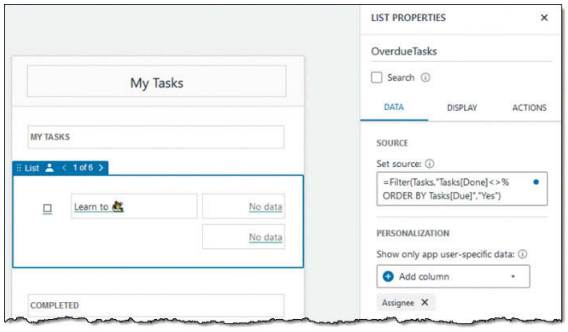
Regardons de plus près l'App Builder. Comme je l'ai fait pour les tables, je vais faire le tour du propriétaire, en vous laissant explorer le reste par vous-même. Quand j'ouvre l'application Simple To-Do, voici ce que je vois à l'écran :



Cette application contient quatre écrans (**My Tasks**, **All Tasks**, **Edit** et **Add Task**). Chaque écran possède une présentation web et une autre pour mobile. Quand vous créez un écran, ainsi que dans cette application d'exemple, les deux présentations sont liées, les changements faits sur l'un sont aussi visibles sur l'autre. Je peux détacher les présentations si je veux contrôler plus finement la mise en page de l'application en fonction du client ou si je veux des mises en pages radicalement différentes pour le web et le mobile :



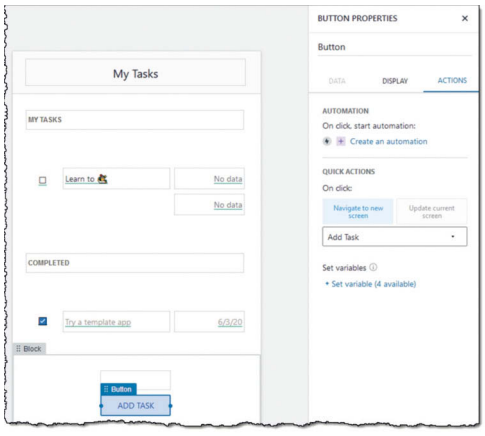
Les composants sur un écran ont accès aux données dans les tables. Par exemple, l'objet **List** sur l'écran **My Task** filtre les lignes de la table **Tasks**. Il permet de sélectionner les tâches non terminées ou de les trier par date :



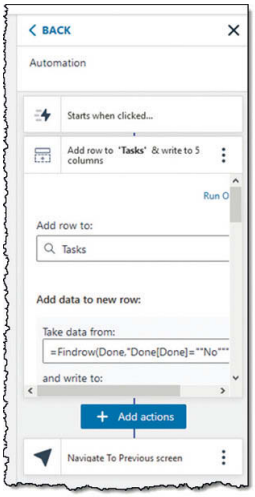
L'expression qui est utilisée est la suivante :

```
=Filter(Tasks,"Tasks[Done]<>% ORDER BY Tasks[Due]","Yes")
```

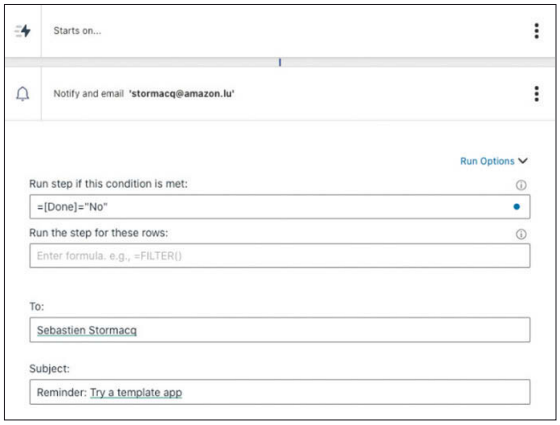
Le signe "%" dans la condition du filtre est remplacé par le second paramètre quand le filtre est évalué. Ce système de substitution permet de construire des expressions riches en utilisant la fonction FILTER(). Quand l'application est lancée, les composants que vous mettez dans la liste sont dupliqués, une copie par ligne dans la table.
Les composants peuvent être utilisés pour déclencher des traitements automatiques ou pour naviguer dans l'application. Par exemple, le bouton **ADD TASK** permet de naviguer vers l'écran **Add Task** :



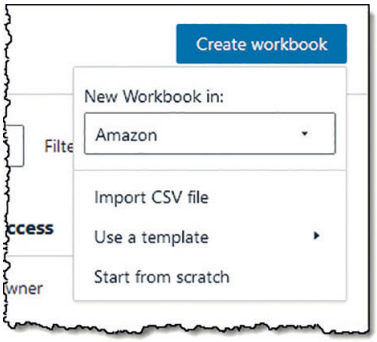
L'écran **Add Task** permet à l'utilisateur d'ajouter une nouvelle tâche et le bouton **ADD TASK** sur cet écran déclenche un traitement automatique qui ajoute les valeurs de l'écran à la table **Tasks** :



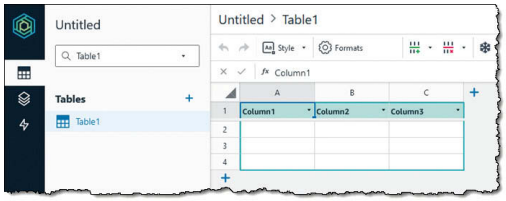
Les traitements automatiques peuvent être déclenchés de quatre manières différentes. Par exemple, ci-dessous vous voyez le traitement automatique qui permet d'envoyer des notifications par email pour les tâches dont la date butoir est dépassée. Le traitement automatique est déclenché pour chaque ligne dans la table.
La notification se déclenche que si la tâche n'est pas marqué comme étant terminée. Vous pouvez aussi utiliser la fonction FILTER()



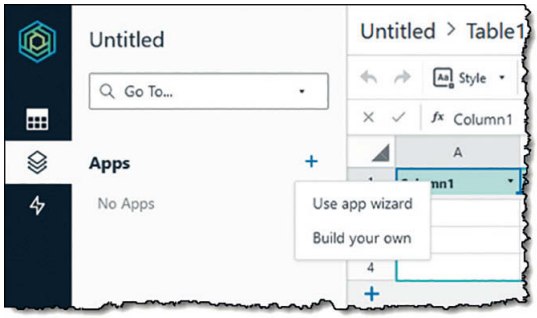
Je n'ai pas la place ici pour détailler la création d'une application à partir de zéro, mais voici quand même un rapide aperçu :
Je clique **Create Workbook** et puis je choisis entre importer des données (**Import CSV File**) or démarrer de zéro (**Start from Scratch**)



Je clique sur l'icône des **Tables** pour créer mes tables de données et de liste de références:



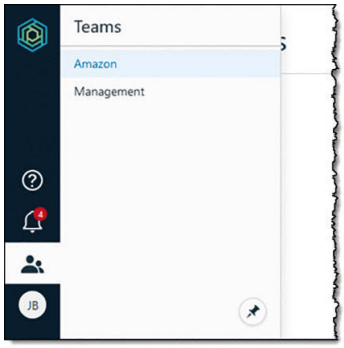
Je clique sur l'icône **Apps** pour construire l'application. Vous pouvez vous faire aider par un guide qui utilise les données de vos tables comme point de départ ou vous pouvez partir d'un écran vierge.



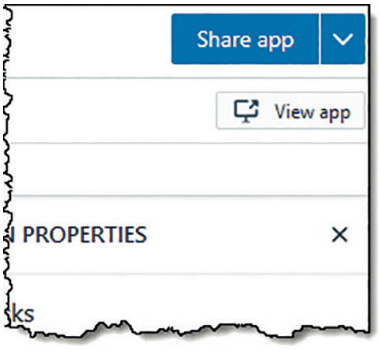
Je clique sur l'icône **Automations** pour ajouter des traitements automatiques qui se déclencheront soit quand vos données changent, soit à intervalles réguliers.
Je peux tester l'application dans mon navigateur et la partager avec mes collègues une fois qu'elle est prête.

Partager mon application

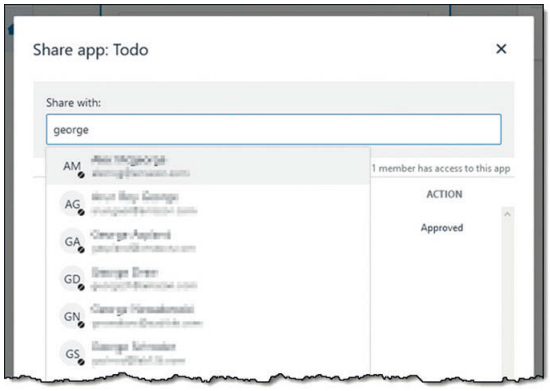
Une fois mon application prête, je peux la partager avec les autres membres de mon équipe. Chaque utilisateur de Honeycode peut être membre d'une ou plusieurs équipes :



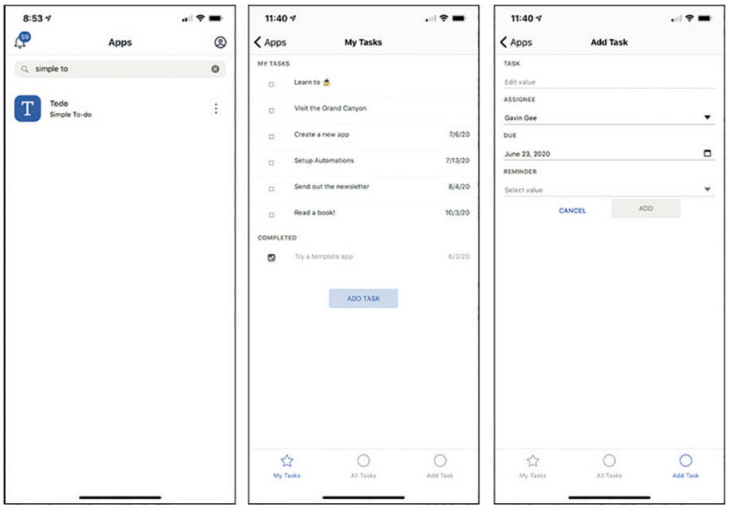
Pour partager mon application, je clique **Share app**:



Ensuite je cherche les personnes avec qui je veux la partager :



Les personnes sélectionnées reçoivent un email de confirmation avec un lien vers l'application et ils peuvent commencer à l'utiliser immédiatement. Si vous souhaitez utiliser l'application sur un smartphone, vous devez d'abord installer l'application Honeycode player (iOS https://apps.apple.com/app/id1502619411, Android https://play.google.com/store/apps/details?id=com.amazon.aws.honeycode). Une fois l'application Honeycode lancée, vous verrez toutes vos applications et celles partagées avec vous :



Les APIs Amazon Honeycode

D'autres applications peuvent utiliser les API de Honeycode https://docs.aws.amazon.com/honeycode/latest/UserGuide/api-calls.html pour interagir avec les applications que vous avez créées. Les deux fonctions sont :

- GetScreenData – pour extraire des données de n'importe quel écran Honeycode
- InvokeScreenAutomation – déclencher un traitement automatique ou une action définie dans un écran.

Conclusion

J'espère vous avoir convaincu que Amazon Honeycode www.honeycode.aws/ est simple à utiliser. Il vous donne l'autonomie pour créer des applications de visualisation et manipulation de données pour augmenter la productivité de vos équipes. Essayez-le dès aujourd'hui, construisez une application cool et dites-moi ce que vous en pensez ! Si vous avez des questions quant à l'utilisation de l'outil, rejoignez les forums dédiés à ce sujet https://honeycodecommunity.aws/.

No-code & Charles-Édouard Turquais : avec Bubble

Charles-Edouard a commencé dans le monde du code avec HTML et CSS. Rapidement, il a été bloqué sur des projets, faute de pouvoir coder l'ensemble des assets. Swift a été un déclic pour retenter l'aventure. Mais le langage reste un langage de programmation. Puis une idée d'application a germé : lancer une plateforme pour recenser les animations dans les commerces. C'était début 2019. C'est à moment-là que notre technophile découvre Bubble.

Pourquoi utiliser le no-code ? Est-ce un besoin spécifique ? De contourner les équipes techniques ?

Le no-code permet de donner vie à ses projets très rapidement. On parle beaucoup de « scaler » et de prévoir sa plateforme pour des utilisations avancées, avec des centaines voire des milliers d'utilisateurs mensuels. Cependant, entre le lancement de l'idée, la diffusion de celle-ci et son adoption, il y a en réalité beaucoup d'itérations et de corrections. Et parfois le risque que "l'idée géniale" ne trouve pas son marché pour diverses raisons. Par exemple, j'ai sorti ma plateforme pour les fêtes de fin d'années 2019 et 3 mois après nous étions tous confinés ce qui a stoppé net et pour une durée encore indéterminée tous les événements de groupe en magasin, comme les dégustations…

Pour les équipes techniques, j'ai rencontré pas mal de développeurs lorsque j'ai cherché à lancer mes précédents projets et ça a été plutôt compliqué de dénicher la perle rare (d'ailleurs, je n'ai pas réussi !) : la bonne techno, la bonne personne, les bonnes disponibilités, le bon prix…

Étudiant, on commence souvent avec très peu de budgets et il faut bien avoir au moins un début de produit pour vendre et lever des fonds afin de rémunérer la personne en question. On cherche donc plutôt un associé, mais l'exercice est tout aussi compliqué, car les jeunes dev cherchent aussi des challenges techniques et ne sont pas trop habitués à l'univers business des étudiants en école.

Avec le no-code, on gère son projet de A à Z dans un moment où il est encore simple et réalisable sans trop de contraintes. C'est vraiment la marche qui manquait pour lancer des projets et les faire grandir !

Tu utilises Bubble. Qu'est-ce qui te plaît dans cet outil ?

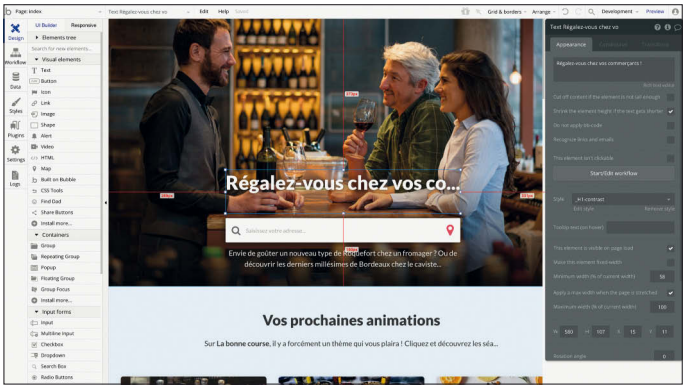
Je suis tombé amoureux de l'outil, car je trouve que c'est une vraie révolution par rapport à tout ce que j'ai pu utiliser précédemment. Bubble (bubble.io) est vraiment clair, et sa logique propre est très cohérente lorsqu'on s'immerge dans la plateforme. Il est conçu pour guider tous les publics, grâce à un signalement des erreurs extrêmement performant, ainsi qu'un manuel très bien intégré aux différentes fonctions dans l'interface. En plus, tous les sites sont réalisables (mobile, responsive, desktop) avec des connexions vers l'extérieur grâce à un configurateur d'API qui étend à l'infini les fonctionnalités et les sources de données.

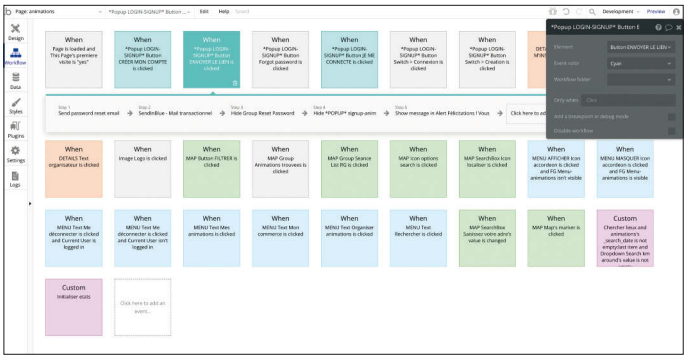
Pour moi, c'est vraiment un outil disruptif qui combine prix, fonctionnalités, confort d'utilisation et possibilités offertes !

Par rapport à de la programmation classique, que t'apporte Bubble ?

Bubble permet visuellement et logiquement de construire vite son produit et de le tester sans perdre de temps sur un stack technique, tout en garantissant une montée en charge progressive avec toutes les fonctionnalités nécessaires à une application moderne.

En tant que développeur d'une idée, il me permet de la matérialiser rapidement et m'aide à y réfléchir en même temps. Même si un regard extérieur est toujours bienvenu pour prendre du recul sur son application et l'améliorer, itérer seul fait gagner beaucoup de temps dans le lancement et le recrutement d'associés sur le projet.





Quelles sont les forces et les faiblesses de l'outil pour toi ? Y a-t-il des subtilités pour certaines fonctionnalités ou dans le debug ou le déploiement ?

Les soucis de l'outil sont sa courbe d'apprentissage et son look un peu démodé qui peuvent rebuter certains. Le SEO n'est également pas incroyable pour les sites sous Bubble, qui se concentrent plus sur un rendu graphique que sur les performances de chargement et la structure du code. C'est assez gênant si le projet nécessite d'être bien référencé pour fonctionner.

On peut cependant améliorer cette limite en passant par un outil externe pour la landing page par exemple. La vraie force de l'outil réside selon moi dans sa conception très orientée vers l'utilisateur et non centrée sur le tarif, ce qui permet de se lancer rapidement et correspond bien aux attentes des outils no-code.

As-tu une idée du ROI ou du TCO des projets faits sur Bubble ?

Bubble permet surtout d'énormes économies de temps et d'argent dans la phase de création. Il permet de donner l'impulsion originelle qui va matérialiser le projet et le faire passer dans une autre phase et ce n'est pas négligeable.

Yann Azoury
créateur de Faveod,
www.faveod.com

Faveod, le « no limit »

Historiquement, les développeurs optimisent leur travail par deux axes principaux : dynamique et statique. Les deux approches sont historiquement complémentaires, au sein d'un même projet comme pour les différents types de projets : la dynamique impose des dépendances plus fortes qui rendent le projet plus sensible à son environnement et aux évolutions. Les projets soumis à de forts enjeux de performance, de sécurité, d'intégration, de propriété intellectuelle ou de dette technique ne seront pas de bons candidats à trop de dépendances multiples ou structurantes. À l'inverse, une approche trop statique pour un projet éphémère sans ces contraintes serait une erreur alors que la roue a déjà été inventée par ailleurs.
Au-delà de ces enjeux d'architecture fondamentaux, le débat de la pertinence du low-code/no-code classique fait écho à la réflexion presque philosophique des relations et des responsabilités entre la Machine et l'Humain.

L'Humain contre la Machine : low-code/no-code

Le low-code/no-code c'est la consécration de la Machine : partant du constat que l'Humain ne sait pas produire suffisamment de logiciels de qualité à un coût faible, il a été décidé d'accumuler des couches réutilisables de runtime dynamiquement, jusqu'à ce que l'Humain n'ait presque plus rien à faire, la Machine ayant désormais une puissance infinie et « presque gratuite » grâce au Cloud (c'est bien connu…), elle pourra bien tenir la charge que cela nécessite. Au final, ces runtimes restent pourtant du recyclage de différentes couches de travail de divers Humains, mais ces couches sont souvent sédimentaires et chaque génération perd un peu de la maîtrise des couches anciennes. Le développeur devient consommateur et perd petit à petit son potentiel de producteur.
L'algorithme le plus rapide sera celui qui consommera le moins de ressources en exploitant au mieux la spécificité de chaque plateforme — matérielle et logicielle — pour le cas précis.
Au-delà du pur aspect pratique — temps de réponse et coût des ressources — les enjeux écologiques exigent des logiciels écoconçus. L'augmentation croissante des failles de sécurité et leurs conséquences de plus en plus visibles imposent une chaîne de sécurité sans maillon faible.
Ces objectifs ne sont pas accessibles pour des low-code/no-code classiques du fait de leur architecture trop dynamique.
On pourrait finalement présenter les low-code/no-code classique, comme une interface simplifiée pour paramétrer des composants prédéfinis et cadrés. La liberté qui en découle peut s'avérer assez large, mais on reste toujours dans des cas prévus et anticipés ; en revanche la créativité est rapidement entravée. Pour autant, que de choses avancées ont pu être réalisées dans de simples tableurs !
Ces outils ne sont donc pas destinés aux développeurs chevronnés, aptes à sortir des sentiers battus, mais plutôt aux « citizen developers » c'est-à-dire aux utilisateurs de bureautique souhaitant se créer des outils plus adaptés et aptes à bidouiller un peu de code simple.
En général ces outils ne sont pas adaptés au développement en équipe, avec un véritable suivi de l'activité de chacun et de son contexte, une gestion de versions fine et pertinente et de déploiement partiel ou incrémental. Cependant ces fonctionnalités peuvent être considérées comme superflues pour le type de projets cibles.

Dans Faveod Designer, la modélisation est agnostique des technologies, il est possible de choisir à tout moment le langage et les frameworks à produire, sans nécessiter de retouche manuelle. La pérennité est alors totale.

Il va de soi que les intégrations avec des flux « legacy » seront aussi fortement limitées ; il faudra soit adapter les interfaces existantes, soit prévoir du fichier plat et quelques tâches manuelles.
Il est aussi recommandé de bien anticiper les exigences de données et de charge utilisateur, car de fortes volumétries poseraient problème, les optimisations bas niveau n'étant pas accessibles. Bref, autant de problématiques bien connues des développeurs, mais qui ne disparaissent pas par magie grâce au low-code/no-code.

L'Humain avec la Machine : Faveod Designer

Même si à première vue Faveod Designer présente les avantages ergonomiques d'un low-code, il a surtout la puissance d'un langage de programmation produisant… du code source normal dans les langages de programmation classiques (Java, JS, Ruby, etc.). Mais au lieu de laisser à la charge de l'Humain d'écrire chaque ligne de code, Faveod Designer va produire l'ensemble des fichiers, full stack, en fonction de la formalisation du besoin (algorithmes, vues, données, etc.) définie par les développeurs.
Le résultat est parfaitement *spécifique* et *optimisé*, à la cible et au besoin, et bien entendu sans les limites techniques et fonctionnelles d'un low-code classique. Le développeur retrouve sa liberté ; sa créativité est décuplée par sa capacité à *programmer les règles produisant (tout) son code*.
Puisque l'ensemble du code est produit, maintenu et mis en cohérence par une machine, il devient aisé de changer du code en masse sans régression : changement transverse d'API, de langages, de frameworks, ou de modèle de données ou de persistance.
Il est désormais incontestable que l'Humain est moins performant qu'une Machine pour écrire du code prenant en compte les dizaines d'exigences techniques et fonctionnelles pour chaque ligne de code. Faveod produit en pic plusieurs centaines de Mo de code… par seconde, l'Humain capable de faire cela n'existe pas encore… En revanche, accumuler des centaines de Mo de middleware ou de runtime pour résoudre le problème ne fait que le repousser et crée un entonnoir fonctionnel et technique, ce qui n'est acceptable que pour des projets éphémères.
Pas de secret, la machine produit le code en masse… mais grâce à l'Humain !
Il garde le contrôle en enseignant à la Machine comment faire mieux : c'est le « Human Teaching ». Faveod Designer accumule tout ce savoir-faire Humain depuis 15 ans et le retranscrit précisément

sous forme de code dans les cas cibles spécifiques, quelles que soient les technologies, les exigences et la complexité générale des projets.

Cela signifie que Faveod peut produire à partir d'une conception formelle le meilleur code qui correspond à tout moment, sans avoir à le retoucher ou à le corriger à la main. Si une nouvelle API, un nouvel algorithme ou une nouvelle exigence arrive, l'Humain lui enseigne formellement, et Faveod réécrit instantanément tout le code source concerné, car il sait quels sont les effets de bord et les impacts qui correspondent.

L'Humain garde le privilège de la créativité, de l'astuce et de la pertinence ; il a à sa disposition la Machine pour les tâches lourdes, répétitives et rébarbatives. Cette alliance permet d'avoir le meilleur résultat, respectant toutes les contraintes et exigences, et par expérience cela permet de créer et maintenir des applications irréalisables en pratique avec des outils classiques ou low-code.

Développer en restant agnostique

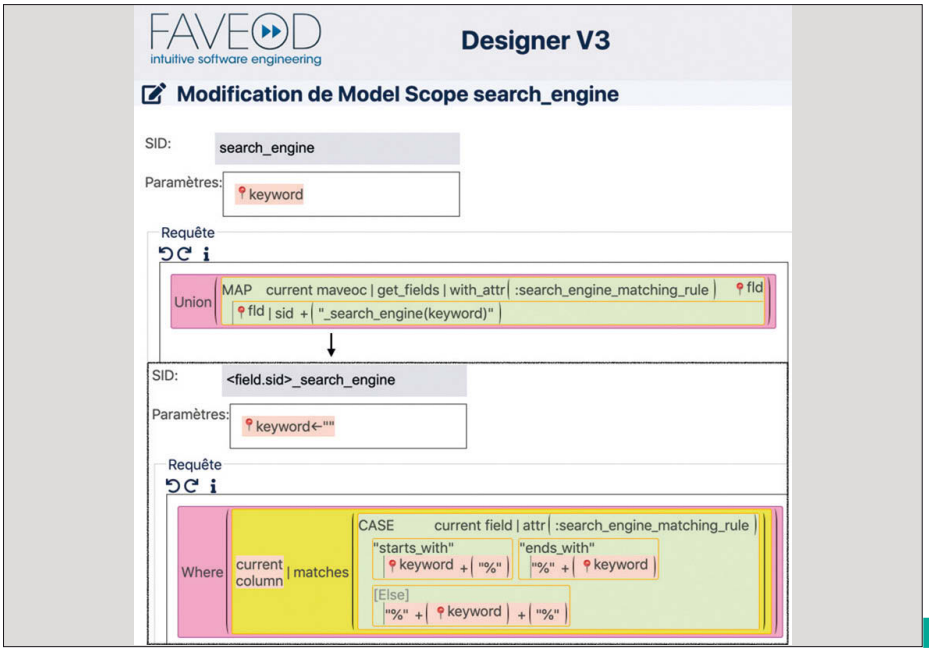
Les technologies et les modes technologiques évoluent vite et sans véritable logique industrielle. Bien des projets n'ont pas le temps d'aboutir alors que les technologies sur lesquelles ils reposent sont déjà dépassées. Le seul moyen d'éviter ces aléas est soit d'avoir un projet éphémère, c'est la cible logique des low-code, soit d'éviter de dépendre de la moindre technologie, c'est-à-dire ne rien écrire qui n'est propre à une technologie. 1

Faveod correspond donc à un ensemble de DSL graphiques et formels, qui ont l'élégance de disparaître totalement dans le code et donc à l'exécution. La différence avec les compilateurs habituels est la possibilité de définir aisément toutes les règles de production de code, y compris pour des cibles exotiques ou maisons.

Il y aura bien évidemment besoin de middleware (OS, bibliothèques, protocoles, etc.), mais aucun en particulier, l'architecture pourra les définir et s'adapter au fil des années. Alors que sinon, l'architecture retenue par les premiers développeurs lors de la définition du besoin initial sera à assumer jusqu'à la prochaine refonte.

Le véritable collaboratif temps réel

Les outils low-code n'atteignent pas en général les capacités collaboratives d'un Git, ils ne sont pas vraiment prévus pour des projets avec un grand nombre de concepteurs ni pour une gestion avancée des versions, de branches ou de feature toggling. Et finalement même Git ne permet pas de faire du véritable collaboratif temps réel : au final chacun travaille sur son poste local jusqu'au push,



2

rien n'empêche qu'un malentendu persiste jusqu'au conflit technique ou fonctionnel. Tout cela fait sens, car les modifications logiques conséquentes d'une nouvelle exigence concernent en général plusieurs fichiers qui doivent être mis en cohérence et poussés en même temps.

Faveod Designer évite cette problématique en s'abstrayant complètement de la logique de fichiers de code source : les modifications étant formelles et les impacts calculés, tous les changements sont poussés en cohérence et en même temps. Ce ne sont plus les fichiers qui sont versionnés, mais bien les évolutions qui correspondent aux exigences. En conséquence, tout changement dans Faveod est directement appliqué, visible et testable sur l'environnement de développement. En cas de malentendu, toute l'équipe le verra tout de suite et évitera que les divergences persistent.

Faveod se base sur les spécifications formelles, indépendamment des fichiers que cela concerne, il est donc tout à fait possible de travailler en parallèle à plusieurs sur ce qui sera le même fichier, sans lock ou autre artifice.

Exemple : algorithme de moteur de recherche

Sur le fond, les algorithmes se définissent formellement, sans subir les spécificités et les complexités propres à chaque langage ; seuls les fondamentaux fonctionnels sont nécessaires. Sur la forme, cela se fait graphiquement, de manière bien plus synthétique et généraliste qu'un code source textuel.

Résultat : créer un moteur de recherche ne se fait pas en écrivant autant de code que de types de recherches et de sources de données nécessaires, mais en définissant des méta-algorithmes qui produisent automatiquement ce code à partir de la formalisation statique des règles. Le code produit ne peut pas régresser techniquement, il est automaintenu et améliorable en masse, dans son entièreté.

Concrètement, le moteur de recherche (simplifié pour l'exemple) fonctionne comme suit. Lors de la conception d'une application, chaque champ participant à la recherche est déclaré via l'attribut formel « search engine matching rule ». À partir d'une unique directive de champ générique sera alors automatiquement produite une routine spécifique pour chaque champ, créant pour chacun sa propre règle de recherche en fonction de la valeur de l'attribut. Une directive de modèle fait ensuite l'union des résultats de recherche pour chacun de ces champs, quels que soient leur nombre et leur type. Enfin, la recherche globale réunit les résultats de tous les modèles participants. Le figure 2 montre les écrans de définition des deux directives génériques : recherche par champ, et par modèle. L'application produite dispose alors d'implémentations dédiées à chaque champ et chaque modèle, comme si elles avaient été écrites à la main.

Conclusion

Le low-code/no-code a été créé pour répondre aux besoins de « Business Analysts » frustrés par les nombreuses limitations et défauts d'Excel et consorts. Les développeurs classiques ont les compétences pour dépasser les limites de ces outils, mais ils ne sont pas assez nombreux pour répondre à toutes les petites demandes simples et agiles, il est plus intéressant pour eux de se concentrer sur les projets critiques et durables.

Nicolas de Mauroy.
J'ai passé 20 ans à gérer des projets informatiques avec des logiciels qui ne me convenaient pas. J'ai créé Open-Lowcode pour résoudre les problèmes techniques et économiques rencontrés pendant toutes ces années.

Open Lowcode : l'open source pour garder son indépendance

Open Lowcode permet de créer une application spécifique en quelques heures de travail. Le développeur ou l'analyste fonctionnel commencent par définir le modèle de l'application : des objets métiers auxquels on ajoute des briques puissantes. Cela commence par le modèle de données, mais les briques utilisés proposent clé en main tous les basiques de l'informatique de gestion : sécurité, traçabilité, calculs, rapports, workflows, planification... Le "designer" génère ensuite une application standard.

Vous pouvez ensuite facilement rajouter des logiques complexes. L'automatisation de groupes d'actions élémentaires correspondant à un cas métier fréquent permet des gains de productivité significatifs. Dans une application récente de gestion d'incidents, les utilisateurs souhaitaient ainsi pouvoir, en un écran, fermer un incident avec un commentaire et ouvrir un autre incident lié.

Pour ces cas, la plateforme propose de nombreuses possibilités pour rajouter une logique ou des écrans spécifiques dans l'application standard. Vous programmez ces extensions en java, avec tout le confort d'un vrai langage et l'accès facile à toutes les fonctions du serveur. 1

Les applications générées s'exécutent sur le serveur applicatif Open Lowcode, et les utilisateurs accèdent à l'application via un client dédié. Ce dernier est optimisé pour les besoins des "power users".

La plateforme est simple d'utilisation. Elle est open source, basé sur des technologies standard : Java 8, SQL, les librairies Apache PDFBox et POI. L'utilisation de l'open source aide à réduire les coûts et à ne pas s'enfermer dans une technologie. Comme tout projet open source, il est possible de committer du code et de proposer de nouvelles fonctions.

Mettons la main à la pâte

Construisons une application minimaliste gérant des post-it virtuels. Pour cela, nous écrivons le fichier de modèle ci-dessous :

[Tutorial.java] : voir code complet sur programmez.com / GitHub

Nous déclarons le module « Tutorial », un gestionnaire d'actions. Ce module possède un « Data Object ». On ajoute à cette entité métier des fonctions avancées d'audit en rajoutant simplement les propriétés CreationLog, UpdateLog, et Iterated. Cette dernière enregistre tous les états passés de l'objet, permettant un accès facile à l'historique. Cette application comprend également une gestion basique des droits, avec deux profils : writer et reader. Ils permettent respectivement de modifier ou seulement de voir les notes.

À partir de ce modèle, le « designer » génère l'application complète. On utilise pour cela un script ant (BuildTutorial.xml) qui va compiler le fichier modèle, générer puis compiler l'application. 2

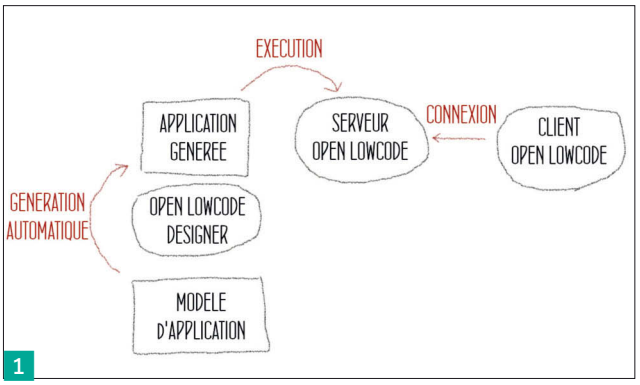
Il suffit ensuite de configurer le serveur (voir tutorial.cfg). Nous utilisons pour le tutoriel la base Apache DB qui sera lancé directement par le serveur (mode « embedded »), puis de lancer le client. Nous avons alors accès à l'application, et pouvons utiliser l'utilisateur admin (mot de passe admin) pour nous connecter.

On accède à l'écran de recherche et de création de notes dans le menu « Open Lowcode Tutorial ». Nous allons créer une nouvelle note, puis effectuer un changement. Ensuite, nous accéderons à l'écran d'audit à travers le menu « Manage » puis « Show History ». Nous avons alors accès à l'intégralité des changements effectués sur l'objet. 3

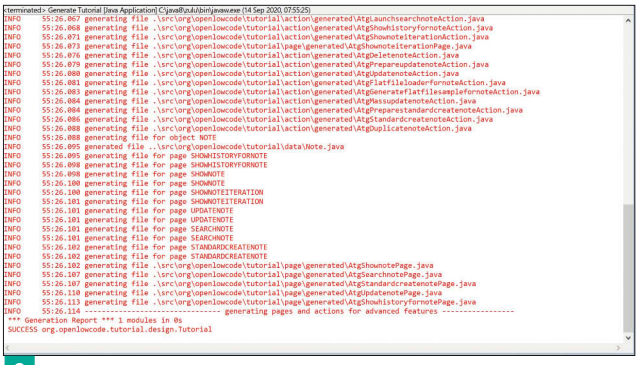
Nous pouvons aussi configurer des droits en rajoutant des utilisateurs. Ceux-ci peuvent être locaux ou liés à votre annuaire d'entreprise LDAP. Pour notre exemple, nous créons des utilisateurs avec le menu « Start with Application User du menu "System". Il est seulement nécessaire de renseigner le nom du compte et le mot de passe. Ces utilisateurs peuvent ensuite être affectés aux deux groupes, "Reader" et "Writer".

En une vingtaine de minutes, nous avons construit une application, certes très modeste, mais elle permet de partager des notes dans un environnement d'entreprise collaboratif réaliste.

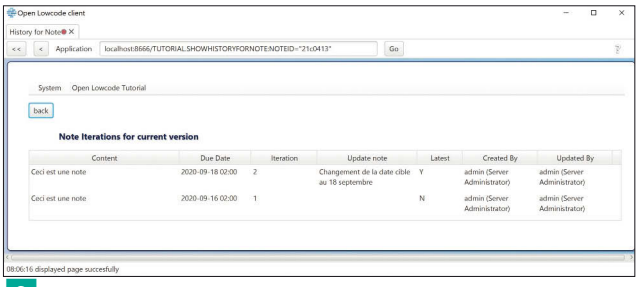
À vous de jouer !



1



2



3

PRÉREQUIS TECHNIQUES POUR LE TUTO

- JDK Java 1.8 avec JavaFX (installation Zulu recommandée)
- Apache DB (base de données locale)
- Apache PDF Box et Apache POI
- La dernière version d'Open Lowcode (tutoriel réalisé avec la version 1.11.6)
- Récupération de Java Mail
- Tous les jars doivent être installés dans le sous-répertoire "./lib" du tutoriel
- Questions et support : https://openlowcode.com/fr/contacter-open-lowcode/

Walter Almeida
Ingénieur ESIEE promotion 1997. Après 13 années d'expérience en tant que consultant développeur puis architecte des systèmes d'information dans plusieurs grands comptes et éditeurs, il fonde Generative Objects (www.generativeobjects.com), éditeur depuis d'une plateforme low-code (GO-Lowcode), utilisée par plusieurs acteurs majeurs (Naval Group, Thales, le CEA, Météo France, APF France handicap, Inserm, etc.).

Generative Objects : les promesses d'une plateforme low-code open source

Le marché du low-code est en pleine expansion, une étude de MarketsAndMarkets prévoit une croissance de ces plateformes de 13,2 M$ en 2020 à 45,5 M$ en 2025. La course est engagée pour déterminer quels en seront les acteurs clés, les éditeurs logiciels nouvelle génération. C'est une course pour le profit et non pas pour donner accès à tous à cette technologie. Une alternative open source serait le moyen de garantir une démocratisation de l'accès au low-code, qui n'est pas sur la table pour le moment. Nous présentons ici la plateforme GO-Lowcode, et ce que cette alternative open source apporte aux développeurs et porteurs de projets.

Une plateforme open source

Une success-story d'une technologie open source qui a révolutionné internet est Wordpress. Wordpress a démocratisé la création de sites internet pour tous, et est totalement open source, ce qui a garanti sa diffusion large. Aujourd'hui, si d'autres acteurs non ouverts se partagent le marché de la création de sites internet, Wordpress a ouvert le chemin et reste largement en tête avec une couverture de 38 % des sites internet de la planète.

Les plateformes low-code du marché restent particulièrement adaptées pour des développements d'applications métiers d'entreprise, et n'ont pas le même potentiel de diffusion large et accessible qu'a eu Wordpress. Plusieurs contraintes rendent notamment leur utilisation non adaptée pour des startups internet ayant pour ambition de créer de nouvelles plateformes et services. En effet, de telles startups ont besoin d'un contrôle complet sur leur code applicatif, et ne peuvent ou ne veulent pas être liées au modèle économique d'une plateforme low-code propriétaire, certainement pas dans le cadre d'un modèle économique lié à la montée en charge du service proposé et la croissance de la base utilisateurs.

L'ambition de la plateforme GO-lowcode de Generative Objects est d'aller au-delà de ces limitations, et d'apporter une alternative complètement ouverte, aussi bien sur le code généré, ouvert et de qualité humaine, que sur la plateforme de génération et modélisation, pour devenir un incontournable tel que Wordpress pour le low code.

Comment cela fonctionne-t-il ?

La plateforme GO-Lowcode est basée sur le paradigme de génération de code. 1

Le métamodèle de GO-Lowcode permet à l'utilisateur de décrire son application sous un format fonctionnel, non lié à un langage ou une technologie. Ce métamodèle permet ainsi de définir le modèle de données, les processus et les interfaces utilisateurs.

La création du modèle de l'application cible se fait sur une interface web, ici une partie de l'interface de création du modèle de données : 2

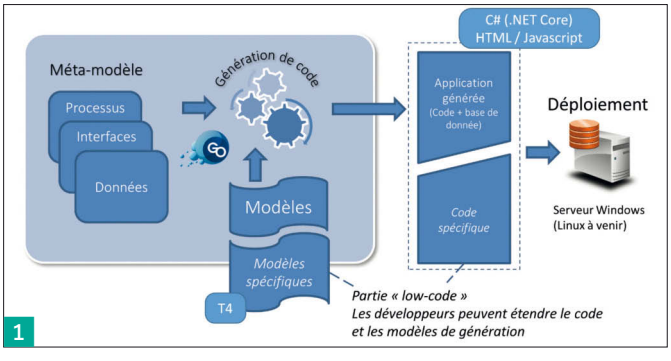
Nous y voyons un modèle pour une application de gestion de contrats, avec ici une vue sur l'entité « Client Entreprise » qui est une entité dérivée de l'entité « Client », avec les champs « Raison Sociale », « Siret », et une relation vers l'entité « Personne » pour la « Personne de contact » pour ce client. À savoir que l'interface web de manipulation du modèle applicatif n'est qu'une façon de faire : il est également possible d'utiliser directement les APIs fournies pour créer ou modifier un modèle applicatif.

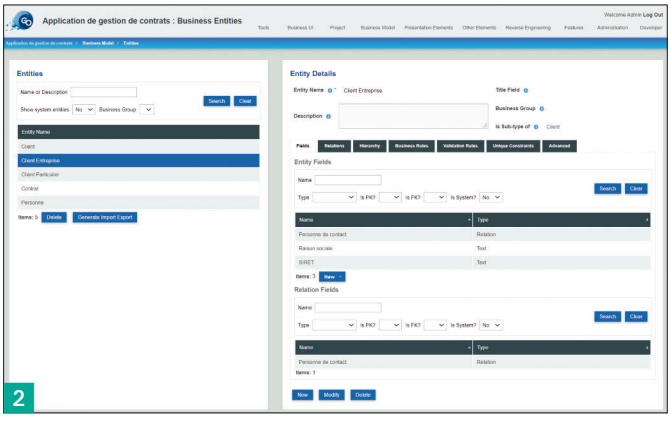
Il en va ensuite de même pour la modélisation des processus, des règles de sécurités, de l'interface utilisateur, l'intégration de services tels que l'envoi d'emails, l'import / export des données, la gestion documentaire, etc. Pour la partie front, GO-Lowcode intègre un modeleur HTML et CSS pour contrôler le rendu et la disposition des pages et des différents widgets.

Le moteur de génération de code va alors générer l'intégralité du code applicatif et scripts de création et/ou de mise à jour de la base de données, compiler et déployer l'application sur l'environnement cible, et déployer ou mettre à jour le schéma de la base de données.

La génération s'appuie sur les modèles de génération de code de GO-Lowcode. Ces modèles permettent de faire la translation entre le métamodèle fonctionnel agnostique de toute technologie, et une stack technologique spécifique.

Ici un modèle de génération simple pour la génération du code associé (une classe C#) à un élément de modèle de type Liste de valeurs (exemple, Civilité : « Monsieur, Madame, Mademoiselle ») 3





```
<#@ include file="references.tt" #>
<#@ output extension=".cs" #>
<#@ include file="FileHeaderInclude.tt" #>
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using GenerativeObjects.Practices;

namespace <#=_currentProject.RootNameSpace#>.Data.DataObjects
{
    public enum <#=_currentEnumeration.Name#>Enum : int
    {
<#  foreach (EnumerationValueDataObject enumerationValue in _currentEnumeration.EnumerationValues.OrderBy(e => e.Value))
    {
#>      <#=enumerationValue.InternalName#> = <#=enumerationValue.Value#>,
<#  }
#>  }

    public static class <#=_currentEnumeration.Name#>EnumDisplayNames
    {
        public static readonly List<EnumDisplayString<<#=_currentEnumeration.Name#>Enum>> Items
            = new List<EnumDisplayString<<#=_currentEnumeration.Name#>Enum>>
                {
<#                  foreach (EnumerationValueDataObject enumerationValue in _currentEnumeration.EnumerationValues.OrderBy(e =>
 e.Value))
                    {
#>                  new EnumDisplayString<<#=_currentEnumeration.Name#>Enum>
 (<#=_currentEnumeration.Name#>Enum.<#=enumerationValue.InternalName#>, "<#=Utilities.EncodeXamlString(enumerationValue.DisplayName)
 #>"),
<#                  }
#>                };
    }
}
```

3

Ces modèles s'appuient sur le langage de templating T4 de Microsoft, enrichis de l'accès direct au métal-modèle de GO-Lowcode, et donc au modèle fonctionnel de l'application à générer. Un orchestrateur de modèles permet ensuite d'ordonner et exécuter les nombreux modèles nécessaires à la génération de l'application complète.

Ce fonctionnement par modèles techniques de génération, au-delà de découpler totalement le modèle fonctionnel de l'application de son implémentation technique, et ainsi d'offrir à l'utilisateur non-développeur un langage de haut niveau d'abstraction, garantit une flexibilité et évolutivité technologique des applications générées. Il suffit en effet de faire évoluer les modèles techniques de génération pour faire évoluer technologiquement les applications générées ou bien de complètement intégrer une nouvelle stack, pour générer les applications sur une cible différente.

La stack actuelle implémentée par les modèles de génération est HTLM/CSS/JavaScript, en mode « Single Page Application » pour le front ; C# pour le back et SQL Server, MySQL ou Oracle pour la base de données. L'architecture technique générée est de qualité entreprise avec un niveau de performance et scalabilité à la hauteur pour supporter des scénarios d'usage à forte charge.

Le paradis du développeur

La partie « low-code », et donc le terrain de jeu du développeur est plus large que sur les plateformes low-code propriétaires telles que nous les connaissons.

Le développeur pourra directement étendre le code généré, en respectant les règles d'extension permettant de continuer à faire évoluer le modèle applicatif sans perdre les extensions apportées. Suite à génération de code, GO-Lowcode pousse le code généré sur Git, pour permettre au développeur d'apporter ses extensions ; puis à chaque génération la plateforme fait un pull du code pour intégrer à la prochaine génération les extensions apportées par le développeur.

En complément, le développeur pourra apporter des extensions aux modèles de génération techniques, afin d'augmenter les capacités de génération automatique apportées par GO-lowcode, pour tout besoin spécifique au projet en cours.

Et à un niveau au-dessus, le développeur pourra faire évoluer le métamodèle fonctionnel de description des applications et virtuellement aller au-delà de toute limitation et enrichir le périmètre fonctionnel qui pourra être couvert par la plateforme GO-Lowcode. L'évolution du métamodèle de GO-Lowcode est possible, car la plateforme est réflective : GO-Lowcode est modélisée et générée par GO-Lowcode elle-même ! Donc pour faire évoluer le métamodèle, il suffit d'ouvrir et enrichir avec GO-Lowcode le modèle de GO-Lowcode et re-générer GO-Lowcode !

Avec la plateforme GO-Lowcode, le développeur n'est donc pas oublié, il endosse un nouveau rôle et devient à la fois l'architecte de la solution ainsi que l'expert qui saura étendre les modèles de génération de code et le métamodèle fonctionnel, offrant de nouvelles perspectives de développement.

Une objection fréquente du développeur envers les plateformes low-code est le manque de contrôle et de visibilité sur le fonctionnement de la plateforme. Avec GO-Lowcode, le développeur se réapproprie ce regard d'expert, mais également sa capacité à comprendre et étendre la plateforme elle-même en devenant contributeur de la technologie, pour le bénéfice à la fois de la communauté d'utilisateurs de GO-Lowcode, mais également pour le bénéfice de son propre projet.

Quels usages ?

L'utilisation première de la plateforme GO-Lowcode est historiquement pour des applications de gestion de données, internes et externes, pour lesquelles le niveau de génération de code atteint les 95 à 100 % et permet donc des gains de productivité excellents. La plateforme GO-Lowcode a ainsi été utilisée pour réaliser l'intégralité du système d'information du Haut Conseil du Commissariat aux Comptes, mais également le réseau social interne de Thales de relation avec les Startups d'intérêt pour le groupe, utilisé par plusieurs milliers de collaborateurs Thales ; l'application interne de suivi des marchés publics de Météo France et de l'EPT Grand-Orly Seine Bièvre ; l'application de gestion des imputations projets et gestion de temps des équipes R&D du CEA. Citons également l'application d'APF France handicap pour la gestion des dossiers de Legs et donations. Au-delà des applications internes, le cas d'usage d'application B2C est l'évolution naturelle des plateformes low-code, et nous travaillons sur un métamodèle revisité de la modélisation de l'interface utilisateurs ainsi que les modèles techniques de génération, qui s'appuient maintenant sur la technologie VueJS, pour apporter une grande souplesse dans les interfaces pouvant être réalisées.

En tant que premier cas d'usage simple d'application B2C, GO-Lowcode a été utilisée pour l'INSERM pour réaliser la plateforme de gestion des abonnements au magazine de l'INSERM, avec la dimension compte utilisateur pour la gestion par l'internaute de son abonnement au magazine.

Où en est la plateforme GO-Lowcode ?

Aujourd'hui nos équipes travaillent sur la finalisation de l'ouverture open source de la plateforme, prévue pour cette année. L'appel est lancé à tout développeur intéressé pour rejoindre la core team ainsi qu'à tout acteur souhaitant un accès « early adopter » à la plateforme.

Rappelons que l'ambition va au-delà du simple défi technologique. Notre engagement et intention est de participer à une révolution d'internet et une transformation de nos usages, en donnant le pouvoir à tous les citoyens d'être acteurs et créateurs des services et plateformes de demain.

Thibault Falque
Je m'appelle Thibault Falque, j'ai 24 ans. Je suis diplômé d'un master en informatique spécialisé en Intelligence Artificielle. Je réalise actuellement une thèse dans ce domaine et plus précisément dans le domaine de l'apprentissage et de la contrainte. Je réalise cette thèse chez Exakis Nelite en collaboration avec le laboratoire de recherche en informatique de Lens (CRIL).

Python et sa bibliothèque standard

Python est un langage assez simple à apprendre. Il possède des structures de données de haut niveau et a une approche simple mais efficace de la programmation orientée objet. Le typage dynamique et sa nature interprétée le rendent idéal pour l'écriture de script. Il se révèle être aussi un langage extrêmement efficace dans des projets à plus grande échelle grâce aux nombreuses librairies et frameworks qui viennent l'enrichir.

La version actuelle de Python est la version 3.8. La version 2 est dépréciée depuis le 1er janvier 2020, il ne faut donc plus utiliser cette version pour tout nouveau projet.

Cet article n'a pas vocation à être un tutoriel pour apprendre à coder en Python. Nous supposons donc que la syntaxe, et les structures de contrôles sont connues.

Python comme beaucoup de langages possède une *bibliothèque standard*. Il s'agit d'un ensemble d'outils, de fonctions, de classes effectuant des calculs, des opérations et actions de base. Le but étant de ne pas réinventer la roue ! Si les développeurs de Python ont implémenté ce que vous souhaitez faire ce sera très généralement plus efficace que si vous l'écrivez vous-même (en particulier d'un point de vue temps). Si l'on souhaite faire une racine carrée, nous n'allons pas écrire une fonction pour faire une racine carrée, non, nous allons utiliser la fonction *sqrt* de la bibliothèque *math* de la bibliothèque standard de Python. Idem si l'on souhaite trier une structure de données, nous pouvons utiliser la fonction *sorted* ou la fonction *sort* de la structure elle-même. Dans cet article je présenterai d'abord l'interaction avec le système grâce aux modules *sys*, *os* et *shutil*, la gestion des expressions régulières avec le mode *re*, nous enchaînerons avec un peu d'aléatoire, la manipulation des dates avec le module *datetime*. Ensuite nous ferons quelques requêtes *HTTP*. Nous verrons également les structures de données que le module *collections* apporte. Enfin nous terminerons par un exemple de programme multithreadé.

L'interaction avec le système et les fichiers

Une première interaction entre votre script et l'extérieur peut se faire avec les arguments de la ligne de commande.

python3.8 mon_script.py one two three

Pour récupérer les arguments de la ligne de commande, il faut utiliser le module *sys*. 1

Les arguments seront disponibles dans le tableau *argv* du module *sys*. Remarquons que le premier élément du tableau est le nom du script. L'utilisation de *sys.argv* est très bien pour un usage simple. Nous verrons plus loin l'utilisation d'un autre module pour faire une *CLI* (command line interface) plus riche.

C'est dans le module *sys*, que nous retrouvons les fichiers que chaque processus ouvre à son lancement, à savoir : *stdout*, *stdin* et *stderr*, respectivement la sortie standard, l'entrée standard et la sortie standard des erreurs. 2

Pour écrire dans les différentes sorties, nous pouvons simplement utiliser la fonction *print*. Par défaut cette dernière utilise *sys.stdout*, mais si nous précisons l'option *file*, il est possible de changer vers la sortie standard des erreurs ou vers un fichier qui aurait été préalablement ouvert avec la fonction *open*.

Il est possible d'utiliser la fonction *write* comme présenté dans le code précédent pour écrire dans une ou l'autre des sorties. Cependant cette méthode est dite bas niveau (au sens système). Il vaut mieux donc, lorsque c'est possible, utiliser la fonction *print*.

Enfin pour lire des lignes depuis *sys.stdin* il suffit d'utiliser *input()*. L'ajout d'une chaîne de caractères, comme présenté dans le code plus haut, permet d'afficher cette chaîne juste avant l'entrée clavier et ainsi précisez ce que l'utilisateur doit rentrer.

La fonction *open* permet justement d'ouvrir un nouveau fichier. Celle-ci est directement disponible, il n'est pas nécessaire d'importer une librarie spécifique. Cette fonction ne doit pas être confondue avec la fonction *open* du module *os* (i.e *os.open*). En effet cette dernière propose une interaction plus bas niveau avec le fichier, comme nous le ferions en **C**. Ici je me limiterai à la fonction *open* normale. 3

Dans ce code, nous ouvrons un fichier en écriture (option *mode=w*), puis grâce à la fonction *print* et à l'option *file* nous écrivons dans le fichier "file.txt". Par défaut l'option *mode* est en mode lecture (i.e *mode='r'*). Les autres options possibles sont *mode='a'* lorsque nous souhaitons écrire dans le fichier sans supprimer le contenu (l'option *w* supprime ce qui était dans le fichier). La dernière option possible est *mode='x'* permettant de créer un nouveau fichier et de l'ouvrir en écriture. L'option *x* lancera une *FileExistsError* si un fichier existe déjà.

Maintenant nous savons comment faire des opérations de base avec des fichiers, à savoir créer, lire et écrire. Voyons comment nous pouvons faire les opérations de copie, de déplacement et de modification de propriétaire.

Le module permettant de faire cela est *shutil*. 4

Le code précédent parle de lui-même. Pour plus d'information il ne faut pas hésiter à

```
import sys
print(sys.argv)
['mon_script.py', 'one', 'two', 'three']
```
1

```
import sys

# Pour stdout
print("Du texte").
sys.stdout.write("Du texte")

# Pour stderr
print("Du texte",file=sys.stderr)
sys.stderr.write("Du texte")

# Pour stdin
name = input("name ? ")
```
2

```
f = open("file.txt",mode="x")
print("Du texte2",file=f)
```
3

```python
import shutil

# COPIE
shutil.copyfile('data.db', 'archive.db')

# DEPLACEMENT
shutil.move('data.db', 'archive.db')

# MODIFICATION PROPRIETAIRE
shutil.chown('data.db', user="dupont")

# MODIFICATION GROUPE
shutil.chown('data.db', group="dev")
```

4

```python
import os

# variables
print(os.name) # 'posix' pour unix et 'nt' pour windows

print(os.sep) # / ou \\

print(os.linesept) # '\r' ou '\n' ou '\r\n'
```

5

```python
import argparse

parser = argparse.ArgumentParser(prog = 'top',description = 'Show
    top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
```

6

```python
# source: https://docs.python.org/3/tutorial/stdlib.html
import re
re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
#['foot', 'fell', 'fastest']
re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
#'cat in the hat'
```

7

```python
# https://docs.python.org/3/tutorial/stdlib.html
import random
random.choice(['apple', 'pear', 'banana'])
# 'apple'
random.sample(range(100), 10)   # sampling
#[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
random.random()    # random float
# 0.17970987693706186
random.randrange(6)    # random integer chosen from range(6)
# 4
```

8

taper la commande suivante *help(shutil)* dans un interpréteur Python (en ayant importé le module au préalable).

Le module os propose l'accès à différentes variables et fonctions intéressantes. 5

En fonction du système sur lequel le script va être exécuté les valeurs des variables du module os vont être différentes.

Ce module dispose également de beaucoup de fonctions. Certaines sont aussi présentes dans le module *shutil* vu plus haut. Dans ce module c'est la version bas niveau qui est présente. Par exemple la fonction *chown* est présente mais elle ne prend en paramètre uniquement le *uid* ou *guid* de l'utilisateur et du groupe. Tandis que *shutil.chown* peut prendre également le nom. Nous retrouvons dans ce module toutes les fonctions exec* qui permettent de lancer des processus externes. Globalement dans ce module nous retrouvons un grand nombre de fonctions qui ne sont en réalité que des *wrappers* aux fonctions existantes en **C**.

Il existe un module en Python permettant de créer une véritable CLI, c'est à dire avec des arguments courts (i.e "-"), longs (i.e "–"), positionnels, des valeurs par défaut, un nombre d'arguments obligatoires, etc.

python3.8 top.py --lines=5 alpha.txt beta.txt

Pour faire ce genre de chose il faut utiliser le module *argparse*. 6

Le code précédent présente la création d'un objet *ArgumentParser*. Nous lui indiquons le nom de notre programme ainsi qu'une description (ligne 3). Nous ajoutons ensuite à notre objet la gestion d'arguments positionnels (ligne 4). L'argument *nargs='+* indique que nous voulons avoir au moins un argument positionnel. Ligne 5 nous ajoutons une option pouvant être courte (*-l*) ou longue (*–lines*). Si cette option n'est pas indiquée alors ce sera la valeur par défaut *10* qui sera utilisée.

Cette librairie se révèle extrêmement pratique lorsque l'on souhaite une interaction via la ligne de commande riche.

Les expressions régulières

Quel serait un langage sans une gestion des expressions régulières. Un langage très puissant et connu pour faire de la manipulation de fichiers et de chaîne de caractères est le Perl, en Python ce sera le module *re* qui nous apporte cette fonctionnalité et qui se révèle efficace. 7

Nous n'allons pas forcément rentrer dans le détail complet de ce module. Le code précédent présente deux méthodes du module *findall* et *sub* qui permettent de chercher l'ensemble des sous-chaînes "matchant" l'expression régulière et de remplacer le résultat d'une expression régulière par le résultat d'une autre respectivement.

Random

Nous allons voir ici des exemples d'utilisation du module *random*. 8

La première fonction *choice* présentée à la ligne 3, permet de choisir un élément dans un *iterable* (i.e *array*,*tuple*…).

La fonction *sample*, utilisée à la ligne 5 permet, ici, de générer un tableau de 10 cases dont les valeurs seront comprises entre 0 et 99 (*range* permet de générer des valeurs entre 0 et max-1).

La méthode *random* permet de simplement générer un nombre flottant aléatoire.

Enfin la méthode *randrange* permet de choisir un nombre parmi *range(max)*, max étant la valeur que l'on donne à la fonction *randrange*.

Requêtes HTTP

En Python, faire des requêtes *HTTP* est presque quelque chose de naturel. Nous avons à notre disposition deux modules fort

pratiques pour faire cela : *urllib* et *http*. 9
Dans le code précédent, nous faisons une requête *HTTP* vers un nom de domaine fictif. L'utilisation du mot clé *with* permet de faire en sorte de fermer automatiquement la ressource. En effet, ici, *response* peut être vu comme fichier, et comme tout fichier après l'avoir ouvert il faut le fermer. Python fournit un mot clé permettant de le faire pour nous.
La fonction *urlopen* permet donc de lancer une requête (ici de type GET) vers une url. Le retour est un objet de type *HTTPResponse*. Cet objet nous donne accès à plusieurs méthodes et champs. La variable *status* comme présenté dans le code, permet de récupérer le code de retour de la réponse. Il existe également un module en Python permettant de récupérer un ensemble de constantes représentant les codes de retours *HTTP* standards.
Cette librairie reste assez bas niveau dans la communication *HTTP*. Il est possible de faire autre chose que des requêtes de type *GET* mais ce ne sera pas détaillé ici. Si l'on souhaite faire des requêtes un peu plus complexes assez facilement, il faudra alors se tourner vers le module *requests* qui est disponible via la commande *pip*.
Un dernier exemple de code où l'on récupère cette fois du JSON : 10

Les dates avec *datetime*

Quel que soit le langage, les dates sont souvent un problème pour les manipuler: *api* mal pensée, arithmétique sur les dates compliquées... En Python cela reste, comme souvent, simple et agréable à utiliser. 11
Dans le code précédent, nous retrouvons les opérations de base que l'on souhaite faire avec les dates : les créer depuis une chaîne, ou via un objet, faire des calculs dessus et enfin restituer dans un format la date.
L'API nous propose d'abord la fonction *today* permettant de récupérer un objet *date* représentant la date du jour. Il est possible de générer une chaîne de caractères depuis cette date en utilisant la méthode *strftime* dont une partie des options de formats est présentée dans le code.
Nous pouvons bien évidemment faire des calculs sur les dates. Il est donc possible de soustraire une date à une autre. Si nous souhaitons faires des calculs plus précis, comme ajouter un jour, une semaine, un mois, des minutes ou autre, il faudra passer par le module *timedelta*.
Il existe le strict équivalent du module *date* mais pour les heures qui est *time*. Enfin si l'on souhaite un modèle permettant de mixer les deux, il suffira d'utiliser datetime. Notons d'ailleurs que la transformation d'une chaîne en *object* dans le code précédent se fait avec la fonction *strptime* du module *datetime* et retourne un objet *datetime*.

Structures de données

Les structures de bases

Python fournit de nombreuses structures de bases qui ne nécessitent pas d'import : 12

Le module collections

Le module *collections* apporte de nouvelles structures en plus des classiques dictionnaires, listes, ensembles et tuples. Je vais présenter deux structures de ce module.

```python
from urllib.request import urlopen
import sys
from http import HTTPStatus

with urlopen('http://www.domaine.com') as response:
    if response.status != HTTPStatus.OK:
        print("il y a un soucis", file=sys.stderr)
        for line in response:
            ....
    else:
        print("C'est ok")
```
9

```python
import urllib
import json
response = urllib.urlopen("http://api.domaine.com/user")
data = json.loads(response.read())
print data
```
10

```python
# https://docs.python.org/3/tutorial/stdlib.html
# dates are easily constructed and formatted
from datetime import date, datetime
now = date.today()
now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
# output: '12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of
    December.'

# dates support calendar arithmetic
birthday = date(1964, 7, 31)
age = now - birthday
age.days
# 14368


# datetime from string
unix_birthday = datetime.strptime("1970-01-01", "%y-%m-%d")
```
11

```python
# array
a = [i for i in range(10)]  # tableau 0-9

b = [0] * 10  # un tableau de 10 cases valant 0

# Dict
d = {"k": "v", "k2": 10}
print(d.items())  # output: [('k', 'v'), ('k2', 10)]

print(d.keys())  # output: ["k","k2"]

print(d.items())  # output: ["v",10]

# Tuple
point2d = (10, 20)
point3d = (10, 20, 30)

# set
s = set('a', 'b', 'c')
```
12

La première structure que propose le module *collections* est *namedtuple*. Cela permet de faire comme un *tuple* classique mais en nommant les champs. 13

Le code précédent présente ce qui est possible avec un *namedtuple*. Notons la réécriture d'un point en utilisant un *namedtuple*, cela permet de rendre le code plus lisible.

Enfin ce type de structure peut se révéler extrêmement utile lorsque nous avons un nombre fixe de champs, car contrairement à un objet ou à un dictionnaire il ne sera pas possible d'ajouter de nouveaux champs. Ainsi si nous souhaitons lire dans un fichier CSV, où en général le nombre de colonnes est fixe alors ce type de structure peut se révéler intéressante.

Une autre structure fort intéressante est *Counter*. Cette structure, comme son nom l'indique, permet de gérer facilement des compteurs. Cette structure est en réalité une extension de la structure de base *dict*. 14

Threading

Threading est une technique de découpage des tâches qui ne sont pas séquentiellement dépendantes. Les threads peuvent être utilisés pour améliorer la réactivité des applications qui acceptent les entrées utilisateur pendant que d'autres tâches s'exécutent en arrière-plan. Le code suivant montre comment le module de *threading* de haut niveau peut exécuter des tâches en arrière-plan pendant que le programme principal continue de s'exécuter : 15

Le principal défi des applications multithread est la coordination des threads qui partagent des données ou d'autres ressources. Pour cela, le module de *threading* fournit un certain nombre de primitives de synchronisation, notamment des verrous, des évènements, des variables de condition et des sémaphores.

Conclusion

Cet article a présenté les nombreuses fonctionnalités de Python, disponibles en standard sans rien installer de plus. Comme dit dans l'introduction, il se révèle idéal pour l'écriture de scripts, y compris de scripts système. Python ne se limite pas à l'écriture de script et peut si on le souhaite écrire une application plus complète de type Web avec le framework Django couplé à des frameworks comme Angular pour réaliser des applications Web.

13
```
# https://docs.python.org/3.8/library/collections.html#collections.namedtuple
import csv
import collections

# Basic example
Point = collections.namedtuple('Point', ['x', 'y'])
p = Point(11, y=22)     # instantiate with positional or keyword arguments
p[0] + p[1]             # indexable like the plain tuple (11, 22)
#output: 33
x, y = p                # unpack like a regular tuple

print(p.x + p.y)                # fields also accessible by name
#output: 33


# CSV

EmployeeRecord = collections.namedtuple(
    'EmployeeRecord', 'name, age, title, department, paygrade')

for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)
```

14
```
# https://docs.python.org/3.8/library/collections.html#collections.Counter
# Tally occurrences of words in a list
import re
from collections import Counter
cnt = Counter()
for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
    cnt[word] += 1

Counter({'blue': 3, 'red': 2, 'green': 1})

# Find the ten most common words in Hamlet
words = re.findall(r'\w+', open('hamlet.txt').read().lower())
Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

15
```
# https://docs.python.org/3/tutorial/stdlib2.html#multi-threading
import threading
import zipfile


class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)


background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

Félix Billion
Microsoft MVP - Developer Technologies

Sylvain Pontoreau
Microsoft MVP - Developer Technologies

Quoi de neuf dans TypeScript 4 ?

TypeScript est devenu très populaire dans le développement web ces dernières années. Peut-être que vous n'en avez jamais entendu parler, nous allons donc commencer par rappeler les principes de base de ce langage. Avant de parler des dernières évolutions.

3 principes :

1 - TypeScript est un sur ensemble de JavaScript. Une ligne de JavaScript est donc une ligne de TypeScript (l'inverse n'étant pas vrai) ;
2 - L'intérêt principal du langage est de rajouter des types et d'avoir ainsi une étape de vérification lors de la compilation ;
3 - L'étape de compilation proposée par TypeScript permet aussi de choisir la version d'ECMAScript qui sera produite à la fin. TypeScript peut donc générer du code supportant les dernières fonctionnalités de JavaScript dans des versions antérieures (on appelle ce concept le "downgrade" de code).

Toutefois, il est important de garder en tête que TypeScript ne garantit pas les types lors de l'exécution. C'est un point qu'il faut absolument retenir lorsque l'on débute avec le langage, si vous respectez correctement le typage et que vous ne forcez pas la main au compilateur (chose qui est possible via l'utilisation de certains mots-clés du langage), votre base de code JavaScript après compilation sera de meilleure qualité. De plus, lors de l'exécution de votre programme, vous n'allez pas avoir de bugs liés à des problèmes sur les types. Cependant, si vous lui forcez la main sur le typage, TypeScript ne vous garantira rien, car il ne fera pas certaines vérifications lors de la compilation, laissant potentiellement dans le code produit des bugs de typage. Les spécifications du langage sont étroitement liées à celle d'ECMAScript. Vous disposerez donc de toutes les capacités des dernières versions de JavaScript (classes, opérateurs rest/spread, arrow functions, bigint, optional chaining, ...). Pour démarrer avec TypeScript, tout dépend de ce que voulez faire. Si vous commencez un nouveau projet, rien de plus simple ! Il suffit d'installer le compilateur via NPM (globalement ou localement en tant que dépendance de développement) :

```
$ npm install -g typescript
```

Et ensuite, initialiser votre projet avec le compilateur (via la commande "tsc") :

```
$ tsc --init
```

Vous avez déjà un projet JavaScript ? Deux options s'offrent à vous :

- Initialiser TypeScript dans votre projet et changer l'extension de vos fichiers JavaScript en ".ts" ;
- Utiliser des commentaires dans votre code avec des types. TypeScript prend en charge l'interprétation de la JSDoc, et si vous utilisez Visual Studio Code, vous n'avez même pas à initialiser quoi que ce soit ! Il vous suffit simplement d'ajouter la directive @ts-check en commentaire dans un fichier pour que la vérification des types s'active sur celui-ci. Voici un exemple pour illustrer l'utilisation de cette directive :

```
// @ts-check
/**
* @param { number[] } numbers
*/
function sum(numbers) {
return numbers.reduce(() => {
  return previous + next;
});
}

/*
* Les valeurs "true" et "3" seront marqués en erreur par Visual Studio code,
* car elles ne sont pas du type number.
*/
const result = sum([true, 2, "3"]); //result === "33"
```

Généralement il est plus simple d'utiliser cette technique lorsque l'on a déjà un projet, car TypeScript n'aura pas d'impact direct sur le code que vous avez déjà produit. La vérification des types dans ce cas est donc plus une suggestion pour vous permettre de résoudre vos erreurs de typage au fil de l'eau (afin de vous préparer au mieux à faire le grand saut vers TypeScript).

Au moment de l'écriture de cet article, la version 4.0 du langage vient d'être publiée. Bien que ce soit une version considérée comme majeure, l'équipe derrière TypeScript ne suit pas forcément le versionnage sémantique. En conséquence, certaines versions mineures contiennent parfois des "breaking changes". De manière générale, les développeurs de TypeScript essayent au maximum de prendre en compte la rétrocompatibilité. Ce genre de changement est alors plus contraint par l'ajout de fonctionnalités au cœur du langage ou la fin du support de certaines capacités. On peut citer par exemple le retrait du support de la propriété "origin" dans les types de bases de TypeScript, propriété obsolète qui est uniquement supportée par Internet Explorer (dont le support va s'arrêter dans les prochains mois). Pour conclure sur ce point, une version majeure de TypeScript indique plutôt l'aboutissement d'un objectif après l'ajout de fonctionnalités au cours de la version précédente. La version 4.0 marque l'aboutissement de plusieurs mois de travail autour de l'expressivité, la productivité et la mise à l'échelle du code.

Mettre à jour de son projet vers TypeScript 4

Comme abordé précédemment, certaines versions mineures de TypeScript contiennent parfois des "breaking changes". En conséquence vous pouvez potentiellement vous retrouver avec des erreurs lors de la mise à jour vers la version 4 (tout dépend de quelle version vous utilisez actuellement). Avant de parler de nouveauté, nous allons aborder quelques "breaking changes" apparues depuis la version 3 du langage :

- Unknown : le type "unknown" a été introduit en version 3.0 en tant que mot-clé réservé dans le langage. Si vous aviez déclaré des variables avec ce nom, vous allez devoir les renommer ;

- This : lors de son utilisation dans le scope global, "this" n'est plus typé avec "any", mais avec "typeof globalThis". Ce changement introduit en 3.4 peut déclencher des erreurs si l'option de compilation "noImplicitAny" est activée, ce qui n'était pas le cas dans les versions précédentes ;
- Générique : à partir de la version 3.5 du langage, les génériques n'ayant pas de contraintes sont implicitement contraints avec "unknown". Auparavant ils étaient considérés comme des objets vides et donc typés avec "{ }". Ce changement peut avoir des conséquences, car il vous sera alors impossible d'utiliser certaines méthodes liées du type "Object" (par exemple ".toString()") ;
- Omit : ce type utilitaire est très utilisé par les développeurs React, mais il n'était pas fourni de base dans TypeScript. Il a ensuite été introduit en 3.5 et provoque donc maintenant des erreurs si votre code contient déjà une implémentation de ce type ;
- "constructor" : la spécification ECMAScript a légèrement évolué sur les constructeurs de classe. À présent une méthode nommée "constructor" est automatiquement considérée comme un constructeur. Ce changement a été implémenté en version 3.6. Si vous utilisez ce genre de déclaration en plus d'un constructeur de classe, vous allez être contraint de trouver une autre solution, car il n'est plus possible de compiler ce type de code :

```
class Person {
  // Error: Multiple constructor implementations are not allowed.
  "constructor"(name: string) {
    console.log(`Hello ${name}`);
  }

  constructor() {

  }
}
```

- Get/set : depuis la 3.9, les assesseurs générés dans une classe ne sont plus énumerables. Sur ce point, TypeScript n'était pas conforme à la spécification ECMAScript ;
- Delete : à partir de la version 4.0, l'utilisation de l'opérateur "delete" ne peut être appliquée qu'à des propriétés de type "any", "unknown", "never" ou optionnelle. Cela permet de garantir que le typage reste intègre après la suppression d'une propriété à l'exécution :

```
interface Person {
name: string;
email: unknown;
}

function deleteSomeProps(x: Person) {
// error (The operand of a 'delete' operator must be optional.)
delete x.prop;
// ok
delete x.email;
}
```

Cette liste n'est bien sûr pas exhaustive ! Durant ces deux dernières années, l'équipe de TypeScript a introduit beaucoup d'autres "breaking changes", notamment sur le typage du DOM, mais aussi dans le système de typage avancé du langage. Il faudrait un article entier rien que pour aborder tous ces changements, il est donc important d'aller en prendre connaissance via le blog TypeScript (https://devblogs.microsoft.com/typescript/). Un post est disponible pour chaque version et contient une section avec l'ensemble des "breaking changes" introduits.

Quelques nouvelles features de TypeScript 4

Labeled tuple

En TypeScript, les tuples ont pour objectif de déclarer des tableaux contenant des valeurs dont les types sont différents. Les valeurs récupérées ensuite via l'index du tableau (ou par déstructuration) sont ensuite correctement typées :

```
type Person = [string, string, number];

function logPerson(person: Person) {
 const [
  firstName, // string
  lastName, // string
  age // number
 ] = person;

 console.log(`
  FirstName: ${firstName}
  LastBane: ${lastName}
  Age: ${age}
 `);
}

logPerson(['Bill', 'Gates', 64]);
```

Ils sont utiles pour créer des structures de données sous la forme d'un tableau typé, mais leur utilisation est encore limitée par le fait qu'ils sont par nature peu lisibles. C'est pourquoi TypeScript 4.0 introduit une nouvelle capacité liée au tuple, le concept des "labeled tuples". Il est à présent possible de définir un libellé pour chaque type contenu dans le tuple (et dès qu'un type dispose d'un libellé l'ensemble des autres types doivent aussi en avoir un) :

```
type Person = [firstName: string, lastName: string, age: number];
```

En termes de développement cela ne change rien, mais cet ajout fait la différence au niveau de votre environnement de développement. Les libellés seront par la suite utilisés par Visual Studio Code pour vous donner des informations plus précises sur la structure du tuple. Voici un exemple :

```
type Person = [firstName: string, lastName: string, age: number];

function logPerson(...person: Person) {

}
logPerson()
```

logPerson(firstName: string, lastName: string, age: number): void

Sans les libellés, Visual Studio Code vous propose ceci via l'auto-complétion :

```
type Person = [string, string, number];

function logPerson(...person: Person) {

}
logPerson()
```

logPerson(person_0: string, person_1: string, person_2: number): void

En termes d'expérience développeur, l'amélioration de lisibilité sur la structure du tuple est un gain vraiment intéressant et cela ne coûte pas plus cher de les mettre. Gageons qu'il y aura probablement d'autres fonctionnalités autour de ce concept dans les prochaines versions de TypeScript !

Variadic tuple

Dans les précédentes versions de TypeScript, certaines déclarations pouvaient être compliquées à typer. Prenons un exemple :

```
type Concat<T extends unknown[], U extends unknown[]> = (t:T, u: U) => unknown[];
```

Ce type accepte deux génériques ayant pour contrainte d'être des tableaux. La fonction implémentant ce type doit retourner la concaténation des deux tableaux. Dans cet exemple, il est compliqué de connaître exactement les types contenus dans le tableau qui est retourné. Pour améliorer cela, l'équipe de TypeScript a introduit le concept de "Variadic Tuple". Cette nouvelle fonctionnalité ajoute aux tuples la capacité d'extirper les types contenus dans un générique via l'opérateur "spread". En reprenant l'exemple précédent, on obtient avec les "Variadic Tuple" le résultat suivant :

```
type Concat<T extends unknown[], U extends unknown[]> = (t:T, u: U)
=> [...T, ...U];
```

Une fois cette capacité mise en place dans le retour de la fonction, le contenu du tuple retourné sera typé correctement :

```
type Concat<T extends unknown[], U extends unknown[]> = (t:T, u: U) => [...T, ...U];

type Person = [firstName: string, lastName: string, age: number];
type Address = [street: string, city: string, zipCode: string];

    type PersonAndAddress = (t: Person, u: Address) => [firstName: string, lastName:
    string, age: number, street: string, city: string, zipCode: string]
type PersonAndAddress = Concat<Person, Address>;
```

Inférence du type des propriétés depuis le constructeur

Le compilateur de TypeScript peut désormais déduire le type des propriétés d'une classe qui ne possède pas d'annotation de type, à condition que celles-ci soient initialisées dans le constructeur :

```
class Person {

    (property) Person.firstName: string
    firstName;

    constructor(firstName: string) {
        this.firstName = firstName;
    }
}
```

Avant la version 4, les propriétés d'une classe ne possédant pas d'annotation et n'étant pas initialisées, étaient implicitement typées avec "any" :

```
class Person {

    (property) Person.firstName: any
    Member 'firstName' implicitly has an 'any' type.
    Quick Fix...    Peek Problem
    firstName;

    constructor(firstName: string) {
        this.firstName = firstName;
    }
}
```

Pour rappel, le type "any" est un type propre à TypeScript qui permet d'indiquer au compilateur de ne faire aucune vérification de type sur un élément, ce qui peut potentiellement laisser passer des erreurs qui auraient pu être détectées lors de la compilation.

Nouveaux opérateurs d'assignement

Grâce au "downgrade" de code lors de la compilation, TypeScript permet d'utiliser des instructions avant même qu'elles ne soient intégrées dans les spécifications d'ECMAScript et donc avant d'être implémentées dans les navigateurs. Dans la version 3.7 de TypeScript (paru en novembre 2019), deux nouveaux opérateurs ont ainsi fait leur apparition : "optional chaining" et "null coalescing". Ces opérateurs ont par la suite été intégrés dans les spécifications d'ECMAScript 2020. Il existe nombre d'opérateurs d'assignement permettant de condenser l'écriture en JavaScript :

```
// Addition
// a = a + b;
a += b;

// Soustraction
// a = a - b
a -= b;

// Multiplication
// a = a * b
a *= b;

// ...
```

Dans la version 4.0 de TypeScript, plusieurs nouveaux opérateurs d'assignement sont utilisables. Il est désormais possible d'utiliser les syntaxes raccourcies avec les opérateurs ||, && et ??.

```
// a || (a = b);
a ||= b;

// a && (a = b);
a &&= b;

// a ?? (a = b);
a ??= b;

// ...
```

Ces nouveaux opérateurs ne sont que des sucres syntaxiques visant à réduire et simplifier la lecture du code :

```
// TypeScript 4.X
obj1.obj2.obj3.x ??= 42;

// TypeScript 3.x
obj1.obj2.obj3.x = obj1.obj2.obj3.x ?? 42;
```

Deprecated

La JSDoc permet d'annoter une déclaration comme étant dépréciée et par conséquent ne devant plus être utilisée :

```
class Person {

  /** @deprecated */
  logPerson() {
    // ...
  }
}
```

Avec la version 4.0 de TypeScript, les IDE ont désormais accès à cette information. Dans Visual Studio Code, les méthodes dépréciées d'une instance sont affichées de manière barrée :

```
class Person {

    /** @deprecated */
    logPerson() {
      // ...
    }
}

const p = new Person();
```

(method) Person.logPerson(): void

@deprecated

'(): void' is deprecated ts(6385)

No quick fixes available

```
p.logPerson();
```

Template literal type et la clause 'as' des mapped type

Prévu pour la prochaine version de TypeScript (v4.1), le concept de "template literal type" permet d'utiliser les templates littéraux au sein des définitions de type.

```
type Async<T extends string> = `${T}Async`;
type ReadFileAsync = Async<'read'>; // 'readAsync'
type FileAsync = Async<'read' | 'write'>; // 'readAsync' | 'writeAsync'
```

Des mot-clés spécifiques comme : "uppercase", "lowercase", "capitalize" ou encore "uncapitalize" seront utilisables afin d'appliquer une transformation sur une chaîne passée en paramètre.

```
type TextTransform<T extends string> = `${uppercase T} ${lowercase T}`;
type Text = TextTransform<'Hello'>; // 'HELLO hello'
```

Les "mapped types" permettent d'itérer sur les propriétés d'un type et de les modifier, créant ainsi un nouveau type. Grâce au "template literal type", la nouvelle clause "as" permettra d'exprimer des transformations sur le nom des membres d'un objet.

```
type Async<T> = {
  [P in keyof T & string as `async${capitalize P}`]: Promise<T[P]>
};
// { asyncRead: Promise<string>, asyncWrite: Promise<boolean> }
type FileAsync = Async<{ read: string, write: boolean }>;
```

Ces deux nouvelles fonctionnalités très attendues par la communauté permettront de simplifier l'écriture de certains types complexes et d'exprimer de nouveaux types plus précis. Il sera par exemple possible de typer une requête SQL inline, le type permettrait de vérifier que certain mots-clés sont présents dans la requête.

Refonte du site et de la documentation

L'ancien site web officiel de TypeScript ayant plus de huit ans, il était temps qu'une refonte plus moderne soit proposée. Ce nouveau site officiel offre donc une charte graphique entièrement revue, une nouvelle navigation, mais surtout une nouvelle documentation ainsi qu'un nouveau Playground (bac à sable permettant de tester du code TypeScript). 1

Depuis sa sortie en 2012, la documentation officielle de TypeScript a longtemps été critiquée et décrite comme peu accessible et mal organisée. Cette nouvelle documentation a été structurée de façon plus intuitive, possède des exemples directement validés par le compilateur de TypeScript et sera régulièrement mise à jour afin de suivre les évolutions du langage. Le précédent Playground était très limité : difficultés à configurer correctement les options du compilateur, peu d'exemples prêts à l'emploi, pas de sortie console, … Le tout nouveau Playground corrige les défauts de son prédécesseur et offre aussi quelques nouveautés :

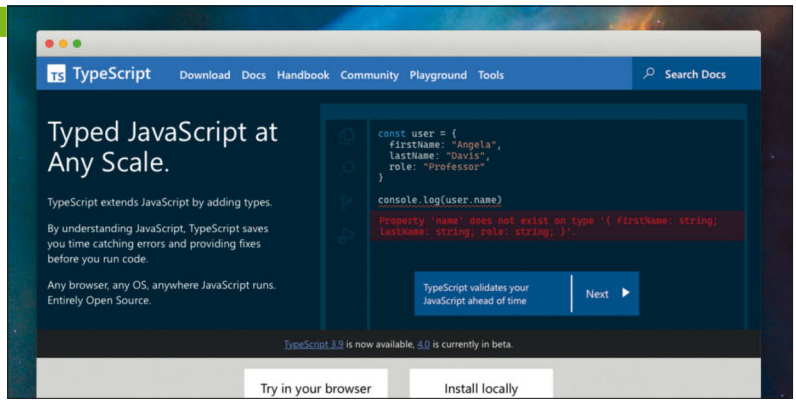
- Aide pour exporter son code ;
- Plusieurs sorties possibles dont : la console, les fichiers de définitions générés, les erreurs, … ;
- Une interface revue pour configurer ses options de compilations ;
- La possibilité d'ajouter des plugins afin d'étendre le comportement du Playground.

Conclusion

Bien que la popularité de TypeScript ne soit plus à prouver, le langage ne se repose pas sur ses lauriers. Dans cette version 4.0, beaucoup de fonctionnalités très demandées par la communauté ont été ajoutées, permettant ainsi de simplifier l'expression de certains types et de créer des types jusqu'alors impossibles.

Beaucoup d'autres nouveautés sont attendues dans la suite de la roadmap (types conditionnels récursifs, support du mot-clé override, …). La prochaine version mineure (4.1) est prévue pour novembre 2020 et les nouvelles fonctionnalités qu'elle va apporter commencent déjà à avoir un impact important dans les communautés, qui se projettent déjà dans de nouveaux scénarios de typage (jetez un coup d'œil à ce dépôt GitHub pour vous en convaincre : https://github.com/codemix/ts-sql) ! Si vous souhaitez échanger sur ce sujet n'hésitez pas à rejoindre le slack de la communauté Paris TypeScript (https://slackin.typescript-paris.now.sh/) ainsi qu'à vous inscrire sur le meetup (https://www.meetup.com/Paris-Typescript/). La communauté se réunit tous les deux mois (actuellement en ligne), nous aurons donc probablement l'occasion de revenir sur toutes ces nouveautés.

1



Jon Mikel Inza
Référent IoT et Azure
De formation technique mais avec un parcours varié, je travaille aujourd'hui en tant que Référent IoT et Azure. Mon rôle principal est d'aider nos clients à construire des solutions cohérentes et optimisées par rapport à leurs besoins et objectifs.

Série IoT : « Welcome to the real world »

PARTIE 1 : DÉFINITION

Dans un numéro précédent, nous avons présenté l'IoT depuis une perspective générale, en décrivant certaines des approches SaaS (Software As A Service) et PaaS (Platform As A Service) disponibles sur le marché.

Ce nouvel article vient enrichir cette première vision en décrivant la segmentation des domaines dans l'IoT et le potentiel de cet univers. L'article décrit aussi ce que représente la transformation digitale générée par le monde des objets connectés et évoque certaines des problématiques rencontrées lors de ces transformations. Pour terminer, une section pratique expose l'implémentation d'une solution PaaS. Ce cas pratique s'ajoute à la mise en place d'une solution SaaS décrite dans l'article antérieur. Les deux approches seront ainsi couvertes par l'ensemble des publications. Tout comme pour le premier article, le socle technologique de fond sera Azure et les services IoT d'Azure. Il existe d'autres possibilités sur le marché mais, sans faire ici un comparatif d'Azure et de ses concurrents, nous continuons sur Azure pour assurer la cohésion dans la rédaction des articles et pour la valeur ajoutée apportée par ces services.

Valeur de l'IoT

L'IoT est l'un des sujets d'actualité depuis quelques temps et pourrait soulever différents types de questions. Par exemple : est-ce juste un effet de mode ? Est-ce que l'IoT est de passage ? Si le sujet est vraiment important, pourquoi ? Où est son potentiel ou sa valeur ?
A vrai dire, connecter des objets ou des capteurs dans le but de récolter de l'information ou faire transiter des messages (ou commandes dans certains cas) n'est pas tout nouveau. En effet, des domaines comme l'industrie connaissent bien ce type de solutions depuis déjà plusieurs dizaines d'années (solutions M2M, Machine To Machine ou autres). Néanmoins, depuis quelques années, l'IoT prend une autre dimension. L'évolution technologique permet désormais des scénarii très riches, avec des TTM (Time To Market) plus courts et à des coûts d'acquisition qui changent le décor. Mais encore une fois, quelle est la vraie valeur de l'IoT ?

Où est le potentiel ?

L'aspect immédiat qui en ressort est la valeur que représente le pouvoir de récolter de la donnée de qualité et d'interagir avec les objets très facilement. Néanmoins, l'intérêt va bien au-delà, en étendant le spectre des perspectives business et fonctionnelles.
Certains des éléments qui caractérisent l'IoT actuel sont :

- Récolte de données (volume, qualité et rapidité) ;
- Possibilités de communication bidirectionnelle (D2C – Device To Cloud – et C2D – Cloud To Device-) ;
- Analytique, décisionnel et stratégies business plus riches, précis et dynamiques ;
- Scénarii fonctionnels riches et dotés d'intelligence (même dans les objets) ;
- Diminution des coûts par rapport aux solutions antérieures ;
- Réduction importante du TTM (Time To Market) ;
- Variété dans les solutions de communication (protocoles, supports non filaires, etc.) ;
- Sécurité qui fait partie des solutions dès les phases de design ;
- Abstraction et absorption des complexités par les différentes technologies et solutions ;
- Différents types de matériel permettant de couvrir des spectres fonctionnels larges ;
- Richesse, évolutivité et ouverture des architectures
- Plateformes, solutions et outils de développement matures ;
- Possibilités d'intégration des systèmes.

Tout ceci crée un écosystème riche avec un certain niveau de maturité où pouvoir évoluer. Comme mentionné précédemment, l'apport en valeur est concret. Son impact peut atteindre des amplitudes importantes. Cela donne lieu à des transformations digitales d'envergure, souvent qualifiées de « révolutions silencieuses ».

Secteurs

L'IoT offre trois axes de valeur :

- De la donnée ;
- De la réactivité dans les systèmes (décisionnel inclus) ;
- De l'intelligence dans les systèmes.

Ces axes sont d'un intérêt indéniable dans de nombreux secteurs d'activité, ce qui a rendu l'intégration de l'IoT relativement naturelle (parfois même très profonde et/ou verticale).

CIoT, Consumer IoT

Il s'agit probablement du secteur le plus accessible et le plus connu par le grand public. Ce domaine englobe des secteurs comme les wearables (ex : montres connectées) et la domotique (ex : maison connectée).

IIoT, Industrial IoT

L'IoT fait partie de l'industrie depuis un certain temps. Initialement sous la forme de M2M (M2M, Machine To Machine), l'IoT n'a cessé d'évoluer et aujourd'hui, les objets connectés sont devenus indispensables dans ce secteur.

Energie

L'énergie est un autre domaine où le taux de pénétration est très grand. La richesse des solutions d'aujourd'hui ont débloqué des scénarii et étendu l'horizon des possibilités en termes de création de valeur. Quelques exemples :

- Des « fermes » de panneaux solaires ont aujourd'hui la possibilité d'optimiser le rendement en jouant sur l'inclinaison et l'orientation de panneaux solaires en fonction de la météo et position du soleil.
- Gestion des parcs éoliens (maintenances prédictives, optimisation des rendements, etc.)

Logistique et Retail
Les nouveaux types de capteurs, protocoles, objets autonomes et riches en intelligence, architectures, capacités d'intégration avec les SI existants apportent une valeur sans précédent. Les informations extraites des données récoltées sont riches en valeur pour améliorer les processus, apporter de l'intelligence de proximité (Edge) et augmenter radicalement les niveaux de réactivité des solutions.

Transport
Le transport est encore un secteur où l'IoT s'est développé de manière naturelle. La localisation et la gestion de flottes en temps réel avec en plus, de la récolte d'informations de types divers permet des modes opérationnels (optimisation des ressources, diminution des interruptions de service, diminution des temps d'attente, maintenances prédictives, etc.) et des business models plus efficients.

Automobile
Pour l'instant, l'IoT n'a pas fait une percée généralisée sur le marché automobile de grande consommation. Néanmoins, on sent qu'elle est en cours de préparation et que la vague arrive. Les véhicules haut de gamme intègrent déjà de nombreux services reposant sur l'IoT. La démocratisation de ces services n'est qu'une question de temps.

AIoT
L'AIoT peut faire référence à deux domaines différents : AI IoT, Agricultural IoT

AI IoT
L'IoT et l'AI (Artificial Intelligence) sont très liés et s'alimentent réciproquement. La création d'intelligence a lieu aussi bien côté cloud (Intelligent Cloud) que côté équipement (Intelligent Edge).
Certains modèles d'intelligence cloud peuvent même être déportés vers l'Intelligent Edge de manière relativement simple. Ces mécanismes permettent de créer des scénarii à forte valeur.

Note
L'intelligence à laquelle nous faisons référence peut être sous forme d'AI mais d'autres formes également.

Agricultural IoT
Le secteur agricole ne cesse de se moderniser et a très rapidement intégré les axes de valeur proposés par l'IoT.
Quelques cas d'utilisation :
- Serres « intelligentes » ;
- Systèmes d'irrigation « intelligents » ;
- Tracteurs totalement contrôlés à distance (ou auto pilotés) en optimisant leurs parcours ;
- Intégration de services météo pour optimiser la gestion des opérations ;
- Sondes contrôlant différents indicateurs chimiques dans le sol, dans l'air ou dans la production ;
- Intégration de systèmes d'imagerie (caméras fixes ou drones) permettant d'analyser les sols ou la production ;
- Utilisation de drones pour des opérations de reconnaissance ou de gestion de semences.

HIoT, Health IoT
La santé est un autre secteur où l'IoT trouve une place naturelle. Les problématiques sont très particulières (vie privée, sécurité, etc.) mais des solutions existent aujourd'hui.
Quelques cas d'utilisation :
- Récolte de signaux biométriques ;
- Télémédecine ;
- Gestion intelligente et autonome de dispositifs pour diabétiques.

Ces types d'usages peuvent prendre une importance particulière dans les contextes que nous vivons actuellement.

IoMT, Internet of Medical Things
Ce domaine est très lié au domaine précédent (HIoT). Nous en faisons une mention à part car la verticalisation est forte et il existe des standards propres à ce domaine.

Smart Buildings
Au-delà des aspects purement domotiques, l'IoT a permis aux architectes et au secteur de la construction d'évoluer vers des bâtiments récoltant des données, « s' adaptant et agissant » par rapport à un contexte donné et exploitant au mieux les ressources (ex : salles, parkings, bureaux, ascenseurs, consommations, ventilation, climatisation, éclairage, sécurité, incidents, qualité de l'air, bien être, etc.). On parle de bâtiments « intelligents » (Smart Buildings).

Smart Cities
Au-delà des bâtiments et par extension logique, l'IoT s'est également développé à une échelle plus grande au niveau des villes. Encore une fois, les possibilités offertes par l'IoT rendent les villes « intelligentes « (Smart Cities) et offrent des possibilités sans précédents :
- Gestion des ressources et infrastructures ;
- Optimisations des coûts/ressources/infrastructures ;
- Meilleure gestion des incidents (réactivité, anticipation, diminution) ;
- Automatisations ;
- Analytique et décisionnel plus précis ;
- Maintenances prédictives ;
- Anticipation et gestion des pics de pollution par croisement d'informations (météo, saisonnalités, accidents, travaux, etc) ;
- Etc.

Synthèse
L'Internet des Objets d'aujourd'hui crée de la valeur à différents niveaux :
- De la donnée (volume, diversité et rapidité) qui alimente une analytique de qualité et une meilleure extraction d'intelligence ;
- Gestion et interactions bidirectionnelles avec les objets ;
- Intégration entre les différents systèmes ;
- L'extraction et la propagation d'intelligence.

L'IoT est venu pour rester et fait déjà partie intégrante de notre quotidien dans différents domaines.

Problématiques
L'IoT couvre un spectre large en termes de possibilités, ce qui l'expose naturellement à une diversité importante de problématiques. Dans le passé, certains de ces sujets ont été un frein à son adoption. Aujourd'hui, le contexte technologique offre des possibilités pour résoudre ou contourner certains de ces obstacles.

Techniques
Les problématiques les plus immédiates sont souvent techniques.

Provisioning

Le provisioning est le procédé d'enregistrer un objet IoT dans un système ou solution IoT. Lors de cette étape, la solution IoT génère une identité digitale unique pour l'objet connecté. Ce dernier peut ensuite utiliser cette identité pour communiquer et interagir avec le système.
Cette étape soulève certaines questions (techniques et fonctionnelles) :

- Est-ce que l'utilisateur final va devoir faire cette opération ?
- Si oui, comment ?
- Si non, doit-on le faire à l'usine ? Est-ce possible ?
- Doit-on privilégier un provisioning à distance ?
- Doit-on passer par un service de provisioning dédié permettant de mettre des niveaux d'industrialisation plus évolués ?
- Comment gérer un reprovisionning ?
- Quel niveau et type de sécurité devons-nous mettre en place (stockage des credential, type de communication, etc.) ?
- Envisage-t-on de permettre la migration d'objets connectés. Si oui, comment ?
- Etc.

Aujourd'hui et dans la mesure du possible, le provisioning devrait pouvoir être fait à distance, le plus simplement possible (frein potentiel d'adoption si trop compliqué) et de manière scalable.
Le marché est déjà mature sur ces sujets et répond aujourd'hui à toutes ces questions :

- Directement dans la passerelle IoT (ex : Azure IoT Hub) ;
- Via un service de provisioning (ex: Azure IoT Hub Device Provisioning Service).

Mise à jour des devices (firmware)

La nature très variée des objets connectés et leur éventuelle distribution géographique peuvent devenir un vrai sujet quand il s'agit de mettre à jour les firmwares (ou autres composants applicatifs). La standardisation des protocoles de communication et leur intégration dans les plateformes technologiques minimisent l'impact éventuel de ce point. Un service comme Azure IoT Hub permet de gérer les notifications de mise à jour de firmwares par « flottes » de devices avec une gestion des status (en fonction des possibilités de chaque device et type de communication réseau). Les Twins d'Azure IoT Hub et la communication C2D (Cloud To Device) permettent d'imaginer des scénarii riches comme la gestion de flottes par le biais de critères techniques et/ou métier.

Note

L'exécution de la mise à jour du firmware doit être gérée par l'objet connecté lui-même. Le rôle d'Azure IoT Hub est de transmettre la notification (ou informations complémentaires pour la mise à jour comme la version, prérequis, emplacement du nouveau firmware) et récolter les status de mise à jour.

Mise à jour des configurations des devices

De manière similaire au point précédent, certains scénarii peuvent reposer sur la modification de la configuration d'un ou plusieurs objets. La gestion de ces configurations sans un accès physique, de manière sécurisée et scalable est vitale. Les Twins et la communication C2D sont l'un des éléments permettant de répondre à cette problématique.

Opérations à distance

De plus en plus de cas d'utilisation exigent la possibilité de déclencher des actions dans les devices sans qu'ils ne soient accessibles physiquement. Exemples d'actions :

- Redémarrage ;
- Commandes opérationnelles (prise de mesure, actionner un moteur/interrupteur ou tout autre mécanisme, etc.).

Les Direct Methods dans Azure IoT Hub permettent d'implémenter ces cas d'utilisation.

Volumétrie des données

Au-delà des problématiques liées directement aux devices, les projets IoT peuvent être confrontés à d'autres types de difficultés.
La quantité d'informations émises par les objets génère des volumétries très importantes. Quoi stocker, comment stocker et où stocker (et à quel coût) deviennent du coup de vrais sujets. Azure inclut différents services adaptés à chacune des problématiques.

Ingestion de données

Lié à la volumétrie mais sous un autre aspect, l'IoT peut générer des millions de messages dans des fenêtres temporelles courtes (saisonnalité, incidents, etc.). Même si la plupart du temps les messages IoT ne font que quelques Ko, concevoir des solutions capables de recevoir des milliers, centaines de milliers voire millions de messages dans un espace de temps réduit est un vrai sujet. En plus de cette notion de pics de charge, il est également nécessaire de réfléchir à la réactivité fonctionnelle de notre solution dans de tels contextes. Azure IoT Hub, Azure Event Hub, Azure Stream Analytics, Azure Service Bus, Azure Functions, App Services (ou de plus en plus, Azure Kubernetes Services) sont certains des services dans la plateforme Azure pouvant être utilisés pour répondre à ces problématiques.

« Smart storage » Vs « Full storage »?

Certaines des architectures IoT gardent tous les messages pour qu'ils soient traités ultérieurement (processus de contrôle, validation, purge, enrichissement, etc.). Une autre approche consiste à intégrer une partie de ces traitements au fur et à mesure que les messages arrivent. L'impact sur le type de repository data construit et tout ce qui en dépend est significatif. Quelle que soit l'option choisie, Azure intègre des services pour répondre aux deux types d'architecture.

Cold path, Warm path, Hot path

Lié aux points précédents, comprendre quels sont les besoins fonctionnels et quelle est la stratégie business autour des données est vitale. Certaines données devront être structurées et accessibles très rapidement, d'autres pourront être structurées d'une autre façon et les temps de latence seront moins importants ou dans d'autres cas encore, on aura juste besoin de la donnée brute pour alimenter des data lakes ou autres. Cette distinction de flux a donné lieu à des concepts architecturaux (cold path, warm path ou hot path) en fonction du niveau de latence acceptable pour l'accès aux données. Une nouvelle fois, Azure inclut des solutions techniques dans les trois cas. Le sujet devra être adressé par l'analyse et la conception de l'architecture.

Sécurité

Historiquement, la sécurité a été l'un des freins à l'adoption généralisée de l'IoT et il reste aujourd'hui un sujet très important dans la plu-

part des projets IoT. Les services Azure IoT permettent aujourd'hui d'implémenter des mécanismes de sécurité à différents niveaux :
- Device ;
- Communication entre le device et la solution IoT (dans les 2 sens) ;
- Stockage des données ;
- Accès à la solution, aux données et au device ;
- Détection d'anomalies ou d'éventuelles failles de sécurité.

Certains des services Azure permettant de mettre en place ces mécanismes sont :
- Azure Sphere (OS, hardware et service implémentant une sécurité très forte) ;
- Azure IoT Hub (communications et accès sécurisés) ;
- Azure Key Vault (service pour protéger tout type de secrets) ;
- Azure IoT Security Center (supervision, détection d'anomalies, failles, etc.) ;
- Azure Maps (isolation et privatisation des données) ;
- Les différents services de stockage qui sécurisent la donnée au repos ;
- Etc.

Canaux bidirectionnels

L'évolution technologique et des cas d'usage rendent les exigences de plus en plus élevées. En ce sens, il est courant que les projets IoT requièrent la possibilité de mettre en place des canaux de communication bidirectionnels :
- Device vers le cloud (D2C – Device To Cloud) ;
- Cloud vers le device (C2D – Cloud To Device).

Azure IoT Hub intègre ceci de manière native.

Note

Les flux C2D restent dépendants du type de communication choisi (SigFox, LoRa, Wifi, etc.). Certains de ces choix peuvent ne pas être compatibles ou limiter le C2D.

Protocoles de communication

De même, la nature des communications IoT exige aujourd'hui que les protocoles de communication soient robustes et fiables.
Azure IoT Hub repose sur MQTT, AMQP et HTTPS.

Intelligent IoT Solutions

Nous avons l'habitude de voir des systèmes d'information où l'intelligence est centralisée. L'IoT amène des scénarii et des problématiques additionnels où la possibilité d'exploiter certains niveaux d'intelligence à proximité de la source de données serait avantageuse. Ceci a donné naissance à la notion d'Intelligent Edge.
Azure IoT intègre des réponses à ce sujet par le biais de :
- Azure IoT Edge, destiné aux devices IoT ;
- Azure Stack Edge, reposant sur du matériel nettement plus puissant (rack loué à Microsoft) et permettant d'enrichir les scénarii Intelligents Edge.

Une notion complémentaire est en train de se concrétiser : le Fog Computing, qui enrichit le Edge Computing par des éléments riches de « proximité (ex : processing de data dans une gateway) et sans aller jusqu'au Cloud Computing.

Production

Certaines des phases d'un projet IoT peuvent avoir besoin d'outils adaptés. Quelques exemples :
- Debug ;
- Packaging ;
- Déploiements (plateforme et solution) ;
- Tests ;
- Détection et correction d'anomalies ;
- Templates ;
- Etc.

Visual Studio et Visual Studio Code offrent de nombreuses extensions pour couvrir ces besoins.
De plus, il existe des outils ou publications open source qui complètent les possibilités offertes par les deux Visual Studio.

Dette technique et évolutivité

Les solutions IoT font souvent l'objet d'investissements importants (surtout les approches PaaS) qui doivent être suffisamment pérennes dans le temps. La dette technique et l'évolutivité doivent donc être gérées avec une attention particulière. Azure IoT offre un socle technique robuste et évolutif permettant de contenir cette dette technique.

Problématiques business

En plus des problématiques techniques, les projets IoT sont également confrontés à des problématiques business.
Ces sujets ne sont pas techniques et cela peut sembler étonnant de les inclure dans une publication technique. Néanmoins, il est important de comprendre l'ensemble des problématiques autour d'un projet afin de produire des architectures optimisées sur tous les aspects (création de valeur, coût, business, fonctionnel, technique, risques, évolutivité, dette technique, etc.).

Création de valeur

La création de valeur peut se concrétiser sous différentes formes mais elles sont toutes fondamentales.
Dans les projets IoT, il est vital de comprendre où est la valeur pour imaginer des solutions qui la créent au plus vite et à des coûts optimisés. Certains projets auront besoin d'un focus très important sur la donnée ; d'autres sur les devices ou d'autres encore, sur l'intégration des systèmes, l'analytique, l'intelligence, la sécurité ou l'Intelligent Edge. L'aspect métier et l'intégration des systèmes peuvent s'avérer très différents d'un projet à un autre.
L'utilisation d'accélérateurs peut aider sur ce point.
Certaines initiatives Open Source proposent des solutions intéressantes (cf plus loin dans l'article).

Note

L'utilisation d'accélérateurs peut avoir d'autres avantages comme :
- *L'industrialisation ;*
- *La contention du risque ;*
- *La réduction des coûts et des TTM (Time To Market) ;*
- *La gestion de la qualité ;*
- *Etc.*

Des éléments qui libèrent du temps pour concentrer l'effort autour de la production de valeur.

Business models

En complément à la création de valeur, la conception de business models adaptés est déterminante. Cette affirmation n'a rien de révolutionnaire. Or, la réalité montre qu'elle finit par ne pas être aussi « évi-

dente ». L'élasticité du cloud et les différentes possibilités offertes par les services Azure IoT permettent d'imaginer différentes solutions.

Equipes

En continuant notre parcours des différentes problématiques, après les aspects techniques et un passage rapide par la partie business, nous arrivons aux équipes de production.
La réalisation de projets IoT a naturellement besoin d'équipes aux compétences spécifiques. Dans le passé, l'expertise nécessaire était extrêmement poussée sur l'ensemble des disciplines couvertes par un projet IoT (en particulier, sur l'aspect technique). Aujourd'hui, les prérequis ont changé en atténuant certains des aspects de tels projets. Les différentes technologies et outils de production absorbent une partie importante des niveaux de complexité.
Les acteurs techniques continueront à être des éléments clé : d'abord les architectes, car ils interviendront à la conception de la solution et ensuite les développeurs, qui doivent intégrer l'ensemble des exigences techniques des projets IoT. Il faut néanmoins garder en tête que les business analysts, Product Owner, testeurs et autres membres des équipes IoT joueront également un rôle important.

Synthèse

L'IoT est un domaine très riche en opportunités. La richesse des problématiques techniques, fonctionnelles et business à couvrir est importante. Le contexte technologique actuel permet d'adresser la plupart des problématiques avec sérénité.
Les compétences restent un vrai sujet. Des outils, patterns, architectures et accélérateurs apportent des éléments sécurisants. Désormais, la technologie est rarement un élément limitateur ou maillon faible dans les projets. Les personnes et leurs compétences restent un élément clé, mais qui est maîtrisable.
La suite de l'article décrit comment créer une solution IoT PaaS qui repose sur Azure IoT Hub. Avant de démarrer la partie purement pratique, l'article expose ce service angulaire afin de vous apporter une meilleure perspective de la construction de la solution.

Azure IoT Hub

Azure IoT Hub est le service principal de l'offre IoT de Microsoft Azure. Il s'agit d'un service PaaS qui permet de :

- Gérer les objets connectés
  - Provisioning (éventuellement avec le service complémentaire Azure IoT Hub Device Provining Service, en fonction du besoin) ;
  - Activation/désactivation ;
  - Configuration ;
  - Twins ;
  - Sécurité ;
  - Jobs.
- Gérer la communication des devices
  - Device To Cloud (D2C) :
    - Messages ;
    - Routes ;
    - File uploads (extrêmement utile pour les devices ou gateways qui transmettent des messages par lots, médias, etc.) ;
    - Twin (Reported properties).
  - Cloud To Device (C2D) :
    - Messages ;
    - Direct Methods, qui consiste à "appeler" des méthodes dans les devices ;
    - Twin (Desired properties).
- Gérer l'ingestion des messages
  - Réception des messages D2C ;
  - Routes ;
  - Enrichissement des messages ;
  - Custom endpoints (Event Hub, Storage, Service Bus Queue, Service Bus Topic).

Il s'agit de la brique centrale de toute solution IoT sur Azure.
Le service est robuste et scalable. Il offre différents types de configuration (SKUs) permettant d'optimiser les coûts à la volumétrie (device et messages) et les fonctionnalités nécessaires (Twins, C2D, etc.). Le service intègre aussi des fonctionnalités de failover. À l'heure où ces lignes sont écrites, la gestion de cette fonctionnalité est encore manuelle. Nous espérons tous qu'elle puisse être industrialisable dans un avenir proche.
Azure IoT Hub Device Provisioning Service (DPS) est un complément très utile à Azure IoT Hub. Il est capable de prendre en charge le provisioning (ou reprovisioning) et la répartition des devices sur différents IoT Hubs en fonction de différents critères.
IoT Plug and Play est l'une des dernières fonctionnalités intégrées à Azure IoT Hub. Il cherche à rendre agile l'intégration des devices IoT dans les solutions en proposant un mécanisme similaire au Plug and Play que nous connaissons tous (du moins, au niveau de l'expérience utilisateur).
En termes de flux de données, l'input dans l'IoT Hub est représenté par les messages envoyés par les objets connectés. Par défaut, Azure IoT Hub inclut un output de type EventHub. Tous les messages sortent par ce canal par défaut. Il est possible de rajouter des routes vers des endpoints personnalisés de types différents.
Les services vers lesquels les routes peuvent pointer sont :

- Azure Storage ;
- Azure Event Hub ;
- Azure Service Bus-Topic ;
- Azure Service Bus-Queue.

Chaque type de sortie a une utilité bien définie et les choix dépendront des besoins de chaque projet.
Azure IoT Hub permet de gérer deux types de devices :

- Device « normal » ;
- Device Edge, avec un runtime IoT Edge.

Chaque device peut intégrer un ou plusieurs modules, ce qui donne une certaine flexibilité au moment d'imaginer les solutions IoT. Azure IoT Hub communique avec les devices via les protocoles suivants :

- MQTT ;
- AMQP ;
- HTTPS ;
- MQTT via WebSockets ;
- AMQP via WebSockets.

Pour les solutions qui auraient besoin d'ingérer des messages par lots ou des médias, Azure IoT Hub peut exposer une API permettant de faire de l'upload de fichiers vers un Azure Storage (blob). Le mécanisme est intégré à IoT Hub.

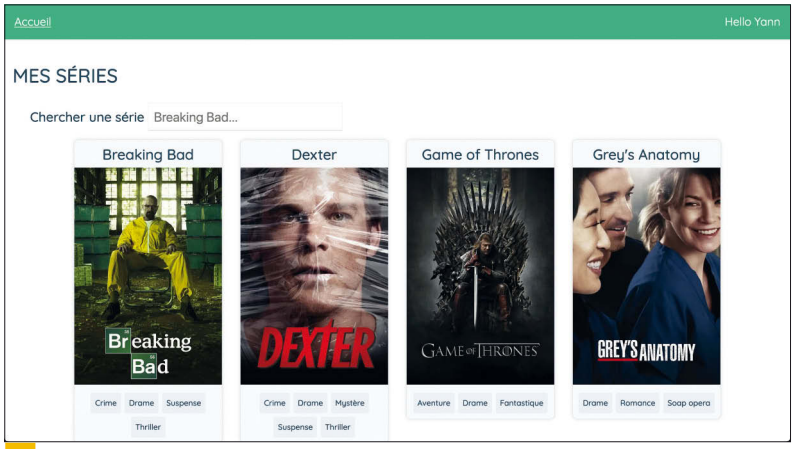
Dans la partie 2, nous réaliserons un projet complet.

La suite dans le n°244

Yann Bertrand
Consultant et Formateur
Zenika Nantes

Développer sa première PWA

Les applications mobiles natives ont des fonctionnalités que nos sites internet ne peuvent fournir : installation sur l'écran d'accueil, présence dans les stores, processus externe au navigateur, fonctionnement hors ligne, lecture/écriture dans le système de fichiers, accès au hardware branché ou via bluetooth, manipulation des contacts et événements… Beaucoup de fonctionnalités indispensables pour sembler faire partie du téléphone.



1 *Le site qui nous servira d'exemple*

Pour faire des applications mobiles natives, il faut du code spécifique pour chacune des plateformes. Ce code est donc dupliqué et demande des compétences différentes. Grâce aux nouveautés du web, les Progressive Web Applications (PWA) nous permettent de développer des expériences de plus en plus proches du natif. Avec le gros avantage de pouvoir partager une bonne partie du code entre les versions desktop, mobiles Android et iOS. À travers ce guide, essayons-nous au développement de notre première PWA.

Qu'est-ce qu'une Progressive Web App ?

Une PWA c'est "juste" une application web. À l'instar des applis "natives multi-plateformes" react-native/flutter ou des "hybrides" Ionic/Cordova, elles ne sont pas packagées et dépendantes de l'App Store iOS ou du Play Store Android. Détaillons rapidement ces différentes visions avant de nous lancer.

Application native

Au plus proche du matériel, les applis natives permettent de profiter au maximum des performances et des fonctionnalités bas-niveau du smartphone (réalité augmentée, machine learning). Ces applications sont téléchargeables sur les stores officiels après validation par Google ou Apple et peuvent être mises à jour par ce biais. Ces mises à jour peuvent être compliquées, car dépendantes de la politique choisie par les propriétaires du store. Par exemple Apple vérifie chaque nouvelle version manuellement. La compatibilité multi-plateforme est également complexe, car les langages et SDK utilisés sont différents.

Application native multi-plateforme

De nombreux outils se sont montés pour combler le souci de compatibilité. Les raisons sont évidentes : baisser les coûts de développement en facilitant la réutilisation. Dans cette catégorie, on retrouve React Native, Flutter ou Xamarin qui permettent de partager le code de l'interface entre Android et iOS, la partie fonctionnelle tourne à part. Pour aller plus loin, je vous conseille l'article d'Andréas Hanss de l'édition de mai (React Native vs Flutter - Les vrais arguments).

Application hybride

Celles-ci sont des applications web packagées dans une webview. Une webview est un composant natif disponible sur Android et iOS pour afficher du HTML, ce qui nous permet d'éviter d'écrire notre application en langage natif. Puisqu'elles sont packagées, elles peuvent elles aussi être envoyées sur les stores. Elles donnent accès à certaines fonctionnalités bas-niveau grâce à un système de plugins fournissant des API JavaScript pour accéder au presse-papier par exemple. On y retrouve Cordova et Capacitor pour le "bas niveau", et la possibilité d'y ajouter une surcouche graphique avec Ionic.

Application web

Accessibles à travers un navigateur sur mobile, mais aussi sur ordinateur, elles ne permettaient historiquement pas d'interagir avec les fonctionnalités bas-niveau de nos smartphones. Une bonne partie de ces fonctionnalités est maintenant accessible, en particulier sur Android avec le navigateur Chrome. Une application web est considérée Progressive quand elle respecte trois critères :

- Utilise HTTPS ;
- Présente un web Manifest ;
- Fournit un Service Worker.

Elles sont également censées être disponibles sur les moteurs de recherche, installables sur l'écran d'accueil, capable de gérer les liens, indépendantes du réseau, fonctionner sur les téléphones récents, mais aussi anciens, capables de gérer les notifications, responsives, et sécurisées. Je m'appliquerai par la suite à détailler et mettre en pratique les trois critères principaux.

Préparer son site à agir comme une application

Pour illustrer cet article, je travaillerai en pas-à-pas en partant d'un site basique : les source sont disponibles sur https://github.com/yannbertrand/pwa-demo et l'application finale est déployée sur https://pwa-demo-yann.netlify.app. 1

Premier prérequis pour notre PWA et première règle à respecter : **notre site doit pouvoir être déployé en HTTPS**. Certaines API web ne sont accessibles que dans un contexte sécurisé, par exemple la géolocalisation ou les Service Workers (que l'on creusera plus loin). Certaines API accessibles sont critiques vis à vis du respect de la vie

privée des utilisateurs. C'est pourquoi cette sécurité est requise, pour éviter par exemple, des attaques de type MITM (man-in-the-middle). Une des difficultés que l'on rencontre fréquemment lorsqu'on crée une PWA est de s'assurer que notre application est rendue correctement sur mobile. Même si le sujet n'est pas spécifique aux PWA, il me semble important d'en parler un peu. Chrome et Firefox nous fournissent un super outil intégré dans les DevTools pour simuler ce rendu, respectivement : le mode responsive et la vue adaptative. Dans notre cas et pour une page basique, il est très probable que vous rencontriez un premier souci : la page est dézoomée et tout y paraît très petit. Il faut régler une métadonnée dans le header pour préciser que notre conteneur (le viewport) doit prendre toute la largeur de l'écran :

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

La présence de cette métadonnée est très importante pour l'aspect graphique, mais également pour la SEO : l'algorithme de Google va mieux positionner les pages où le viewport est correctement renseigné. Globalement cet algorithme encourage à fournir une bonne expérience utilisateur sur mobile, sujet également mis en avant dans l'initiative Google web Vitals. Pour aller plus loin sur l'aspect graphique je vous invite à creuser le sujet du responsive design (en particulier le positionnement avec flexbox et le grid layout) et les media queries (dark mode, cacher les animations...). Attention tout de même, gardez bien en tête qu'en fonction de votre cible, il faudra faire attention à la compatibilité navigateurs mobile et desktop des fonctionnalités CSS que vous utiliserez.

Construire un web App Manifest

Le Manifest va permettre de donner des informations génériques sur notre application dans un fichier JSON. Le but est de pouvoir installer l'application sur l'écran d'accueil du téléphone pour fournir à l'utilisateur un accès plus rapide au contenu et une expérience plus riche. Il n'est donc pas obligatoire de passer par une boutique d'application pour les installer. Pour installer une PWA il faudra utiliser la fonctionnalité "Add to home screen", en réalité il n'y a rien de particulier à faire pour Android qui nous montrera une boîte de dialogue pour y accéder (si le navigateur le permet), mais il faudra que l'utilisateur fasse une action manuelle sur iOS.
Techniquement, notre manifest sera dans un fichier avec l'extension ".webmanifest" qu'il faut référencer dans notre header HTML :

```
<link rel="manifest" href="/manifest.webmanifest">
```

Dans ce fichier, il y a énormément de choses que l'on peut préciser, voici la liste des clés les plus importantes à mon sens.

short_name et/ou name

Une des deux clés est au moins obligatoire. Si les deux sont fournies, name sera utilisée dans la plupart des cas et short_name sera utilisée si l'espace est limité (page d'accueil, launcher). Je vous conseille de préciser les deux pour un maximum de compatibilité.

icons

À l'installation, il est possible de définir un ensemble d'icônes que le téléphone pourra utiliser sur l'écran d'accueil, le launcher, le sélecteur d'apps, comme splash screen, etc. La propriété icons prends en paramètre un tableau d'images. Pour Chrome sur Android, il faut au moins fournir une icône en 192x192 pixels et une en 512x512 pixels. Chrome est capable de les retailler en toutes les tailles nécessaires par l'OS. Pour plus de précision, il est possible de les fournir par incréments plus fins.
Il existe de nombreux outils permettant d'automatiser le découpage de ces icônes comme https://realfavicongenerator.net.

start_url

start_url est obligatoire et indique au navigateur où votre application doit démarrer lorsqu'elle est lancée, et l'empêche de démarrer sur la page sur laquelle l'utilisateur se trouvait lorsqu'il a ajouté votre application à son écran d'accueil. Pensez à ce que l'utilisateur voudra faire une fois qu'il aura ouvert votre application, et utilisez la page la plus adaptée.

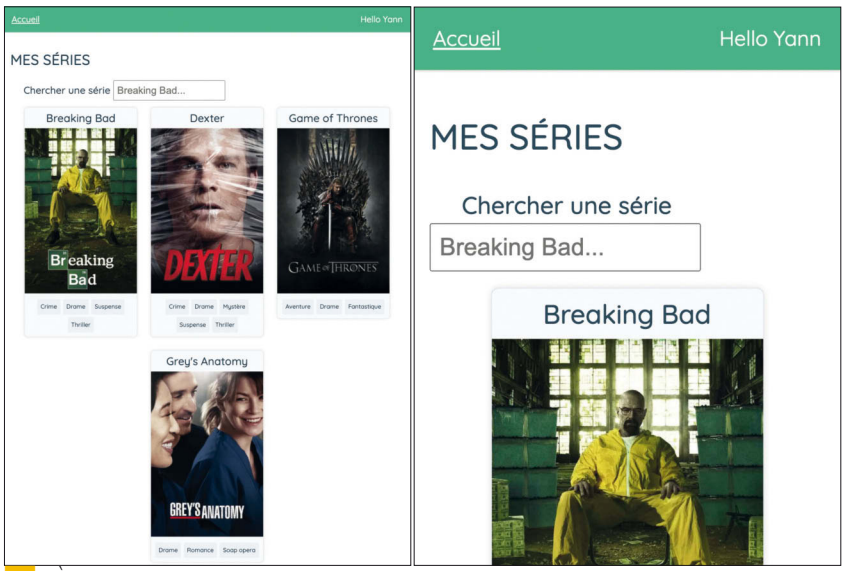
background_color

La propriété background_color est utilisée sur le splash screen de l'application (écran de lancement). Sur Android, l'icône (512x512 minimum, format PNG) sera affichée au-dessus du nom de l'application avec la couleur background_color en fond sur le reste de l'écran.

display

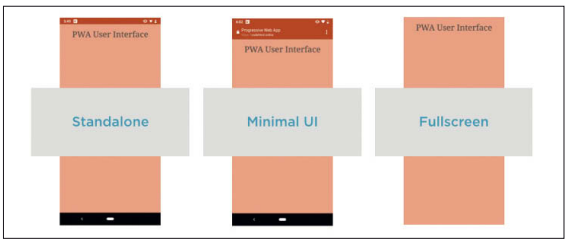
Elle permet de personnaliser l'interface qui sera utilisée quand l'appli est lancée. Quatre valeurs sont possibles :

- browser : l'application s'ouvre dans un nouvel onglet ou une nouvelle fenêtre. C'est la valeur par défaut.
- minimal-ui : l'application va ressembler et se comporter comme une application autonome, mais elle aura quelques éléments d'interface permettant de contrôler la navigation. Cela peut inclure que l'application ait une fenêtre différente, sa propre icône dans le lanceur d'applications, etc... Si le choix est indisponible, le type de display précédent sera utilisé.
- standalone : l'application va ressembler à une application autonome et se comporter comme telle. Dans ce mode, l'agent utilisateur va exclure les éléments d'interface qui permettent de contrôler la navigation, mais peut inclure d'autres éléments comme une barre de statut.



2 *À gauche : sans avoir précisé la meta viewport*
À droite : en l'ayant précisée

- fullscreen : toute la zone d'affichage disponible est utilisée. Ce mode est plutôt adapté pour les jeux vidéo.



Les différentes valeurs de display disponibles

scope

Le scope définit l'espace où l'utilisateur peut naviguer. S'il essaye d'accéder à une URL externe au scope, il sera redirigé vers une nouvelle page de navigateur. Attention à ce que la start_url soit bien comprise dans le scope.

theme_color

La theme_color définit la couleur de la barre d'outils, et peut se refléter dans l'aperçu de l'application dans le sélecteur d'apps. Elle doit être accompagnée de la métadonnée theme-color dans le header de notre page HTML. De la même manière que pour les icônes, il existe foultitudes d'applications pour générer et valider son manifest. Pour mon exemple j'ai utilisé **https://app-manifest.firebaseapp.com** pour générer mon manifest et mes icônes et **https://www.pwabuilder.com** pour le valider. Dans les DevTools Chrome, l'onglet Application permet aussi de vérifier tout ça. Voici le fichier que j'ai généré de mon côté :

```
{
  "name": "Démo de PWA",
  "short_name": "Ma PWA",
  "theme_color": "#00c58e",
  "background_color": "#ffffff",
  "display": "standalone",
  "scope": "/",
  "start_url": "/",
  "icons": [
    {
      "src": "/icons/icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png"
    },
    ...
    {
      "src": "/icons/icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ],
  "lang": "French"
}
```

Le cas iOS

Maintenant qu'on a vu la théorie, il faut voir la réalité des choses : le support des web Manifest par iOS est loin d'être parfait. Malheureusement, Apple a préféré gérer les choses différemment du standard W3C. Il y a donc quelques astuces à connaître pour supporter correctement iOS. Le chemin vers l'icône doit être précisé dans un link apple-touch-icon, sa taille doit être de 180x180 pixels pour gérer un maximum de devices :

```
<link rel="apple-touch-icon" href="/icons/apple-touch-icon.png">
```

Il existait autrefois une métadonnée apple-mobile-web-app-capable pour que le site soit utilisable en dehors du navigateur. Elle est aujourd'hui dépréciée, et la seule manière de faire la même chose est de donner la valeur standalone à la clé display. Si on lui donne n'importe quelle autre valeur, l'application s'ouvrira systématiquement dans un navigateur. Attention à correctement gérer la navigation dans le site donc. Il est toujours intéressant d'ajouter la métadonnée pour conserver le support des anciennes versions d'iOS.

```
<meta name="apple-mobile-web-app-capable" content="yes">
```

Apple ne propose pas de générer le splash screen à partir des clés icons + name + background_color. Il faut préciser un link pour chaque type d'appareil où l'on voudra que l'application soit utilisable. Exemple pour l'iPhone X :

```
<link href="splashscreens/iphonex_splash.png"
    media="(device-width: 375px) and (device-height: 812px) and (-webkit-device-pixel-ratio: 3)"
    rel="apple-touch-startup-image" />
```

Elles peuvent être également générées depuis des services comme l'iOS splash screen Generator d'Appscope : **https://appsco.pe/developer/splash-screens.** Il est aussi possible de préciser si la barre de statut en haut doit être noire ou noire translucide (texte blanc sur fond sombre transparent) avec la métadonnée apple-mobile-web-app-status-bar-style et les valeurs black (default) ou black-translucent.

```
<meta name="apple-mobile-web-app-status-bar-style" content="black-translucent">
```

Pour supprimer des effets qu'on aurait sur navigateur, il est possible de spécifier du CSS pour supprimer la sélection et le highlighting de texte par exemple :

```
body {
    -webkit-user-select: none;
    -webkit-tap-highlight-color: transparent;
}
```

Dernier point concernant le support : du côté de Windows 10, il est aussi possible d'ajouter une métadonnée pour une bonne gestion de la couleur de fond de notre tile. Car oui : Windows 10 sait aussi installer les PWA comme des applications.

```
<meta name="msapplication-TileColor" content="#00c58e">
```

Vous l'aurez compris, on s'éloigne un peu de l'esprit de communion que l'on aurait souhaité avoir… Heureusement que tout cela ne bouge pas souvent.

Notre premier Service Worker

Après avoir passé notre site en HTTPS, l'avoir rendu responsive et l'avoir rendu installable sur les différentes plateformes, il est temps de concevoir notre Service Worker. Les Services Workers sont un type de Workers permettant entre autres de servir de proxy entre le navigateur et le réseau pour conserver une bonne expérience de navigation si le mobile est hors ligne. 3

C'est ce que nous allons construire dans la suite de l'article en profitant de l'API Cache. Ils permettent également d'envoyer des notifications push et d'utiliser des API de synchronisation en arrière-plan. Ce dernier point est tellement vaste qu'il mériterait à lui tout seul un article entier. Retenez tout de même que les notifications push sont pour le moment impossible sur les PWA iOS.

Enregistrer un Service Worker

Avant de commencer le développement de notre Service Worker, il va falloir l'enregistrer dans notre page d'accueil HTML. Pour ce faire, il suffit de demander au navigateur de s'en occuper dans un petit script JavaScript :

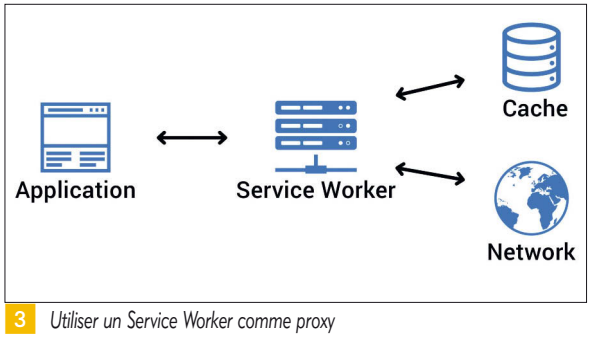
```
<script>
  if ('serviceWorker' in navigator) {
    window.addEventListener('load', () => {
      navigator.serviceWorker.register('service-worker.js')
        .then(registration => {
          console.log('😎 Le Service Worker est enregistré', registration);
        })
        .catch(error => {
          console.error('😢 L\'enregistrement ne s\'est pas bien passé :', error);
        });
    });
  }
</script>
```

La première ligne du script permet de vérifier si le navigateur gère les Services Workers, le support est aujourd'hui très bon, mais il est important de ne pas générer d'erreur si le navigateur de l'utilisateur est trop ancien. Lorsqu'il est prêt, on demande au navigateur d'enregistrer le fichier qui contient le code de notre Service Worker. N'oubliez pas de créer ce fichier qui restera sans contenu pour le moment. navigator.serviceWorker.register(...) nous renvoie une promesse avec un objet registration une fois que tout s'est bien passé, ou rejette une erreur s'il y a eu un souci. 4

Le cycle de vie du Service Worker

Ajoutons ce bout de code dans le fichier service-worker.js :

```
self.addEventListener('install', event => {
  console.log('👷 Installation du Service Worker...');
});

self.addEventListener('activate', event => {
  console.log('🤖 Activation du Service worker...');
});
```



3 *Utiliser un Service Worker comme proxy*

```
});

self.addEventListener('fetch', event => {
  console.log('🎁 Interception d\'un fetch vers :', event.request.url);
});
```

Après avoir rechargé la page, on peut constater dans nos logs que le Service Worker s'installe. À noter que les logs de l'enregistrement peuvent apparaître en dessous ou au-dessus, car le Service Worker tourne dans un processus séparé et ne garantit donc pas qu'il sera exécuté avant ou après. En retournant dans l'onglet Application, on peut constater qu'une nouvelle version de notre Service Worker est disponible. Le navigateur ne l'active pas de lui-même pour éviter des mises à jour alors que le Service Worker est occupé, ici on peut cliquer sur skipWaiting pour le mettre à jour manuellement. 5

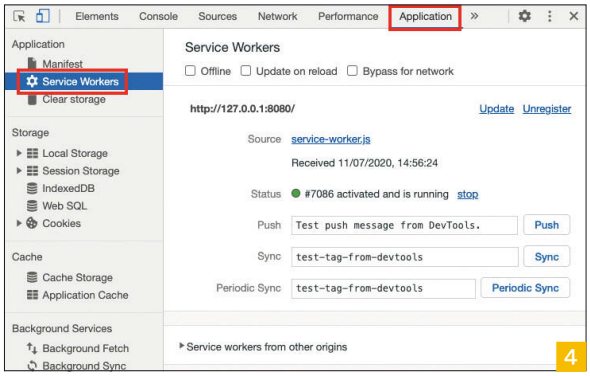
Pour simplifier le développement, je vous conseille de cocher la case "Update on reload" qui mettra à jour notre Service Worker à chaque rechargement de page. 6

Une fois activé et si l'on recharge la page, on peut constater de nombreux logs d'interception de l'événement fetch. Cet événement est déclenché à chaque fois qu'une requête part de notre application. C'est ici que l'on va pouvoir récupérer ces requêtes et aiguiller la réponse soit pour réellement récupérer la ressource et la mettre en cache, soit pour servir ce qui est dans le cache. 7

Mettre en cache les ressources pour fonctionner hors ligne

Il est temps de s'attaquer à la vraie valeur ajoutée de notre Service Worker : utilisons l'API Cache pour stocker nos ressources sur l'appareil de l'utilisateur ! Par ressource j'entends n'importe quel type de fichier qu'on peut habituellement récupérer sur une page web : fichier HTML, fichier CSS, image, police d'écriture... Ici nous utiliserons la stratégie de "Cache first" : on récupère en priorité depuis le cache, sinon depuis internet. Mais il est tout à fait possible d'implémenter d'autres stratégies de mise en cache.

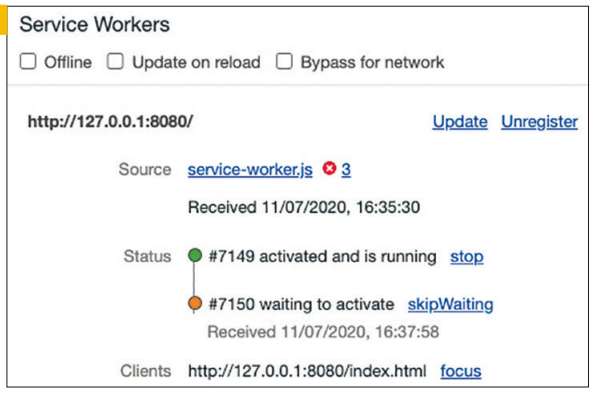
```
const CURRENT_CACHE_NAME = 'v1';

...

self.addEventListener('fetch', event => {
  console.log('🎁 Interception d\'un fetch vers :', event.request.url);
```



Les DevTools de Chrome nous permettent de vérifier que le log d'enregistrement est bien présent et l'onglet Application > Service Workers qu'il s'est bien enregistré.

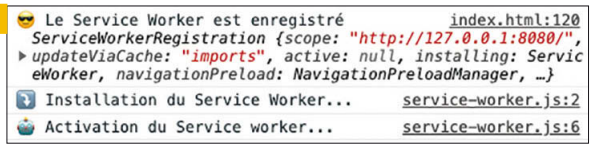
La nouvelle version de notre Service Worker en attente d'activation

5



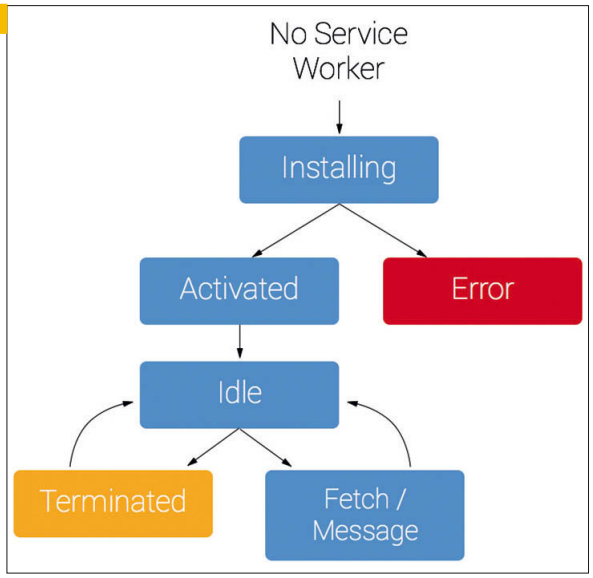
Le cycle de vie du Service Worker : installation puis activation

6



Le cycle de vie complet du Service Worker lors de sa première installation

7



Récupération des ressources depuis le Service Worker dans l'onglet Network des DevTools

8

Name	Type	Initiator	Size	Time	Waterfall
index.html	document	Other	(ServiceWorker)	4 ms	
quicksand.woff2	font	index.html	(ServiceWorker)	6 ms	
bb-poster.jpg	jpeg	index.html	(ServiceWorker)	6 ms	
index.css	stylesheet	index.html	(ServiceWorker)	5 ms	
dexter-poster.jpg	jpeg	index.html	(ServiceWorker)	5 ms	
got-poster.jpg	jpeg	index.html	(ServiceWorker)	6 ms	
ga-poster.jpg	jpeg	index.html	(ServiceWorker)	7 ms	

```
event.respondWith(caches.match(event.request)
    then(cachedResponse => {
        if (cachedResponse) {
            console.log('Réponse depuis le cache pour :', event.request.url);
            return cachedResponse;
        }

        return fetch(event.request).then(response => {
            if (!response || response.status !== 200 || response.type !== 'basic') {
                return response;
            }

            const responseToCache = response.clone();
            caches.open(CURRENT_CACHE_NAME).then(cache => {
                console.log('Mise en cache de :', event.request.url);
                cache.put(event.request, responseToCache);
            );

            return response;
        });
    })
  );
});
```

La callback de l'événement fetch nous envoie un event qui nous permet de surcharger la réponse avec la méthode respondWith.
L'API Cache nous fournit un objet caches qui contient des méthodes pour stocker, récupérer et ouvrir le contenu d'un cache. Respectivement put, match, et open. On précise un nom de cache pour pouvoir l'invalider quand ça sera nécessaire.
La première partie du code est plutôt simple, on vérifie si un cache contient déjà la réponse associée à une requête, si c'est le cas : on la renvoie telle quelle.
Si le cache ne contient pas la réponse attendue, on lance la requête grâce à la méthode fetch (qui est disponible sur tous les navigateurs mobiles récents). Si la réponse est valide et provenant de notre domaine, on la stocke dans notre cache puis on la retourne.
Deux petites choses à noter :

- On versionne notre cache avec CURRENT_CACHE_NAME. Si une ressource est mise à jour, on change la valeur de cette variable ce qui aura pour effet de rafraîchir toutes les réponses à l'activation du nouveau Service Worker.
- La réponse étant un stream, il faut absolument la cloner pour pouvoir la renvoyer à la fin. Le contenu d'un stream ne peut être consommé qu'une seule fois ! 8

Il est aussi tout à fait possible de filtrer les ressources qui nous intéressent et de leur appliquer différentes stratégies de mise en cache.

Ne pas oublier de nettoyer derrière soi

Pour réellement faire fonctionner le versionning, il faut également penser à bien nettoyer les caches inactifs que l'on aurait pu laisser traîner. Voici comment on peut s'y prendre :

```
self.addEventListener('activate', event => {
    console.log('Activation du Service worker...');
    event.waitUntil(
        caches.keys().then(cacheNames => {
            return Promise.all(
                cacheNames.map(cacheName => {
                    if (cacheName !== CURRENT_CACHE_NAME) {
                        return caches.delete(cacheName);
                    }
                })
            );
        })
    );
});
```

Les méthodes install et activate attendent qu'on leur donne la permission de se terminer avant de passer à la suite. Ici, je fournis

une liste de promesses à exécuter avant de terminer l'activation : je parcours chaque cache et s'il est inutilisé, je le vide grâce à la méthode delete.

Stratégies de mise à jour

Si vous avez essayé de votre côté, la question qui doit venir maintenant est : comment est-ce que je mets réellement à jour mon application ? En effet, pour le moment nous trichons avec l'upload on reload. Le plus simple est de forcer l'activation d'un nouveau Service Worker directement après l'installation :

```
self.addEventListener('install', event => {
    console.log('🔽 Installation du Service Worker...');
    self.skipWaiting();
});
```

Ainsi l'utilisateur récupérera la nouvelle version au prochain chargement de la page. Attention cependant, car notre cache est vidé à l'activation, donc si l'utilisateur ne recharge pas la page, il n'y aura plus accès sans connexion ! Heureusement, il est possible d'écouter des événements sur l'objet registration et d'afficher un message à l'utilisateur pour qu'il recharge la page.

Enfin, nous aurons tendance à éviter ce genre de mises à jour si l'utilisateur ne souhaite pas être interrompu. Donc attention à bien réfléchir à la stratégie qui nous convient le mieux.

Grâce à tout ce travail, vous devriez obtenir de très bons scores Lighthouse PWA et PWABuilder ! 9

Aller plus loin

Nous avons pu voir comment construire un Service Worker basique au plus bas niveau, mais il existe de nombreux outils pour nous simplifier le travail. Le package npm register-service-worker améliore l'API serviceWorker en fournissant des callbacks sur les mises à jour de ressources ou le passage en offline par exemple. Workbox permet de faciliter le développement des différentes stratégies de mise en cache selon le type de fichiers et la durée souhaitée. PouchDB permet d'avoir une vraie expérience offline avec une file de messages à envoyer au serveur quand la connexion est récupérée. Pour un framework utilisable directement, je vous conseille le projet open source bento-starter qui combine astucieusement Vue.js et Firebase pour se lancer rapidement avec une super expérience développeur. Même si vous n'avez pas l'intérêt d'installer un site statique (comme celui de l'exemple) sur votre écran d'accueil, l'API Cache et les Service Workers améliorent grandement l'expérience de l'utilisateur en fluidifiant le rechargement après notre première visite.

Le futur des PWA

Alors que la part d'utilisation des ordinateurs de bureau diminue pour laisser place à la navigation mobile, le web se place comme un concurrent sérieux aux applications natives. De nouvelles API web ne cessent d'apparaître pour s'approcher de l'expérience native : paiement en ligne, partage natif (web share), géolocalisation, communication temps réel WebRTC, presse-papier...

Certaines sont mêmes poussées par la communauté à travers le Project Fugu. On y trouve par exemple des API de détection de formes (visage ou texte) ou d'autorisation d'accès par empreinte digitale/reconnaissance faciale (WebAuthn) qui arrivent tout juste avec iOS 14. D'ici quelques années, on pourra même se poser la question de l'intérêt des applications hybrides. Côté plateformes, elles gèrent de mieux en mieux les PWA. Chrome et Microsoft Edge permettent de les installer sur les OS comme Windows, Linux ou macOS. Certains OS les gèrent même nativement. On peut maintenant les distribuer sur le Play Store Android (on parle en fait de Trusted web Activity). Il existe également une vitrine des meilleures expériences de PWA : AppScope.

Malheureusement, un acteur important du monde mobile ne joue pas le jeu : Apple. Alors que Steve Jobs était un des premiers promoteurs des applis mobile web (l'AppStore n'était pas initialement prévu), Apple refuse d'entendre parler de PWA leur préférant le nom d'HTML5 App ou d'Homescreen WebApp.

Avec iOS 14 fraîchement sorti, les enthousiastes avaient des grosses attentes pour rattraper le retard : en particulier le support des notifications push et une meilleure expérience d'installation. Malheureusement il n'en est toujours rien, Apple fait faux bond et repousse le standard. On ne peut plus entendre ce discours ; ils refusent d'implémenter des nouvelles API pour des raisons de respect de la vie privée : Bluetooth, NFC, niveau de batterie... Le support est amélioré chaque année, mais vraiment au goutte à goutte.

J'en ai terminé pour cette intro au monde des PWA en 2020 que j'espère suffisamment concrète. Tout n'est pas rose, mais le monde du web a le mérite d'avoir un support assez incroyable même si tout n'y est pas encore disponible. Je suis persuadé que les PWA auront une place importante dans le futur et je ne perds pas espoir pour que le standard regroupe enfin tous les acteurs. J'espère aussi vous avoir donné envie d'essayer les Service Workers pour profiter de vos sites sans connexion, et plus globalement de développer votre première PWA !

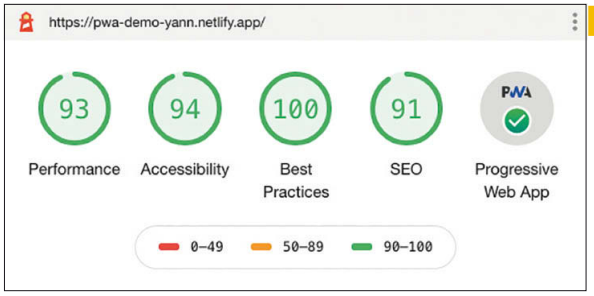
Sources

MDN https://developer.mozilla.org/fr/docs/Web/Progressive_web_apps
https://web.dev/progressive-web-apps/
https://developers.google.com/certification/mobile-web-specialist/study-guide/pwas
La référence des PWA, principalement sur iOS : Maximiliano Firtman (@firt) en particulier les articles :
https://medium.com/@guillaumeandre/progressive-web-app-pwa-fichier-web-app-manifest-7292db378af5
https://medium.com/appscope/designing-native-like-progressive-web-apps-for-ios-1b3cdda1d0e8

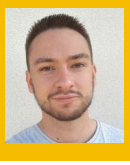
et aussi :

https://www.julienpradet.fr/fiches-techniques/pwa-declarer-un-service-worker-et-gerer-son-cycle-de-vie/

Merci à toutes les personnes qui m'ont aidé à la rédaction de cet article.



9 *Les scores Lighthouse de l'application de démo*



Thomas Boutin
Développeur Mobile chez
Ippon Technologies
https://github.com/Thomas-Boutin

Vincent Treve
Architecte Data et Solution chez **Ippon Technologies**

Introduction aux Fonctions d'ordre supérieur avec Kotlin

Kotlin, par son approche simple et concise du développement dans la JVM, s'appuie sur certains principes de programmation fonctionnelle et en particulier sur les Higher Order Functions (HOF pour les intimes). Pour rappel, ou pour celles et ceux qui découvrent le sujet, les Fonctions d'ordre supérieur sont des fonctions qui prennent une ou plusieurs fonctions en entrée et/ou qui produisent une fonction en sortie.

Normalement, en seulement une phrase d'introduction, le sujet de la programmation fonctionnelle a déjà rebuté une (grande) partie du public de cet article… La programmation fonctionnelle a trop souvent été expliquée par des mathématiciens (ils l'ont inventée).

Nous avons tous le souvenir d'un prof de math en amphi qui fait l'appel le premier jour et qui est capable de saluer dans les couloirs toute la promo le lendemain :

- "Bonjour M·me [Insert ici ton nom et ton prénom] !"

et intérieurement, on se dit :

- "Non, je ne deviendrai pas comme ça, je veux faire de la pratique, pas seulement de la théorie…"

Et à voix haute :

- "Bonjour M … heu … Jaumet" [Le nom a été changé]

Plus tard, bien après cet épisode, nous nous sommes orientés vers le C/Java/Python en laissant sur le bas-côté de notre scolarité les langages fonctionnels chers à nos professeurs de math-info comme Haskell ou OCaml en se disant qu'il est quand même temps d'apprendre à utiliser SpringBoot/Pandas !

Ne tombons pas dans le travers de l'approche théorique sans cas d'utilisation réel et commençons par étudier le fonctionnement d'une HOF, nous reviendrons sur les bases théoriques ensuite.

Google ajoute de nombreuses HOF dans Android que vous utilisez sans forcément vous en rendre compte. Une partie d'entre elles sont amenées par *Android KTX* (https://developer.android.com/kotlin/ktx). Pour reprendre leurs mots, l'idée est de "*Fournir des utilitaires Kotlin concis et idiomatiques à Jetpack, à la plateforme Android et aux autres APIs*". En somme, faciliter la vie du développeur.

Le lien spécifié ci-dessus fournit un exemple d'utilisation de KTX. On nous montre comment on peut utiliser plus simplement l'API SharedPreferences, qui sert à stocker simplement dans son application des couples clé / valeur.

On passe alors de (sans KTX) :

```
sharedPreferences
    .edit()  // create an Editor
    .putBoolean("key", value)
    .apply() // write to disk asynchronously
```

à (grâce à KTX) :

```
// SharedPreferences.edit extension function signature from Android KTX - Core
// inline fun SharedPreferences.edit(
//     commit: Boolean = false,
//     action: SharedPreferences.Editor.() -> Unit)

// Commit a new value asynchronously
sharedPreferences.edit { putBoolean("key", value) }

// Commit a new value synchronously
sharedPreferences.edit(commit = true) { putBoolean("key", value) }
```

Une affaire de HOF

On profite ici d'une manière de développer qui ressemble plus à du Kotlin, via l'utilisation d'une closure. Pour notre sujet, la partie qui nous intéresse se trouve dans l'implémentation de cette fonction edit :

```
inline fun SharedPreferences.edit(
  commit: Boolean = false,
  action: SharedPreferences.Editor.() -> Unit
) {
  val editor = edit()
  action(editor)
  if (commit) {
    editor.commit()
  } else {
    editor.apply()
  }
}
```

On a en fait ici affaire à une HOF. Le paramètre *action* est une fonction qui prend en paramètre un SharedPreferences.Editor.() . Le premier intérêt, c'est de concentrer le code sur l'action que l'on va faire dans cette base de données clé / valeur (putBoolean). On force surtout que l'action terminale soit ou un *commit()* ou un *apply()*, sans quoi les modifications ne seraient pas sauvegardées !

On peut se demander si on peut quand même chaîner les modifications. La réponse est oui !

Puisque putBoolean est un *SharedPreferences.Editor* qui lui-même retourne un *SharedPreferences.Editor*, on pourrait très bien écrire :

```
sharedPreferences.edit {
    putBoolean("key", true)
    putBoolean("key2", false)
    putInt("key2", 1)
}
```

On voit très bien dans cet exemple que les HOF permettent de factoriser des comportements autour d'un traitement. Cela rend le code plus lisible. Le développeur se concentre ainsi sur la logique métier, pas sur les contraintes techniques. On diminue aussi le nombre de copier-coller, ce qui va mener à un code de meilleure qualité qui sera plus facile à tester. Essayons de profiter des avantages des HOF vus dans l'exemple ci-dessus pour notre base de code. Nous allons repartir d'un contexte Android. Si vous avez déjà développé des apps, vous avez sûrement déjà dû faire des appels réseaux ou en base de données. La plupart de ces traitements résultent soit par la restitution de la valeur (Succès), soit par une exception (Erreur). Dans un contexte Kotlin, la capture d'exception se fait via l'encapsulation de notre code dans un bloc try / catch. Bloc qui est potentiellement à recréer pour chaque appel que vous avez à faire. C'est le premier souci : la répétition de code. Le second souci, c'est que Kotlin n'a pas le concept de checked exception. Le compilateur ne nous avertit donc pas si un bloc de code peut lever une exception. La bonne pratique est alors d'utiliser des classes qui représentent des résultats de traitement. Voyons plutôt :

```
fun <T> wrapInResource(block: () -> T): Resource<T> {
  return try {
    block().let {
      Resource.Success(it)
    }
  } catch (e: Exception) {
    Timber.e(e)
    Resource.Error(e)
  }
}
```

Cette fonction d'ordre supérieure un peu particulière prend un bloc de code qui retourne un type générique T. Ce bloc de code est exécuté dans un try / catch. Deux issues sont possibles :

- Succès. Auquel cas on retourne un Resource.Success
- Exception. Auquel cas on log l'erreur avec Timber et on retourne un Resource.Error

À l'utilisation :

```
wrapInResource {
  val fichierId = getStringValue(FichierDao.FICHIER_ID)
  val fichierDocument = getMapValue(FichierDao.FICHIER_ALIAS)

  LocalFichier(
    fichierId,
    fichierDocument
  )
}
```

Le wrapInResource retourne un Resource<LocalFichier> qui est soit de type de Success, soit de type Error. Comment peut-on en être sûr? Eh bien c'est grâce au concept de *sealed class*. Ce modificateur de classe permet d'assurer que Resource ne pourra s'étendre qu'au sein de la closure qui la définit :

```
sealed class Resource<T> {
  data class Success<T>(val data: T) : Resource<T>()
  data class Error<T>(val error: Throwable) : Resource<T>()
}
```

Une Resource.Success contient une *data*, qui est le résultat du traitement. Une Resource.Error contient une *error* qui est le *Throwable* (ie: l'exception) associé au traitement. Grâce à ce procédé, nous serons obligé de vérifier que notre résultat est de type Resource.Success pour accéder à la *data* associée.

La programmation fonctionnelle apporte de belles promesses. Mais, le prix à payer est l'apprentissage d'un nouveau paradigme basé sur des concepts mathématiques. C'est parfois complexe à manipuler pour des développeurs habitués à la programmation impérative, en particulier si notre dernière interaction avec un professeur de mathématiques ne date pas d'hier !

Les fonctions pures

Pour faire plaisir à ce bon vieux M. Jaumet, je vous propose de vous arrêter sur un concept qui vous permettra peut-être d'éviter les pièges classiques : les fonctions pures.

Une fonction pure est une fonction :

- qui renvoie toujours le même résultat pour une même liste d'arguments ;
- qui est sans effet de bord. Une fonction à effet de bord est une fonction qui modifie une donnée à l'extérieur de son scope. Les effets de bords sont très utilisés en programmation, pour la gestion des Entrée/Sortie par exemple ou la modification d'un état par exemple.

C'est ici qu'apparaissent les premières difficultés : comment écrire un serveur web qui écrit dans une base de données avec des fonctions pures ? Comment modifier un attribut d'une instance ?

C'est bien sûr possible, mais ces problèmes simples peuvent devenir complexes en programmation fonctionnelle. Si le dernier point peut facilement être contourné avec l'utilisation du pattern update-as-you-copy (une sorte de copy-on-write). Le premier nécessite l'utilisation de concepts spécifiques à la programmation fonctionnelle. Une solution pragmatique est de voir les HOF comme les commandes d'un pipeline de commandes Linux, où seules les fonctions aux extrémités ont des effets de bords. C'est ce que nous avons fait dans notre exemple avec wrapInResource, qui encapsule le point d'entrée avant de venir appliquer nos traitements.

On retiendra les bénéfices suivants :

- Les fonctions pures sont faciles à appréhender, rien ne se passe en dehors de leur scope ;
- Les fonctions pures facilitent les tests unitaires, en étant faciles à mocker :
- Les fonctions pures peuvent être parallélisées et distribuées facilement ;
- Les programmes sont plus faciles à lire, en séparant la logique métier du code lié à la gestion des éléments techniques ;
- Les signatures des fonctions sont remplies d'informations.

Si la bascule peut être complexe à aborder, les transformations dans la manière de programmer se répercutent positivement dans tout le code qu'on écrit ensuite. Les choix faits par les créateurs de Kotlin permettent de facilement créer des HOF, sans avoir à connaître l'état de l'art de la programmation fonctionnelle. État de l'art que les plus curieux peuvent retrouver dans le projet Arrow-kt (https://arrow-kt.io/), une librairie qui apporte pléthore d'autres concepts que les fonctions d'ordre supérieur !

Pascale Bailly
Au sein des Labs d'Orange, évoluant entre marketeurs, ingénieurs et technologies informatiques, je développe les méthodes d'innovation par les business models et les algorithmes. Ces méthodes poussent vers des écosystèmes d'affaires aux impacts sociétaux et environnementaux vertueux. Démystifier, publier des articles et faire des conférences dans nos Ecoles sont une suite logique pour transmettre les clés des devenirs numériques… Et il IA un max de travail !

Partie 2

Algorithmes génétiques pour clusteriser un nuage

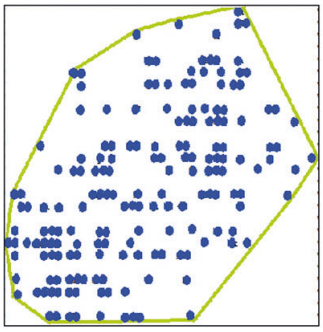
Ma petite nièce m'a demandé : « pourquoi le chameau a-t-il 2 bosses et le dromadaire une seule alors qu'ils font le même travail dans les mêmes déserts ? » J'ai répondu : parce que c'est préférable de… heu… ». En effet, quelle optimisation dame Nature cache-t-elle derrière cette différence ? Y a-t-il forcément unicité de l'optimum ? Avant de lui répondre j'ai décidé de tester des algorithmes génétiques pour l'optimisation de clusters ; trouve-t-on deux clusterisations différentes qui ont la même note de performance ? Nos clusters sont-ils vraiment optimaux ? Quels sont nos critères de jugement ?

Utilisation d'algorithme génétique pour une clusterisation

Ce champ d'application étant neuf, il est à peine effleuré sur le web : [https://en.wikipedia.org/wiki/Cluster_analysis, §Recent developments ou https://en.wikipedia.org/wiki/Genetic_algorithm].

Contexte professionnel et conditions d'utilisation

Nous utilisons régulièrement des méthodes de clustering en petites dimensions dans le cadre de traitement d'idées issues de *workshops* à Orange Labs Services. Le clustering étant lui aussi victime de l'explosion combinatoire des partitions possibles, je peux envisager une heuristique génétique pour son optimisation.
Les données considérées dans nos ateliers sont des nuages de quelques centaines de points, en 6 dimensions, qu'on cherche à clusteriser en une dizaine de clusters (ce nombre sera déterminé par l'algorithme génétique, pas par moi).
Dans le but de dessiner des résultats dans le présent papier, j'ai adopté un nuage simplifié de démonstration : il comporte 178 points en seulement deux dimensions, « x » et « y » cadrées entre 0 et 1. En admettant que l'on souhaite partitionner en 10 clusters, il existerait 10 puissance 177 possibilités à examiner ; l'**explosion combinatoire** est très impressionnante, d'où la nécessité de trouver des algorithmes plus rusés qu'un examen exhaustif.



6 *Nuage en deux dimensions.*

Ce nuage cobaye n'est pas aisé, il possède des trous en certains endroits, est aéré à l'est et très dense au sud-ouest. À vue d'œil on ne saurait pas trop comment le clusteriser manuellement, voyons ce que donnerait une méthode génétique.

Définition de notre codage

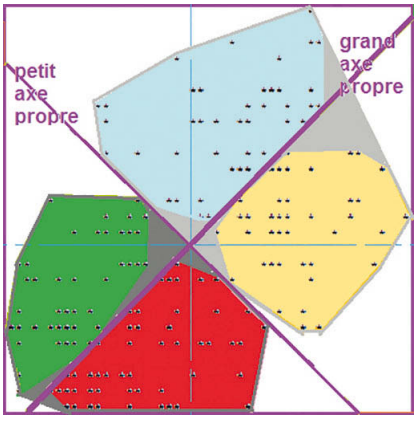
Tous les identifiants de cette démo sont des noms alphanumériques pour la facilité de lecture.
La population courante est un dictionnaire Python {nom du chromosome :son codage}.
J'ai opté pour des chromosomes lisibles par un développeur humain qui voudrait suivre le déroulement des opérations.
Un chromosome est une liste ordonnée d'allèles ; chaque allèle étant un duo [nom du point, nom de son cluster]. Ainsi les allèles sont toujours ordonnés selon l'ordre alphabétique du nom de leur point.
Cette caractéristique favorisera les croisements de chromosomes puisque tous les allèles seront positionnés de la même façon.
Exemple :
"Archiv1_ancien_essai": [["1", "clust_1"],["10", "clust_1 »],…., ["86", "clust_2"],…., ["98","clust_2"], ["99", "clust_2"]].

Initialisation aléatoire

L'objectif de notre algorithme est d'affiner un clustering, sans connaître le nombre de clusters idéal. Nous avons fait le choix d'initialiser nos individus au hasard, comme pour pi.

Deux chromosomes initiaux non-aléatoires

Nous avons rajouté à cette liste une partition à 4 clusters découpée sur la base de la théorie du calcul matriciel (7).



7 *Clustering en quatre parties*

Explication. On positionne le barycentre global du nuage (ici: x=0.4539 y=0.40886). On calcule la matrice 2x2 des

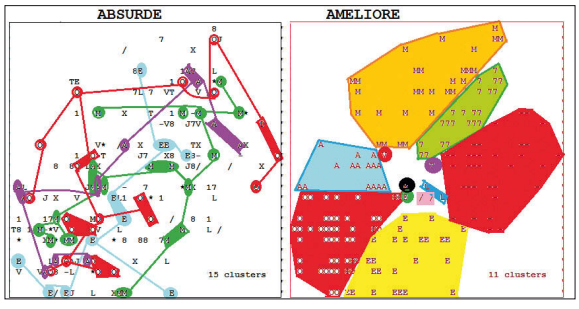
covariances, on calcule les deux valeurs propres, la grande et la petite, de cette matrice, puis les deux vecteurs propres associés, qui sont toujours orthogonaux. Sur la figure on trace à partir du barycentre, les deux axes propres qui en découlent. Ceci détermine les 4 clusters optimaux, jaune clair, bleu clair, vert foncé et rouge foncé.
Le but de notre exercice n'est pas de trouver l'optimum en 4 clusters ; en l'introduisant dans la population initiale de chromosome je compte seulement qu'elle pourra engendrer des enfants plus intéressants.

Métrique à utiliser pour la fonction de fitness

Rappelons que le mot anglais *fitness* désigne l'adaptation génétique des espèces à leur milieu naturel. Ici une partition est bien adaptée, présente une bonne *fitness*, s'il n'y a pas trop de clusters, ni trop peu, si chaque cluster est bien compact et bien écarté des autres, si, si, si...
On conçoit qu'il existe de nombreuses métriques qui servent à valider la pertinence d'un clustering donné.
Dans un premier temps j'avais choisi l'indice de Dunn, inventé par la statisticienne Olive Jean Dunn (1915–2008), qui est le rapport entre le diamètre maximum d'un cluster et la distance minimum qui sépare deux éléments classés séparément. Plus il est faible, meilleure est la partition : clusters bien compacts, grande séparation entre eux. Mais il y a un inconvénient : cet indice tend à augmenter fortement le nombre de clusters ce qui donne des petits diamètres de clusters ainsi que des bonnes séparations de points de clusters voisins. C'est la raison pour laquelle ensuite j'ai adopté pour fitness un panachage de divers indices.

Croisement de deux chromosomes

Précisons d'abord que le nom d'un point du nuage ("1","10","11"....,"117"...,"97","98","99") n'a rien à voir avec ses coordonnées ; il en résulte que deux points voisins par ordre alphabétique ne sont pas voisins sur le dessin en 2D.
Ceci entraîne un sérieux problème de vraisemblance des chromosomes car si on se contentait de croiser brutalement deux chromosomes on obtiendrait souvent des partitions absurdement entremêlées telles que celle-ci.



8 *Un clustering absurde et un raisonnable.*

À gauche, le coloriage d'un chromosome brut engendré par croisement. Les points d'un cluster sont situés n'importe où, c'est un véritable labyrinthe dont je n'ai dessiné que les clusters rouge, vert, violet et bleu clair, le coloriage des 11 autres serait fastidieux et inutile puisque l'on réordonnera tout ceci.
Voici le réarrangement du chromosome.

- on part de la partition plus ou moins absurde résultant d'un croisement brut ;
- on calcule le barycentre de chaque cluster, en lui donnant un nom provisoire ;
- on affecte chaque point au centre de gravité le plus proche (méthode de centroïde) ;
- on obtient ainsi une partition plus rationnelle du point de vue géométrique ;
- ensuite on donne un nom séquentiel à chaque cluster ("clust_1", "clust_2", "clust_3", etc.) ;
- enfin on code ce chromosome au format de liste ordonnée d'allèles, chaque allèle étant un duo [nom du point, nom de son cluster] comme ceci :

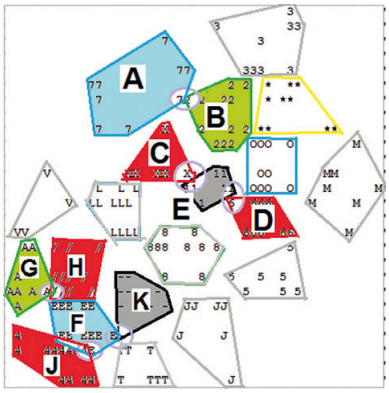
[["1", "clust_1"],["10", "clust_1 »],...., ["86", "clust_2"],....
on obtient ainsi le chromosome à droite, qui contient 11 clusters, dont 6 gros et 5 minuscules. Cette partition est devenue raisonnable, même si elle a peu de chances d'être la championne de sa génération (déséquilibre de la taille des clusters).
Pour info, voici les coordonnées des 15 barycentres qui serviront de points d'accrochage pour la remise en ordre.

Nom provisoire du barycentre	Coordonnées du barycentre
"384615384615":	"x": 0.384615,"y": 0.4111111},
"39102564102564":	"x": 0.391025,"y": 0.4027777},
"39142592592":	"x": 0.391025,"y": 0.3425925},
"425213675213":	"x": 0.425213,"y": 0.4629629},
"425641025641":	"x": 0.425641,"y": 0.350000},
"43356643356":	"x": 0.433566,"y": 0.386363},
"433760683760":	"x": 0.433760,"y": 0.314814},
"439102564102":	"x": 0.439102,"y": 0.416666},
"446886446886":	"x": 0.446886, "y": 0.61111},
"46270396270":	"x": 0.462703,"y": 0.368686},
"470695970695":	"x": 0.470695,"y": 0.416666},
"516483516483":	"x": 0.516483,"y": 0.420634},
"529914529914":	"x": 0.529914,"y": 0.524691},
"534516765285":	"x": 0.534516,"y": 0.431623},
"58974358974":	"x": 0.589743,"y": 0.444444

Mutations généralistes

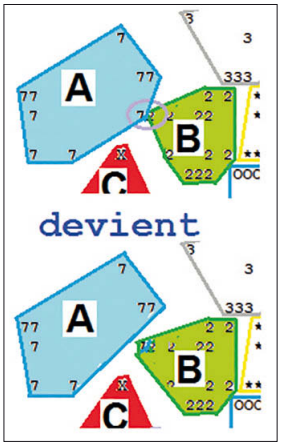
La première idée de mutation consiste à considérer qu'un rayon cosmique frappe un chromosome aléatoire de la population courante, aléatoirement sur un allèle et change le nom de son cluster (en tirant aléatoirement parmi les noms de clusters présents dans ce chromosome) ; au total une cascade de trois aléas (un chromosome, un allèle, un cluster). Ensuite ce nouveau mutant est reconditionné selon la méthode précédente. Ce re-calcul des centroïdes et des affectations est quelquefois susceptible de modifier le nombre et la disposition des clusters. Si oui, ce serait une bonne chose qui favoriserait la diversité.
Des mutations orientées-clusterisation.
Notre but est d'obtenir des bonnes clusterisations. Donnons un petit coup de pouce au destin en créant des mutations pas du tout aléatoires, mais sciemment orientées dans un sens favorable à un beau découpage des clusters. Voici celles que j'ai imaginées, les lecteurs pourront en trouver d'autres.

Mutation qui corrige un seul point d'une frontière



9 *Points litigieux de frontières (qui sont trop proches).*

Un coup d'œil à cette figure interceptée au cours d'un run nous interpelle. Nous voyons des endroits où un cluster semble marcher sur les pieds d'un voisin. Par exemple B vert qui heurte A bleu pourrait très bien céder un point à A, ou l'inverse, la frontière serait plus nette. Il en est de même pour les autres conflits :
C rouge et E gris ;
D rouge et E gris ;
G vert et H rouge ;
K gris et F bleu ;
J rouge et F bleu.
Exemple d'une telle modification locale si le point x est dans le cluster A bleu :



10 *Rectification d'une frontière (le point bleu est passé au vert).*

Le cluster vert est amélioré, la frontière bleu/vert est plus nette.

Mutation qui re-dessine légèrement toutes les frontières
Il s'agit d'une procédure facile à comprendre :
On prend la clusterisation championne ; pour chaque point x à tour de rôle, on l'enlève du nuage, on recalcule les barycentres des clusters, et on rattache x au barycentre le plus proche. Ainsi on peut espérer que les rattachements seront globalement meilleurs, donc que la fitness sera meilleure.

Mutation pour accroître le nombre de clusters
Lorsqu'on photographie les générations au cours de runs, on s'aperçoit que le champion de chaque génération se stabilise sur un nombre constant de clusters, par exemple 17. Je n'ai pas trouvé d'explication dans la littérature. Alors il faut imaginer des mutations qui font varier le nombre de clusters, en plus ou en moins. Pour ce faire je choisis aléatoirement des chromosomes, et pour chaque allèle je transforme aléatoirement le nom de son cluster en rajoutant le caractère "1" ou "2" ou rien, par exemple ["1", "clust_1"],["10", "clust_1 »],…., ["86", "clust_2"],… Deviendra : ["1", "clust_12"],["10", "clust_1 »],…., ["86", "clust_21"],...
Ceci, déjà dans la présente génération, augmente sensiblement le nombre de clusters du chromosome muté, mais de plus, à la génération suivante, un croisement pourra nous donner des enfants inattendus.

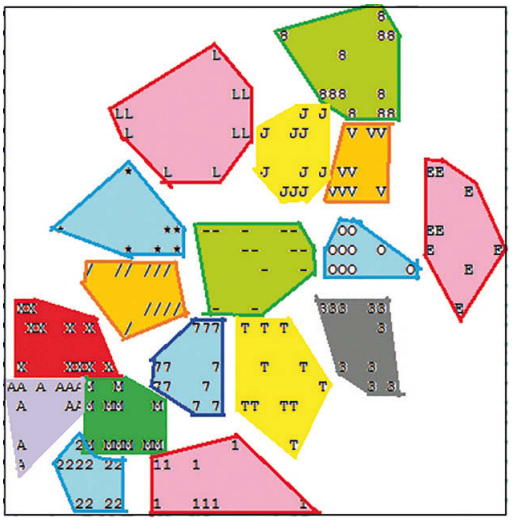
Mutation pour diminuer le nombre de clusters
Pour un chromosome, cette mutation choisit 2 clusters au hasard et les fusionne. Ceci indépendamment de leur éloignement. Donc il y a un cluster de moins, et j'espère que lors de la prochaine génération, un croisement pourra nous donner des enfants inattendus et performants.

Algorithme global

Il a la même structure que celui de pi. Ce qui change, c'est la grande variété des mutations et le méta-paramètre N_SELEC.
Tous les calculs sont d'un niveau mathématique simple.
Les fonctions de fitness, que nous verrons plus loin, consistent à calculer beaucoup de distances — j'ai choisi la distance euclidienne, mais vous pouvez tenter une autre — ou calculer beaucoup de moyennes et d'écarts-types.
Les fonctions de mutations consistent très souvent à manipuler des dictionnaires, des listes, faire des tris, éventuellement calculer des barycentres et des distances.

Bilan de cette tentative

J'ai effectué de nombreux runs. Exemple dans l'un d'eux, la partition gagnante fut Figure 11.



11 *Chromosome gagnant.*

Il y a 17 clusters relativement bien séparés, sauf au sud-ouest qui est une zone très dense ; et où le bleu clair gêne trop le vert sombre. Personnellement je trouve que 17 clusters c'est un peu trop pour 178 points. Ceci dit des générations suivantes pourraient proposer un meilleur chromosome gagnant.

Panachage de métriques de fitness

Ceci est une remarque valable pour tous les algorithmes de clusterisation, génétiques ou pas. Il existe dans la littérature tellement de métriques avec leurs avantages et inconvénients qu'on est en droit d'hésiter. En voilà une liste non exhaustive dont on trouve les définitions sur le web :

- L'indice Silhouette : S : [-1 ; 1]. Plus elle est proche de 1, plus chaque objet a de chance d'être dans le bon cluster ;
- L'inertie inter-clusters ;
- L'Indice de Dunn déjà mentionné ;
- L'indice de Davies-Bouldin (DBI) : positif, plus l'indice est proche de 0, plus les clusters sont éloignés les uns des autres, et plus les éléments d'un cluster sont proches les uns des autres ;
- La variance du nombre de points par clusters (V) : positive, plus elle est basse, plus les clusters sont équilibrés, mais attention ! Avec 178 clusters de 1 point, on obtient la variance optimale qui est 0.

Une bonne clusterisation ?

À propos, qu'appelle-t-on une bonne clusterisation ? Il semble que nous courons après plusieurs lièvres ; nous voulons des clusters :

- de tailles équilibrées ;
- pas trop nombreux ;
- fortement séparés ;
- de forme plutôt ronde ;
- bien denses.

Mais dans la vie réelle, notre nuage de points est rarement compatible avec tous ces objectifs, comme on peut le voir au sud-ouest très dense du nuage cobaye. Il faut faire quelques compromis entre les indices.

Mon choix actuel

Après un nombre certain de tentatives, j'ai adopté sept indices à combiner.

autokmeans_soustract : indice inventé par A. Letois, Automeans, magazine MISC HS018, novembre 2018 ;

C'est l'inertie inter-clusters à laquelle on soustrait le carré du rayon moyen du nuage multiplié par le nombre de clusters. Le but recherché est de freiner l'augmentation abusive du nombre de clusters.

écarttype_taillesclusters : comme son nom l'indique, cet indice vise à obtenir des clusters de tailles voisines.

autok_quotient : même philosophie que autokmeans_soustract, sauf qu'au lieu de soustraire le terme correctif on le place au dénominateur.

dunnsur_n : l'indice de Dunn est trop brutal, je prends son inverse et je le tempère en le divisant par une fonction affine du nombre de clusters.

maillage_somme : plus les clusters sont distants entre eux, meilleur c'est. Alors cet indice est carrément la somme de toutes les distances entre couples de barycentres des clusters, divisés par la même somme où l'on remplace la distance inter-barys par le rayon moyen du nuage.

Nbclusters : je considère arbitrairement que le nombre « idéal » de clusters est la racine carrée de la taille du nuage, donc ici environ 13 clusters de 13 points : de cette manière mon indice nbclusters mesure l'écart entre ce nombre idéal et le nombre de clusters du chromosome.

Dunn : déjà mentionné.

Quant à la combinaison de ces 7 indices, c'est une tout autre histoire ; je leur affecte des poids différents, dans l'ordre décroissant de leur énumération ci-dessus. Les valeurs précises des poids ont été ajustées en faisant un grand nombre de runs de combinaisons. Cette question à elle seule pourrait faire l'objet de plusieurs articles.

Conclusion globale sur l'algorithmique génétique

Heuristique, hasard et biomimétisme, telle est la devise des algorithmes génétiques.

Au fil des essais ces petits exemples nous ont appris huit choses.

1) Même si on ne connaît presque rien sur un sujet, on peut quand même tenter un algorithme génétique qui rendra un résultat plus ou moins bon et nous suggérera des pistes.
2) Mais l'efficacité du traitement est fortement liée à notre manière de coder un chromosome, d'effectuer un croisement, d'effectuer les mutations et de choisir les méta-paramètres de pilotage (taille d'une population, taux de mutations, nombre limite de générations).
3) La vitesse est aussi une question cruciale. Elle résultera d'un compromis entre nombre de générations, taille de la sélection, indice de fitness, et durée des calculs d'une fitness. Nous souhaitons converger en peu de générations, ce qui peut être favorisé par l'abondance des mutations imaginatives ; mais plus on imagine des mutations et plus la durée des calculs sera importante. À nous de tester divers paramétrages.
4) L'évitement du coinçage dans un sous-optimum local (en : *to be stuck on a*...). Problème commun à tous les algorithmes d'approximations successives, il se rencontre souvent. Il n'y a pas de remède miracle, le développeur doit imaginer des mutations ingénieuses liées à la nature du problème, quitte à alourdir les calculs.

Conclusion spécifique à la clusterisation

5) Concernant la clusterisation, nous savons qu'il n'existe pas de consensus universel sur le meilleur indice de jugement de la qualité des partitions. Chacun devra se faire une opinion, par exemple en testant divers panachages de diverses métriques pour finalement adopter la fonction de *fitness* correspondant à un panachage que l'on juge « bon ». Ceci est une problématique générale de la clusterisation qui se pose aussi bien avec l'algorithme K-means qu'avec le co-clustering spectral, rien à voir avec la méthode génétique.
6) Il est clair que les calculs de divers indices de fitness sont toujours chronophages.
7) On ne doit pas hésiter à inventer d'autres mutations dans le but d'augmenter la diversité.
8) Tout dépend du problème que l'on traite, de l'urgence d'un résultat attendu par vos clients ou collègues de travail ; si vos données sont volumineuses et si votre commanditaire est impatient, optez pour un spectral coclustering. Si le tableau des données est modeste et si on vous laisse un certain délai, tentez donc des variantes d'algorithme génétique.

Un rêve qui s'appelait no-code

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
Rédacteur en chef : François Tonic
Secrétaire de rédaction : Olivier Pavie
Ont collaboré à ce numéro : ZDnet, Franck Dubois
Les contributeurs techniques : A. Caussignac, A. Roman, L. Juliard, S. Ducasse, N. Bouraqadi, L. Fabresse, G. Polito, P. Tesone, C. Hermandez Philips, S. Stormacq, Y. Azoury, N. de Mauroy, W. Almeida, T. Falque, F. Billion, S. Pontoreau, J. Mikel Inza, Y. Bertrand, T. Boutin, V. Treve, P. Bailly
Photo de couverture : © RichVintage - **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, Octobre 2020

Ce numéro comporte un encart jeté PC Soft pour les abonnés.

Abonnement :
Programmez - service abonnement,
57 rue de Gisors, 95300 Pontoise
abonnements@programmez.com

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
39 € (monde entier) souscription sur www.programmez.com

NOUVEAU !

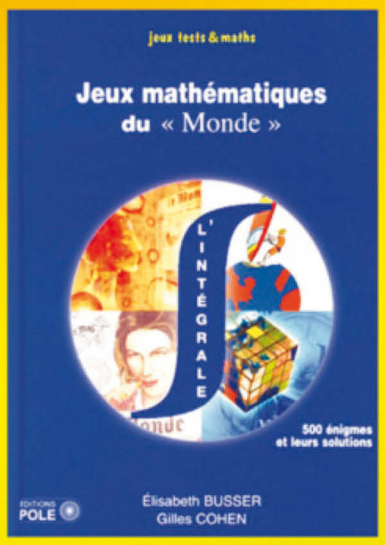
l'abonnement numérique

INTÉGRAL MATHS

Et aussi...

sur

affairedelogique.com



400 problèmes du Monde et leurs solutions

28 nos de Tangente

sur

tangente-mag.com

17 hors séries

53 nos de Tangente Éducation

sur

tangente-education.com

pour seulement

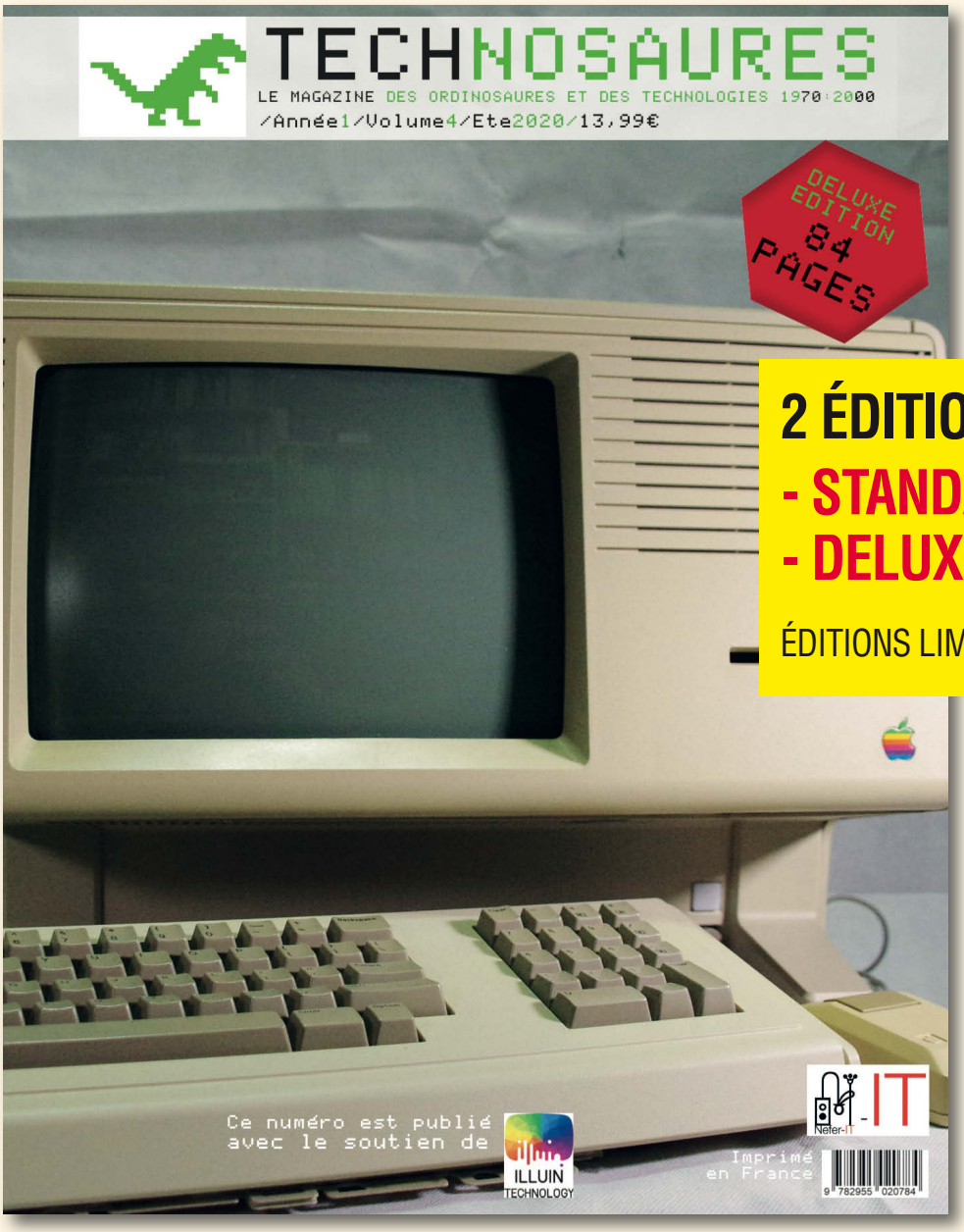
3,99 €

par mois

www.infinimath.com/librairie

NUMÉRO EXCEPTIONNEL

100 % APPLE LISA



2 ÉDITIONS :

- STANDARD *52 PAGES*

- DELUXE *84 PAGES*

ÉDITIONS LIMITÉES

Commandez directement sur **www.programmez.com**

Standard édition : **8,99 €** *

Deluxe édition : **13,99 €** *

** Frais de port : 1,01 €*

TECHNOSAURES

le magazine du **rétro-computing**