RUST Python Smalltalk Cloud Computing

PROGRAMMEZ!

PROGRAMMEZ!

Le magazine des dév

NOUVELLE FORMULE

N°244
01/02
2021

TOUT SAVOIR SUR LE LANGAGE RUST

- syntaxe
- mes premiers codes
- les outils compatibles
- déploiement

INITIATION

LE CLOUD COMPUTING

Dans ce numéro,
trouve ton bonheur de codeur :

spaCy

SMALLTALK, partie 2

CORDOVA : TECHNOLOGIE À OUBLIER !

Le seul magazine écrit par et pour les développeurs

M 04319 - 244 - F: 6,50 € - RD

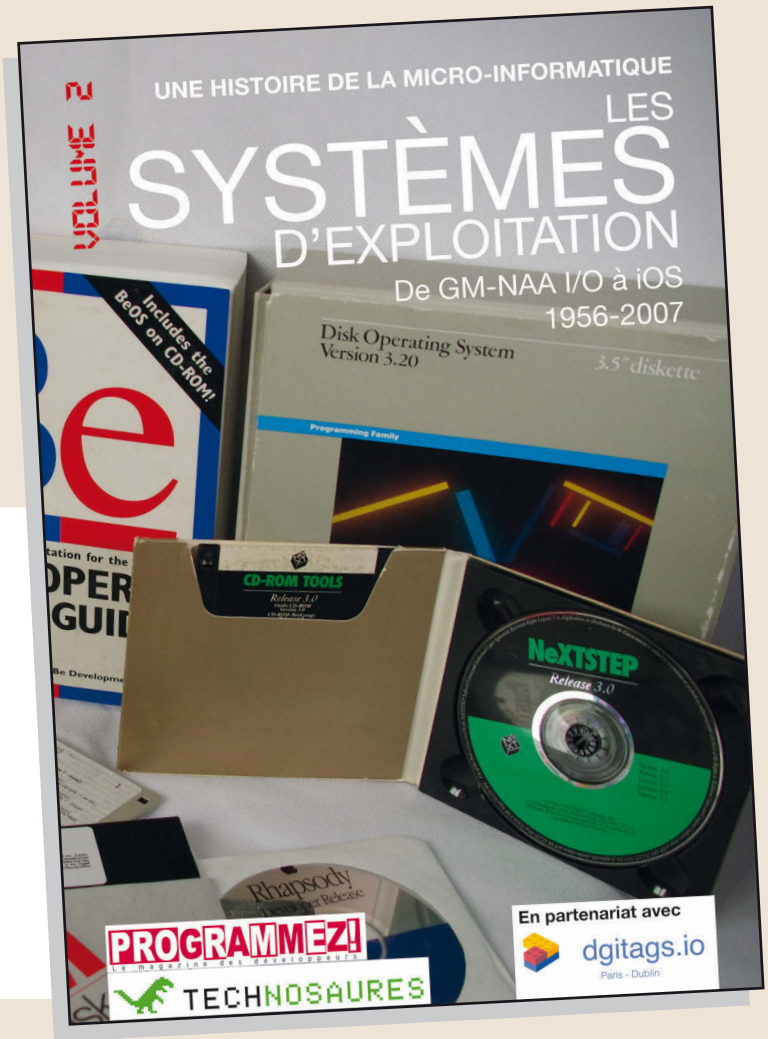
Printed in EU - Imprimé en UE - BELGIQUE 7 € - Canada 9,80 $ CAN - SUISSE 13,10 FS - DOM Surf 7,50 € - TOM 1020 XPF - MAROC 55 DH

Le CADEAU idéal pour toutes/tous les geeks !

UNE HISTOIRE DE LA MICRO-INFORMATIQUE

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

116 pages. Format mook.



UNE HISTOIRE DE LA MICRO-INFORMATIQUE

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

100 pages. Format mook.

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

Contenus

Divers

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

#244*

Destination : RUST

Préparer un numéro de Programmez!, c'est prendre le temps de la réflexion et durant quelques jours, je regarde les blogs, les échanges dans les communautés. Même si les principaux thèmes sont fixés depuis plusieurs semaines, je ne perds jamais ce besoin de faire de la veille technologique. C'est indispensable, c'est incontournable. Cette veille techno est un des commandements du développeur. C'est comme le RTFM, un réflexe à appliquer tout le temps et pas seulement quand on a le temps.

Après la disponibilité de .Net 5, et notre hors-série #2, la pression est un peu retombée. Nous pouvons préparer plus calmement 2021. Mais, on va peut-être mieux lire les CGU et regarder de plus près les warnings à la compilation.

Honnêtement, nous avons forcément raté un truc pour les binaires de 2020. Un mauvais test unitaire ou une gestion des exceptions mal codée… Ou alors, les développeurs ont raté une spec projet quelque part. Bon OK, quand on nous a présenté le concept 2020, je me suis dit, ouais, c'est pas mal, j'aime bien les couleurs, l'interface. Mais j'aurai dû me méfier entre le beau prototype et le livrable. Et les mises à jour ont mis le bordel un peu partout. Ah, encore des tests de non-régression oubliés… Quand on vous dit de ne jamais mettre en prod un vendredi soir…

« SAVOIR, C'EST POUVOIR »

Avant d'exécuter 2021, on peut donc se montrer un peu sceptique, car le code reste le même, avec quelques modifications ici et là. En fait, 2020 c'est un peu comme Windows Millenium : un emballage connu, mais un logiciel mal fini. Est-ce que 2021 sera notre Windows XP ? Réponse d'ici l'été prochain avec le Service Pack 1. Espérons simplement que 2021 ne soit pas un reboot mais un new project.

Dans ce numéro, comme vous le lirez dans quelques minutes, nous avons plusieurs gros dossiers. Le principal concerne le langage RUST. C'est la nouvelle vedette des développeurs et des géants de l'informatique. Le fait qu'Amazon Web Services s'y intéresse, montre le potentiel du langage. Microsoft s'y intéresse aussi et on en parle pour les conteneurs et les outils de virtualisation.

« LE CODE C'EST LA VIE »

RUST, LE langage incontournable de 2021 ? Pas forcément, mais je vous conseille de le suivre et même de coder un peu avec pour comprendre son potentiel. Va-t-il suivre l'exemple de Python qui s'est imposé comme le langage pour le machine learning et le deep learning ? Réponse dans quelques mois.

Pour 2021, nos résolutions sont simples :

- Continuer à parler codes et technologies
- Prédire la mort du Cobol (comme chaque année)
- Lire que le C++ est dépassé
- Quel rachat dépassera les 20 milliards de $?
- Espérer que l'impression 3D prouvera son utilité
- S'améliorer à Tétris (je n'ai jamais été bon)
- Que Matrix 4 soit à la hauteur des attentes
- Impatient de voir Fondation sur Apple TV+
- Impatient de voir Dune, le Seigneur des anneaux (version Amazon) et la suite de The Mandalorian
- Voir, ou pas, House of the Dragon

Bonne année 2021 !

François Tonic
Rédacteur en chef
ftonic@programmez.com

**en réalité le # 246 avec les hors-séries*

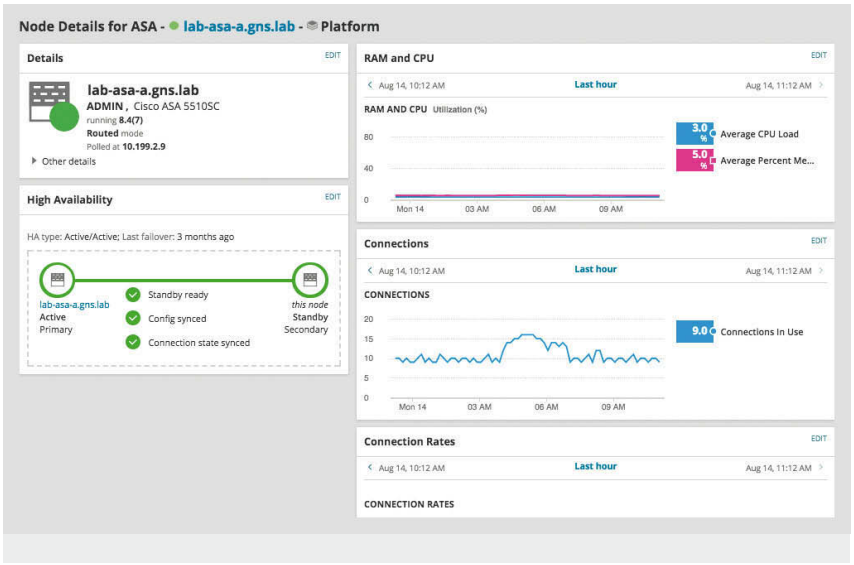
LES PROCHAINS NUMÉROS

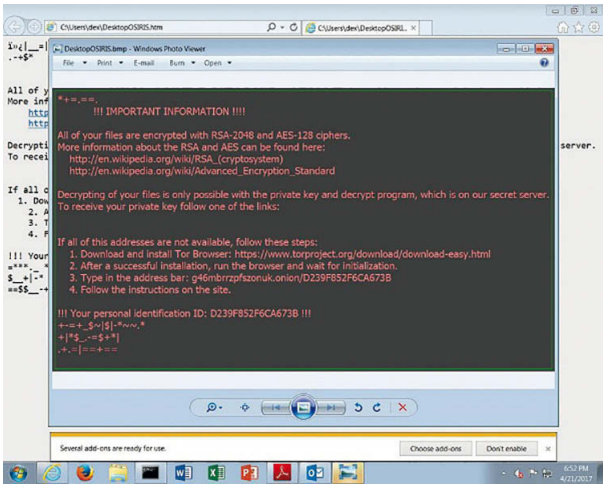
Hors série #3 spécial froid
100 % Amazon Web Services
Disponible dès le 29 janvier

Programmez! n°245
PHP 8.0 : overview & 1er bilan
Retour sur Drupal 9
Disponible dès le 5 mars

SolarWinds : attaque en cascade pour le gouvernement américain

Tout a commencé avec le piratage de la société américaine FireEye : ce géant de la cybersécurité a annoncé au début du mois de décembre s'être fait pirater. Les attaquants en ont profité pour dérober plusieurs outils internes. La société a fait preuve d'une grande transparence, mais restait peu loquace sur la technique utilisée par les attaquants pour infiltrer ses systèmes. La réponse est venue une semaine plus tard au travers d'un communiqué de la société SolarWinds : un groupe d'attaquant a piraté son système afin de diffuser une mise à jour malveillante de son logiciel phare, la plateforme Orion. Et ce piratage a eu de lourdes conséquences, puisque c'est cette même porte dérobée que les pirates ont exploitée pour pirater le ministère américain des finances, le ministère de la sécurité intérieure, le ministère de la santé, le département d'État ou encore l'équivalent américain de l'Anssi, la CISA. Un coup à trois bandes dont les États-Unis se souviendront longtemps.





Vinnik : condamné, mais blanchi pour Locky

Le tribunal a rendu son jugement dans le procès du russe Alexander Vinnik. Celui-ci était accusé par les autorités françaises d'être le donneur d'ordre et le principal bénéficiaire du rançongiciel Locky, qui a sévi en France en 2016 et fait de nombreuses victimes. Mais pour les juges, les accusations liées à Locky n'étaient pas suffisantes : Alexander Vinnik a été condamné à 5 ans de prison et 100 000 euros d'amende pour le blanchiment des sommes d'argent extorquées au travers de ce logiciel malveillant, mais a été reconnu innocent de toutes les charges liées à la conception, à l'utilisation et à la diffusion du rançongiciel. Il fait néanmoins l'objet d'une demande d'extradition aux États Unis et en Russie, et n'est donc pas au bout de ses peines.

L'Europe veut pouvoir juguler les GAFA

La Commission européenne a présenté la première version de son nouveau cadre réglementaire sur les services numériques. Cette refonte vise à donner aux Européens des outils visant à s'assurer de la loyauté des fournisseurs de services numérique, en imposant par exemple des obligations de transparence aux acteurs majeurs du secteur, ainsi qu'aux organisations suspectées de « verrouiller » certains marchés. Sans directement les nommer, les grands acteurs américains tels que Apple Google ou Facebook sont évidemment dans le viseur. Et pour s'assurer d'avoir toute leur attention, la commission propose des sanctions pouvant monter jusqu'à 10 % des revenus globaux de la société sanctionnée. Le texte doit encore être adopté par le parlement et le conseil de l'UE, qui pourront l'amender au passage. Le lobbying bruxellois s'annonce donc corsé dans les prochains mois.

Soriano : Goodbye Arcep, hello IGN

Au terme d'un mandat de six ans, le directeur de l'Arcep Sebastien Soriano annonce son départ. Son successeur n'a pas encore été annoncé, mais Soriano partira prendre la tête de l'institut national de l'information géographique et forestière dès les premiers jours de janvier. Un choix assez logique pour celui qui avait multiplié les initiatives de cartographie des équipements et antennes de réseaux mobiles à la tête du régulateur des télécoms.

Cryptomonnaie : l'anonymat ne passera pas

Le ministre de l'économie Bruno Lemaire a annoncé un changement dans la régulation des acteurs français opérant dans le secteur de la cryptomonnaie : au travers d'une ordonnance, le gouvernement impose à l'ensemble des acteurs des contrôles stricts sur l'identité des utilisateurs, visant à lutter contre le blanchiment d'argent et le financement du terrorisme. Une régulation qui désole les entreprises du secteur. Elles déplorent une régulation bien trop sévère pour un secteur encore balbutiant en France, mais qui doit faire face à une concurrence mondiale.

Google et Amazon : des cookies difficiles à digérer

La CNIL a sanctionné Google et Amazon, leur infligeant des amendes de 100 millions d'euros et 35 millions d'euros, pour des infractions à la législation en vigueur sur les cookies. Dans les deux cas, le problème relevé par la CNIL est sensiblement le même : le dépôt de cookies publicitaires sans en informer l'internaute et donc sans son consentement. Sans grande surprise, les deux entreprises sanctionnées ne sont pas tout à fait d'accord avec le verdict de l'autorité, et rappellent que la protection des données des utilisateurs est depuis toujours au cœur de leurs préoccupations. Pas suffisamment, aux yeux de la CNIL, mais ça passe toujours mieux en râlant un peu.

Meetups Programmez!

19 janvier : être développeur en 2021, avec la participation d'Arolla
23 février : les nouveautés dans le monde Java, avec la participation d'Arolla
23 mars : l'art du refactoring, avec la participation d'Arolla

Où : virtuel pour le moment
à partir de 18h30

DevCon by Programmez !

25 mars : conférence sur l'informatique quantique, à partir de 13h30.
Sous réserve de la possibilité d'un événement physique.

Informations & inscription : programmez.com

Janvier

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

Février

1	2	3	4	5	6	7
				webstoriesconf	Fosdem (Belgique)	Fosdem (Belgique)
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
	Meetup Programmez!					

Mars

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

A l'heure où nous partons en impression, peu de conférences ont été annoncées pour le 1er trimestre 2021.

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

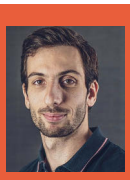
100% code

DISPONIBLE !
dès le
29 janvier 2021

Couverture provisoire

Kiosque - Abonnement - Versions papier & PDF

www.programmez.com



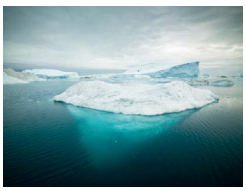
Renaud-Alexis Vol
Expert Technique
SQLI

SQLI DIGITAL EXPERIENCE

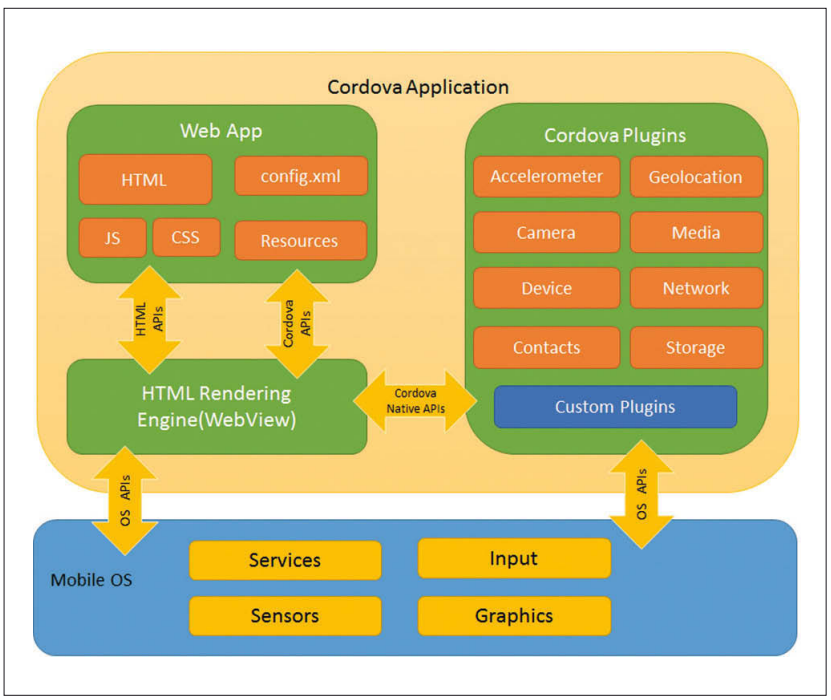
Cordova : pourquoi ne faut-il plus l'utiliser dans un développement mobile hybride ?

Pour nombre d'entreprises, le besoin de développer des solutions mobiles est toujours aussi présent. Le mobile fait partie intégrante de leur stratégie marketing afin d'atteindre leurs cibles et de s'adapter à leurs comportements. La plupart des entreprises optent pour les applications cross-plateformes, en ayant recours en l'occurrence à Cordova. Cette technologie est considérée comme l'une des solutions les plus accessibles et bon marché pour le développement de ces applications. Pourquoi cela ? Et pourquoi je vous suggère de ne plus l'utiliser. Voici le sujet que je vous propose de découvrir.

Cordova est une technologie permettant d'adresser **plusieurs plateformes** en ayant l'avantage de produire **une seule base de code**. Elle est principalement utilisée pour développer des **applications mobiles sur iOS et Android**. La majorité du code applicatif est basée sur des technologies **JavaScript** (~95 %). Il est donc possible de choisir n'importe quel **framework JavaScript** pour se lancer dans le développement.



Cordova expose une importante **API en JavaScript permettant de communiquer avec les fonctionnalités matérielles** des appareils mobiles telles que l'appareil photo, la géolocalisation, les notifications, etc. avec la possibilité de rajouter un tas de plug-ins selon les besoins pour accéder à d'autres API.



Architecture d'une application Cordova
(source : https://cordova.apache.org/docs/en/latest/guide/overview/index.html)

La face émergée de l'iceberg

Comme il s'agit d'une des premières technologies cross-plateformes du marché, on y voyait auparavant plusieurs avantages à utiliser Cordova en comparaison aux technologies natives :

- Il n'était plus nécessaire d'écrire deux codes différents en Objective-C (ou Swift) pour iOS et Java (ou Kotlin) pour Android, ce qui laissait penser que les coûts pouvaient être divisés par deux ;
- Le besoin sur le mobile grandissant, il était très rare de trouver des profils de développeurs avec des compétences sur une, voire deux technologies natives. Avec Cordova, il suffit de trouver des profils avec une appétence sur le front, ce qui est plus fréquent ;
- Les technologies natives imposent des règles strictes en termes de style et d'expérience utilisateur. Cordova permet de s'en affranchir et de laisser libre cours à l'imagination des designers.

Des avantages, mais pas seulement…

C'est bien beau tout ça, mais il se cache derrière une face plus sombre que j'ai pu expérimenter lors d'un projet mobile dans le domaine bancaire :

- **Des coûts divisés par deux ? Pas forcément !**
  - Il est souvent demandé de **reproduire des comportements natifs** en termes de style et d'animation, pour lesquels ils sont déjà implémentés sur les technologies natives. Mais pour Cordova, cela nécessite de **réécrire des feuilles de style adaptées** ;
  - Il est toujours **nécessaire de tester sur toutes les plateformes,** car en réalité, j'ai souvent pu constater que pour un même cas de tests, il apparaissait des **comportements différents entre iOS et Android.** Des corrections spécifiques doivent donc être développées.

- **Des profils front, oui, mais…**
  - Comme il est possible de choisir parmi **un large choix de framework JS** (Angular, ReactJS, framework propriétaire,

etc.), il faudra tout de même prévoir la plupart **du temps des formations** ;
- **Le développeur front n'a pas forcément une culture du mobile** et ne prend pas en considération un certain nombre de paramètres lors de ses développements ; par exemple, les particularités de navigation, le réseau, la status bar, et j'en passe ;
- Il est tout de même nécessaire d'avoir **au moins un développeur expérimenté sur iOS et Android**.

- **Réinventer les concepts de style et d'interaction sur mobile n'est pas forcément une bonne chose pour l'expérience utilisateur**. En effet, celui-ci pourra se voir proposer une expérience différente à celle offerte par le natif habituellement, et ainsi abandonner l'utilisation de votre application mobile.

Quelles sont les alternatives à Cordova ?

Si vous souhaitez tout de même rester sur une solution mobile hybride, je vous conseille de vous diriger vers la solution **Ionic** qui présente les avantages suivants en comparaison avec Cordova :

- Ionic est une surcouche de Cordova, mais elle contient déjà une **riche librairie de composants** déjà pensés pour le mobile permettant de **préserver l'expérience utilisateur** ;
- Ces composants assurent normalement déjà une **compatibilité avec les différentes plateformes**. Il est donc plus rare d'avoir des surprises sur une plateforme spécifiquement. **On y gagne en termes de coûts** ;
- Le choix parmi 3 frameworks dont Angular2+. Les profils front les plus courants sur le marché sont sur cette technologie. Ionic a rendu également compatible sa librairie de composants avec React et Vue.

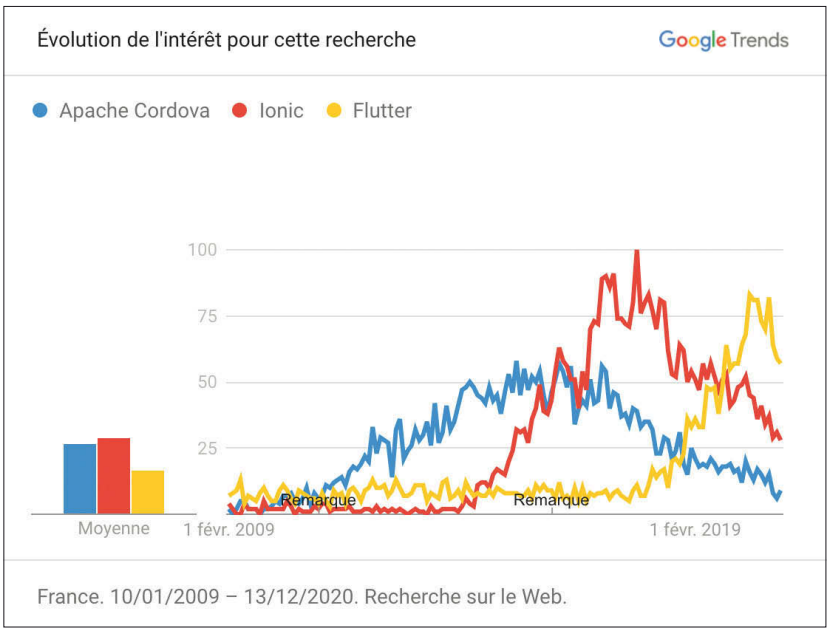
Capacitor

Celui qui se proclame successeur « spirituel » de Cordova s'appelle **Capacitor**. Ce dernier a d'ailleurs été produit par les équipes de Ionic. Même s'il est aussi possible grâce à cet outil de **construire une application mobile à partir de n'importe quelle application web**, il peut se coupler au framework Ionic, ce que je recommande afin d'éviter de repartir dans les travers de Cordova vus précédemment. Ce qui diffère avec Capacitor est sa manière de gérer la partie native qui est **plus moderne et performante**.

Capacitor propose **une stratégie de migration depuis Cordova** ainsi que la marche à suivre au niveau technique. Il est possible de migrer la plupart des plugins Cordova vers des plugins Capacitor, ceux pour lesquels ce ne serait pas possible, pas d'inquiétude, Capacitor les supporte. La version 2 de Capacitor est sortie début avril 2020.

Mais si vous pouvez investir un peu plus tout en restant sur une solution cross-plateformes, il sera préférable de ne pas choisir de technologie se basant sur une webview. **On y gagne ainsi en performance**, comparable au natif, **et en stabilité** en ce qui concerne les développements.

Les solutions que je propose sont **React Native** et **Flutter** avec une préférence particulière pour le second. **L'apprentissage** sur ce type de frameworks est peut-être **un peu compliqué** au départ, mais vous assura **un bon rendement sur le développement ensuite**.



Tendance des recherches Google sur Cordova, Ionic et Flutter sur les 10 dernières années en France

Conclusion

En conclusion, avec de l'expérience sur Cordova, nous nous sommes aperçus qu'un tas d'inconvénients émergeait lors des développements. Si cette solution reste encore d'actualité, je conseille toutefois de la laisser de côté au profit d'autres technologies cross-plateformes accompagnées d'un kit adapté aux contraintes du mobile. Les solutions en vogue en ce moment sont **React Native et Flutter** qui nécessitent un peu plus d'investissement qu'**Ionic**, qui elle représente un bon compromis.

SITUATION EN 2020

Adobe a annoncé, en août dernier, la fin de son soutien (en développeurs et en $) à PhoneGap, PhoneGap Build et à Cordova (Apache). Adobe se désengage donc totalement. Il préconise de migrer sur Framework7, NativeScript, Ionic, PWA, etc.

Côté Apache, Cordova continue à être maintenu et développé. Début octobre, Cordova Electron 2 .0 était annoncée.

La rédaction

Karbon Platform Services : des interfaces de données simplifiées pour les déploiements Edge et IoT

À travers sa plateforme, Nutanix permet de simplifier la gestion de nombreux aspects du développement, du déploiement et de la gestion du cycle de vie des applications, et apporte aussi des solutions pour gérer les problématiques liées à l'IoT et au Edge.

En septembre dernier, Nutanix a dévoilé Karbon Platform Services, une solution de Platform-as-a-Service (PaaS) multicloud basée sur Kubernetes avec des fonctions de sécurité intégrées pour accélérer le développement et le déploiement des applications basées sur des microservices.

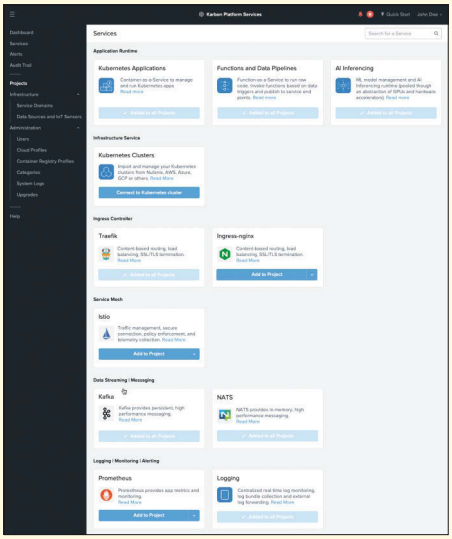
Karbon Platform Services est conçu pour permettre aux développeurs de créer rapidement des applications « cloud native » et de les déployer en quelques clics que ce soit dans le cloud, on premises ou en edge. Pour le déploiement d'applications IoT sur le Edge, Karbon Platform Services s'appuie sur les acquis de Xi IoT et ajoute des fonctions innovantes d'abstraction, permettant la mutualisation entre conteneurs et fonctions des capacités d'accélération AI et ML des GPU et autres cartes accélératrices.

KPS fournit un véritable plan de contrôle SaaS, accessible via une interface web simple d'emploi, pour gérer le cycle de vie de l'infrastructure multicloud et des applications critiques de l'entreprise. Il introduit le concept de « domaine de service », une puissante abstraction, qui permet aux clients de déployer les services de la plateforme sur n'importe quelle infrastructure cloud, tout en leur permettant de les piloter via des API et une interface utilisateur cohérentes.

Karbon Platform Services offre un ensemble riche de services, que les développeurs peuvent déployer à la volée en fonction de leurs besoins pour supporter leurs applications conteneurisées :

Services de Runtime applicatifs :

- Container as a service : permet de provisionner, exécuter, gérer et superviser des applications Kubernetes



comme des pods de conteneurs isolés sans avoir à gérer la complexité sous-jacente de Kubernetes

- Kubernetes as a service : le service d'administration de clusters Kubernetes permet de contrôler et de superviser des clusters Kubernetes à distance
- Services de Fonctions et de datapipeline : ils permettent d'exécuter des logiques métiers et s'appuyant sur les runtimes applicatifs embarqués dans la plateforme ou en s'appuyant sur les frameworks de type Functions-as-a-service choisis par le client. Ils peuvent aussi déclencher l'invocation de fonctions en fonction d'événements (comme la transmission d'une donnée), puis de transmettre les données transformées par une fonction vers un « service domain » local ou externe via les connecteurs incorporés à KPS.
- Services d'inférence AI et ML : la gestion des modèles de machine learning et les runtimes d'inférence AI sont mutualisés entre conteneurs et fonctions grâce à la mise en œuvre d'une couche d'abstraction des services d'accélération matériels (GPU ou cartes d'accélération AI).

Contrôleurs d'entrée/répartition de charge :

- Ngnix : routage de contenu, répartition de charge, terminaison SSL/TLS
- Traefikroutage de contenu, répartition de charge, terminaison SSL/TLS

Service Mesh :

- Istio : gestion du trafic, sécurisation et observabilité

Streaming et messagerie de données :

- Kafka : système persistant de message et de gestion d'événements à hautes performances
- NATS : agent de message en mémoire à hautes performances

Gestion des logs, surveillance et alerte :

- Prometheus : monitoring de la performance et des métriques applicatives
- Logging : monitoring centralisé et en temps réel des logs, audit et transfert des logs vers des plates-formes tierces (comme AWS CloudWatch)

Nutanix Karbon Paas Platform s'adapte à de nombreux cas d'usages. Dans cet article, nous avons choisi de détailler la mise en place des interfaces de

IOT, INTELLIGENCE ARTIFICIELLE ET SYSTÈMES DE CONTRÔLE INDUSTRIEL

Dans le cas présenté, le besoin consiste à transmettre des données en continu afin de générer des informations à partir d'un automate Omron fonctionnant dans une usine. Il s'agit d'améliorer la disponibilité de la machine et de fournir une visibilité des opérations en lisant les données de température et de pression moteur d'une machine de moulage par injection.

Le directeur de l'usine doit pouvoir surveiller ces données de manière centralisée pour une meilleure visibilité et un temps de réaction réduit. Notez qu'un changement mineur des paramètres de température et de pression génère un profil thermique qui peut causer des défauts sur les pièces.

Bien que de nombreux protocoles tels que MQTT, RTSP, etc. soient pris en charge de manière native dans la pile technologique de KPS, il a été choisi de tirer parti de Litmus Automation afin d'ingérer les données des automates. Le but était d'ingérer les données de l'automate Omron, de les filtrer et de les transformer, puis de les pousser vers un cluster S3 fonctionnant sur AWS. La question qui se pose est de savoir comment transmettre les données de Litmus Edge vers Karbon Platform Services. La solution repose sur l'utilisation du service Litmus fourni par la plate-

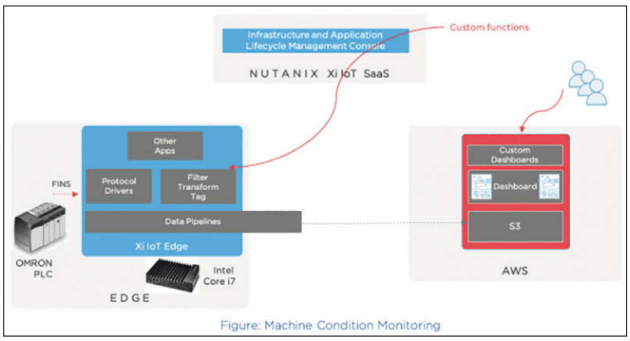


Figure: Machine Condition Monitoring

forme KPS qui permet d'exposer une source Litmus comme une interface de données consommable par KPS. Développé à l'origine pour répondre au besoin d'un client CaaS Litmus est disponible pour tous les cas utilisateurs de type PLC.

1

```
kind: dataSource
name: ex_nats
svcDomain:
protocol: DATAINTERFACE
type: Sensor
authType: CERTIFICATE
ifcInfo:
  class: DATAINTERFACE
  img: registry.hub.docker.com/xiiot/nats:v1.0
  kind: IN
  protocol: nats
  ports:
    - name: nats
      port: 4222
edge: myprovideredge
fields:
- name: topic
  topic: "mock-topic"
- name: host
  topic: "natshost-10.0.0.1"
- name: port
  topic: "natsport-4222"
```

données pour la configuration et le déploiement des sources de données afin de connecter capteurs IoT et automates à la plateforme.

Dans le cadre d'un déploiement Edge Computing, Nutanix Karbon Platform Services (KPS) libère les développeurs de la complexité du déploiement de périphériques et de flux de données et leur permet de se concentrer sur la création d'une logique utile d'analyse et de transformation des données grâce à des applications Kubernetes faciles à utiliser, des pipelines de données réutilisables, des fonctions « serverless » et une architecture de machine learning enfichable.

Modèle d'interface de données des services de la plateforme Karbon

KPS est une plateforme PaaS multicloud, qui permet de créer facilement des solutions pour collecter des données à partir de capteurs et périphériques distants et le traitement en temps réel de ces données. Une solution IoT se compose typiquement de passerelles ou de serveurs Edge, connectée à des capteurs, qui génère des données. Les flux de données (transformés préalablement des passerelles de traduction de protocoles si nécessaire) sont configurés comme des sources de données, qui peuvent être connectées en entrée/sortie à des applications ou à des pipelines de données pour l'analyse et la transformation. C'est sur cette composante de KPS que nous allons nous concentrer aujourd'hui.

Notions de base sur les interfaces de données

L'environnement KPS déployé sur le Edge permet d'exécuter des sources de données personnalisées appelées interfaces de données. Sur KPS, les interfaces de données sont simplement des conteneurs Docker fonctionnant dans un environnement Kubernetes hébergé. Les interfaces de données déployées dans le cadre d'une solution KPS sont principalement utilisées pour combler le fossé entre les dispositifs IoT, les services fonctionnant sur la plateforme et les services externes sur site ou dans le cloud comme AWS Kinesis. Pour les développeurs déjà familiers avec l'utilisation de Docker et de Kubernetes, le déploiement d'une interface de données sur le Edge avec Karbon Platform Services est tout aussi simple qu'un déploiement dans le datacenter.

Pour créer et gérer des interfaces de données dans les services de la plateforme, il est possible d'utiliser l'interface de ligne de commande ou des API REST.

L'interface de ligne de commande kps permet la création, la suppression et la modification des ressources de Karbon Platform Services, y compris les sources de données. Il est possible de créer des interfaces de données via l'interface de ligne de commande avec un fichier yaml comme celui-ci : 1

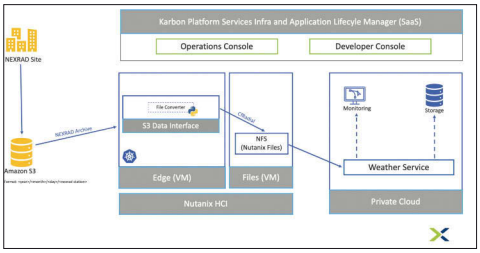
Pour plus de détails, Nutanix propose un ensemble de recettes sur Github : https://github.com/nutanix/karbon-platform-services/tree/master/how_to/data_sources

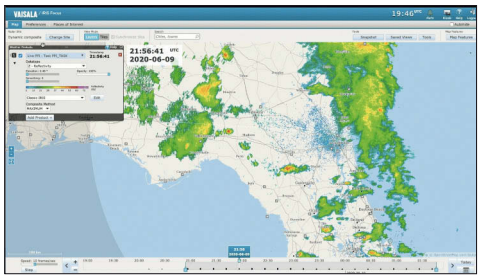
GESTION DES CONDITIONS CLIMATIQUES ET S3

Une branche de l'armée US a engagé Nutanix pour fournir une plateforme permettant d'injecter des données météorologiques provenant des buckets S3 NEXRAD dans un service d'analyse météorologique fonctionnant sur leur cloud privé.

Le bucket S3 source contient un ensemble de données en temps réel et d'archives provenant de NEXRAD, un réseau de 160 sites de radars Doppler à haute résolution qui détecte les précipitations et les mouvements atmosphériques et diffuse les données à intervalles d'environ 5 minutes à partir de chaque site. Les données peuvent être utilisées par les services de visualisation météorologique à des fins d'analyse et de prévision. L'architecture visée était la suivante :

Le but était d'ingérer et de transformer les fichiers téléchargés du bucket S3, puis de les livrer vers un serveur de partage NFS (Nutanix Files). Comme les données sont mises à jour en temps réel, il était impossible de télécharger directement les fichiers vers le partage NFS. L'interface de données mise en œuvre permet de traiter le contenu du bucket par lot et de tirer parti du traitement en temps réel pour pousser les données dans le logiciel d'analyse météorologique.



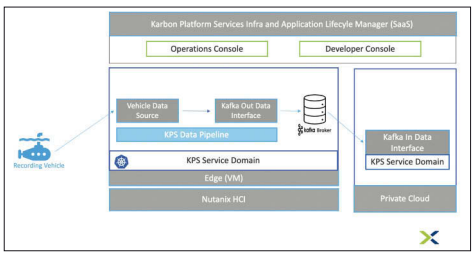


Le système a été mis en place en quelques semaines et le client a pu utiliser l'interface de données pour ingérer les données NEXRAD de quatre sites différents. Cela lui a permis de surveiller les données radar en temps réel pour prendre des décisions précises.

INTERCONNEXION DE DOMAINES DE SERVICES AVEC KAFKA

Un client existant du secteur du pétrole et de l'énergie a engagé Nutanix avec un cas d'utilisation pour fournir une interconnexion entre les domaines de service KPS existants. L'objectif était de transférer les données enregistrées sur le Edge vers leur cloud privé. Les domaines de service fonctionnaient sur des réseaux différents, de sorte que des entités telles que les Data Pipelines ne fonctionnaient pas bien. La solution choisie a alors été d'utiliser le streaming d'événements pour faire passer les données du Edge vers le cloud. Kafka a été choisi en raison de sa capacité à fournir un haut débit avec une faible latence pour traiter les flux de données en temps réel. L'architecture visée était la suivante.

L'architecture de Kafka reposant sur un principe producteur/consommateur, deux interfaces de données ont été mises en place. La première fait tourner un producteur Kafka pour publier des données à travers le broker Kafka de KPS fonctionnant sur le domaine de service. Cette interface de données fonctionne sur le Edge et ingère des données provenant du véhicule d'enregistrement du client. Une autre interface de données fonctionne dans le datacenter du client avec un consommateur Kafka pour ingérer les données publiées par le broker. Avec les deux interfaces de données en fonctionnement, le client est capable de transmettre des données/messages à travers plusieurs domaines de service et dispose d'un système véritablement interconnecté.



Roadmap des langages

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

Attention : à l'heure où nous bouclons ce numéro, certaines roadmaps n'étaient pas fixées.

Déjà disponible

PHP 8.0

La version finale a été annoncée fin novembre ! Cette version apporte de nombreuses évolutions, 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.

Avant toute migration ou tout déploiement de la v8, testez la plateforme sur un petit projet. Attendez les premiers retours. On peut s'attendre à des fix bugs rapidement.

Kotlin 1.4.20

Kotlin 1.4.20 améliore les concaténations de chaînes. Depuis Java 9, la concaténation de chaînes sur la JVM est plus efficace, notamment grâce à l'appel d'une méthode dynamique qui se fait via l'instruction de bytecode invokedynamic. L'analyse d'échappement du compilateur de Kotlin 1.4.20 est améliorée. Il s'agit d'une technique utilisée par le compilateur pour décider si un objet peut être alloué sur la pile ou s'il doit « s'échapper » vers le tas. L'allocation sur la pile est beaucoup plus rapide et ne nécessite pas de travail ultérieur du ramasse-miettes. Notons encore une nouvelle fonctionnalité, expérimentale, qui permet d'ignorer les erreurs de compilation. Cette fonctionnalité vous permet d'essayer votre application même installable. Par exemple, quand vous effectuez une refactorisation complexe. Le compilateur ignore tout code erroné et le remplace par des exceptions d'exécution au lieu de refuser de compiler.

React 17

Les équipes React ont annoncé dès cet été que cette version n'apportait pas de nouvelles fonctionnalités. Le focus est mis sur la stabilité et le nettoyage du code. Cette version n'est qu'une étape intermédiaire vers les futures évolutions notamment sur les mécanismes de mise à jour des modules internes.
Pour en savoir plus :
https://reactjs.org/blog/2020/10/20/react-v17.html

Rust 1.48

Cette version propose des évolutions de rustdoc, l'outil de génération de documentation. Jusqu'à présent il n'était pas possible, en documentant un type, d'établir un lien avec d'autres types pouvant être utilisés de concert avec lui. Dans cette version, vous pouvez utiliser une syntaxe pour informer rustdoc que vous voulez créer un lien vers un type, et il générera les URL pour vous. Cela permet aux utilisateurs de la documentation de naviguer facilement d'un type à l'autre.

Python 3.9.x

La première mise à jour (bugfix) est sortie début décembre. D'autres suivront. La 3.10 est annoncée.

Deno 1.5

Deno continue de monter en puissance. La version 1.5 a pour but d'améliorer les performances par un facteur 3 sur les bundles. Cela signifie aussi des améliorations significatives du compilateur. La taille des bundles sera réduite. Un gros travail a été fait sur le compilateur swc utilisé par la plateforme. Plusieurs API arriveront sur la plateforme dont confirm et prompt. La partie REPL a elle aussi été améliorée.

mars

Java 16

Le prochain OpenJDK est prévu pour mars. Les équipes travaillent sur plusieurs évolutions et nouveautés :

- Migration vers Git
- Disponibilité des fonctions de C++14 dans OpenJDK
- Disponibilité sur Alpine Linux & Windows/AArch64, sans oublier Apple Silicon

Cette version, et la v17, doit aussi apporter plusieurs projets, dont Panama et Amber. Panama doit pouvoir assurer l'interconnexion entre la JVM et le code natif. Il doit faire des appels natifs depuis la JVM, accès natif aux données, optimisation du JIT (côté natif), librairies dédiées.

octobre

Python 3.10

Python change sa cadence de mise à jour. La 3.10 devrait sortir en octobre prochain. Parmi les nouveautés prévues : opérateur Union, nouveauté sur le TypeAlias, amélioration sur plusieurs modules (base64, codecs, py_compile). Il n'est pas prévu d'ajouter de nouveaux modules.
Les listes des retraits et dépréciations sont assez longues : checker la liste. Plusieurs changements sont indiqués sur les API. Là encore, soyez vigilant(e) sur la migration. Mais, il y a largement le temps pour le portage / la migration des codes actuels.

Date inconnue

PHP 8.1

Les développeurs travaillent à la 8.1 depuis plusieurs semaines. Cette version doit apporter des évolutions par rapport à la 8.0 et des corrections. On s'attend au support du MurmurHash (en v3). Pour le moment, la liste des modifications / ajouts n'est pas fixée.

Swift 5.4

Le langage Swift continue d'évoluer avec l'annonce de la version 5.4. Les nouveautés ne sont pas claires. Plusieurs dizaines de propositions ont été acceptées par la gouvernance du langage. La 5.4 sera dans la lignée de la précédente version et de la 5.x : stabilité des API, performances. L'arrivée officielle de la version Windows est un réel avantage pour le langage, surtout depuis le retrait d'IBM.

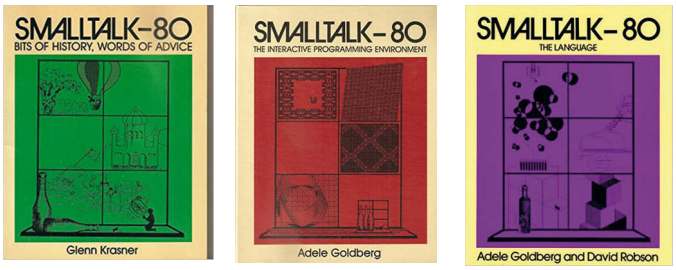
Angular vNext

Les prochaines versions apporteront beaucoup d'évolutions :

- 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)

Smalltalk : le retour d'un langage mythique

PARTIE 2



Dans le 243, nous avions parlé des origines de Smalltalk et brièvement de Pharo, l'implémentation moderne de SmallTalk. Dans cette suite, nous allons nous concentrer sur Pharo. Vous verrez qu'il s'agit d'un très puissant langage.

La rédaction

Dossier relu et coordonné par **Laurent Juillard** *et* **Stéphane Ducasse.**

PharoJS : Contrôlez JavaScript avec Pharo

Noury BOURAQADI

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. Il occupe les fonctions d'enseignant-chercheur en informatique à l'IMT Lille Douai.

PharoJS [1] est un ensemble d'outils et de libraires libres (licence MIT) qui permettent de bénéficier de l'environnement de Pharo pour la conception d'applications JavaScript. Le développement est effectué entièrement en Pharo. Le passage en production consiste à convertir ce code pour obtenir in fine une application JavaScript.

Exemple d'application PharoJS

Afin de présenter PharoJS, considérons un exemple qui simule un jeu de dés **(Figure 1)**. L'interface comporte des éléments qui permettent de modifier le nombre de dés, de lancer les dés, d'afficher le nombre correspondant à chaque dé, ainsi que le score total.

L'interface est décrite sous la forme d'un fichier HTML dont un extrait est donné ci-dessous. Il comporte les balises correspondant à la structure de l'interface graphique et une référence vers un fichier JavaScript *index.js*. Ce dernier est totalement généré à partir de code Pharo. En effet, lors du passage en production, la totalité de la logique de l'application est convertie en JavaScript par PharoJS.

```
<!DOCTYPE html>
...
<body>
...
 <span id="diceView"></span>
...
 <button id="rollDiceButton">Roll Dice</button>
...
 <script type="text/javascript" src="js/index.js"></script>
</body>
</html>
```

Intégration de JavaScript dans l'univers Pharo

La totalité du code écrit par les développeurs d'applications PharoJS est du code Pharo. Il est organisé autour d'une classe principale qui crée et connecte les objets applicatifs. Ces objets peuvent être issus aussi bien du monde Pharo que du monde JavaScript. Quand il s'agit de réutiliser des librairies existantes, celles-ci peuvent être aussi-bien écrites en Pharo qu'en JavaScript. Mais, pour ce qui est du nouveau code, il est naturellement écrit en Pharo, du fait de la souplesse et du confort de l'environnement. Reprenons notre exemple du jeu de dés présenté plus haut. Nous réalisons cette application suivant le modèle de conception MVC (*Model, View, Controller*). Le navigateur web génère les objets DOM qui correspondent aux "vues" à partir des balises HTML. Ce sont les seuls objets nativement JavaScript de cette application. En revanche, les contrôleurs et les modèles sont codés en Pharo. Par exemple, nous allons définir une nouvelle classe *Dice* pour les dés, et réutiliser la classe *OrderedCollection* de Pharo.

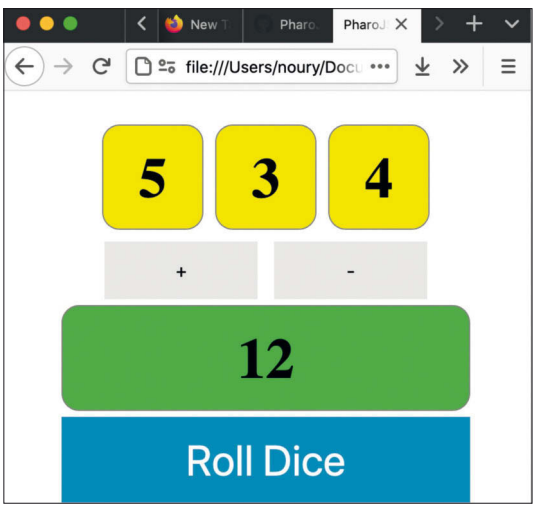


Figure 1. *Exemple d'une application simple réalisée avec PharoJS*

La **figure 2** donne un extrait de code Pharo du contrôleur de notre application de jeu de dés. Le modèle est une instance de la classe *Dice* qui dispose d'un champ *faceUp* qui correspond à la face supérieure du dé. La vue d'un dé est un objet DOM que nous retrouvons par l'identifiant unique associé à sa balise HTML (attribut *id*). Nous appelons la méthode *getElementById()* sur la variable globale *document* de JavaScript. En effet, PharoJS permet d'accéder à n'importe quelle variable globale de JavaScript. Il permet également d'appeler des méthodes JavaScript (en rouge dans la Figure 2) dans du code Pharo.
Ainsi, nous pouvons réaliser le schéma de conception observateur (*Observer Pattern*) de notre application dans Pharo. Un changement de la face supérieure du dé, va conduire à réviser le contenu textuel de la vue *diceView*. De même, un événement HTML clic sur le bouton *rollButton* va faire virtuellement rouler le modèle de dé, grâce à l'envoi du message *roll*.

Cohabitation d'objets Pharo et JavaScript pendant les tests

Durant les tests, l'application est répartie entre la machine virtuelle Pharo et l'interprète JavaScript. Dans notre exemple, les objets, contrôleurs et modèles sont gérés par la machine virtuelle Pharo. Les vues (par exemple les boutons), résident quant à elles dans la mémoire de l'interprète JavaScript (Voir la **figure 3**).
Les tests sont écrits en Pharo et font abstraction de la localisation des objets, comme illustré dans l'exemple du listing suivant. La variable *app* est pré-initialisée par le *framework* de test de PharoJS de manière à référencer l'objet "point d'entrée" de l'application. Cet objet est localisé dans la mémoire gérée par Pharo. Il en est de même pour la liste des dés *diceList*. En revanche, le bouton référencé par la variable *addButton* référence un objet du monde JavaScript. En effet, on y a accès via la variable globale *document* de JavaScript. **(figure 4)**
Le *middleware* PharoJS permet de lier les 2 mondes et d'acheminer les appels de méthodes ou les résultats. La communication se fait par l'intermédiaire de *web sockets*. Lors de l'exécution de tests, un serveur est lancé côté Pharo. Puis, le client du middleware -écrit en Pharo- est traduit vers JavaScript et exporté dans le fichier index.js référencé par le fichier HTML de l'application. Enfin, PharoJS fait un appel système pour ouvrir le fichier HTML dans un navigateur web. Le JavaScript généré est ainsi exécuté et la connexion est établie avec Pharo. Les objets des 2 mondes sont ainsi connectés et peuvent communiquer.
Comme montré par vidéo [2], un test d'une application PharoJS peut inclure des actions telles que des simulations de clics sur les éléments de l'interface HTML. Il peut également vérifier des assertions sur l'état des objets JavaScript. Le développeur dispose de toute la puissance de l'environnement Pharo. Il peut poser des points d'arrêt, déboguer du code, ou inspecter des objets, à volonté. PharoJS permet même d'inspecter des objets JavaScript. Un *Playground* dédié permet également de contrôler l'application et les objets JavaScript de manière interactive.

Figure 2. *Code Pharo illustrant les références vers le monde JavaScript (en rouge)*

```
| dice diceView rollButton |
diceView := document getElementById: #diceView.
dice := Dice new.
dice
  afterChangeOf: #faceUp
  do: [ :newFaceUp | diceView textContent: newFaceUp ].
rollButton := document getElementById: #rollDiceButton.
rollButton
  addEventListener: #click
  action: [ dice roll ].
```

Figure 3. *Pendant les tests, les objets Pharo cohabitent à avec les objets JavaScript*

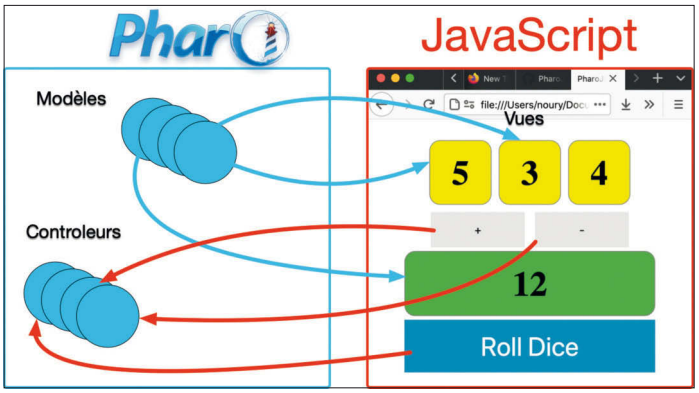


Figure 4

```
testAddDiceButton
  | initialNumberOfDice addButton |
  initialNumberOfDice := app diceList size.
  addButton := document getElementById: #addDiceButton.
  addButton click.
  self assert: app diceList size equals: initialNumberOfDice + 1.
```

Utilisation des API Pharo pour les objets JavaScript

Une fois l'application validée par les tests, tout le code Pharo est exporté vers JavaScript. Cette opération est réalisée pour les classes Pharo accessibles à partir de la classe principale de l'application. Au-delà de l'aspect syntaxique, l'export passe aussi par la conversion d'éléments du langage Pharo vers JavaScript. C'est le cas par exemple de l'objet indéfini *nil* ou du *retour non-local* dans les block.
De plus, PharoJS opère une fusion de certaines classes fondamentales de Pharo avec leurs homologues JavaScript. C'est le cas par exemple, des classes : *Array*, *String* ou *Number*. La fusion consiste à enrichir les classes JavaScript avec les champs et les méthodes provenant de Pharo. Cette opération permet d'utiliser convenablement des librairies JavaScript, avec l'API Pharo. Par exemple, dans Pharo, le premier indice d'un tableau est 1, alors que dans JavaScript c'est 0. L'export PharoJS permet de faire cohabiter les deux conventions.

Conclusion

Dans cet article, nous avons présenté PharoJS qui permet de développer en Pharo des applications exécutables par un interprète JavaScript : NodeJS ou un navigateur web. On code en Pharo, puis on génère du JavaScript pour obtenir des applications qui ciblent entre autres : les serveurs et les clients web, ainsi que les smartphones iOS et Android [3].
Il est possible de réutiliser et d'intégrer, de manière transparente, des librairies provenant aussi bien du monde JavaScript que de celui de Pharo.

Références

[1] Site officiel de PharoJS https://pharojs.org/
[2] Vidéo illustrant l'exécution de tests en PharoJS https://youtu.be/tGP4FTij5q8
[3] Exemple d'applications basées sur PharoJS https://pharojs.org/successStories.html

Richard Uttner : un développeur Smalltalk pour toujours !

Interview réalisée par Laurent Julliard

Est-ce que vous pouvez nous décrire votre activité et qui sont vos clients ?
RU : Dans les années 1980, les ordinateurs personnels ont commencé à gagner en popularité et ont fait décoller la demande pour de petites applications d'entreprise nécessitant de la persistance de données. Mais à l'époque développer ce genre d'applications restait difficile et j'ai donc commencé par me concentrer sur les frameworks et les outils permettant le développement de ce genre d'applications. Lorsque mes projets clients m'ont amené à Tübingen, j'ai rencontré plusieurs petites sociétés prêtes à joindre leurs efforts pour développer ce type d'outils. C'est alors qu'ensemble, en 1988, nous avons fondé Project Software Tool Systems.

Comment êtes-vous venu à Smalltalk et qu'est-ce qui vous a attiré ?
RU : En 1990, Basic et C étaient les langages les plus populaires pour développer des applications professionnelles sur ordinateur personnel, mais c'est aussi l'époque où Smalltalk a fait une percée commerciale. Notre équipe a aimé les concepts du langage et en particulier le ramasse-miettes automatique gérant l'allocation de la mémoire. Smalltalk proposait aussi un modèle bien défini permettant de s'interfacer avec la machine virtuelle et d'y attacher nos propres fonctions primitives (NDLR: des fonctions qui en général discutent avec le système d'exploitation sous-jacent). C'est ainsi que nous avons conçu un système proche conceptuellement de Smalltalk, mais dans une implémentation assez différente et incompatible avec les Smalltalk commerciaux de l'époque. Mais pour nos utilisateurs cibles et nos financeurs, c'était le but recherché à la fois pour des raisons techniques, mais aussi pour s'affranchir des coûts de licence élevés des principales distributions Smalltalk de l'époque (NDLR : les versions Open Source de Smalltalk comme Squeak ou Pharo sont arrivées plus tard).

Quel rôle Smalltalk joue-t-il dans vos produits et services ?
RU : Depuis 1992, je propose des services et du conseil pour le développement Smalltalk. De 1998 à 2019 uniquement sur des distributions commerciales et en 2019 j'ai commencé à travailler avec Pharo (NDLR l'Association Pharo et son consortium industriel veillent à ce que Pharo réponde bien aux besoins des entreprises).

Quelles sont les caractéristiques de Smalltalk qui le rendent utile pour vous et vos clients ?
RU : Pour des applications desktop, je pense que la raison qui a amené mes clients à faire le choix de Smalltalk est la même: lorsqu'ils ont démarré le développement de leurs produits dans les années 90, les environnements Smalltalk constituaient le choix idéal pour arriver rapidement à des résultats. Certaines de ces applications ont crû très rapidement, atteignant une ampleur et une sophistication qui les rendaient difficiles à porter en Java et en C#; un choix que mes clients auraient pourtant aimé faire lorsque Smalltalk a soudainement perdu son leadership sur le marché (NDLR : par suite de décisions stratégiques calamiteuses des éditeurs Smalltalk à l'époque - Voir l'article sur l'histoire de Smalltalk). Néanmoins Smalltalk a survécu au travers de plusieurs distributions commerciales même si elles ont dû revoir leurs ambitions d'innovation à la baisse pour franchir cette mauvaise passe.
Pour ce qui concerne les applications Web je n'ai pas d'expérience personnelle dans ce domaine avec Smalltalk, mais ce que je peux dire c'est que j'ai détesté ce que j'ai vu chez mes clients qui utilisaient Java Server Pages ou Enterprise Java Beans. De ce fait je comprends pourquoi Smalltalk reste populaire aujourd'hui dans le développement Web plus que dans le développement d'application desktop (NDLR : voir les articles sur Zinc et Seaside)

Comment voyez-vous le futur de Smalltalk ?
RU : Aujourd'hui je ne vois pas d'implémentation Smalltalk ayant atteint un niveau totalement satisfaisant. Certes, nombre des fantastiques concepts originellement créés pour Smalltalk (NDLR: en référence au Système Smalltalk-80 créé et publié par Xerox PARC au début des années 80 - Voir l'article sur l'histoire de Smalltalk) ont déjà été implémentés. Mais dans la routine quotidienne, je dois encore aujourd'hui faire plus de compromis que je ne le souhaiterais au niveau professionnel. Quand j'ai découvert l'initiative Pharo, cette implémentation de Smalltalk possédait encore plusieurs points faibles en particulier pour le développement d'application desktop. Malgré ça, j'ai décidé de rejoindre l'Association Pharo en tant que membre industriel et j'ai convaincu mes clients d'en faire autant. À mes yeux, la communauté grandissante de Pharo avec sa détermination à améliorer ce qui doit l'être a déjà enrichi Pharo. À ce rythme, dans moins de 2 ans, je m'attends à ce que Pharo atteigne un haut niveau de professionnalisme tout en restant fidèle aux remarquables design et concepts d'origine de Smalltalk. Je suis sûr qu'alors Pharo attirera l'attention d'une communauté encore plus large qu'aujourd'hui.

Pour en savoir plus : https://projector.de/

Richard Uttner

Je suis né en Autriche en 1957. En 1976 j'ai démarré des études de psychologie. Comme j'étais déjà bien accro à l'électronique et à la programmation sur les calculatrices de poche (NDLR: la grande époque des premières calculatrices programmable HP et TI), je me suis arrangé pour suivre aussi des cours et travaux pratiques sur les microprocesseurs ce qui m'a permis de développer des logiciels en assembleur sur les microprocesseurs disponibles dans les années 1970 (NDLR: essentiellement des 8 bits comme le Z80, Motorola 6800...). En 1980 j'ai décidé d'abandonner mes études et de démarrer une carrière de free-lance comme développeur en Allemagne.

projector software

Alexandre Bergel

Professeur associé à l'Université du Chili. Il dirige des lignes de recherches en génie logiciel et intelligence artificielle. Alexandre est auteur de Agile Artificial Intelligence in Pharo: Implementing Neural Networks, Genetic Algorithms, and Neuroevolution (Apress, 2020).

Visualisation agile avec Roassal

La capacité à visualiser des données est cruciale dans de nombreux domaines allant de l'intelligence artificielle, à la simulation et à l'analyse de données. Retranscrire des données sous forme visuelle n'a jamais été aussi simple, mais certains aspects restent, aujourd'hui encore, mal résolus. Comment (1) permettre à l'utilisateur de définir ses propres visualisations, avec des interactions particulières et spécifiques (2), construire incrémentalement des visualisations et avoir un retour immédiat (3), pouvoir intégrer des visualisations dans un environnement de production existant (4), être capable de connecter plusieurs visualisations dans un environnement donné ? La visualisation agile, telle que promue par le moteur de visualisation Roassal pour Pharo, propose une solution à tous ces problèmes. C'est l'objet de cet article.

Roassal est à la fois une bibliothèque de constructions de visualisations, un DSL (domain specific language) et un outillage pour prototyper rapidement vos visualisations.
Roassal est distribué sous licence MIT et disponible pour l'environnement Pharo depuis le site GitHub de Roassal [1].

Cet article suppose que vous êtes relativement à l'aise avec Pharo. Nous créerons de manière incrémentale des visualisations du code d'un composant logiciel. Du point de vue des données, un code source logiciel est une donnée complexe, et les techniques d'analyse de données standard ne peuvent pas être appliqués sur le code d'un logiciel.

Ce type de visalisation a de nombreuses applications : allant de l'évaluation de la qualité du code à la réingénierie de l'architecture. Cette pratique n'est pas une tâche triviale. Cet article utilise un composant central du runtime Pharo comme illustration de la manière de procéder. Des analyses de code ou runtime d'applications écrites en Java, Android, JavaScript ou Powerbuilder et Postgresql suivent le même procédé en utilisant un méta modèle dédié comme ceux proposés dans Moose [3].

Visualisation de classes par l'exemple

Considérons le code qui suit, exécutable dans le Playground de Pharo (ndlr: voir l'article sur Pharo):
"Stockons dans la variable classes les classes que nous voulons visualiser"

```
classes := Collection withAllSubclasses.
```

"Un canevas Roassal est une surface pour accueillir des formes graphiques"

```
c := RSCanvas new.
```

"Chaque classe est représentée par une boîte"

```
classes do: [ :aClass | c add: (RSBox new model: aClass) ].
```

"La largeur de la boîte indique le nombre de variables d'instances de la classe."

```
RSNormalizer width shapes: c shapes; from: 6; to: 20;
normalize: #numberOfVariables.
```

"La hauteur représente le nombre de méthodes de la classe"

```
RSNormalizer height shapes: c shapes; normalize: #number
OfMethods.
```

"La couleur d'une classe va de gris à rouge, indiquant le nombre de lignes de code"

```
RSNormalizer color shapes: c shapes;
    from: Color gray; to: Color red; normalize: #number
OfLinesOfCode.
```

"Les lignes verticales représentent la relation d'héritage."

```
RSEdgeBuilder orthoVertical
    canvas: c; withVerticalAttachPoint; color: Color lightGray;
    connectFrom: #superclass.
```

"Pour placer les classes, on utilise un arbre"

```
RSTreeLayout on: c nodes.
```

"Nous rendons les classes manipulables et ajoutons une aide de survol"

```
c nodes @ RSDraggable @ RSPopup.
```

"Enfin grâce au contrôleur RSCanvasController, la visualisation est tout entière zoomable, manipulable et les formes peuvent être recherchées"

```
c @ RSCanvasController.
```

Après avoir copié-collé ce code dans le Playground de Pharo, vous pouvez l'exécuter et l'inspecter en utilisant les touches Cmd-i sur Mac or Ctrl-i sur Windows/Linux. Le résultat devrait ressembler à ça : **figure 1**

Cette visualisation représente la hiérarchie des classes du composant Collection de Pharo. Elle utilise une métaphore simple dans laquelle chaque boîte est une classe. Les arêtes indiquent l'héritage, et en particulier, une superclasse est

située au-dessus de ses sous-classes. La hauteur d'une boîte indique le nombre de méthodes définies dans la classe représentée, tandis que la largeur représente le nombre de variables. La couleur de chaque case va du gris au rouge, indiquant le nombre de lignes de code de la classe représentée. Une boîte longue et rouge indique une classe avec de nombreuses méthodes et est définie par de nombreuses lignes de code.

À l'opposé, une petite boîte grise indique une classe qui n'a pas beaucoup de méthodes, pas beaucoup de variables et seulement quelques lignes de code. Cette visualisation donne un aperçu de la distribution du code dans la hiérarchie des classes. Il permet également de repérer des entités exceptionnelles, c'est-à-dire des classes significativement différentes des autres classes.

Détaillons le script : la variable «classes» fait référence à un ensemble de classes Pharo. Chaque classe est représentée comme un objet `RSBox`, ajouté dans le canevas `c`. Une caractéristique importante de Roassal permet de prendre en charge la connexion entre une forme graphique et le modèle objet représenté par la forme. Cette connexion est exprimée à l'aide du message `model:`, comme dans l'expression `RSBox new model: aClass`. Pour définir la forme et la couleur de chaque boîte, nous utilisons des normalisateurs, qui utilisent l'objet modèle. Une classe dans Pharo répond à de nombreux messages. Par exemple, l'expression «String numberOfMethods» donne 382, qui est le nombre de méthodes définies dans la classe «String». De même, on peut envoyer le message `numberOfVariables`,`numberOfLinesOfCode`, `superclass` à une classe pour obtenir, respectivement, le nombre de variables d'instance, le nombre de lignes de code et la superclasse. Des lignes sont construites entre les classes pour indiquer l'héritage à l'aide d'un objet dédié, le `RSEdgeBuilder`. Les classes, positionnées dans le canevas en arborescence sont déplaçables et le nom de la classe apparaît sous forme de fenêtre contextuelle lorsque le curseur de la souris survole une classe.

Cet exemple intégré dans Pharo

L'exemple de code fourni semble relativement standard et la même visualisation pourrait être créée à l'aide d'un moteur de visualisation moderne. Un lecteur familier avec une bibliothèque telle que D3.js ou Matplotlib pourrait dire: «Eh bien, je peux en faire autant». C'est vrai jusqu'à un certain point. Le véritable avantage de Roassal ne se limite pas à son API de création de visualisation. Roassal propose des fonctionnalités pour rendre la visualisation navigable et faciliter l'intégration dans l'environnement Pharo.

Une forme graphique dans Roassal est liée à un modèle d'objet. L'expression simple «RSBox new model: aClass» fait d'une boîte une façade d'une classe, et cette façon de représenter les données est une position très différente de celle des autres moteurs. En ayant la connexion entre une forme graphique et le modèle objet explicite, on peut cliquer sur une forme pour inspecter le modèle objet. Dans ce cas, on peut cliquer sur cette case pour ouvrir un inspecteur sur la classe cliquée. L'inspecteur donne pour un objet inspecté un certain nombre de représentations visuelles intégrées.

Définissons, par exemple, une nouvelle représentation visuelle du graphe d'appel des méthodes définies dans une classe. Considérez la méthode suivante définie sur la classe `Class`:

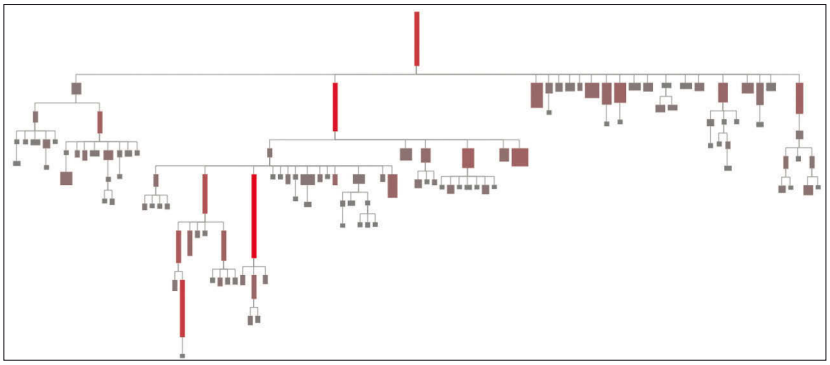
```
Class >> visualizeCallGraph
    "Visualize the call graph of the class' methods"
    | c eb shapes |
    shapes := self methods
collect: [ :cm | RSBox new
color: #blue;
size: cm numberOfLinesOfCode;
model: cm ]
as: RSGroup.
    shapes @ RSPopup @ RSDraggable.
    c := RSCanvas new.
    c addAll: shapes.
    eb := RSEdgeBuilder arrowedLineWithOffset: 0.2.
    eb moveBehind.
    eb shapes: c nodes.
    eb canvas: c.
    eb withVerticalAttachPoint.
    eb connectToAll: #dependentMethods.
    RSTreeLayout on: c nodes.
    shapes @ RSHighlightable withEdges.
    ^ c @ RSCanvasController
```

Cette méthode peut être invoquée sur n'importe quelle classe. Par exemple, l'inspection de l'expression `String visualizeCallGraph` visualise les graphes d'appel des méthodes de `String`. La méthode `visualizeCallGraph` crée d'abord une boîte pour chaque méthode. Des arêtes sont ensuite ajoutées pour indiquer les appels entre les méthodes. Nous devons maintenant informer le framework de l'inspecteur d'appeler `visualizeCallGraph` lors de l'inspection d'une classe. L'inspecteur peut facilement être étendu avec la méthode suivante:

```
Class>>gtInspectorViewCallGraphIn: composite
    <gtInspectorPresentationOrder: -10>
    composite roassal3
        title: 'CallGraph';
        initializeCanvas: [ self visualizeCallGraph ]
```

La méthode `gtInspectorViewCallGraphIn:` crée une visualisation montrant le graphe d'appel lorsqu'une classe est ins-

Figure 1

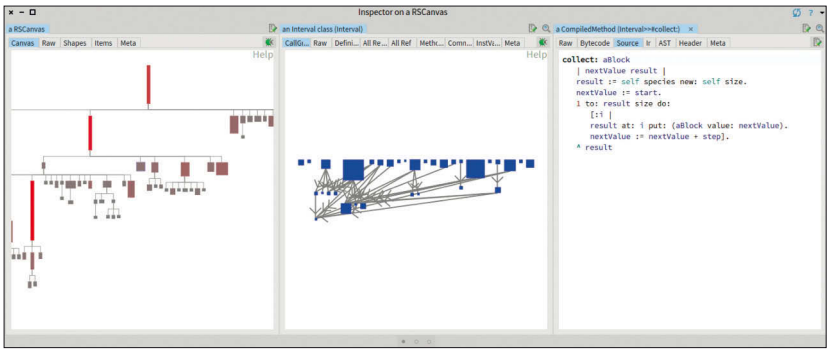


pectée. Les deux visualisations sont donc connectées en cliquant sur les classes et en affichant le graphe d'appel. De même, une méthode peut être sélectionnée pour afficher son bytecode et son code source. Prenons l'exemple suivant : **figure 2**

Le volet le plus à gauche représente la visualisation des classes, donnée dans le script initial. Dans cette première visualisation, la classe «Interval» est sélectionnée. Au centre, la sélection affiche le graphe d'appel de méthodes. Dans le graphe d'appel, la méthode «collect:» est sélectionnée, ce qui montre le code source de la méthode dans le volet droit.

Pour résumer, nous avons construit indépendamment deux visualisations. Le framework inspector les expose de manière très pratique tout en permettant la navigation parmi les objets visualisés: on peut passer du premier au second, bien que leurs scripts soient complètement indépendants.

Figure 2



Conclusion

Imaginez être doté d'un outil permettant de prototyper facilement des visualisations et de les intégrer dans votre environnement de travail. Si le coût de création des visualisations et de leur intégration est suffisamment bas, vous pourriez écrire plus de visualisations et plus fréquemment. Dès que les visualisations sont faciles à écrire et à utiliser dans les tâches quotidiennes, des visualisations naissent tels des champignons dans votre environnement de production. C'est précisément l'objectif de Roassal de vous donner cette facilité.

Nous définissons la notion d'agilité dans la visualisation de données comme le fait 1) d'abaisser le coût de production d'une visualisation, et 2) réduire le coût d'intégration de la visualisation dans un environnement de production. L'agilité transforme une belle image en un outil permettant aux praticiens d'agir sur un domaine précis et de bénéficier d'un retour immédiat.

Pour aller plus loin avec Roassal rendez-vous sur AgileVisualization [4] et dans le channel Roassal sur le serveur discord de Pharo [5].

Ressources :

[1] Roassal: https://github.com/ObjectProfile/Roassal3
[2] Pharo: https://pharo.org
[3] Moose: http://moosetechnology.org
[4] AgileVisualization: http://AgileVisualization.com
[5] Communauté Roassal sur Discord: https://discord.gg/QewZMZa

Applications Web avec Seaside

Johan Brichau
PhD
Cofondateur et CTO de Yesplan, une application de gestion de salles en ligne. Il a passé plus de 20 ans dans l'ingénierie logicielle et Smalltalk. Il est actuellement l'un des principaux contributeurs de Seaside.

Seaside est le principal framework open source de développement d'applications Web pour Smalltalk. Il utilise des composants et le principe de continuation avec un rendu des pages côté serveur.

Le développeur construit son application comme une composition de composants réutilisables et conçoit des flux de contrôle comme pour une application desktop. Seaside cohabite très bien avec les librairies et frameworks JavaScript pour la mise en œuvre d'interface graphique dynamique exécutée directement dans le navigateur (ndlr: voir aussi l'article sur PharoJS).

Seaside aujourd'hui

De nombreuses applications Web commerciales s'appuient sur Seaside. Citons le CMS www.cmsbox.ch, la fintech www.finworks.biz, le gestionnaire de lieux culturels www.yesplan.be, les services financiers en ligne www.mercapabbaco.com, la gestion de notes de frais www.spesenfuchs.de, le gestionnaire d'entrepôt www.orderflow.be, la comptabilité en ligne www.kontolino.de, l'ERP www.koretrax.com, etc.

La première version de Seaside remonte à 2002. La version actuelle (3.4) s'appuie sur un profond remaniement qui a eu lieu en 2010. Seaside est supporté par plusieurs versions de Smalltalk : Pharo [2], GemStone/S [3] et Squeak, ainsi que des portages indépendants sur Visualworks et Instantiations.

Un "Hello World" Seaside: le compteur Web

Illustrons la construction d'une application Web avec un petit "Hello World" : un simple compteur. C'est un exemple que vous trouverez dans tous les supports d'introduction à Seaside et cet article ne fait pas exception. La page Web que nous vous proposons affiche un compteur et l'utilisateur pour cliquer sur '++' pour incrémenter le compteur et sur '--', oh surprise, pour le décrémenter. Vous suivez toujours ? :-) Si vous téléchargez et installez Seaside [1], vous trouverez cet exemple dans l'application de bienvenue de Seaside et vous pourrez l'essayer. **Figure 1.**

Les composants

Toutes les applications Seaside sont implémentées sous forme d'un assemblage de composants en interaction. C'est cette structuration à base de composants qui, au fil du temps, s'est avérée être la fonctionnalité la plus caractéristique et la plus puissante pour le développement d'applications Web en Seaside.

Un composant encapsule un état. Il peut effectuer son propre rendu en HTML et sait répondre aux actions de l'utilisateur. Un composant Seaside est toujours une sous-classe de WAComponent et se doit d'implémenter certaines méthodes clés qui seront appelées par Seaside pour gérer son état et générer son rendu en HTML.

L'exemple "Compteur" est une application à un seul composant représenté par la classe WACounter qui possède une seule variable d'instance 'count'. Lorsqu'un utilisateur accède à l'URL de notre application, Seaside crée une instance de WACounter et appelle la méthode `renderContentOn:`, qui produit son rendu en HTML et le renvoie vers le navigateur.

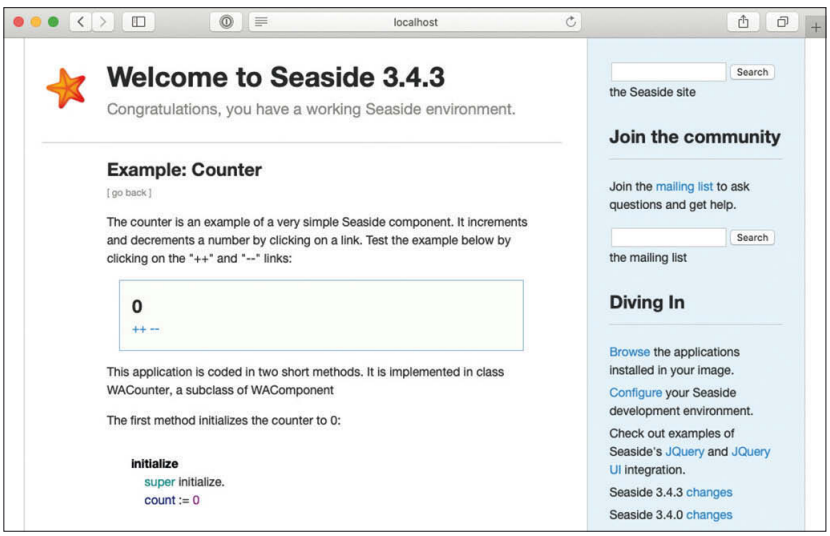
Le rendu HTML

Nous n'utilisons ni HTML ni langage de markup pour définir le rendu d'un composant. Le rendu est en fait généré en envoyant des messages à un objet "canvas". L'objet en question est créé par Seaside et passé en argument à la méthode `renderContentOn:'. Pour notre application Compteur, cette méthode se présente ainsi:

```
renderContentOn: html
 html heading: count.
 html anchor
  callback: [ count := count+1 ];
  with: '++'.
 html space.
 html anchor
  callback: [ count := count-1 ];
  with: '--'
```

L'instruction `html heading: count` effectue le rendu de la variable d'instance `count` sur la page Web dans le heading de la page HTML en envoyant simplement le message `heading: count` à l'objet canvas passé en argument (ici c'est 'html'). Dans les instructions qui suivent, 2 ancres html ('++' et `--') sont rendues sur la page séparées par un espace insécable. Nous expliquerons le rôle du callback dans la prochaine section.

Figure 1.
L'application de bienvenue de Seaside et son compteur.



Ce ne sont là que quelques exemples de messages destinés au rendu d'une page HTML. L'objet canvas de Seaside propose des messages pour toutes les balises html et leurs attributs. De cette façon, Seaside garantit le rendu d'un HTML conforme à la norme et évite au développeur de travaillier directement en HTML. Seaside gère aussi l'échappement de toutes les entités qui en ont besoin, une fonctionnalité importante pour se préserver des attaques de type XSS.

L'autre avantage de Smalltalk pour générer le rendu tient au fait qu'on peut utiliser toute la panoplie du langage pour effectuer des rendus aussi sophistiqués qu'on le souhaite. On peut ainsi tronçonner le rendu d'un segment HTML complexe en plusieurs méthodes ; des sections répétées peuvent être générées via des itérations ; les paramètres des méthodes et autres variables peuvent être utilisés pour paramétrer le rendu et, enfin, les méthodes peuvent être surchargées pour construire des hiérarchies entières de composants exprimant des variantes d'un rendu.

Les actions de type callbacks

Le code de notre composant doit aussi indiquer comment répondre à des actions de l'utilisateur. En Seaside, les actions sont mises en œuvre via des callbacks. Ce sont des blocs Smalltalk qui sont attachés à des éléments HTML durant l'exécution du rendu pour une exécution ultérieure lorsque l'utilisateur déclenchera une action particulière. Dans notre exemple, on trouve deux callbacks attachés aux ancres '++' et '--'. Quand un utilisateur clique sur l'une des ancres dans la page Web de l'application, Seaside va évaluer (exécuter) le bloc de code attaché à cette ancre. Ainsi, un clic sur '++' va exécuter le callback qui modifie l'état de notre composant en ajoutant 1 à la variable d'instance count.

Lorsque l'exécution du callback s'achève, Seaside effectue à nouveau l'opération de rendu en intégralité et renvoie le résultat vers le navigateur montrant ainsi le nouvel état du compteur augmenté de 1.

Composition de composants

Une véritable application Seaside est presque toujours faite de multiples composants. Les composants permettent en effet de penser l'application en morceaux indépendants et réutilisables. Il est facile d'écrire un composant puis de le réutiliser plus tard y compris dans une autre application. Par exemple, la page d'accueil de Seaside (voir copie d'écran plus haut) montre le composant compteur intégré dans la page. Il existe deux façons de réutiliser un composant: on peut embarquer (embedding) le composant ou on peut l'appeler (calling).

Embarquer un composant

Un nouveau composant peut être créé en assemblant d'autres composants embarqués. Pour illustrer ce point, nous allons réutiliser le composant 'WACounter' pour créer une application Web qui affiche plusieurs compteurs. Pour cela, nous allons créer le nouveau composant `WAMultiCounter` qui est une composition de 5 composants `WACounter` et qui va donc conserver dans son état 5 variables d'instance de `WACounter` (les 5 instances sont ici conservées dans le tableau counters)

```
initialize
  super initialize.
  counters := (1 to: 5)
    collect: [ :each | WACounter new ]
```

Nous devons aussi écrire la méthode de rendu de `WAMultiCounter`. Elle itère simplement sur les variables d'instance et délègue l'opération de rendu à chacune d'elle. On réutilise ainsi le comportement existant défini dans WACounter. En complément, WAMultiCounter insère dans le rendu une ligne horizontale de séparation entre deux compteurs.

```
renderContentOn: html
   counters
   do: [ :each | html render: each ]
   separatedBy: [ html horizontalRule ]
```

En embarquant ainsi des composants, on crée au final une application sous la forme d'un arbre de composants. Il existe toujours un composant racine qui est une composition d'autres composants qui eux-mêmes peuvent être à leur tour faits de plusieurs composants et ainsi de suite. Notre nouvelle application a pour composant racine `WAMultiCounter` et les composants `WACounter` s'y trouvent embarqués.

Appeler des composants

Une autre façon d'assembler des composants dans une application consiste à "appeler" un composant depuis un autre composant. Cette méthode d'assemblage est unique à Seaside et elle est typiquement utilisée dans l'implémentation du flot de contrôle d'une application.

Illustrons ce concept dans notre application compteur en l'étendant avec une action 'reset'. Dans la méthode de rendu de `WACounter` nous ajoutons un lien 'reset' qui, à l'occasion d'un clic, va tout d'abord demander confirmation à l'utilisateur avant de remettre le compteur à 0.

```
html anchor
   callback: [ (self call: WAYesOrNoDialog new) ifTrue:
[ count := 0 ] ];
   with: 'reset'
```

Comme vous le voyez ce flot de contrôle est totalement spécifié dans le callback associé à l'ancre 'reset'. Le callback dit à notre composant WACounter qu'il devra tout d'abord instancier un objet de type WAYesOrNoDialog avec l'instruction self call: WAYesOrNoDialog new`. Cela aura pour effet d'afficher une boîte de dialogue avec deux boutons Yes et No immédiatement après le clic sur le lien. Si l'utilisateur confirme en cliquant sur Yes, le composant WAYesOrNoDialog renvoie true ce qui aura pour info d'exécuter le bloc de code en argument de ifTrue: et donc de remettre la variable d'instance count à 0. Facile et concis, non ?

Si vous regardez le code du composant WAYesOrNoDialog, vous verrez que la réponse est produite dans un callback comme suit:

```
html button
  callback: [ self answer: true ];
  with: 'Yes'
```

Lors d'un appel de composant, le composant appelé va temporairement remplacer le composant appelant dans l'arbre des composants de l'application. Seaside effectuera ensuite le rendu de l'arbre, ce qui dans notre exemple signifie que Seaside va effectuer le rendu de `WAYesOrNoDialog` à la place de WACounter. Lorsque le composant appelé a répondu, le composant d'origine est remis en place dans l'arbre et Seaside poursuit l'opération de rendu.

Seaside propose aussi la notion de tâches qui sont en fait des composants qui ne font aucun rendu, mais décrivent simplement la logique d'interaction entre les composants et donc la logique métier de l'application. Il faut aussi rappeler ici que rien n'interdit au développeur d'utiliser toutes les possibilités de la programmation objet, dont les sous-classes, pour implémenter des variations autour de composants déjà existants.

Parmi les exemples de librairies de composants Seaside les plus populaires, on peut citer Willow [10], les wrappers de Bootstrap [11] et de Material Design Lite [12].

Et les continuations dans tout ça ?

A tout moment, Seaside s'assure que l'utilisateur d'une application peut utiliser les boutons navigation avant, navigation arrière et rafraîchissement du navigateur. Sur un clic du bouton arrière, Seaside présentera l'état précédent de l'application à l'utilisateur. Dans l'application compteur , un clic sur back va effectivement remettre le compteur dans son état précédent. Et de la même façon, un clic sur le bouton avance renverra le compteur à l'état où il était.

Seaside gère donc de façon transparente pour le développeur le comportement à adopter en cas d'utilisation de ces boutons de navigation. Il y parvient en sauvegardant l'état des composants de chaque page et en incluant une clé unique pour chaque page dans l'url. Il est possible pour le développeur de personnaliser ce qui est mémorisé et quelle doit être l'étendue de l'historique. Par ailleurs, Seaside utilise les continuations (un mécanisme de programmation qui permet de suspendre et reprendre un flot d'exécution) pour le l'appel de composants. Les continuations étant très consommatrices de mémoire, un mécanisme dit de "continuations partielles" est en place depuis la version 3 de Seaside.

Étant donné que Seaside gère les transitions entre pages web ainsi que les boutons de navigation du navigateur de façon transparente, le développeur peut implémenter le flot de contrôle de l'application de façon très naturelle en utilisant simplement des callbacks qui changent l'état d'un composant ou appellent d'autres composants plutôt que de penser en termes de requête http et de pages HTML.

Applications Web hybride: Javascript & Seaside

Aujourd'hui, de nombreuses applications Web sont écrites en JavaScript et s'exécutent directement dans le navigateur. Cela permet de proposer des interfaces utilisateur infiniment plus dynamiques et interactives en comparaison d'un rendu qui serait effectué uniquement du côté serveur nécessitant donc un aller-retour entre navigateur et serveur à chaque nouvelle interaction. En Seaside, il est très facile d'étendre votre application avec du code tournant sur le client (JavaScript) permettant ainsi de bénéficier du meilleur des deux mondes.

Une application peut faire du rendu de code JavaScript à peu près comme du rendu HTML. Ce code JavaScript généré dynamiquement peut référencer un callback en Seaside permettant ainsi à du code JavaScript exécuté sur le client de faire appel à du code d'un composant Seaside résidant côté serveur. Comme vous avez aussi accès à des "wrap callbacks", vous pouvez générer de petits morceaux de code qui relient entre eux du code client en JavaScript et du code serveur en Smalltalk.

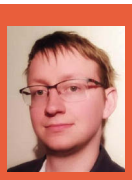
Il existe plusieurs extensions de Seaside qui habillent (wrap) des librairies JavaScript en composants Seaside réutilisables ou en extension des API de rendu html de l'objet canvas. Seaside est aussi livré en standard avec un wrapper de JQuery, ce qui facilite l'intégration d'autres librairies qui s'appuient sur ce dernier.

Pour en savoir plus sur Seaside, le meilleur point de départ est l'ouvrage Seaside [5] et le tutoriel en ligne [6]. Un chapitre lui est aussi consacré dans l'ouvrage Pharo par l'exemple [7]. Si vous êtes intéressés par Seaside sur Gemstone/S et sa solution native de persistance native vous pouvez aussi suivre le tutoriel de Seaside [8]. J'espère que cette courte promenade en bord de mer vous aura mis en appétit !

[1] https://www.github.com/SeasideSt/Seaside
[2] https://pharo.org/
[3] https://gemtalksystems.com/small-business/gsdevkit/
[4] https://jquery.com/
[5] http://book.seaside.st/book
[6] http://www.swa.hpi.uni-potsdam.de/seaside/tutorial
[7] https://books.pharo.org/updated-pharo-by-example/
[8] http://seaside.gemtalksystems.com/tutorial.html
[9] https://github.com/ba-st/Willow
[10] https://github.com/astares/Seaside-Bootstrap4
[11] https://mdl.ferlicot.fr/

Scripter VLC avec Pharo FFI

VLC est le logiciel le plus connu pour lire des fichiers audio et vidéo. Il permet aussi de lire des streams, et donc de lire, par exemple, le stream diffusé par des caméras de sécurité. Nous verrons comment utiliser Pharo pour s'interfacer avec la librairie développée pour VLC et donc de contrôler VLC depuis Pharo.



Benoît Verhaeghe

Doctorant et travaille dans le service recherche et développement de Berger-Levrault. Il se concentre sur la mise à jour des anciennes applications et utilise pour cela Pharo. À ses heures perdues, il développe des solutions permettant d'interfacer Pharo avec d'autres frameworks ou langages de programmation.

Lorsque je travaille, j'ai souvent envie d'écouter de la musique. Mais ma musique est stockée à différents endroits: en local sur mon ordinateur portable, sur mon serveur personnel, sur YouTube et sur des comptes premium de musique en ligne. Bien que je puisse passer d'un système à un autre facilement, je rêvais d'un gestionnaire de musique où tout se retrouverait combiné à un seul endroit.

Jouer de la musique avec Pharo

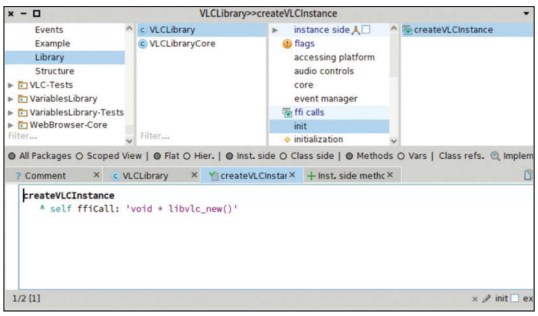
Pharo permet déjà de jouer de la musique directement via un plug-in intégré dans la machine virtuelle (VM). Malheureusement ce plug-in est vieillissant. VLC est capable de lire tous ces formats audio et vidéo. Afin de pouvoir jouer de la musique contrôlée par Pharo, il suffit donc de manipuler VLC. Et c'est ce que va nous permettre de faire uFFI ! En effet, le projet VLC est supporté sur les trois plateformes les plus utilisées (Linux, Windows et macOS) et fournit les librairies partagées libvlc et libvlccore. Une librairie partagée a pour objectif d'être utilisée par d'autres programmes. Elles définissent une API permettant à un programme d'interagir avec un autre. Et c'est précisément ce que permet de faire uFFI à savoir appeler depuis Pharo les méthodes définies dans une librairie partagée.

Mettre en place FFI

Mettre en place FFI n'a jamais été aussi simple. Il n'y a en effet que très peu de prérequis : installer le projet que l'on veut utiliser (pour moi c'est VLC), s'assurer que les librairies partagées sont dans le PATH du système. Et voilà pour la première étape !Une fois le projet installé, nous passons au développement. Tout d'abord, nous créons une classe qui sert de lien entre Pharo et VLC. Pharo venant avec un framework complet pour mettre en place FFI, il nous suffit de créer une sous-classe de FFILibrary (VLCLibrary) et de lui indiquer le nom des librairies externes en fonction du système que l'on utilise (.so sur Linux, .dll sur Windows et .dylib sur OSX). Nous avons fini la mise en place de FFI !

```
VLCLibrary>>#unix64LibraryName
  #('/usr/lib/i386-linux-gnu' '/usr/lib32' '/usr/lib')
, ((OSEnvironment current at: 'LD_LIBRARY_PATH' ifAbsent:
[ '' ]) substrings: ':')
    do: [ :path |
        | libraryPath |
        libraryPath := path asFileReference / 'libvlc.so'.
        libraryPath exists ifTrue: [ ^ libraryPath fullName ] ].
  self error: 'Cannot locate vlc library. Please check if it
installed on your system'
```

Figure 1.
Appel FFI de la méthode libvlc_new()



Mappage des fonctions

Nous allons maintenant pouvoir mapper les fonctions de VLC. Pour pouvoir utiliser une fonction définie dans la librairie, nous déclarons dans notre classe VLCLibrary les méthodes que nous allons utiliser. Pour connaître les fonctions, deux options s'offrent à nous, utiliser des outils qui vont extraire des librairies les fonctions disponibles ou… utiliser la documentation ! Heureusement pour nous les librairies de VLC sont bien documentées ce qui simplifie grandement le travail. En accédant à la documentation des librairies, nous avons accès à la signature des fonctions disponibles. Il suffit de copier-coller la signature des fonctions dans Pharo pour pouvoir ensuite les utiliser. Les types primitifs sont compris par Pharo tandis que nous remplacerons les structures astucieusement par void * afin de les manipuler comme des objets opaques.

```
VLCLibrary>>#createVLCInstance
  ^ self ffiCall: void * libvlc_new()'
```

Un premier test

Nous allons maintenant présenter les premières méthodes que nous avons dû implémenter pour utiliser VLC. Tout d'abord, libvlc_new permet de créer une instance de VLC. Nous allons ensuite utiliser cette instance pour scripter VLC.

Figure 1

Ensuite, nous allons développer le code nécessaire pour jouer une musique hébergée sur notre ordinateur. Pour cela, nous avons besoin de trois méthodes: création d'un média, création du média player correspondant et la méthode play du média player. Les trois méthodes sont décrites dans la documentation par : libvlc_media_new_path, libvlc_media_player_new_from_media et libvlc_media_player_play. Pour leurs implémentations, il suffit de copier les signatures, et de créer des méthodes dans la classe VLCLibrary comme nous l'avons fait pour le new.

```
VLCLibrary>>#mediaNew: aVLCInstance path: aStringPath
  "Create a media for a certain file path."
  ^ self ffiCall: void * libvlc_media_new_path(void *
aVLCInstance, String aStringPath)'

VLCLibrary>>#mediaPlayerPlay: aMediaPlayer
  "Play"
  ^ self ffiCall: 'int libvlc_media_player_play(void
* aMediaPlayer)'
```

Enfin, quelques lignes dans le playground vont nous permettre de scripter VLC pour jouer une musique. **Figure 2**

Mappage des structures - Programmons en Pharo !

En écrivant en Pharo toutes les méthodes publiques des librairies de VLC, il est possible de complètement scripter VLC. Cependant, nous obtenons une seule abstraction avec plein de méthodes alors qu'il serait nettement préférable de créer des objets qui représentent chacun des aspects de VLC.
Il est ainsi possible d'améliorer notre mapping avec une librairie VLC en effectuant le mappage des structures C vers des objets Pharo. Encore une fois, nous allons pouvoir le faire grâce à VLC. Les structures peuvent être divisées en quatre catégories : les énumérations, les callbacks, les structures C et les structures opaques. FFI offre une solution pour le mappage de chacune d'elles. Pour les énumérations, nous devons étendre la classe Pharo « FFIExternalEnumeration », ensuite, nous implémentons la méthode « enumDecl » en écrivant un tableau qui contiendra le nom de l'élément de l'énumération suivi de sa valeur, etc. Puis, nous exécutons la méthode « rebuildEnumAccessors » sur la classe, étape qui va générer les accesseurs à toutes les valeurs de l'énumération. Enfin, nous initialisons la classe. Il est maintenant possible d'utiliser l'énumération en utilisant : nom de l'énumération + nom d'un des éléments de l'énumération.

```
VLCMediaType class>>#enumDecl
"self rebuildEnumAccessors"

  ^ #(libvlc_media_type_unknown 1
libvlc_media_type_file 2
libvlc_media_type_directory 3
libvlc_media_type_disc 4
libvlc_media_type_stream 5
libvlc_media_type_playlist 6)
```

Pour les callbacks, uFFI vient aussi avec une implémentation facile à utiliser. Cette fois-ci, nous devons étendre la classe « FFICallback ». Nous allons ensuite définir la signature de fonction à laquelle notre callback correspond en C et le block Pharo qui sera exécuté lorsque le callback sera appelé. Pour la signature de fonction, nous définissons deux méthodes : la première « fnSpec » qui retourne la signature de fonction comme nous l'avions fait lors du mappage de l'API, et la seconde « on: » prend un block en paramètre qui sera exécuté lors de l'appel du callback. C'est pour les structures opaques qu'il est le plus simple de définir un mappage FFI. Il suffit d'étendre la classe Pharo « FFIOpaqueObject ». Cette étape permet ainsi d'utiliser cette structure en ne considérant que ses méthodes. C'est une stratégie employée en C pour cacher le fonctionnement interne de la structure. Enfin, il reste le cas des structures C. Comme pour les cas précédents, nous étendons cette fois-ci la classe « FFIExternalStructure ». Ensuite, nous définissons côté classe la méthode « fieldsDesc » qui retourne un tableau contenant l'ensemble des variables de la structure et leurs types. En exécutant la méthode « rebuildFieldAccessors » sur la classe, nous créons aussi les accesseurs de ces attributs automatiquement.

```
VLCTrackDescription>>#fieldsDesc
    "self rebuildFieldAccessors"
    ^ #(int i_id;
    String psz_name;
    VLCTrackDescription * p_next;)
```

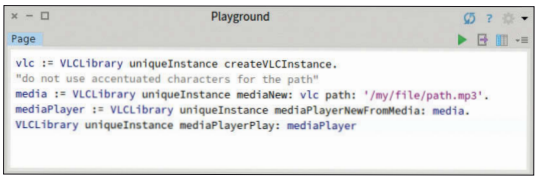


Figure 2.
Script pour jouer le fichier path.mp3

Nous pouvons maintenant complètement utiliser la librairie VLC comme si c'était une librairie Pharo.

Et graphiquement ?

Comment rapidement créer une interface dans Pharo qui va nous permettre de contrôler VLC ? Pour ce faire, nous allons utiliser la nouvelle version de Spec avec Pharo 9. L'objectif final est de créer une interface de lecteur audio utilisable pour l'utilisateur. Plus précisément, nous allons ajouter une extension à l'inspecteur de Pharo afin de pouvoir regarder et contrôler l'état de nos player VLC. Pour cela, nous allons faire deux choses : ajouter une extension et créer l'interface utilisateur de l'extension. Pour ajouter une extension, il suffit de créer une méthode avec le pragma « inspectorPresentationOrder: title: » qui retourne l'interface que nous voulons créer. Pour définir l'interface utilisateur, nous utilisons Spec2 (le framework graphique de référence depuis Pharo 8+). Nous avons défini une progress bar qui nous montre l'avancement d'un morceau de musique et deux boutons « start » et « pause » pour contrôler la lecture. L'exemple complet est disponible sur le repository github de Pharo-LibVLC en chargeant la groupe « inspector » de la baseline.

```
Metacello new
 baseline: 'VLC';
 repository: 'github://badetitou/Pharo-LibVLC';
 load: 'inspector'
```

Puis il est possible de lancer une musique et d'obtenir l'interface suivante en inspectant :

```
vlc := VLCLibrary uniqueInstance createVLCInstance.
media := vlc createMediaFromPath: '/path/to/file.mp3'.
mediaPlayer := vlc createMediaPlayerFromMedia: media.
mediaPlayer play
```

Conclusion

Nous avons vu qu'il est possible de contrôler VLC depuis Pharo. Nous pouvons ainsi lire des morceaux qui proviennent de notre ordinateur, mais aussi depuis d'autres services (comme YouTube) en utilisant la capacité de VLC à jouer un stream audio. Nous avons présenté une première interface pour contrôler le player audio de manière graphique. Alors, pourquoi ne pas continuer dans cette direction en créant un media center complet en Pharo, soit sous forme d'application desktop ou bien d'une application web avec le framework web Seaside.

Ressources

http://mooc.pharo.org
http://www.pharo.org
https://en.wikipedia.org/wiki/Pharo
https://github.com/SquareBracketAssociates/Booklet-uFFI/
https://github.com/badetitou/Pharo-LibVLC (projet binding FFI VLC avec Pharo)
https://www.videolan.org/ (VLC homepage)
https://www.videolan.org/developers/vlc/doc/doxygen/html/group__libvlc.html (libvlc documentation)
https://github.com/pharo-spec/Spec (Créer une interface en Spec2)
http://seaside.st/ (Faire du web avec Pharo)
https://github.com/pharo-media-center (Créer un media center en Pharo - développement en cours)

Marc Bojoly

Après avoir été développeur, architecte, formateur AWS et consultant senior, je suis désormais Engineering Manager des équipes Hive et Fairways Dette chez Finance Active.

Notre logiciel de gestion de la dette en SaaS est basé sur des technologies, Java, Angular, Docker. Nous étudions actuellement l'opportunité de sa migration sur le cloud.

Démarrer dans le développement cloud

LE CLOUD COMME ACCÉLÉRATEUR DES DÉVELOPPEMENTS

Qui se souvient du temps où la mise en ligne d'un site dynamique avec PHP (ou une techno équivalente) était complexe et limitée par les fonctionnalités de son fournisseur d'accès ? Déployer n'importe quelle technologie signifiait héberger un serveur et nécessitait de très fortes connaissances en infrastructure. Le Cloud Computing (ou tout simplement cloud), ou «l'informatique dans les nuages», a révolutionné la possibilité de développer et mettre en ligne une application. Mais comment définir le Cloud ? Comme un nuage est constitué de gouttelettes, le Cloud est constitué de l'ensemble des services web donnant accès, de façon programmable, à des ressources informatiques sur internet.

Derrière le terme ressource informatique se cachent des serveurs, des disques durs, mais également tout un ensemble de services de plus haut niveau permettant de faire fonctionner des applications. Ces services, comme des bases de données, des services d'authentification sont les briques de bases de ces plateformes, car elles permettent de déployer ces applications, qui prennent désormais le nom d'Applications Cloud Native.

La **figure 1** résume les principaux types de services que l'on peut rencontrer dans une architecture Cloud.

Ils peuvent se décomposer en trois grandes catégories : les services d'Infrastructure As Code (IaaS) qui exposent du matériel (machine, réseau, virtuel), des services de Container As A Service (CaaS) qui constituent une Plateforme As A Service (PaaS) : le matériel est masqué et on orchestre des applications packagées dans des conteneurs Docker. Et des Software As A Service (SaaS) exposant directement des fonctionnalités logicielles. Une plateforme de contrôle de code source comme GitHub peut être dans cette catégorie.

Le Cloud est une plateforme de choix pour développer et déployer un site web, une application web ou mobile dont la partie serveur (tout ce qui est back-end donc) peut s'exécuter dans un conteneur (les technologies Java, .Net, Node.js le permettent aujourd'hui), car tous les services nécessaires sont disponibles. Le cas d'une application historique (ou legacy), même si son déploiement n'est pas impossible reste beaucoup moins recommandé (plus difficile dirions-nous, NDLR). Nous prendrons un exemple de ce type pour illustrer notre première application Cloud Native et nous reparlerons ensuite des autres types d'application.

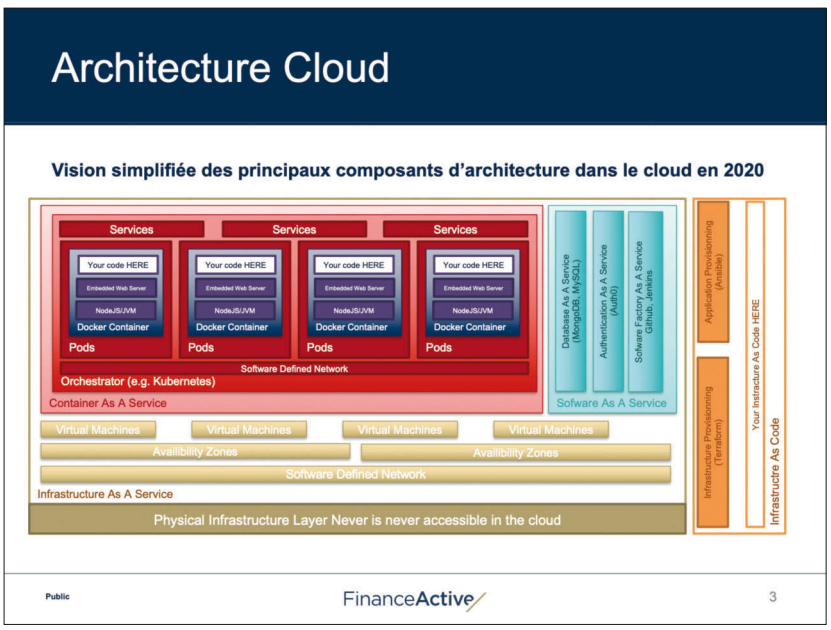


Figure 1

Check-list pour démarrer

Votre application est-elle accessible via un navigateur ? **O**

Partez-vous d'un existant ? **X**

Ou développez-vous une nouvelle application ? **O**

L'un des composants principaux que vous souhaitez utiliser (base de données, moteur de machine learning,...) est-il disponible comme un service natif dans le Cloud ? **O**

Les autres composants que vous utilisez sont-ils packagés
- Sous forme d'images Docker **OOO**
- Sous forme de paquets Linux (.deb, .rpm...) ? **O**
- Disponibles uniquement sur des systèmes d'exploitation non basés sur Linux (Windows, Solaris) **XX** (voire AS/400 ou S/360 si vous savez à quoi cela ressemble) ? **XXX**

L'utilisation de votre application se fait-elle sur des plages de temps bien définies ? **O**

Si vous avez plus de **O** que de **X** alors je vous recommande de fortement de déployer votre application sur le Cloud.

Choisir les outils pour écrire ma première application Cloud Native

Le Cloud a l'avantage d'être quasi agnostique des langages de programmation(1). Pour illustrer le développement dans le Cloud, nous verrons comment développer une application Angular entièrement dans le Cloud en utilisant le langage TypeScript.

La première étape pour déployer son application est de choisir sa plateforme, donc son fournisseur Cloud. Le choix peut être important et parfois avec des contraintes. Pour l'instant, plaçons-nous dans la peau d'un développeur voulant construire sa première application cloudisée. Pour développer une application, nous avons besoin de 3 éléments : un éditeur de code ou un IDE, un compilateur et/ou un outil de dépendance et de packaging (exemples : npm en JavaScript/TypeScript et maven en Java) et un serveur ou un PaaS.

Le Cloud est d'abord et avant tout une plateforme de déploiement. AWS se définit d'ailleurs comme un fournisseur de ressources informatiques programmables, disponibles dynamiquement et payables à la demande. Pour déployer une application Angular, il faut un simple service : un stockage objet (S3 chez AWS, Azure Object Storage chez Azure, Object Storage chez OVH Cloud, OSU chez Oustcale, etc.). Ces services sont managés. On ne s'occupe de rien.

Nous fournissons les fichiers constituant notre application et le Cloud les met à disposition sur internet et donc au monde entier ! Comment cela fonctionne ? Lorsqu'un utilisateur se connecte à l'URL fournie, son navigateur va interroger le service de système de stockage objet. C'est l'un des services fondamentaux du Cloud. Les données stockées sont accessibles (avec des vérifications de droit bien sûr) directement via cette URL comme un immense serveur web au nombre virtuellement illimité de pages. Nous stockerons dans celui-ci les fichiers produits par Angular qui constituent notre application telle qu'elle sera chargée par le navigateur.

Pour le côté codage, nous utilisons la version en ligne de l'IDE de Microsoft : Visual Studio Code. Sa version locale est gratuite, open source et disponible sur les principaux systèmes. La version en ligne est disponible sur Azure Visual Studio Codespaces. L'outil est en train de devenir l'IDE proposé par GitHub. L'architecture de Visual Studio Code est intéressante, car elle prépare l'arrivée de la version en ligne. Elle sépare la partie affichage du code de la partie analyse syntaxique comme cela est expliqué dans cet article **https://code.visualstudio.com/docs/remote/remote-overview.** Cela permet d'abord de faire de Visual Studio Code un IDE dont l'affichage peut être réalisé entièrement dans un navigateur. Il fournit les mêmes fonctionnalités sur un PC et dans le Cloud.

Enfin nous ferons l'hypothèse d'une application basée sur Angular dans sa dernière version avec tous les outils qui l'accompagnent. Il existe de très bons articles sur ce sujet et nous n'entrerons pas plus dans le détail. Passons à la pratique.

Passons à la pratique !

Nous allons développer puis déployer sur le cloud une petite application qui se limitera à un écran et utilisera un seul service cloud : le stockage objet. Cette application implémentera le Hello World d'une application de gestion de dette : le calcul d'un échéancier à taux fixe et amortissement constant ! Nous l'implémenterons totalement avec la technologie Angular pour nous concentrer sur la prise en main du Cloud.

Avant toute chose, vous pourrez trouver le code source de cet article dans les deux repositories Github mbojoly/programmez-cloudapp et mbojoly/dotfiles. La convention consistant à utiliser un repository public nommé *dotfiles* dans son compte est importante pour la suite.

La première étape est de créer notre environnement de développement sur le Cloud. Nous aurons pour cela besoin d'un dépôt de code source (repository en anglais) Github qui contiendra notre code source et assurera le contrôle de source. Nous aurons également besoin d'un second repository Github sur lequel je reviendrai plus tard que nous appellerons *repository Dotfiles*. Je vous laisse dérouler le formulaire Github correspondant. **Figure 2**

Nous créons un compte sur Azure pour l'utilisation de Visual Studio Code. Rassurez-vous, comme Github, le service est gratuit pour l'utilisation que nous en ferons. Si vous disposez déjà d'un compte Microsoft vous pourrez l'utiliser pour vous logger. **Figure 3**

Nous allons désormais pouvoir créer notre environnement de travail dans le Cloud qui se nomme un codespace. Ce service est en cours de migration de Azure vers Github. A l'heure où j'écris ces lignes le service Github est encore en bêta privé, mais à l'heure de la mise en kiosque le service Azure n'accep-

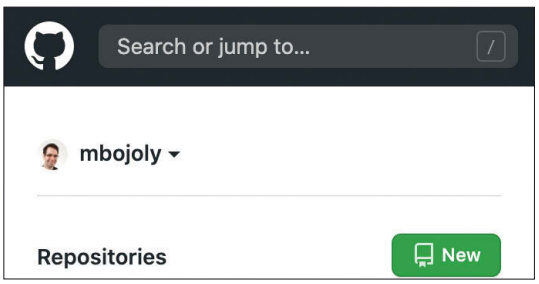


Figure 2

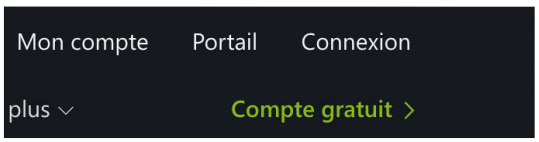


Figure 3

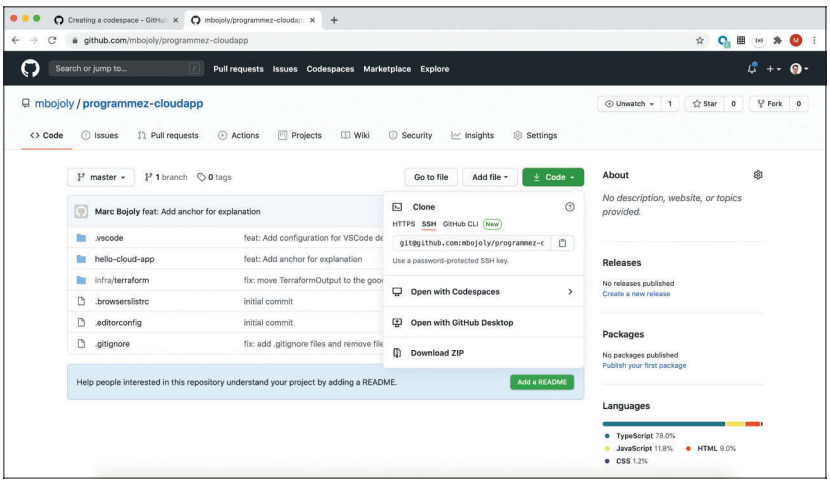


Figure 4

(1) NDLR : il faut tout de même que les langages soient supportés via des runtimes / SDK. Les principaux langages le sont.

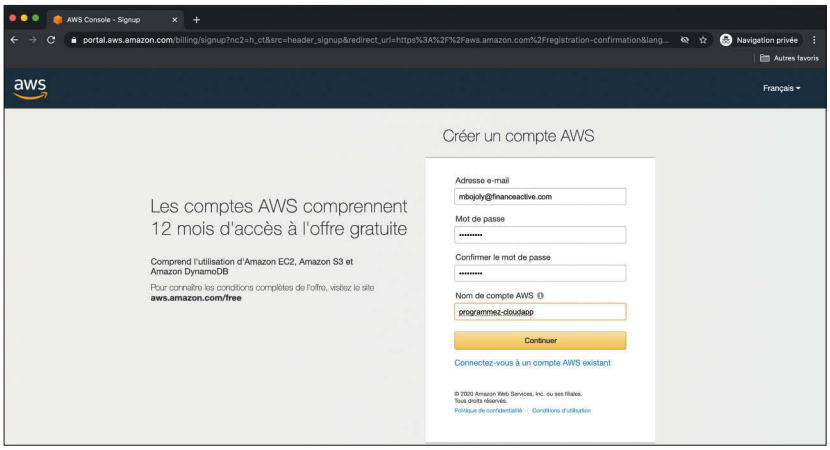


Figure 5

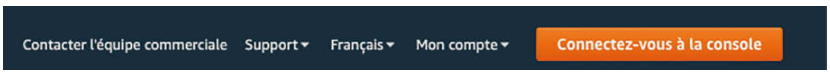


Figure 6

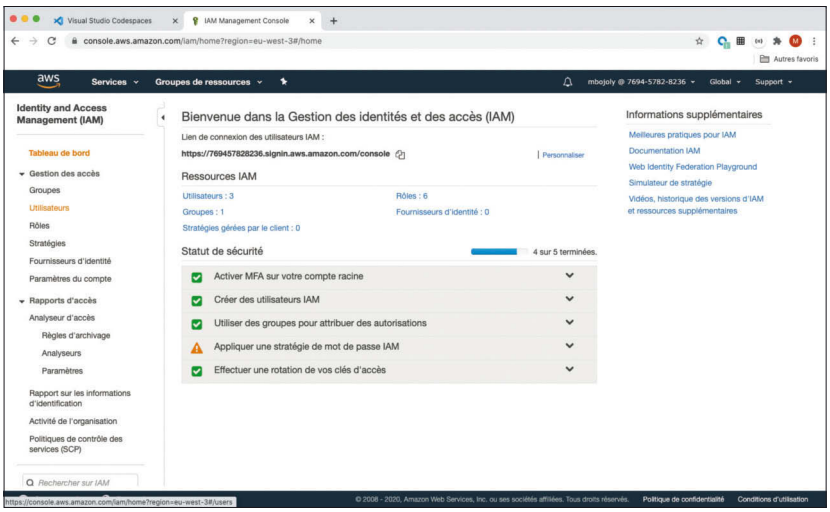


Figure 7

Figure 8 et 10

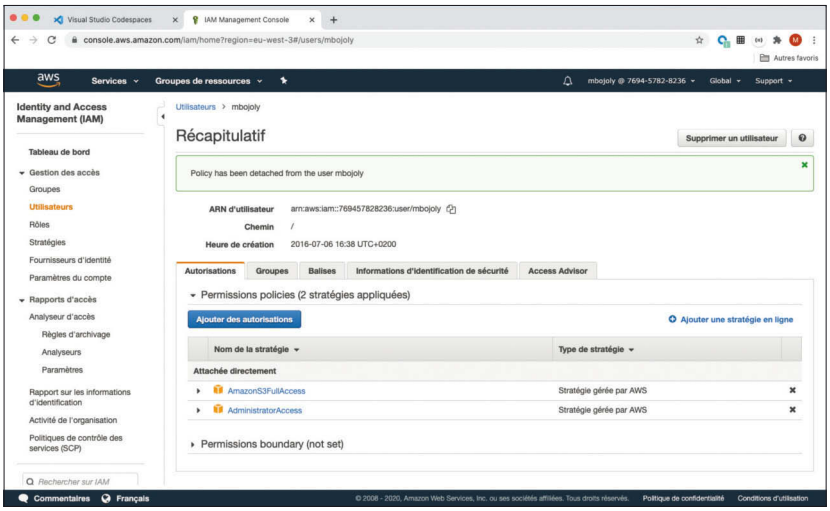


Figure 9 et 11

tera plus de nouveaux codespaces pour très longtemps. Nous allons donc présenter les deux interfaces.

Dans l'interface Azure nous spécifions les deux repositories git que nous avons créés et les informations suivantes :

- Codespace Name : programmez-cloudapp
- Git Repository : https://github.com/mbojoly/programmez-cloudapp
- Instance Type : Basic (Linux) 2 cores, 4 GB RAM
- Suspend idle Codespace after : 5 Minutes
- Dotfiles Repository : https://github.com/mbojoly/dotfiles
- Dotfiles Install Command : ./install.sh
- Dotfiles Target Path : ~/dotfiles

Dans l'interface Github, nous créons directement le codespace à partir du menu Code de Github. Par convention, il recherchera un repository public nommé dotfiles dans notre compte GitHub. **Figure 4**

Nous pouvons alors démarrer notre environnement de développement dans un navigateur. Plus aucun besoin d'installer de logiciel en local. S'il est nécessaire d'avoir un bon écran et un navigateur récent, une bonne partie du travail consommateur réalisé par un environnement de développement est déporté sur le Cloud. La compilation du code notamment s'effectue dans le cloud Azure et plus précisément dans un conteneur Docker. Cet environnement est facturé à l'heure d'utilisation 9 centimes de l'heure ET lorsque l'environnement est en pause, mais non détruit à 1 centime de l'heure. La capacité à recréer son environnement à la volée apporte tout de suite de l'argent !

Avant de commencer le développement, nous allons créer le dernier compte nécessaire : le compte AWS qui nous permettra d'héberger les fichiers de l'application sur le service AWS S3 et de les exposer sur internet. **Figure 5**

Le compte AWS ne propose malheureusement pas de période d'essai et l'étape angoissante de laisser la carte bleue arrive à grands pas.

Mais là encore à la création d'un nouveau compte vous disposez de crédit gratuit suffisant pour héberger votre application pendant quelques mois. Si vous oubliez de supprimer ce que nous allons créer, l'utilisation que nous allons faire de ce service coûtera au plus quelques centimes par mois.

Nous pouvons ensuite nous connecter à la console AWS : **Figure 6**

Afin de protéger ses comptes Cloud il est fortement recommandé de se créer un utilisateur et d'activer l'authentification multi-facteurs. En effet, le compte créé à la première connexion est l'équivalent d'un compte root sous un système Linux : un compte qui a tous les pouvoirs et qu'on ne peut pas restreindre. Cela est également vrai pour Azure, mais plus problématique sur AWS où notre programme d'infrastructure as code pourra créer des ressources donc débiter hors période de gratuité notre carte bleue ! Nous utiliserons pour cela un premier service AWS via la console : Identity and Access Management. **Figure 7**

Nous cliquons sur Gestion des accès *Utilisateurs* dans le menu de gauche. **Figure 8**

Puis nous cliquons sur *Ajouter un utilisateur*.

J'ai créé un utilisateur *mbojoly* et je vous laisse choisir le nom qui convient. Nous attacherons ensuite à cet utilisateur des autorisations via des stratégies gérées par AWS c'est-à-dire des ensembles de droits. **Figure 9**

Je ne vais pas décrire en détail l'activation de l'authentification multi-facteurs pour cet utilisateur comme pour l'utilisateur racine (l'identifiant/mot de passe qui a saisi la carte bleue), vous suivrez pour cela la documentation AWS et la documentation Azure.

S3 peut être manipulé via la console AWS comme nous l'avons fait précédemment. Je recommande d'ailleurs de découvrir tous les nouveaux services à travers la console.

Il est toujours recommandé d'automatiser autant que possible nos actions, car elles devront être répétées très fréquemment. AWS propose une ligne de commande qui constitue une première solution simple pour nous entraîner à l'automatisation. Avant de l'utiliser, il faut nous authentifier pour que cette ligne de commande ait accès à nos données. Dans la console nous nous authentifions avec un identifiant et un mot de passe. Dans la ligne de commande AWS nous nous authentifions avec un autre couple : une *access key* et une *secret access key*. Le concept est un peu différent dans les autres clouds, mais le principe reste le même, une chaîne ou un couple de chaînes aléatoires vont être générées et nous devrons les stocker de façon sécurisée. Ce mécanisme peut dans les grandes lignes être comparé à une clé ssh. Une clé ssh est stockée par convention dans ~/.ssh dans un système Unix, par analogie AWS stocke le couple *access key/secret access key (Identifiant / clé d'accès secrète)* dans ~/.aws/credentials. Nous devons d'abord créer cette clé et cela se passe dans le service IAM.

Nous sélectionnons notre utilisateur : **Figure 10**

Puis nous sélectionnons *Informations d'identification et de sécurité*. **Figure 11**

Nous générons le couple *access key/secret access key (Identifiant / clé d'accès secrète)* en prenant soin de les stocker. **Figure 12**

Nous allons ensuite installer la ligne de commande AWS ce qui va nous faciliter la configuration du fichier ~/.aws/credentials. Pour cela, nous allons créer notre premier *codespace* en précisant le nom des deux repositories GitHub que nous avons créés initialement.

Figure 13

Nous obtenons ce type d'écran très reconnaissable en développement : un environnement de développement avec la liste des fichiers à gauche et au centre l'emplacement où on éditera le code.

Le schéma page suivante va nous aider à en comprendre le fonctionnement qui peut étonner quand on a l'habitude d'avoir un environnement de développement installé sur son PC.

Figure 14

Dans cette architecture, l'environnement de développement est réparti entre deux conteneurs : le navigateur et Docker dans le Cloud. L'architecture sera très similaire dans les versions Azure et Github.

Le navigateur va se charger de toute la partie interaction avec l'utilisateur : navigation dans le code source, affichage du code source et collecte de toutes les séquences de touches que nous pouvons taper. Mais contrairement à un programme IDE classique il ne va pas travailler seul. Il va déléguer toutes les tâches consommatrices à un conteneur Docker dans le cloud Azure. L'indexation des sources, la compilation, toutes les lignes de commande que nous exécutons dans l'IDE s'exécutent en réalité dans ce conteneur Docker. Lorsque dans l'IDE nous naviguons dans le système de fichiers ou nous utilisons la ligne de commande, c'est dans le conteneur Docker que nous exécutons tout cela.

Installons désormais la ligne de commande AWS. La ligne de commande se compose d'instructions à taper derrière un prompt (<nom du répertoire courant>:$ dans les exemples). Ces commandes sont saisies dans un terminal qui se trouve dans le menu *Fichier* > *Terminal* de Visual Studio Online.

Par la suite, lorsque nous indiquons... cela signifie que nous n'avons pas reproduit toute la sortie de la ligne de commande par souci de concision.

Nous devons nous trouver dans un répertoire ~/workspace/programmez-cloudapp sinon nous le créons.

```
codespace:~$ mkdir -p ~/workspace/programmez-cloudapp
...
codespace:~$ cd ~/workspace/programmez-cloudapp
...
```

Puis nous installons la ligne de commande.

```
codespace:~/workspace/programmez-cloudapp$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
codespace:~/workspace/programmez-cloudapp$ unzip awscliv2.zip
...
codespace:~/workspace/programmez-cloudapp$ ./aws/install -i ~/aws-cli -b ~/bin
```

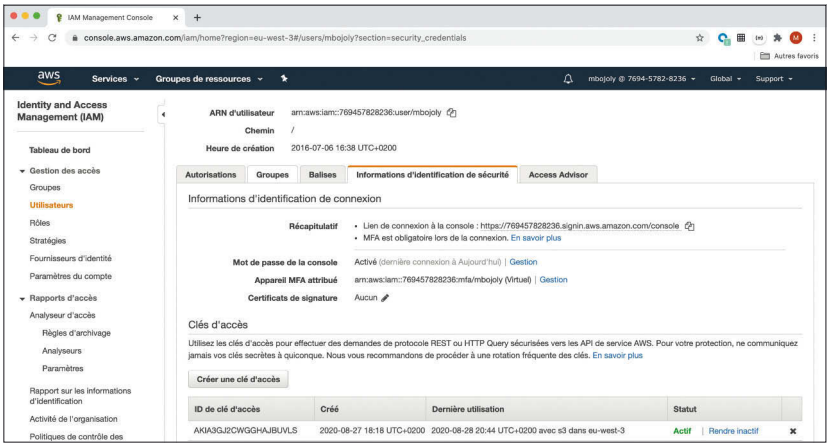


Figure 12

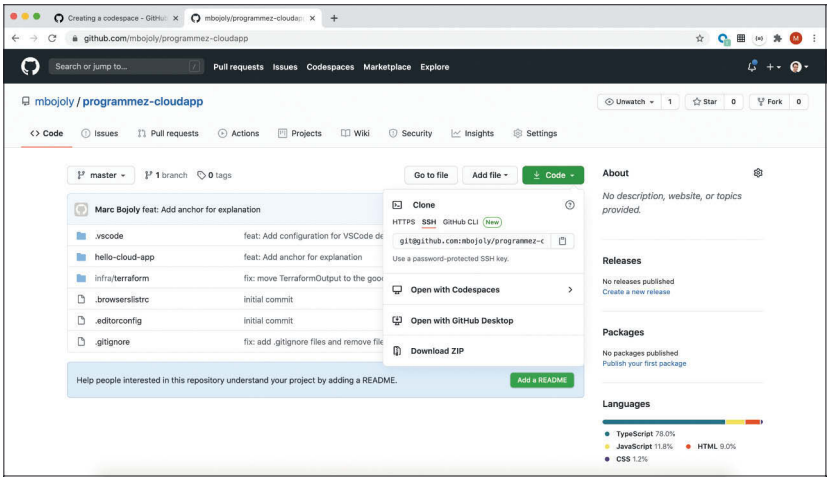


Figure 13

```
...
codespace:~/workspace/programmez-cloudapp$ echo "PATH=$PATH:~/bin" >> ~/.bashrc
codespace:~/workspace/programmez-cloudapp$ source ~/.bashrc
codespace:~/workspace/programmez-cloudapp$
```

Nous pouvons désormais vérifier que l'installation s'est bien passée.

```
codespace:~/workspace/programmez-cloudapp$ aws --version
aws-cli/2.0.33 Python/3.7.3 Linux/5.3.0-1032-azure botocore/2.0.0dev37
```

Les numéros peuvent légèrement varier en fonction des mises à jour. La CLI fournit ensuite un wizzard pour faciliter l'installation de l'identifiant et de la clé secrète que j'ai remplacée ici par xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.

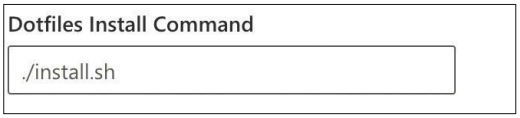
```
codespace:~/workspace/programmez-cloudapp$ aws configure
AWS Access Key ID [None]: AKIA3GJ2CWGGHSEV7D5Y
AWS Secret Access Key [None]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Default region name [None]: eu-west-3
Default output format [None]: json
```

Nous avons désormais accès à tous les services AWS auxquels l'utilisateur mbojoly a accès. **Attention donc** : il convient de protéger cette clé avec beaucoup d'attention. Ne la stockez jamais dans un espace public comme un repository GitHub. Toute personne ou plus exactement tout robot qui s'en empare peut consommer autant de service AWS que les droits de l'utilisateur le permettent ! C'est pour cette raison que notre utilisateur possède uniquement les droits sur le service dont nous avons besoin ici : AWS S3.
AWS S3 expose des buckets qui eux-mêmes contiennent des fichiers. Nous allons donc stocker notre premier fichier.

```
codespace:~/workspace/programmez-cloudapp$ aws s3 mb s3://programmez-cloudapp
make_bucket: programmez-cloudapp
codespace:~/workspace/programmez-cloudapp$ echo "Hello" > hello.txt
codespace:~/workspace/programmez-cloudapp$ aws s3 cp hello.txt s3://programmez-cloudapp
codespace:~/workspace/programmez-cloudapp$ aws s3 cp hello.txt s3://programmez-cloudapp
upload: ./hello.txt to s3://programmez-cloudapp/hello.txt
```

Nous avons créé notre premier fichier dans le cloud que nous pouvons désormais voir à travers la console AWS. **Figure 15**
Il s'agit maintenant de le transformer en un site web puis en une application web.

Avant d'aller plus loin, n'oubliez pas que Codespace est facturé au temps d'exécution et au temps d'inexécution. Bref, soyez attentive/if à vos services clouds… La facture peut vite exploser. Petit souci : notre répertoire Home (~ dans la ligne de commande) se trouve dans le conteneur Docker hébergé dans Azure. Si nous supprimons notre Codespace nous supprimons aussi ce conteneur et donc toute l'installation et la configuration que nous venons de réaliser ! Pour remédier à cela, il y a l'infrastructure as code (IaC). Nous allons écrire un programme - un script en l'occurrence - qui réalise l'installation que nous avons faite pour ne plus avoir à la répéter lorsque nous créerons un nouveau Codespace à notre retour de congé.
Nous allons utiliser pour cela le repository https://github.com/mbojoly/visualstudio-dotfiles que nous avons créé tout à l'heure. Vous vous rappelez peut-être dans la configuration de Codespace qu'il est possible de préciser une commande d'installation. Celle-ci sera exécutée à chaque fois que le Codespace est créé.



Nous allons donc dans Github utiliser la commande Get started by creating a new file et insérer à l'intérieur le contenu suivant (les lecteurs les plus perspicaces noteront que le repository doit se nommer désormais dotfiles et non visualstudio-dotfiles pour être conforme à la convention de Github). **Figure 16**

```
#!/bin/sh

curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
./aws/install -i ~/aws-cli -b ~/bin
echo "PATH=$PATH:~/bin" >> ~/.bashrc
source ~/.bashrc
```

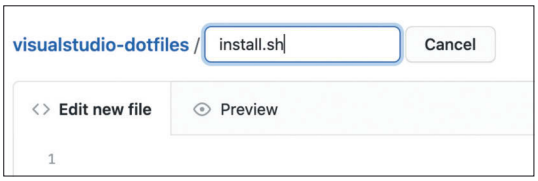
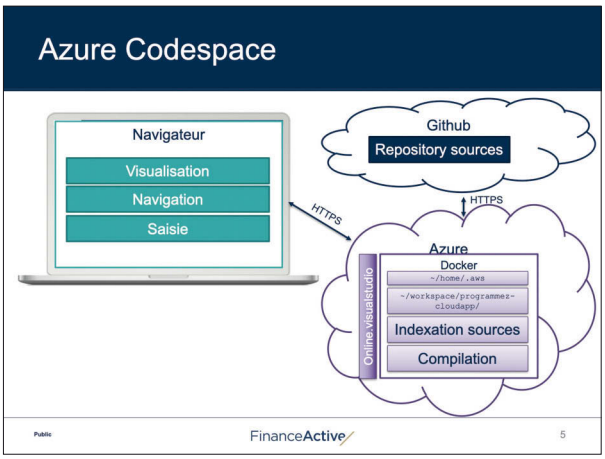


Figure 16

Figure 14



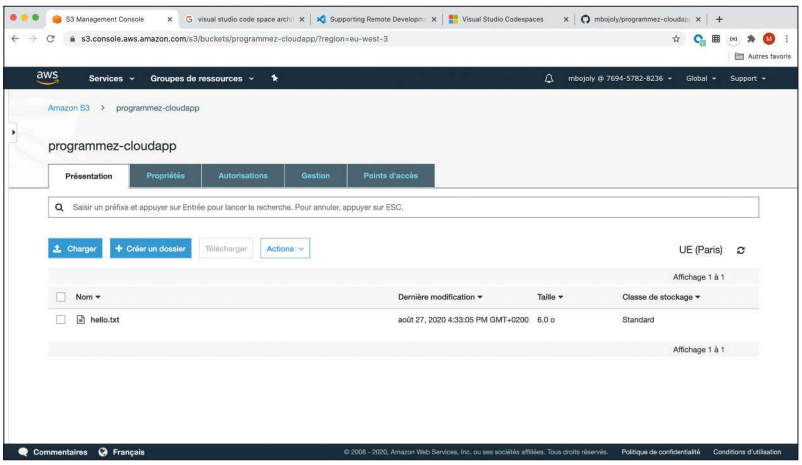


Figure 15

Et nous préciserons en message de commit en bas de l'écran Install AWS CLI.
Notez que je n'ai **pas** inclus la commande aws configure, car c'est ici que l'on saisit notre secret et il ne doit absolument pas être sauvegardé publiquement sous GitHub.
Si nous terminons et redémarrons notre Codespace, la ligne de commande AWS est pré-installée. Il est d'ailleurs nécessaire d'attendre quelques instants la fin de l'initialisation et de l'étape configuration de votre espace de code. Dans la version Github, une très légère différence, probablement l'installation par une autre instance de la ligne de commande, nécessite de taper à l'ouverture de celle-ci la commande suivante source ~/.bashrc, pour que les programmes aws et terraform puissent être localisés.

```
codespace:~/workspace/programmez-cloudapp$ source ~/.bashrc
```

Nous le vérifions avec :

```
codespace:~/workspace/programmez-cloudapp$ aws --version
aws-cli/2.0.43 Python/3.7.3 Linux/5.3.0-1035-azure exe/x86_64.debian.9
```

Et nous configurons notre secret :

```
codespace:~/workspace/programmez-cloudapp$ aws configure
AWS Access Key ID [None]: AKIA3GJ2CWGGHAJBUVLS
AWS Secret Access Key [None]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Default region name [None]: eu-west-3
Default output format [None]: json
```

C'est effectivement un peu fastidieux. D'autres providers de cloud fournissent des solutions un peu plus simples, mais le maintien des secrets dans un environnement entièrement dans le Cloud en est aujourd'hui à ses balbutiements. La gestion de la sécurité est l'un des points qui nécessite le plus d'attention et le plus d'investissement lorsqu'on passe sur le Cloud.
Ce script est très utile et il est suffisant pour installer notre petit environnement de développement. Néanmoins, il présente une faiblesse : il fait l'hypothèse qu'il s'exécute à chaque fois sur un environnement vierge. Que se passe-t-il si le fichier est déjà présent ? Au mieux nous risquons d'utiliser la mauvaise version. On dit de ce script qu'il n'est pas *idempotent*.
Le développement étant une continuelle suite d'écritures de fonctionnalités, de déploiements et de tests, nous sommes certains de devoir exécuter l'installation de notre application de très nombreuses fois. Nous n'allons donc pas utiliser un script de ce type pour déployer l'application Angular.
Là encore il existe une solution décrite dans le schéma ci-contre et implémentée par l'outil que nous allons utiliser : cdktf.
Figure 17
La solution consiste à maintenir un fichier d'état (en rouge sur le schéma) qui va lister l'ensemble des ressources créées sur le Cloud (représentées par l'arbre de formes géométriques). Lorsque l'installation est réexécutée, le programme va d'abord regarder dans le fichier quelles sont les ressources requises par le script d'installation qui sont déjà présentes dans le fichier d'état. Cela signifie qu'elles ont déjà été créées. Celles-ci ne seront alors pas régénérées. Ce fichier peut également être stocké dans le Cloud, par exemple dans AWS S3 afin de pouvoir synchroniser l'état sur un autre poste de développement.
Ce mécanisme est implémenté notamment par un outil très utilisé dans le monde de l'infrastructure as code : Terraform. Nous allons utiliser dans cet article une extension récente de Terraform appelée cdktf. Celle-ci nous permettra d'écrire nos scripts en TypeScript ce qui permettra à ceux qui écrivent leur première application avec ce dossier de n'utiliser qu'un seul langage de développement et aux autres de découvrir cdktf.
Nous allons installer les différents outils nécessaires en commençant par Terraform :

```
~/workspace/programmez-cloudapp$ wget https://releases.hashicorp.com/terraform/0.12.28/terraform_0.12.28_linux_amd64.zip
...
codespace:~/workspace/programmez-cloudapp$ unzip terraform_0.12.28_linux_amd64.zip
...
```

Nous vérifions son installation :

```
codespace:~/workspace/programmez-cloudapp$ terraform --version
Terraform v0.12.28

Your version of Terraform is out of date! The latest version
is 0.13.1. You can update by downloading from https://www.terraform.io/downloads.html
```

Les montées de version étant très rapides, vous obtiendrez potentiellement des messages indiquant qu'une nouvelle version est disponible quelques semaines après la parution sur cette commande ou sur une autre. Ils ne sont pas gênants.
Nous vérifions que des outils de base nécessaires pour cdktf sont bien disponibles dans Codespace.

```
codespace:~/workspace/programmez-cloudapp$ node --version
v12.16.3
codespace:~/workspace/programmez-cloudapp$ yarn --version
1.17.3
```

Puis nous installons cdktf.

```
codespace:~/workspace/programmez-cloudapp$ npm install --global cdktf-cli@0.0.16
/home/codespace/.npm-global/bin/cdktf -> /home/codespace/.npm-global/lib/node_modules/cdktf-
```

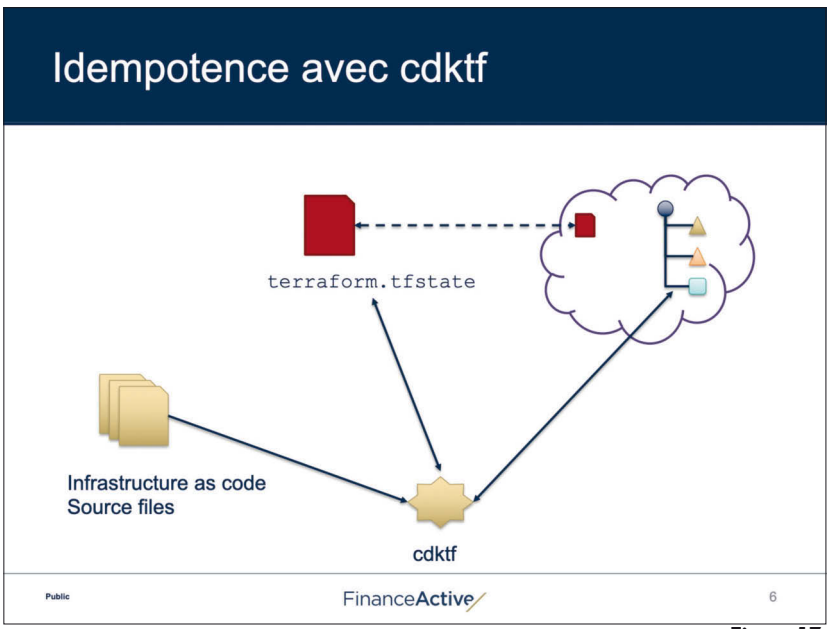


Figure 17

```
cli/bin/cdktf
npm WARN eslint-plugin-react@7.20.6 requires a peer of eslint@^3 || ^4 || ^5 || ^6 || ^7 but none is installed. You must install peer dependencies yourself.

+ cdktf-cli@0.0.16
added 209 packages from 140 contributors in 27.147s
```

Je vous laisse imaginer les modifications à réaliser sur le fichier install.sh ou utilisez directement ma version.
Pour valider notre installation de TypeScript qui sera nécessaire pour le programme d'infrastructure as code et notre application, voici un tout premier programme.

```
codespace:~/workspace/programmez-cloudapp$ mkdir -p hello-cloud-app
codespace:~/workspace/programmez-cloudapp$ cd hello-cloud-app
codespace:~/workspace/programmez-cloudapp/hello-cloud-app$ echo 'console.log("Hello World!")' >> hello.ts
codespace:~/workspace/programmez-cloudapp/hello-cloud-app$ npx tsc hello.ts
npx: installed 1 in 0.777s
codespace:~/workspace/programmez-cloudapp/hello-cloud-app$ node hello.js
Hello World!
```

Ce n'est pas encore une application cloud, mais c'est une application console ! Comme nous l'avons vu pour l'installation, même au temps du Cloud la console qui est la première interface de l'ordinateur est encore utile !
Nous avons donc un environnement de développement avec un script d'installation. Nous avons installé cdktf qui nous permettra de déployer notre future application de façon idempotente ce qui nous permettra de tester plus facilement l'installation. A quoi va ressembler cette future application ? Il s'agira d'une application Cloud Native : une application web, parlant nativement HTTP et déployée sur le Cloud par de l'Infrastructure As Code. Les ressources pourront ainsi être créées à la demande et supprimées lorsqu'elles ne sont plus utilisées.
Dans le bandeau de gauche *Explorer* de Visual Studio, nous allons créer un fichier HTML avec le code suivant.

```
<html>
  <body>
    <h1>Programmez !</h1>
  </body>
</html>
```

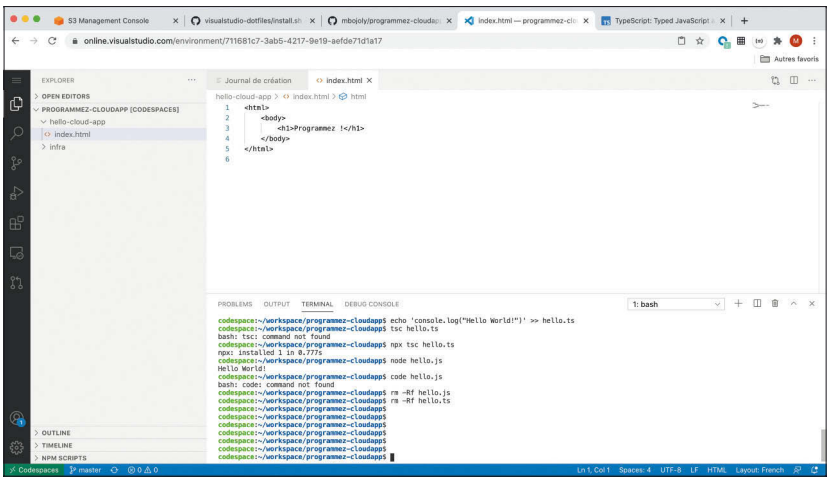


Figure 18

Figure 18
Nous allons désormais chercher à déployer cette application sur le web. Nous créons pour cela le répertoire dédié à notre infrastructure as code.

```
codespace:~/workspace/programmez-cloudapp$ mkdir -p infra/terraform
codespace:~/workspace/programmez-cloudapp$ cd infra/terraform
```

Puis nous initialisons le projet :

```
codespace:~/workspace/programmez-cloudapp/infra/terraform$ cdktf init --template=typescript --local
...
```

Un certain nombre de fichiers ont été créés.

```
codespace:~/workspace/programmez-cloudapp/infra/terraform$ ls
cdktf.json help main.d.ts main.js main.ts node_modules package-lock.json package.json tsconfig.json
```

TypeScript est un langage compilé. Il a besoin des définitions des librairies qu'il utilise à la compilation. Nous demandons à cdktf de les télécharger.

```
codespace:~/workspace/programmez-cloudapp/infra/terraform$ npm install
added 199 packages from 136 contributors and audited 200 packages in 4.214s

43 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Notre environnement d'infrastructure as code est prêt.

Codons notre infrastructure : vive l'IaC !

Nous allons modifier le fichier ~/workspace/programmez-cloudapp/infra/terraform/main.ts de la façon suivante.

```
import { Construct } from 'constructs';
import { App, TerraformStack, TerraformOutput } from 'cdktf';
import { AwsProvider } from './.gen/providers/aws/aws-provider'
import { S3Bucket, S3BucketConfig } from './.gen/providers/aws/
s3-bucket';
import { S3BucketObject, S3BucketObjectConfig } from './.gen/
providers/aws/s3-bucket-object';
import * as fs from 'fs';

interface WebFile { //1
  name: string;
  mimeType: string;
}

const DEPLOY_DIR= '../../hello-cloud-app/dist/hello-angular/' //2

class ProgrammezCloudAppStack extends TerraformStack {
  AWS_REGION = 'eu-west-3';
  AWS_S3_BUCKET_NAME = 'programmez-cloudapp'
  TERRAFORM_RELATIVE_PATH = '../' //Execution takes place in
.cdktf.out

  deployDir: string;

  constructor(scope: Construct, name: string, deployDir:
string, webfiles: WebFile[]) {
```

```
  super(scope, name);

  this.deployDir = deployDir

  new AwsProvider(this, 'aws', { //3
    region: this.AWS_REGION
  });

  const s3BucketConfig: S3BucketConfig = { //4
    website: [{
      indexDocument: 'index.html',
      errorDocument: 'index.html'
    }],
    tags: { 'stack':'programmez-cloudapp-stack' }
  };
  const s3Bucket = new S3Bucket(this, this.AWS_S3_BUCKET
_NAME, s3BucketConfig); //5

  for(let webfile of webfiles) { //6
    console.log('file ' + webfile.name)
    const s3ObjectConfig: S3BucketObjectConfig = {
      key: webfile.name,
      source: this.TERRAFORM_RELATIVE_PATH + this.deployDir
+ webfile.name,
      contentType: webfile.mimeType,
      acl: 'public-read',
      bucket: s3Bucket.bucket ?? this.AWS_S3_BUCKET_NAME,
      tags: { 'stack':'programmez-cloudapp-stack' }
    }
    new S3BucketObject(this, webfile.name, s3ObjectConfig); //7
  }
 }

 new TerraformOutput(this, 'Web site URL', { //8
   value: s3Bucket.websiteEndpoint
 });
}

class FilesLister { //9
 private MIME_TYPE_PER_EXTENSION = new Map([
  ['html', 'text/html'],
  ['css', 'text/css'],
  ['js', 'application/javascript'],
  ['ico', 'image/x-icon'],
  ['txt', 'text/text']
 ]);

 private findMimeTypeBasedOnExtension(files: string[]) :
WebFile[] {
  return files.map(f => {
    const fileNameSplit = f.split('.')
    const extension = fileNameSplit[fileNameSplit.length -1]
    return { name: f,  mimeType: this.MIME_TYPE_PER_
EXTENSION.get(extension) ?? 'text/plain' }
  })
 }

 listFilesToDeploy(deployDirPath:string) : WebFile[] {
  const files: string[] = fs.readdirSync(deployDirPath);
  return this.findMimeTypeBasedOnExtension(files)
 }
}

const app = new App(); //10
const filesLister = new FilesLister();
let webFiles: WebFile[] = filesLister.listFilesToDeploy(DEPLOY_DIR)
new ProgrammezCloudAppStack(app, 'programmezCloudAppStack',
 DEPLOY_DIR, webFiles);
app.synth(); //11
```

(1) nous créons une interface pour décrire la liste des fichiers qui devront être déployés et leur type MIME associé pour que le navigateur exécute correctement leur contenu.
(2) nous précisons où se trouvent les fichiers à déployer. Nous identifions dans un premier temps le répertoire contenant notre fichier index.htm.
cdktf va exécuter le programme TypeScript ci-dessus qui doit en retour lui fournir la liste des ressources AWS à créer. La façon d'écrire ce code est donc un petit peu particulière pour du TypeScript, car très déclarative. Chaque ressource AWS est déclarée en instanciant la classe correspondante et en l'associant à this c'est-à-dire à l'instance de l'application.
(3) nous déclarons un *provider* AWS permettant à cdktf de connaître les instructions pour le Cloud AWS. cdktf comme terraform sont en effet multi-cloud ce qui nous confère l'avantage décisif de nous rendre indépendants de notre fournisseur de Cloud. Nous précisons avec le nom de région eu-west-3 que nous souhaitons utiliser les datacenters Amazon situés dans la région de Paris.
(4) nous créons ensuite la configuration du *bucket* S3 (la traduction officielle est un seau, mais je n'arrive pas à utiliser ce terme). Nous souhaitons que ce *bucket* stocke, mais surtout expose ces fichiers à la manière d'un serveur HTTP pour que tout navigateur puisse y accéder. Nous précisons pour cela une configuration web en indiquant que la page index.html doit être la page par défaut, mais également - dans un souci de simplification - la page renvoyée si une erreur survient (par exemple une page qui n'existe pas). Nous pouvons alors créer notre *bucket* et l'associer à this c'est-à-dire à notre stack en cours de construction.
A l'intérieur de ce *bucket* nous voulons charger les différents fichiers constituant l'application. Nous allons pour cela lister tous les fichiers du répertoire de déploiement et identifier à partir de leur extension quel type MIME le serveur doit renvoyer pour que le fichier soit traité correctement par le navigateur. En **(9)** la classe FilesLister fournit le code nécessaire pour cela. La fonction listFileToDeploy() utilise ici une méthode synchrone par simplicité, là où un programme plus conséquent devrait utiliser une approche asynchrone pour tirer au mieux parti du moteur JavaScript. La méthode publique par défaut listFileToDeploy() sera appelée ultérieurement et le résultat positionné dans le champ webfiles de la classe ProgrammezCloudAppStack.
(6) nous avons notre liste de fichiers et nous pouvons créer pour chacun d'entre eux un objet dans S3. Notons que nous précisons explicitement que chaque objet doit pouvoir être lu depuis internet sans être authentifié.

(10) nous instancions l'application cdktf, notre classe auxiliaire FilesLister puis notre classe principale ProgrammezCloudAppStack à qui nous fournissons la liste des fichiers qu'elle va transformer en objets et la référence à l'application à laquelle elle attachera ces instances de classe.
(11), l'instruction synth va transformer ces instances en instructions terraform puis en appels vers les services AWS. Cela conduit finalement à la création réelle des ressources.
En **(8),** nous avons demandé à cdktf de nous fournir à la fin de l'exécution, l'URL publique sur laquelle est disponible notre bucket. Cela sera l'URL de notre application Cloud Native !
Nous avons complété la souche de code qui a été généré par cdktf. Pour rappel, vous pouvez retrouver ce fichier et l'intégralité du code source de cet article dans le repository Github mbojoly/programmez-cloudapp. Nous pouvons désormais l'exécuter.

```
codespace:~/workspace/programmez-cloudapp/infra/terraform$ cdktf deploy
⠸ synthesizing ...
file index.html
Deploying Stack: programmezCloudAppStack
Resources
✓ AWS_S3_BUCKET programmezcloudapp aws_s3_bucket.
programmezCloudAppStack_programmezcloudapp_A6982D3D
✓ AWS_S3_BUCKET_OBJECT indexhtml aws_s3_bucket
_object.programmezCloudAppStack_indexhtml_8F92135E

Summary: 2 created, 0 updated, 0 destroyed.

Output: programmezCloudAppStack_WebsiteURL_28EE239E = terraform-20200903154627629000000001.s3-website.
eu-west-3.amazonaws.com
codespace:~/workspace/programmez-cloudapp/infra/terraform$
```

Et lorsque nous allons sur l'URL terraform-20200903154627629000000001.s3-website.eu-west-3.amazonaws.com (notez que l'URL change à chaque exécution) nous obtenons une première page web déployée sur le Cloud accessible internet ! **Figure 19**
En allant dans la console AWS nous pouvons voir les objets créés.
Afin de ne pas payer outre mesure, nous pouvons à tout moment supprimer l'ensemble des ressources créées via la commande suivante.

```
codespace:~/workspace/programmez-cloudapp/infra/terraform$ cdktf destroy
⠸ synthesizing ...
file index.html
Destroying Stack: programmezCloudAppStack
Resources
✓ AWS_S3_BUCKET programmezcloudapp aws_s3_bucket.programmezCloudAppStack_
programmezcloudapp_A6982D3D
✓ AWS_S3_BUCKET_OBJECT indexhtml aws_s3_bucket_object.programmezCloudApp
Stack_indexhtml_8F92135E

Summary: 2 destroyed.
```

Figure 19

En route pour une première application

La page que nous voyons est statique. Nous pouvons mieux faire, non ? Nous allons écrire pour cela une application Single Page Application avec Angular. Cette app sera exécutée côté navigateur ce qui nous permettra de la déployer exactement comme la page web précédente avec le seul service S3.
Nous allons pour cela installer le framework *Angular* de façon globale sur notre machine. Puis dans le répertoire hello-cloud-app nous allons initialiser une application Angular.

```
codespace:~/workspace/programmez-cloudapp/infra/terraform$ cd ~/workspace/programmez-cloudapp/
hello-cloud-app
codespace:~/workspace/programmez-cloudapp/hello-cloud-app$ npm install -g @angular/cli
...
codespace:~/workspace/programmez-cloudapp/hello-cloud-app$ ng new hello-cloud-app
...
```

Nous allons ensuite modifier le composant principal pour qu'il implémente ce qui pourrait être le Hello World d'une application de gestion de dette (ce qui est le cœur de mon métier) : le calcul d'un échéancier. Ce calcul consiste à déterminer pour un montant emprunté appelé le *nominal*, pour une durée d'emprunt appelée la *duration* et pour un taux d'intérêt fixe dans notre cas, des échéances mensuelles constituées d'intérêts à payer et de *nominal* à rembourser.
Angular organise l'application en composant. Nous allons écrire notre application dans le composant app. Nous modifions le fichier ~/workspace/programmez-cloudapp/ hello-cloud-app/src/app/app.component.html de la façon suivante.

```
<div>
 <h1>Hello Debt Management Cloud App</h1>
 <h2>Quick loan schedule</h2>
</div>

<div>
 <label for="nominal">Nominal (EUR):</label>
 <input type="number" placeholder="1000000" id="nominal" [(ngModel)]="nominal"
(blur)="clearModel()" /> <!-- 1 -->
 <label for="duration">Duration (months):</label>
 <input type="number" placeholder="12" id="duration" [(ngModel)]="monthsDuration"
(blur)="clearModel()" /> <!-- 1 -->
 <label for="interestRate">Interest rate (%):</label>
 <input type="number" placeholder="2.0" id="interestRate" [(ngModel)]=
"interestRate" (blur)="clearModel()" /> <!-- 1 -->
 <button (click)="renderSchedule()">Compute schedule</button>
</div>

<div>
 <table>
  <tr>
   <th>Month</th>
   <th>Type of Payment</th>
   <th>Payment</th>
  </tr>
  <tr *ngFor="let payment of schedulePayments; index as i"> <!-- 2 -->
   <td>{{payment.date.toDateString()}}</td> <!-- 3 -->
   <td>{{payment.type}}</td> <!-- 3 -->
```

```
    <td>{{payment.amount.toFixed(2)}}</td> <!-- 3 -->
  </tr>
 </table>
</div>

<router-outlet></router-outlet>
```

Angular utilise un mécanisme de templating qui se contente d'ajouter des attributs aux attributs HTML ce qui permet d'obtenir un fichier HTML complètement valide pour les outils d'éditions HTML.

En **(1)** nous déclarons les 3 champs permettant de saisir les caractéristiques de cet emprunt. Les champs sont de type *number* ce qui garantit qu'ils n'acceptent que des valeurs numériques. Le *placeholder* et le *label* donnent une première solution sommaire pour penser à l'ergonomie de notre application. [(ngModel)]="nominal" assure le mapping avec le composant Angular que nous allons présenter juste après. (blur)="clearModel()" indique à Angular d'appeler la méthode clearModel() du composant à chaque fois que le champ en question perd le focus. Cela nous sert à effacer l'échéancier dès lors qu'une des caractéristiques a pu changer.

Nous modifions ensuite le fichier ~/workspace/programmez-cloudapp/hello-cloud-app/src/app/app.component.ts qui contient la logique du composant de la façon suivante.

```
import { Component } from '@angular/core'; //1
import { Schedule, computeSchedule } from './schedule-engine.service' //2

@Component({ //3
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
 nominal:number; //4
 monthsDuration:number;
 interestRate:number;
 schedulePayments:Schedule[] = []

 clearModel(): void { //5
  this.schedulePayments = []
 }

 renderSchedule():void { //6
  this.schedulePayments = computeSchedule({
   nominal:this.nominal,
   monthsDuration:this.monthsDuration,
   interestRate:this.interestRate
  });
 }
}
```

L'architecture d'Angular repose sur un partage des responsabilités entre le fichier template HTML (appelé la vue) - qui se charge d'afficher et de mettre en forme l'interface utilisateur - et la classe TypeScript qui représente les données et contrôle les interactions (le composant qui peut aussi être appelé contrôleur). Pour plus de détails sur les termes en bleu des liens sont fournis sur le github mbojoly/programmez-cloudapp.

Notre composant ~/workspace/programmez-cloudapp/hello-cloud-app/src/app/app.component.ts est donc une classe TypeScript qui en **(3)** va déclarer, à l'aide d'une annotation du framework Angular importée en **(1)**, avec quels autres fichiers le framework Angular doit faire le lien à l'exécution. On reconnaît la vue app.component.html. L'autre fichier est une feuille de style CSS, une technologie incontournable sur le web qui présente comment la structure du HTML doit être rendue visuellement.

La classe définit en **(4)** les variables que le framework va remplir avec les valeurs des champs saisies et la variable schedulePayment qui contient le résultat de notre calcul. Le framework va utiliser cette variable pour remplir la boucle ngFor pour produire le tableau dans la vue HTML.

Lorsque les champs perdent le focus la méthode clearModel() en **(5)** est appelée et réinitialise schedulePayment ce qui vide le tableau.

En **(6)** la méthode renderSchedule() est appelée par le clic sur le bouton et déclenche un appel à la méthode computeSchedule() qui a été importée en **(2)**. Importée signifie qu'elle se trouve dans un autre fichier. Nous avons en effet isolé le code métier - ici le calcul financier - dans un fichier séparé qu'on appelle parfois le modèle. Le contenu de ce fichier ~/workspace/programmez-cloudapp/hello-cloud-app/src/app/schedule-engine.service.ts est le suivant.

```
export enum PaymentType { //1
 DRAWING = 'DRAWING',
 INTEREST = 'INTEREST',
 AMORTIZATION = 'AMORTIZATION'
}

export interface Schedule { //1
 date: Date
 type: PaymentType
 amount: number
}

export interface Deal { //1
 nominal:number;
 monthsDuration:number;
 interestRate:number;
}

function computeAmortization(nominal:number, monthsDuration:number) : number { //2
 return nominal / monthsDuration;
}

function computeInterestAmount(principal:number, interestRate:number, periodNbOf
Months: number) : number { //3
 return principal * interestRate/100 * periodNbOfMonths * 30/360
}

export function computeSchedule(deal: Deal): Schedule[] { //4
 console.log(` ComputeSchedule with nominal ${deal.nominal}, duration ${deal.months
Duration}, interest rate ${deal.interestRate}`)
 let schedule:Schedule[] = []
 const startDate:Date = new Date(Date.now())
```

```
  let principal:number = deal.nominal
  schedule = schedule.concat({
   "date": startDate,
   "type": PaymentType.DRAWING,
   "amount": -1*deal.nominal
  })
  for(let i:number = 1; i <= deal.monthsDuration; i++) {
   const paymentDate:Date = new Date(startDate.getFullYear(), startDate.getMonth()
+i, startDate.getDay())
   const interestAmount:number = computeInterestAmount(principal, deal.interestRate, 1)
   schedule = schedule.concat({
    "date": paymentDate,
    "type": PaymentType.INTEREST,
    "amount": interestAmount
   })
   const amortization:number = computeAmortization(deal.nominal, deal.monthsDuration)
   schedule = schedule.concat({
    "date": paymentDate,
    "type": PaymentType.AMORTIZATION,
    "amount": amortization
   })
   principal = principal - amortization
  }
  return schedule
}
```

En **(1)** les interfaces PaymentType, Schedule et Deal sont exportées ce qui signifie que ces structures de données seront partagées avec le composant app et serviront de contrat d'échange.
(2) et **(3)** constitue notre modèle financier.
En **(2)** nous calculons le montant de nominal à rembourser chaque mois appelé amortissement. La méthode utilise ici un montant constant ce qui aura pour conséquence que l'échéance à payer à chaque fin de mois sera variable du fait de la diminution progressive des intérêts.
En **(3)** nous calculons les intérêts à rembourser qui sont ici un pourcentage fixe du capital restant dû c'est-à-dire le *nominal* décrémenté des amortissements préalablement réalisés.
En **(4)** la fonction computeSchedule(...) est elle-même exportée et constitue le point d'entrée vers le contrôleur.
Notre application ne serait pas complète sans le fichier de feuille de style dont je parlais précédemment ~/workspace/programmez-cloudapp/hello-cloud-app/src /app/app.component.css.

```
table {
 border: 1px solid black;
}

th {
 border: 1px solid black;
}

td {
 border: 1px solid black;
}
```

Celui-ci ajoute simplement des bordures noires au tableau.
Nous pouvons désormais construire notre application, la déployer.

```
codespace:~/workspace/programmez-cloudapp$ cd hello-cloud-app/
codespace:~/workspace/programmez-cloudapp/hello-cloud-app$ npm install
# WARNING omis
removed 1 package and audited 1466 packages in 8.352s

61 packages are looking for funding
 run `npm fund` for details

found 0 vulnerabilities

codespace:~/workspace/programmez-cloudapp/hello-cloud-app$ npx ng build --prod

chunk {} runtime.e227d1a0e31cbccbf8ec.js (runtime) 1.45 kB [entry] [rendered]
chunk {1} main.2241593b93285996d4d0.js (main) 239 kB [initial] [rendered]
chunk {2} polyfills.a4021de53358bb0fec14.js (polyfills) 36.1 kB [initial] [rendered]
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
Date: 2020-09-06T14:53:00.394Z - Hash: 2d05ea6e090ec878bd56 - Time: 17542ms


codespace:~/workspace/programmez-cloudapp/hello-cloud-app$
```

Le répertoire ~/workspace/programmez-cloudapp/ hello-cloud-app/dist/hello-app contient tous les fichiers que nous devons déployer.

```
codespace:~/workspace/programmez-cloudapp/hello-cloud-app$ cd dist/hello-angular/
codespace:~/workspace/programmez-cloudapp/hello-cloud-app/dist/hello-angular$ ls
3rdpartylicenses.txt  index.html          polyfills.a4021de53358bb0fec14.js  styles.3ff695c00d717f2d2a11.css
favicon.ico           main.2241593b93285996d4d0.js  runtime.e227d1a0e31cbccbf8ec.js
codespace:~/workspace/programmez-cloudapp/hello-cloud-app/dist/hello-angular$
```

Nous éditons ~/workspace/programmez-cloudapp/ infra/terraform/main.ts en replaçant la valeur de DEPLOY_DIR par DEPLOY_DIR= '../../hello-cloud-app/dist/hello-angular/.
Tapons la commande cdktf deploy ou pour les plus pressés qui n'auraient pas déroulé tout l'article :

```
cdktf get
npm install
cdktf deploy
```

Après avoir confirmé par yes la console présente ce type de sortie :

```
codespace:~/workspace/programmez-cloudapp/hello-cloud-app/dist/hello-angular$ cd ../../..
codespace:~/workspace/programmez-cloudapp$ cd infra/terraform/
codespace:~/workspace/programmez-cloudapp/infra/terraform$ cdktf deploy
⁝ synthesizing ...
# Liste des resources omise
Summary: 8 created, 0 updated, 0 destroyed.

Output: programmezCloudAppStack_WebsiteURL_28EE239E = terraform-20200906145756204700000001
.s3-website.eu-west-3.amazonaws.com
```

Après une dernière confirmation par YES nous déployons notre application qui fonctionne désormais sur le web !
Figure 20
Retenons que nous pouvons grâce au Cloud développer notre première application Angular, la déployer sur S3. Nous avons enfin vu que des stacks d'infrastructure as code permettent de se passer entièrement d'actions manuelles en décrivant notre infrastructure comme du code qui est l'un des fondements du développement dans le Cloud.

Notre processus de développement et de déploiement reste manuel. Un outil d'usine de développement comme travis-ci permettrait d'automatiser le lancement de `npm` et de la CLI AWS. Pour cette dernière la *secret key* devra être stockée de façon sécurisée en utilisant dans le cas de Travis les clés de chiffrement. La détention des clés, leur chiffrement est un sujet central dans le Cloud.

Cela reste également un embryon d'application. Celle-ci n'est pas sécurisée, car nous utilisons le protocole HTTP et non HTTPS. Nous voudrions sans doute que les calculs de l'échéancier ne soient pas réalisés dans le navigateur, car n'importe qui peut modifier notre code. Nous aimerions probablement pouvoir également enregistrer des informations, fermer notre navigateur et pouvoir les retrouver. Nous voudrions que ces informations soient protégées ce qui signifie à minima que je suis le seul à pouvoir y accéder. Tous ces services sont disponibles sur le Cloud.

La prochaine étape serait de déployer du code côté serveur - expression quasi surannée aujourd'hui - c'est-à-dire du code qui s'exécuterait dans le service Cloud et non dans le navigateur. Dans une architecture Cloud Native ce code s'exécute dans un conteneur Docker comme celui que nous utilisons avec Visual Studio Code. Celui-ci est déployé dans Kubernetes en utilisant un service comme AWS EKS, Google GKE ou OVH Kubernetes. Il nous faudrait d'abord utiliser un framework pour écrire notre code serveur, framework qui exposerait notre code sous la forme d'API REST. Express en TypeScript ou Spring Boot avec le langage Java sont deux exemples parmi les plus populaires. Nous voudrions probablement utiliser un stockage en base de données documentaire comme MongoDB avec Atlas, un service d'authentification comme Auth0. Au lieu d'installer chacun de ces logiciels sur des serveurs nous pouvons désormais les configurer sur le Cloud comme nous l'avons fait avec le système stockage objet S3.

Mettre en place cette partie serveur de notre application mériterait à minima un article aussi étoffé que cette première partie. Et je préfère pour la suite de ce dossier prendre un peu de recul sur cette nouvelle plateforme de développement qu'est le Cloud.

Le Cloud : la plateforme de dév des 50 prochaines années

Aujourd'hui la question n'est plus, lorsqu'on est développeur, d'être développeur Windows ou développeur Linux. Elle est de savoir sur quel Cloud nous allons développer. En anglais une plateforme désigne un quai, quelque chose de plat, d'homogène, afin de permettre une standardisation et ainsi favoriser une activité industrielle. On peut imaginer une plateforme portuaire avec ses grues qui permettent d'automatiser le chargement et le déchargement des navires.

En informatique, les plateformes de développement désignent tous les outils et les frameworks (ou librairies) permettant de construire un logiciel. Au début de l'informatique, l'ordinateur, le matériel lui-même constituait la plateforme. Les premiers développeurs écrivaient en assembleur en s'appuyant sur les instructions du microprocesseur. Puis depuis les années 1970, les systèmes d'exploitation sont devenus les nouvelles plateformes de développement, abstrayant de plus en plus la plateforme matérielle. Aujourd'hui, en tant que développeur nous utilisons des compilateurs et des frameworks qui vont s'emboîter dans le système d'exploitation potentiellement à travers une ou plusieurs couches de machine virtuelle (VirtualBox, Xen, Java, moteur node V8, ...).

Le Cloud est de cette nature-là. Il nous fournit une nouvelle plateforme permettant de développer non plus des applications conçues pour fonctionner sur un seul ordinateur, mais des applications conçues pour utiliser un ensemble de ressources informatiques accessibles de façon programmable sur internet. Ce changement est si profond que je pense qu'il fondera l'informatique des 50 prochaines années. Alors quelles en seront ses caractéristiques ?

Tout d'abord il s'agit de services payables à la consommation. Contrairement aux machines traditionnelles qu'il fallait occuper au mieux, car on les payait en totalité, il va désormais falloir réussir à arrêter un serveur comme une application *Spring Boot* ou une application *node express* lorsqu'elle n'est plus utilisée. Il s'agit ensuite d'un service hébergé ce qui fait se poser la question de la disponibilité. Si les fournisseurs de Cloud aiment bien mettre en avant leur 10 ou 11 9 de disponibilité sur leurs services les plus centraux, il faut néanmoins savoir décrypter leur SLA (Service Level Agreement). Tout d'abord il faut garder en tête que les SLA sur le Cloud varient entre les services et même en fonction des options choisies au démarrage du service ! Tout est à la carte. Cela signifie que c'est nous, en tant que développeurs, qui sommes responsables du SLA de notre application par la combinaison des SLA sous-jacents. On est loin d'un SLA unique d'un ordinateur central. Mais en contrepartie, les disponibilités de la plupart des services offrent un niveau de disponibilité inégalable sans investissements colossaux. En effet, le moindre abonnement Cloud vous donne accès à plusieurs Data Center ce qui représenterait un investissement considérable à louer et encore plus à construire. Dernier cas, qui est arrivé même si c'est rare : vos services sont indisponibles. Si le déploiement que vous avez choisi respecte les bonnes options, vous allez être remboursé... des frais liés au Cloud, mais pas de votre perte d'activité. Cela implique d'autant plus notre responsabilité sur le SLA lorsque nous déployons sur le Cloud.

Pour rappel les services Cloud peuvent se décomposer en trois grandes catégories : les services d'Infrastructure As Code (IaaS) qui exposent du matériel (machine, réseau, virtuel), des services de Container As A Service (CaaS) qui constituent une Plateforme As A Service (PaaS) : le matériel est masqué et on orchestre des applications packagées dans des conteneurs Docker. Et des Software As A Service (SaaS) exposent directement des fonctionnalités logicielles. Une plateforme

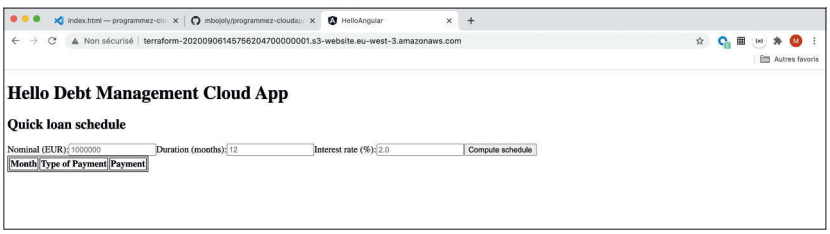


Figure 20

de contrôle de code source comme GitHub peut être dans cette catégorie.

Mais qu'ont-ils en commun ? Dans le Cloud, tout est API REST. JSON est lingua franca du Cloud. Le moteur d'orchestration des conteneurs Kubernetes est ainsi entièrement pilotable à travers des API REST. Et cette tendance se confirme chez l'ensemble des fournisseurs.

Sur les services les plus simples comme le stockage de données, le concept d'API REST est le standard de fait comme en témoigne la convergence des API pour les systèmes de stockage objet. L'URL est devenue le nouveau chemin d'accès au système de fichier et la requête HTTP le nouveau "mega" bloc du disque.

Pour le déploiement d'application serveur, la brique de base de la plateforme est en train de devenir le conteneur, en particulier le conteneur Docker. A l'image des grues d'une plateforme portuaire, la puissance d'une plateforme tient à sa standardisation. Les conteneurs du port ne peuvent être transportés de façon automatisée sur un navire, une grue, un camion, un train que parce qu'ils possèdent une taille standard. L'industrialisation ne peut pas fonctionner sans standard, sans contrainte.

Sur d'autres services les plus avancés (PaaS, SaaS, voir FaaS Fonctions As A Service) au contraire la guerre fait rage. Le Cloud n'échappe pas à la guerre des plateformes comme ont pu avoir lieu la guerre des matériels puis celle des systèmes d'exploitation qui ne sont pas compatibles entre eux. Les formats des machines virtuelles, des disques virtuels ou des Fonctions As A Service ne le sont pas plus. Ainsi, en tant que développeur, choisir la plateforme de Cloud sur laquelle on investit du temps de formation devient un élément différenciateur marquant.

Docker et Kubernetes ont tiré leur formidable succès de leur positionnement comme composants et plateformes portables. Docker a introduit la notion de conteneur qui impose une standardisation de fait dans le packaging des applications. Packager une application avec Docker permet ainsi de la déployer sur toutes les plateformes de Cloud. Kubernetes en tant qu'orchestrateur Open Source a également imposé une certaine standardisation dans l'orchestration de ces conteneurs, c'est-à-dire sa répartition sur les différentes ressources informatiques sous-jacentes au cloud. Un peu à l'image d'un noyau Linux ou d'un packaging JAR en Java, ils se sont imposés comme les compétences de base à maîtriser dans le développement cloud, car partagées entre tous les grands acteurs.

Mon application est-elle prête pour le Cloud ?

En tant que développeur, mon application est-elle prête à être déployée et à tirer parti du Cloud ? Vous l'avez sans doute entendu, le terme Cloud Native envahit les conférences et nombre d'articles. Chaque vendeur de plateforme ou de logiciel vante ses frameworks comme Spring Cloud Native ou, ses services propriétaires, qui permettent de tirer le meilleur parti de sa plateforme.

Chaque plateforme apporte son lot de contraintes sur les développements, car c'est aussi cela qui fait sa force. Le premier prérequis d'un déploiement sur le Cloud est de produire un binaire compatible x86 ou x86_64. Les processeurs RISC ou CICS antérieurs en sont exclus. Je ne parle pas ici des nouvelles architectures de processeurs ARM Neoverse apparues depuis quelques mois dans le Cloud et prévues dans le nouveau Mac, mais des architectures propriétaires IBM Z, SPARC, Itanium pour les principales. Je vous parle là d'un temps que les moins de vingt ans ne peuvent pas connaître, mais ces architectures permettent encore d'exécuter nombre d'applications d'actualité comme la commande des lignes 1 et 14 et de très nombreux systèmes bancaires. Mais rassurez-vous, si votre code s'exécute sur un PC et donc dans une machine virtuelle x86, votre application pourra s'exécuter dans le Cloud !

L'utilisation des services d'infrastructure vous permettra de mettre en œuvre des machines virtuelles ou IaaS (Infrastructure As A Service), de l'infrastructure as code pour l'automatisation et potentiellement d'utiliser certains middlewares proposés sous la forme de SaaS comme Amazon RDS pour les bases relationnelles comme MySQL et PostgreSQL, Cloud Database pour son équivalent OVH ou Amazon DocumentDB pour une base compatible Mongodb. **La première des compatibilités pour que votre application s'exécute dans un Cloud repose sur l'architecture processeur et les protocoles binaires des services.** Mais certaines applications et en particulier les applications Legacy ne respecteront pas ces exigences. Si elles peuvent être installées sur des machines virtuelles IaaS, il reste à notre charge toute l'installation, la mise à jour et la maintenance du système d'exploitation et des serveurs applicatifs. Il reste également à créer une architecture hautement disponible et scalable par de l'infrastructure as code. Le schéma page suivante montre ainsi un déploiement redondé très classique. Son coût d'exécution sur le Cloud sera probablement supérieur à celui d'un environnement classique, car dans le Cloud le coût horaire est souvent plus élevé. Le même schéma montre l'architecture Cloud Native composée de conteneurs exposés par différents services. Le Cloud est responsable de gérer l'orchestration des conteneurs qui idéalement seront éteints dès qu'ils ne seront plus utilisés.

Figure 21

Les fondements du Cloud sont de proposer des services disponibles et payables *à la demande* sur internet. **Le Cloud donnera tous ces fruits si votre application est déployable de façon totalement automatique et capable d'avoir des coûts qui suivent au plus près la demande. C'est en ce sens que je définirais une application Cloud Native.**

Idéalement sur le Cloud, vous devriez pouvoir fournir votre code source et obtenir une application totalement déployée, la gestion de l'infrastructure, du système d'exploitation et des différents middelware étant totalement automatisée. Idéalement sur le Cloud votre application ne devrait consommer des ressources, et donc être facturée, qu'aux instants où vous traitez des requêtes. Les services de Fonction As A Services sont les plus proches de cet idéal. Vous déployez le code de votre fonction et celle-ci n'est facturée que pendant les secondes où votre code s'exécute. Mais ils imposent également le plus de contraintes sur le déploiement de l'application : celle-ci doit fréquemment implémenter des interfaces et le temps de démarrage de votre applicatif est pénalisant, car à chaque

requête HTTP votre application peut avoir besoin d'être déployée, le fournisseur de Cloud ne la maintenant pas en mémoire quand vous ne payez pas. De plus, malgré les approches récentes comme OpenFaaS, ce monde également appelé serverless reste très propriétaire.

Les conteneurs docker couplés aux orchestrateurs comme Kubernetes ont remporté l'adhésion du marché. Le conteneur constitue une unité de packaging des applications standardisées et réutilisables avec un fonctionnement beaucoup plus traditionnel. L'application s'exécute dans des instances de conteneurs appelés Pods. L'orchestrateur Kubernetes agit comme un système d'exploitation de cluster et se charge de déployer ces pods sur les bonnes machines physiques afin d'assurer haute disponibilité et scalabilité. Comme le montre le schéma ci-dessous, le nombre d'opérations pour le provisioning d'infrastructure et le provisioning applicatif en est considérablement réduit. Ce mode de fonctionnement assure également de bien meilleurs échanges entre équipe Ops et équipe de développement avec un partage plus clair des responsabilités.

Alors en tant que développeur, est-ce qu'il suffit que mon application soit une application exécutable dans un navigateur pour le côté client et déployée dans un conteneur docker pour le côté serveur pour qu'elle soit Cloud Native ? C'est un bon début, mais pour que votre application puisse être réellement qualifiée de Cloud Native il est nécessaire qu'elle s'intègre au mieux dans cet écosystème. Dans un unix, tout est processus et fichier. **Dans un Cloud tout est service REST. Le protocole HTTP et le format JSON sont la lingua franca du Cloud**. Toutes les applications Cloud Native seront ainsi constituées de services ou de microservices exposant des API REST. Enfin, les 12 Factor App énoncés par la plateforme Heroku - l'un des clouds avant l'heure - constituent une très bonne base de bonnes pratiques pour développer des applications Cloud Native.

Présenter ces 12 facteurs serait trop long pour cet article : je me contenterais d'en illustrer deux. Le facteur le plus fondamental à mon sens consiste à **exécuter exclusivement des processus sans état**. En effet, le Cloud repose sur une facturation à la demande - à la seconde d'exécution dans bien des cas - et sur des unités d'exécution les plus petites possible. En effet, exécuter plusieurs petites unités d'exécution (conteneur ou machine virtuelle : on parle souvent de nœud) est plus flexible et parfois moins cher qu'exécuter un nœud avec beaucoup de CPU et de mémoire. Afin de déployer plusieurs nœuds et de facturer au plus près de l'utilisation, il est nécessaire de pouvoir arrêter ces unités (les pods dans le vocabulaire Kubernetes) et de les redémarrer à tout moment. Deux requêtes HTTP successives d'un même utilisateur doivent pouvoir être envoyées vers un autre pod de façon totalement transparente. Le traitement de chaque requête doit donc être totalement indépendant de celui de la précédente pour pouvoir être traité sur n'importe quel nœud. Dit autrement, aucune information utile au traitement de la requête, aucun état ne doit être nécessaire sur le nœud. Le second des 12 facteurs auquel je souhaite faire référence concerne **les processus d'administration : ceux-ci doivent désormais être déployés avec l'application**, car ils seront exécutés automatiquement par les orchestrateurs de déploiement. Prenons l'exemple d'une application déployée dans Kubernetes : si des migrations de données doivent être gérées, l'application se chargera de vérifier si la version de la base de données est compatible avec la version exécutée avec un outil comme flyway. Durant cette migration, l'application sera probablement incapable de répondre à tout ou partie des requêtes SQL. L'application exposera donc des **API de monitoring** permettant à l'orchestrateur de connaître l'état de chaque nœud et ainsi de rediriger les requêtes vers les nœuds pertinents. Ces API de monitoring deviennent ainsi des standards de fait entre l'orchestrateur et les applications. Dans notre exemple le readiness state sera en erreur indiquant à l'orchestrateur de ne pas envoyer de requêtes sur ce nœud, mais le liveness state sera positif indiquant à l'orchestrateur qu'il n'est pas nécessaire de redémarrer ce nœud. Les processus de déploiement, l'infrastructure as code font partie de votre application comme nous l'avons vu dans notre exemple. **Une application dépourvue d'infrastructure as code ne peut être qualifiée de Cloud Native.** En tant que développeur, sans être un expert des outils DevOps, il est nécessaire de maîtriser ces mécanismes afin de bien communiquer avec l'équipe Ops, d'exposer les bonnes API de monitoring pour rendre possible un déploiement et un maintien en condition opérationnelle programmable qui sont les promesses du Cloud.

Pour conclure, une application compatible x86, déployée dans un conteneur et exposant des API REST constituera les bases d'une application Cloud. Plus elle pourra être déployée de façon automatique et rapide, plus les instances pourront être arrêtées et redémarrées en fonction de vos besoins et plus votre application sera Cloud Native. Ces caractéristiques sont naturellement incluses dans les architectures modernes de type micro-service. Il est possible de rendre d'autres applications Cloud natives, mais, moins ces caractéristiques auront été prises en compte lors de la conception, plus il sera difficile de les implémenter. Les plateformes FaaS ou Serverless et plus généralement les services PaaS vont encore un cran plus loin en facturant uniquement à l'exécution, mais au prix d'une dépendance forte vers une technologie propriétaire

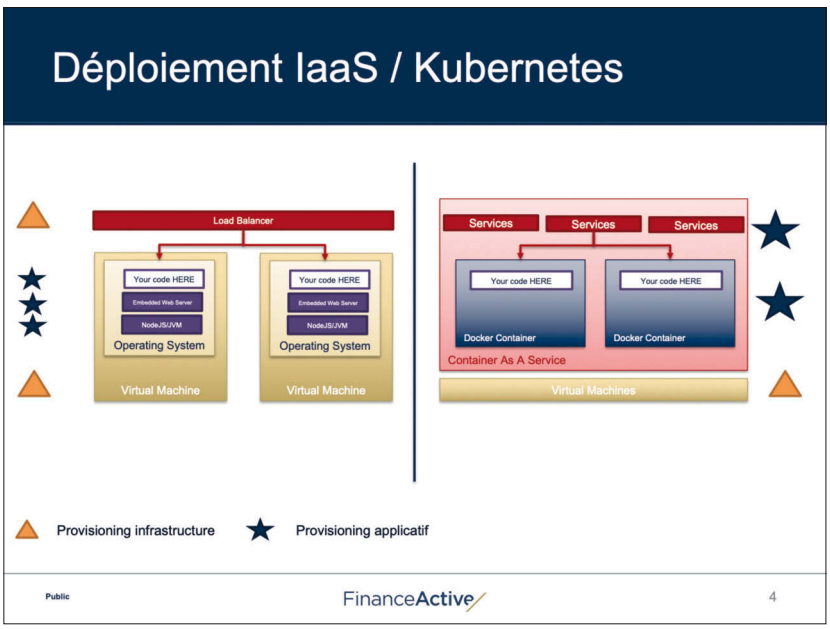


Figure 21

au fournisseur de Cloud et de contraintes que tous les usages ne permettent pas. Plus votre usage est ponctuel et asynchrone, basé sur une technologie moderne, plus le Cloud portera ses fruits. Les lourds traitements d'IA en sont un parfait exemple. Pour des usages plus interactifs, il est impératif de pouvoir en arrêter l'usage quelques heures (par exemple la nuit) pour en espérer facilement un gain significatif. Les services bureautiques sont un exemple de cet autre bout de la chaîne avec l'arrivée de quelques entreprises qui n'installent plus Microsoft Office. Mais pour les développeurs, les postes de développement complètement dans le Cloud comme Visual Studio Code, n'en sont qu'à leur balbutiement. Ces IDE n'ont pas encore toutes les fonctionnalités d'un IDE en local (le débogage est ainsi très limité) et ils évoluent très rapidement. A l'heure de mettre sous presse, la plateforme Visual Studio Code Online est d'ailleurs en train de migrer sur Github. Elles peuvent être étudiées aujourd'hui comme des technologies à surveiller afin de savoir si le Cloud s'étendra aussi demain sur ces usages. Ainsi aujourd'hui quelques usages (portage d'anciennes applications, applications s'exécutant en permanence) sont clairement difficiles à porter sur le Cloud.

Conclusion

Alors avec le développement dans le Cloud les autres types de plateformes vont-elles disparaître ? Non : le temps de porter certaines fonctionnalités est long, voire infiniment long. Prenons l'exemple des systèmes de fichiers bloc. Les systèmes de fichiers objet ou systèmes de stockage objet les ont dépassés sur le Cloud, car leur architecture leur permet d'être scalable contrairement au système de fichier par bloc. Néanmoins les temps d'accès ne sont pas des mêmes ordres de grandeur : un bloc de fichier local sera accessible en quelques millisecondes là où le passage sur HTTP et la granularité de l'objet consommera au moins dix fois plus pour un système de stockage objet. Je pense ainsi que notre système de fichier va progressivement se transformer en un nouveau niveau de cache. C'est par exemple ce que je comprends de l'architecture de Snowflake lorsque Thierry Cruanes explique à QCon que les données sont chargées progressivement du stockage objet (...) vers le SSD et la mémoire.

Le Cloud fait l'objet de beaucoup de promesses et d'enthousiasme. Néanmoins, il est important de bien comprendre les impacts du déploiement de nos applications et de nos données sur ce type de plateforme. Certains outils comme le chiffrement permettent de se protéger et l'innovation se déplace sur ce terrain avec des travaux sur le chiffrement homéomorphique pour qu'idéalement un jour les données soient déchiffrées le moins longtemps possible. Les formules de calcul de prix sont également très complexes. Outre la facturation au temps d'exécution, on trouvera quasi systématiquement une facturation sur les flux réseau. Or ceux-ci sont très difficiles à évaluer. De façon générale, si vous disposez d'un serveur occupé plus de 80 % du temps à plus de 80 % le Cloud sera assurément un surcoût. Les gains dans le Cloud sont majoritairement à rechercher pour des applications qui consomment des ressources pendant une faible portion du temps. Dans quelques cas, le choix du Cloud pourra donc ne pas être pertinent.

Comment se préparer à développer dans le Cloud dans les cas où cela est pertinent ? Fondons nos applications sur des API REST et des principes Cloud Natif (stateless, avec un démarrage et un arrêt rapide pour ne citer que deux principes), investissons fortement sur les conteneurs, et Kubernetes tout en ne négligeant pas de maîtriser en profondeur au moins une plateforme de Cloud parmi les plus grandes (AWS, Google, Azure, OVH et Outscale en France voire Alibaba en Chine ?). En effet, en informatique la standardisation n'arrive jamais à couvrir immédiatement 100 % des besoins. Mettre en œuvre un développement sur le Cloud nécessitera aussi de savoir manipuler certaines technologies propriétaires.

Bon voyage dans les nuages !

Implémenter un double facteur d'authentification par SMS

Les mots de passe ne suffisent plus à protéger efficacement, quelles que soient leur longueur ou complexité. Plus le mot de passe est utilisé/demandé souvent, plus il perd sa capacité de « blocage ». Le changement systématique et trop régulier implique que l'utilisateur se crée des méthodes simplificatrices qui peuvent facilement être découvertes. Une autre manière d'éviter le problème du mot de passe est de ne plus l'utiliser, par exemple en définissant un code PIN (lié au matériel) ou en utilisant une clé de type FIDO2.

Thierry Deman

Architecte sénior chez Exakis Nelite, un partenaire Gold de Microsoft SSII avec plus de 35 ans d'expérience en IT. Par ailleurs, il est régulièrement nommé MVP par Microsoft depuis 18 ans sur les technologies Exchange et Office Apps&Services. Il est occasionnellement speaker sur certaines conférences comme « Identity Days » ou « Rebuild ». Il a publié des ouvrages pour ENI Editions et des articles sur différents supports, ainsi que de répondre sur les forums Microsoft.

Sachant que les mots de passe ne sont plus une sécurité suffisante, on ajoute un facteur d'authentification, dont ne disposera a priori pas le hacker. Ceci empêchera les accès non souhaités à la messagerie ou à toute information qui permettraient au pirate d'accéder à d'autres données sensibles, puis de rebondir vers d'autres comptes ou d'autres sites.

Par ailleurs, toutes les tentatives d'utilisation et d'accès à partir d'emplacements, d'horaires et de périphériques inhabituels vont lever des alertes qui seront détectées et transmises à l'utilisateur et/ou aux administrateurs.

Actuellement, il y a encore trop d'entreprises présentant sur Internet des authentifications « basiques » basées sur la simple authentification Windows ou ADFS plus ou moins bien configurés, permettant ainsi aux hackers de réaliser de nombreuses tentatives de recherche de mot de passe, qui au mieux, ne font que verrouiller les comptes correspondants.

Les méthodes d'authentification

Il existe de nombreuses méthodes proposées par défaut pour vérifier l'utilisateur.

L'URL https://mysignins.microsoft.com/ permet d'accéder directement aux différents moyens définis par l'utilisateur.

DÉFINITION 2FA/MFA, L'AUTHENTIFICATION MULTIPLE

Pour décrire cette implémentation, nous utiliserons Azure AD /Office 365 comme référence principale. L'authentification multifacteur (MFA), aussi appelée vérification en 2 étapes, est un système de sécurité qui nécessite plus d'une méthode d'authentification provenant de plusieurs types d'informations permettant de vérifier l'identité de l'utilisateur au moment d'une connexion ou d'une opération particulière.

L'authentification multifacteur suppose donc au moins deux moyens d'identification ou plus : il s'agit généralement d'un élément que l'utilisateur connaît, son mot de passe, et d'un élément unique dont dispose l'utilisateur, un appareil identifié, une carte/badge, voire un de ses paramètres biométriques (empreinte, œil, visage…).

Source : https://www.lemagit.fr/definition/Authentification-multiple

Ces différentes méthodes peuvent être utilisées pour la connexion ou la réinitialisation du mot de passe, ou renforcer l'accès à une application ou une donnée sensible. **Figure 1**

L'utilisateur peut accéder à son profil et à ses appareils à partir de l'URL : https://myprofile.microsoft.com/

En revanche, c'est l'administrateur qui définit les méthodes d'authentification possibles ou indispensables en fonction des situations. Néanmoins, il est préférable de proposer plusieurs méthodes alternatives d'authentification à l'utilisateur si l'on veut éviter tout blocage ou appel au support.

Voici les options de vérification que l'administrateur peut proposer aux utilisateurs : **Figure 2**

L'appel téléphonique vérifie simplement la possession du téléphone, par une réponse sur le terminal et la validation par le code #. Le SMS envoie un code à 6 chiffres qu'il faut indiquer dans l'interface. La notification via l'application mobile (Microsoft Authenticator) demande une approbation ou un refus si vous n'êtes pas à l'origine de la demande. Le code de vérification demande la saisie de la valeur générée dans l'application mobile.

Quelques préconisations

Le numéro de téléphone doit être intégré au préalable dans le compte de l'utilisateur. La configuration doit être faite bien

Figure 2

Figure 1

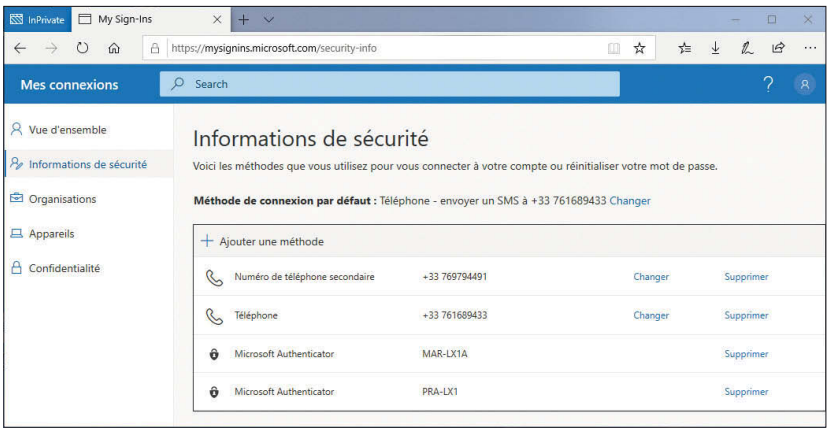


Figure 3

Figure 4

Figure 5

Figure 6

Figure 7

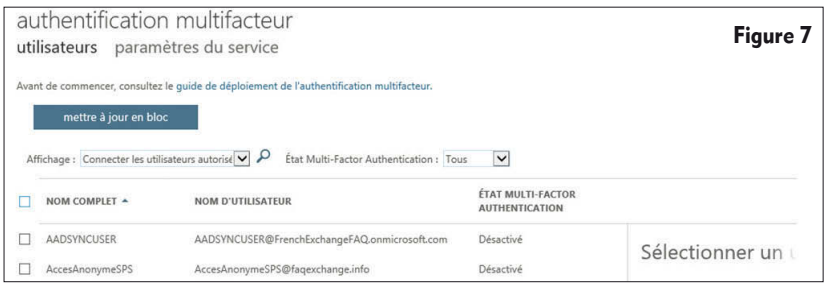


Figure 8

avant la première utilisation. Il est très important de tester chaque besoin et application sur un groupe d'utilisateurs pilotes, afin de vérifier du bon fonctionnement, avant d'appliquer cette nouvelle configuration à la totalité des utilisateurs. Certaines applications, généralement anciennes, ne sont pas compatibles avec l'authentification moderne et le MFA. La génération d'un mot de passe dit applicatif sera nécessaire pour chaque application.

Exemple d'implémentation de l'authentification par SMS sur Office 365

Configuration du MFA

À partir du portail principal d'administration d'Office, sélectionner la gestion des utilisateurs actifs : **Figure 3**

Cliquer sur le choix « **Authentification multifacteur** » proposé dans le menu. **Figure 4**

Cliquer sur « Paramètres du service » **Figure 5 et 6**

À ce niveau, vous pouvez sélectionner la méthode souhaitée, en l'occurrence le mode « SMS par téléphone » puis cliquer sur « enregistrer ».

Les autres choix à ce niveau peuvent alléger les contraintes de l'utilisation du MFA, mais aussi diminuer la sécurité :

- Proposer la mémorisation pendant 30 jours
- Ne pas demander l'authentification MFA à partir des réseaux connus de l'entreprise
- Autoriser les mots de passe applicatifs pour des applications qui ne supporteraient pas l'authentification en 2 étapes.

Après avoir configuré le service, il faut ensuite appliquer cette configuration sur les utilisateurs.

Activation du MFA sur le ou les utilisateurs choisis

Il faut revenir sur le menu « Utilisateurs » pour accéder à la configuration MFA spécifique de chaque utilisateur. **Figure 7**

L'activation peut se faire facilement à partir de l'interface pour les rôles d'administrateur ou pour quelques utilisateurs. En revanche, il faudra choisir l'option « Mise à jour en bloc » si l'on veut modifier un nombre important et précis d'utilisateurs.

Format du fichier à utiliser pour activer ou désactiver le MFA par utilisateur :

Username	MFA Status
tdeman@faqexchange.info	Enabled
nbasset@faqexchange.info	Disabled

Attention à bien garder le séparateur virgule (,) pour que le fichier soit pris en compte.

Après indication du fichier à utiliser, le fichier est validé (par un nombre d'enregistrements) et sera appliqué si l'on clique sur la flèche droite. **Figure 8**

La méthode manuelle consiste à sélectionner un ou plusieurs utilisateurs (que l'on peut rechercher). **Figure 9**

Puis cliquer sur « activer ». Cette activation permet à l'utilisateur de préparer son profil pour le MFA et de saisir les informations nécessaires (téléphones, Application mobile…), ces éléments n'étant validés qu'après vérification. **Figure 10**

L'utilisateur doit avoir l'information qu'il DOIT s'inscrire pour cette authentification. **Figure 11**

Voici un exemple d'authentification par SMS demandée après la saisie du mot de passe classique.

L'application du MFA **Figure 12**

L'application du MFA va forcer l'utilisation de mots de passe d'applications pour celles qui ne sont pas compatibles. **Figure 13**

La communication sur le changement

Il est important que l'utilisateur prépare et comprenne les éléments qui lui seront demandés afin de savoir ce qui est normal ou pas. Certains éléments pouvant ressembler à des tentatives de phishing. Une période de temps est nécessaire entre l'activation du MFA et la mise en place des règles correspondantes afin que tous les utilisateurs aient eu le temps de configurer et mettre à jour leur profil.
À noter qu'il est possible de bloquer et débloquer le MFA d'un utilisateur, ou de l'exclure temporairement dans les conditions d'accès en cas d'urgence, par exemple en cas de vol ou perte du téléphone.

Quelques points d'attention

L'authentification moderne doit être activée. L'option SMS pour le MFA n'est pas considérée comme la plus sécurisée, mais elle est considérée comme un minimum. En effet, elle permet déjà d'éliminer 99% des tentatives de piratage les plus classiques. (Application « SMS Forwarder » vers mail ou autre téléphone). Microsoft préconise l'utilisation de l'application « Microsoft Authenticator », pour éviter les coûts liés aux appels et SMS, notamment lors de déplacements à l'étranger.
Cette configuration suppose que chaque utilisateur dispose bien d'un téléphone mobile, et que celui-ci soit bien « protégé ». Autorise-t-on l'utilisateur à prendre son téléphone personnel ?
Attention : les facteurs de type « biométrique » ne sont pas non plus infaillibles, voire facilement accessibles et reproductibles.

Quelques liens intéressants

https://docs.microsoft.com/en-us/azure/active-directory/authentication/concept-mfa-howitworks
https://docs.microsoft.com/en-us/azure/active-directory/user-help/security-info-setup-signin
https://blog.piservices.fr/post/2016/08/18/Activer-lAuthentification-Multi-Facteur-(MFA)-dans-Office-365
https://docs.microsoft.com/fr-fr/azure/active-directory/user-help/multi-factor-authentication-end-user-troubleshoot
L'activation par défaut du MFA par Microsoft : https://docs.microsoft.com/fr-fr/office365/admin/security-and-compliance/set-up-multi-factor-authentication?view=o365-worldwi

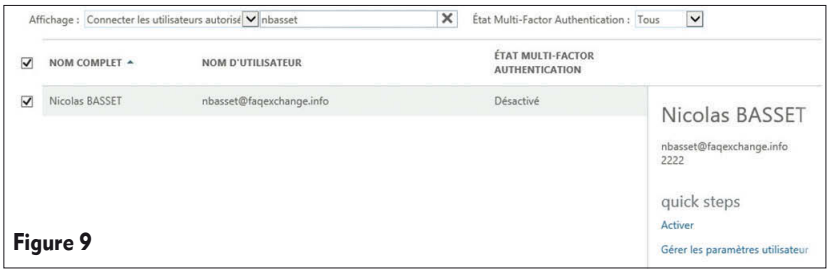


Figure 9

Figure 10

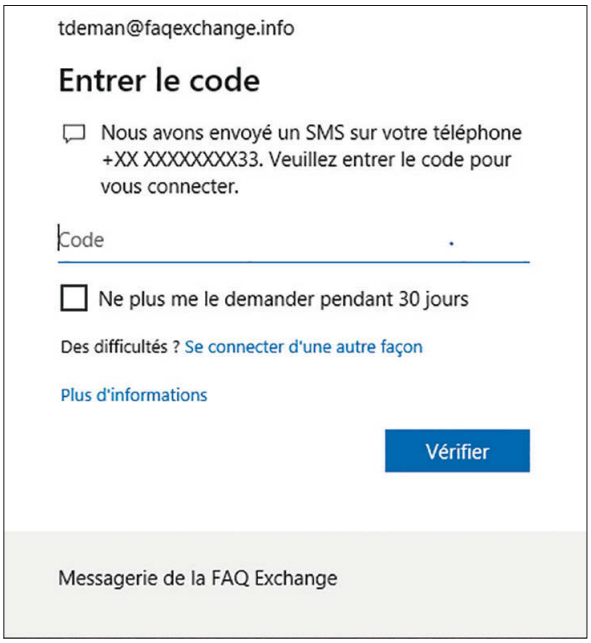


Figure 11

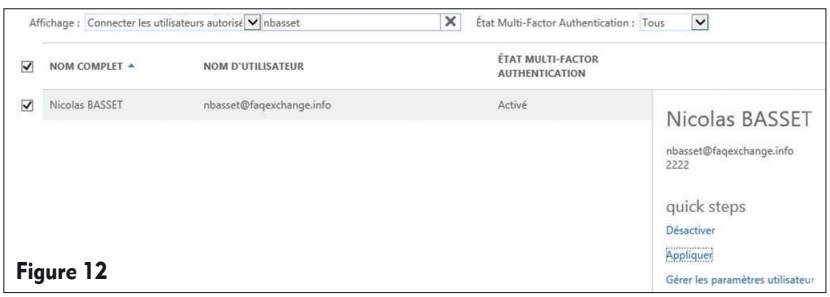


Figure 12

Figure 13

Boutique Programmez!

Les anciens numéros de PROGRAMMEZ!

Le magazine des développeurs

226

236

238

239

240

241

HS 01 été 2020

242

243

Tarif unitaire 6,5 € (frais postaux inclus)

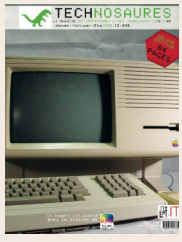

N°1


N°2


N°3

TECHNOSAURES

Le magazine à remonter le temps !



N°4 Standard 10 €

N°4 Deluxe 15 €

Prix unitaire :

7,66 €

(frais postaux inclus)

Histoire de la micro-informatique 1973-2007

Volume 1

12,99 €

(frais postaux inclus)

☐ 226 : ___ ex ☐ 241 : ___ ex
☐ 236 : ___ ex ☐ HS 01 : ___ ex
☐ 238 : ___ ex ☐ 242 : ___ ex
☐ 239 : ___ ex ☐ 243 : ___ ex
☐ 240 : ___ 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 : ___

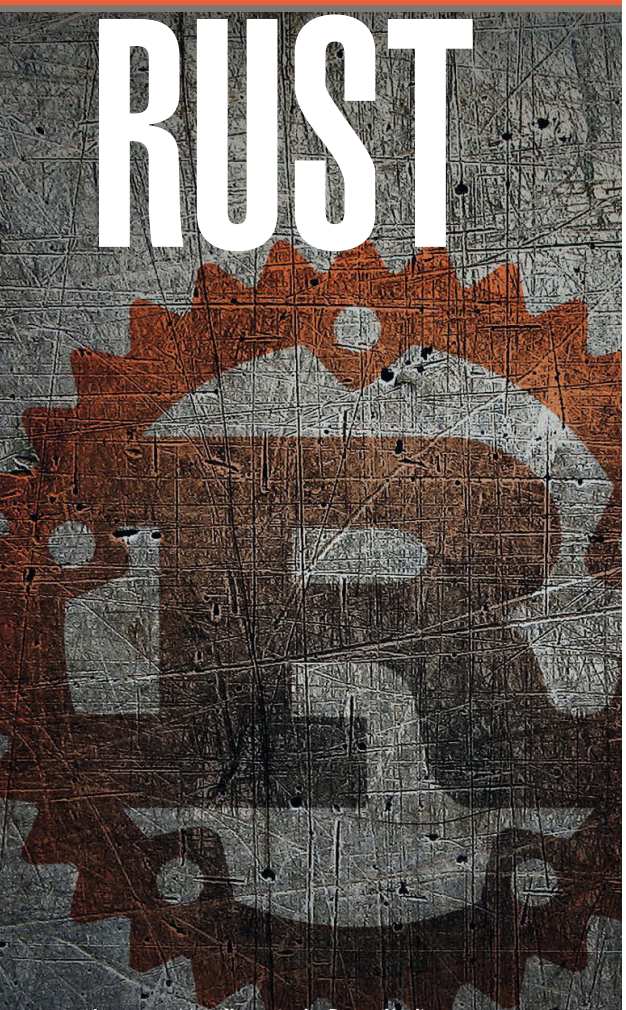
Adresse : ___

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 244

DOSSIER RUST



Lorsque Mozilla crée le Rust, l'éditeur ne se doutait pas que ce langage allait connaître une belle reconnaissance. Le succès n'arriva pas immédiatement. Mais année après année, Rust s'est embelli et stabilisé. Ce processus prend du temps, nous l'avons vu récemment avec Kotlin et Swift. Une petite inquiétude s'est faite jour quand Mozilla a licencié une partie des équipes, le projet a réagi rapidement pour calmer la communauté et rassurer.

Aujourd'hui, Rust est évoqué ouvertement par Amazon Web Services et Microsoft. Plusieurs projets sont en cours et ce n'est qu'un début. En août dernier, un des développeurs de Qemu / KVM avait même évoqué la possibilité de recoder ces outils en Rust. Pour lui, Rust apporte une sécurité, une robustesse et une gestion mémoire de très haut niveau. Mais il a aussi reconnu que migrer le code représenterait plusieurs années de travail.

Quoi qu'il en soit, Rust sera sans aucun doute une des vedettes de 2021. Et pour bien démarrer en Rust, nous vous proposons un dossier spécial. Bonne découverte.

Vous allez en pincer pour Rust

Depuis le début de ma carrière, il y a 20 ans environ, j'ai cette petite voix dans ma tête qui dit : "*Bon tu fais de l'informatique de gestion, au fond, tu utilises beaucoup trop de ressources pour faire des traitements assez simples et communiquer avec des bases de données. Mais la vitesse de développement de langages de haut niveau permet de répondre rapidement à des besoins et à leurs évolutions, tandis qu'un langage de plus bas niveau, plus 'dangereux' (pense à la gestion manuelle de la mémoire), serait plus lent à produire un logiciel stable et donc coûterait beaucoup plus cher en développement.*"

Jérôme Rouaix

Développeur indépendant depuis peu, architecte .Net et C# notamment pendant 6 ans chez BeezUP (gestionnaire de flux e-commerce), je suis, depuis 2 ans environ, obsédé par le potentiel de Rust.

(https://twitter.com/jrouaix/, eventuallyconsultant.com)

Cette petite voix fait des phrases beaucoup trop longues, mais elle a raison. Dans le monde de la programmation il y a ceux qui ont le choix : web, backend, clients lourds, BI, Big Data … Ceux-ci peuvent se permettre d'utiliser beaucoup de ressources ; d'avoir un garbage collector qui, une fois de temps en temps, va mettre le monde en pause ; de stocker des chaînes de caractères en utilisant 2 octets par caractère (.net) ; et par défaut d'utiliser beaucoup de références (pointeurs) vers le tas quand il serait plus efficace de stocker sur la pile. Et il y a ceux qui n'ont pas le choix, qui ont des contraintes telles qu'aucune négociation n'est possible sur les performances : système, embarqué, temps réel, jeu vidéo … Ceux-ci se servent un 4e café, s'équipent de discipline, et choisissent la seule arme disponible : C/C++.

C et C++ sont intrinsèquement enclins à l'erreur de gestion mémoire (dangling pointer, double free, null pointer, …). L'erreur étant humaine : selon Microsoft plus de 2/3 des vulnérabilités sont dues à ce genre de défauts. Ces problèmes sont adressés par Rust et disparaissent pratiquement avec son utilisation.

Sources :

https://thenewstack.io/microsoft-rust-is-the-industrys-best-chance-at-safe-systems-programming/

https://msrc-blog.microsoft.com/2019/11/07/using-rust-in-windows/

Sécurité, rapidité et concurrence

Ce sont les 3 promesses de Rust, mais un langage peut-il viser ces 3 objectifs ? Voyons cela :

- Memory safe : il est vraiment difficile en Rust d'écrire un bug de gestion de la mémoire, ça n'est pas impossible mais il vous faudra utiliser le mot clé `unsafe` qui, comme son nom l'indique est l'équivalent de l'escalade sans corde : c'est à vous qu'incombe alors la responsabilité de ne pas faire d'erreur.
- Zero cost abstraction : comme en C++, utiliser les fonctionnalités les plus évoluées du langage n'aura pas de surcoût, et compi-

lera in fine dans un binaire aussi optimisé que si vous l'aviez écrit sans sucre syntaxique.
- Fearless concurrency : le modèle de gestion de la mémoire élimine aussi les race conditions (une situation rendant un comportement non-déterministe selon la vitesse d'exécution de plusieurs traitements concurrents). Vous ne pourrez pas compiler un mauvais design sur ce plan-là, par conséquent vous éviterez des heures de débogage fastidieux.

Syntaxe

Un autre article de ce magazine aborde en détail la syntaxe Rust, cet article va donc faire une impasse dessus et assumer que le code Rust ressemble suffisamment à d'autre langages bien connus pour vous permettre de suivre… sinon lisez l'autre article en détail d'abord ;-).
Tous les exemples de code de cet article sont téléchargeables à l'adresse : https://github.com/jrouaix/rust-article/.

Rustc, la version 'syndrome de Stockholm' des compilateurs

Quand vous allez commencer à écrire du Rust, si vous avez de l'expérience dans d'autres langages plus permissifs, vous allez probablement rapidement traiter le compilateur de fasciste. Il n'en est rien, le compilateur Rust vous empêche de vous mettre à l'eau sans avoir mis votre gilet de sauvetage, fait vos vaccins, mis des chaussons de baignade au cas où il y aurait des trucs coupants au fond, c'est votre gardien, votre protecteur, il est très à cheval sur la sécurité, mais il a raison et il veut votre bien. Considérant cela, il y a heureusement dans la communauté une grande volonté d'accompagner la montée en compétence et l'expérience de programmation avec Rust, et les messages d'erreur du compilateur sont souvent d'une aide précieuse. Quand vous allez commencer à développer en Rust, vous allez vous manger beaucoup d'erreurs de compilation (miam miam). Plus vite vous apprendrez à dialoguer avec le compilateur et ses suggestions, de manière itérative, interactive presque, plus vite vous allez progresser. Ecrivez quelques lignes, compilez, lisez les erreurs s'il y en a, corrigez, recommencez.
Ci-dessous, je ne me souviens pas quel trait implémenter pour avoir la possibilité de comparer des variables de type `Card` avec l'opérateur `<`. Qu'à cela ne tienne, j'écris ce que je veux `Ace(Club) < King(Club)`, je tente un build, et le compilateur me met sur la voie. **Figure 1**
Autre exemple ici lorsque qu'il me donne la liste des branches manquantes d'un pattern matching. Le compilateur me suggère les matchs manquants. **Figure 2**

Rust est un langage très bien outillé. Nous aborderons quelques outils standards vers la fin de l'article (cargo, clippy, rust-analyser, rustfmt, cargo-watch). Pour une meilleure expérience et pour une meilleure progression, utilisez-les.

La gestion de la mémoire en Rust

Rentrons maintenant dans le vif du sujet, ce qui donne à Rust ses caractéristiques les plus uniques, j'ai nommé, dans la langue de Shakespeare, le "borrow checker" (vérificateur d'emprunts). Il vous permet d'avoir une gestion de la mémoire automatique SANS garbage collector. Il est exécuté à la compilation et implante donc de manière statique et déterministe les endroits dans votre programme où sera libérée la mémoire. Il vérifie pour vous les 3 règles suivantes :

```
error[E0369]: binary operation `<` cannot be applied to type `Card`
  --> examples\typesystem.rs:25:30
   |
25 |         assert_eq!(Ace(Club) < King(Club), false)
   |                    --------- ^ ---------- Card
   |                    |
   |                    Card
   |
   = note: an implementation of `std::cmp::PartialOrd` might be missing for `Card`
```

Figure 1

```
error[E0004]: non-exhaustive patterns: `(&King(_), _)`, `(&Queen(_), _)`, `(&Jack(_), _)` and 1 more not covered
  --> examples\typesystem.rs:58:28
   |
58 |         let result = match (self, other) {
   |                            ^^^^^^^^^^^^ patterns `(&King(_), _)`, `(&Queen(_), _)`, `(&Jack(_), _)` and 1 more not covered
   |
   = help: ensure that all possible cases are being handled, possibly by adding wildcards or more match arms
   = note: the matched value is of type `(&Card, &Card)`
```

Figure 2

- Une valeur est possédée par une variable qui en est propriétaire.
- Il ne peut y avoir qu'un seul propriétaire à la fois.
- Quand le propriétaire sort du scope, la mémoire est libérée (drop).

```
// Cette fonction ne compile pas car la valeur est `moved`
// de s1 vers s2, il n'est donc plus possible d'utiliser s1
fn a_value_owned_by_2_variables() {
  let s1 = String::from("hello");
  // -- move occurs because `s1` has type `std::string::String`,
  // which does not implement the `Copy` trait
  let s2 = s1;
  // -- value moved here
  println!("{}, world!", s1);
  // ^^ value borrowed here after move
}
```

Bon c'est bien mignon tout ça, mais comment fait-on pour passer une valeur à une autre fonction, juste un petit moment, sans en perdre la propriété ? On la prête ; d'où le nom : borrow checker. Il est possible de créer une ou des références à la valeur. Ces références sont à comprendre comme des prêts de valeurs mais dans ce cas de nouvelles règles s'appliquent :

- On peut avoir plusieurs références non modifiables.
- On peut avoir une seule référence modifiable.
- On ne peut pas avoir de référence modifiable et non modifiable en même temps.
- Aucune référence ne peut survivre à sa variable propriétaire.

```
fn main() {
  let mut s = String::from("hello Rust");
```

```
  // un emprunt modifiable dont le scope ..
  mutation(&mut s); // .. se termine ici

  // ou plusieurs emprunt non modifiable
  let a = &s;
  let b = &s;
  print(a, b);
}

fn mutation(s: &mut String) {
  s.push_str("!");
}

fn print(a: &String, b: &String) {
  dbg!(a, b);
}
```

Cette gestion des emprunts corrige même certains problèmes que vous pouvez rencontrer dans d'autres langages. En C# par exemple, il n'est pas possible de modifier une collection en même temps qu'on la parcourt. Mais il est possible de l'écrire et même de le compiler, ce qui aura pour conséquence une exception à l'exécution. **Figure 3**

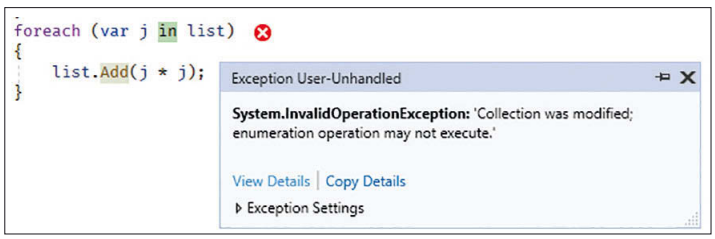


Figure 3

```
error[E0502]: cannot borrow `list` as mutable because it is also borrowed as immutable
  --> examples\borrowchecker.rs:51:9
   |
49 |     for j in &list {
   |              -----
   |              |
   |              immutable borrow occurs here
   |              immutable borrow later used here
50 |         // pas possible ici parce qu'un emprunt est déjà en cours
51 |         list.push(j * j);
   |         ^^^^^^^^^^^^^^^^ mutable borrow occurs here
```

Figure 4

```
root@DESKTOP-7HBM3F4: /home/jrx
root@DESKTOP-7HBM3F4:/home/jrx# sudo apt install cargo -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
cargo is already the newest version (0.44.1-0ubuntu1~20.04.1).
0 upgraded, 0 newly installed, 0 to remove and 103 not upgraded.
root@DESKTOP-7HBM3F4:/home/jrx# cargo new hello
     Created binary (application) `hello` package
root@DESKTOP-7HBM3F4:/home/jrx# cd hello
root@DESKTOP-7HBM3F4:/home/jrx/hello# cargo build
   Compiling hello v0.1.0 (/home/jrx/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.60s
root@DESKTOP-7HBM3F4:/home/jrx/hello# cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/hello`
Hello, world!
root@DESKTOP-7HBM3F4:/home/jrx/hello# cargo version
cargo 1.43.0
root@DESKTOP-7HBM3F4:/home/jrx/hello# cd ..
```

Figure 5

En Rust, parcourir un vecteur nécessite au minimum un emprunt immutable, le modifier nécessite un emprunt mutable : les deux ne sont pas possibles simultanément. Le compilateur va vous stopper net.

```
fn borrow_checker_prevent_from_change_while_iterate() {
  let mut list = vec![1, 2, 3, 4];
  list.push(5);
  for i in 5..10 {
    // Aucun problème ici
    list.push(i);
  }
  // Ici on emprunte la liste pour la parcourir
  for j in &list {
    // Pas possible ici parce qu'un emprunt
    // est déjà en cours
    list.push(j * j);
  }
}
```

Figure 4

A travers ces quelques exemples, vous commencez à entrevoir ce qui rend ce langage unique. Vous commencez aussi à comprendre que cette nouvelle gymnastique mentale n'a rien de trivial, qu'elle va nécessiter un apprentissage. Pour les plus affûtés d'entre vous, vous avez déjà des questions sur la manière de faire retourner une référence par une fonction, de partager des valeurs entre threads… Bienvenue ! Dernier détail, vous voulez voir l'implémentation de la fonction `drop()` de la librairie standard qui permet de libérer la mémoire utilisée par une variable sans attendre que celle-ci ne sorte du scope ?

```
pub fn drop<T>(_x: T) {}
```

C'est tout ? Oui. La fonction `drop()` prend la propriété d'un paramètre qui sera donc détruit à la sortie. Le compilateur s'assurera bien sûr que la fonction appelante ne puisse plus utiliser la variable passée à `drop()`. CQFD.

Se mettre à l'eau

Vous pouvez `jouer` facilement dans le bac à sable Rust : https://play.rust-lang.org/ (très pratique pour partager des exemples). Mais si vous voulez ruster local, l'outil recommandé pour installer la suite d'outils Rust s'appelle rustup. Vous pourrez le trouver ici : https://rustup.rs/.

Vous pouvez aussi faire confiance à votre distribution Linux préférée, mais Rust publiant une nouvelle version toutes les 6 semaines, les mainteneurs de votre distribution auront probablement une paire de versions de retard. **Figure 5**

Donc si vous voulez suivre le train des releases, utilisez rustup, il vous permettra de vous mettre à jour facilement, ainsi que de switcher rapidement entre les versions `stable`, `beta` et `nightly` des outils.

Votre premier projet Rust (pour le créer, exécutez `cargo new <nom_du_projet>`) est un simple répertoire avec un fichier `cargo.toml` et un fichier `main.rs` dans un dossier `src`. Nous allons revenir sur l'utilité du fichier `cargo.toml`. **Figure 6**

Cargo

Une fois Rust installé, cargo est la commande suivante à connaître. C'est LE gestionnaire de paquets (appelés crates,

disponibles sur https://crates.io). Il télécharge, compile, teste, benchmark ... Il permet aussi d'installer d'autres outils et ainsi d'enrichir vos lignes de commandes.
C'est pour cargo qu'existe le fichier de manifeste `cargo.toml`. Ce fichier décrit votre package (application ou bibliothèque), ainsi que ses dépendances. C'est donc dans ce fichier que vous devrez ajouter des lignes lorsqu'il s'agira d'utiliser des `crates` pour aller facilement au-delà de ce que la librairie standard `std` propose. **Figure 7**
Vous pourrez aussi par ce fichier, ajouter des dépendances uniquement pour les tests, définir des profils de compilation, et bien plus.

Genèse influences, paradigmes

Rust est un langage relativement nouveau. Depuis ses débuts, il a subi des influences et a beaucoup changé jusqu'à se cristalliser dans sa version 1 en 2015. Le langage continua d'évoluer à travers la notion d'éditions, notamment l'édition 2018 qui apporta en particulier `async` et `await` comme mot clés réservés, ouvrant la voie aux syntaxes permettant de simplifier grandement la programmation asynchrone, et qui se sont stabilisées fin 2019.
Parmi ses principales influences, citons d'abord C++. Rust s'est beaucoup construit comme un remplaçant à C++ (projet Servo de Mozilla pour arriver à paralléliser une part du rendu dans Firefox). Peut-on parler ici d'une "anti-influence" ? Il ne faut pas jeter le crabe avec l'eau de cuisson. Rust reprend de C++ l'exigence de "0 cost abstraction", les smart pointers, RAII (l'acquisition d'une ressource par l'initialisation d'un objet, et sa libération par destruction de ce même objet), les références et la sémantique `move`.
Rust reprend d'OCaml les types algébriques (enum en Rust), le pattern matching, l'inférence de type ; les attributs du C# ainsi que les mots clés async/await ; les closures Ruby ; et enfin citons Haskell pour les typeclasses (trait en Rust). Disons encore qu'il y a les génériques, que le concept de référence nulle y est inexistant, et que les variables sont immuables par défaut. Pour plus d'exhaustivité :
https://doc.rust-lang.org/reference/influences.html

Rust objet

Il n'y a pas d'héritage en Rust comme on pourrait le trouver en Java ou .Net mais le polymorphisme est possible par les Traits. Un Trait est ce qui se rapproche le plus d'une interface dans d'autres langages bien connus. Vous pourrez implémenter pour vos types des Traits venant d'autres packages, mais vous pourrez aussi implémenter vos Traits sur des types venant d'autres packages.

Laissez moi vous présenter un Trait "bruyant" :

```
trait Noisy {
  fn make_a_noise(&self) -> String;
}
```

Nous allons l'implémenter pour les canards, les enfants, et les notes de musique :

```
struct Duck {}

struct Child {
  age: u8,
}

#[rustfmt::skip]
#[derive(Debug)]
enum MusicNotes{
  A,B,C,D,E,F,G
}

// Et les implémentations

impl Noisy for Duck {
  fn make_a_noise(&self) -> String {
    String::from("Couac couac")
  }
}

impl Noisy for Child {
  fn make_a_noise(&self) -> String {
    match self.age {
      0..=1 => String::from("Ouiiinnnnnnnn!!!!"),
      2..=4 => String::from("Maaaamaaaan!!!"), //parfois papa
      5..=12 => String::from("Aïeaïeaïe!"),
      _ => String::from("I'm too old for this sh.t."),
    }
  }
}

impl Noisy for MusicNotes {
  fn make_a_noise(&self) -> String {
    format!("{:?}", self)
  }
}
```

Maintenant implémentons une fonction qui affiche le bruit de n'importe quel Noisy et utilisons la :

```
fn print_a_noise<T>(noisy: &T)
```

```
C:\_git
λ cargo new hello
     Created binary (application) `hello` package

C:\_git
λ tree hello /f
Folder PATH listing
Volume serial number is FE86-1645
C:\_GIT\HELLO
    .gitignore
    Cargo.toml
└───src
        main.rs
```

Figure 6

Cargo.toml

```
[package]
name = "rustintro"
version = "0.1.0"
authors = ["Jérome Rx <jerome@eventuallyconsultant.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
tide = "0.13.0"
async-std = { version = "1.6.2", features = ["attributes"] }
itertools = "0.9.0"
rayon = "1.3.1"
```

Figure 7

```
where
  T: Noisy,
{
  println!("{}", noisy.make_a_noise());
}

fn main() {
  let baby = Child { age: 2 };
  let duck = Duck {};
  let note = MusicNotes::A;

  print_a_noise(&baby);
  print_a_noise(&duck);
  print_a_noise(&note);
}
```

`print_a_noise` est une fonction générique, c'est à dire qu'elle est utilisable avec des paramètres de différents types, du moment qu'ils vérifient la contrainte d'être `Noisy`. Pour chacun des différents types en paramètre de la fonction générique, le compilateur va créer une version de cette fonction utilisant exactement le type final. Ici notre binaire une fois compilé portera 3 versions finales de `print_a_noise`. Les fonctions génériques sont très utiles pour réutiliser du code et elles n'ont aucun coût supplémentaire à l'exécution, mais elles ont une limite : le type final utilisé doit être connu au moment de la compilation.
Si cela n'est pas possible, par exemple si l'on a une liste de `Noisy` indifférenciés de tous types, il faudra faire appel au boxing. Le boxing est un mécanisme signifiant qu'une structure de données qui pourrait être stockée sur la pile, va être déplacée dans le tas (faisant une allocation mémoire au passage), pour ne stocker finalement que son adresse sur la pile. L'objet peut avoir une taille variable, son adresse aura toujours la même taille, c'est pourquoi en ne manipulant que des adresses, les tailles de structures sont connues et de nouvelles choses sont possibles comme faire des tableaux d'objets de types différents (Abus de langage, en réalité ce sont des pointeurs vers des objets de types différents).
Voici comment créer une liste d'objets de différents types partageant un même trait :

```
  let noisy_boxes: Vec<Box<dyn Noisy>> =
    vec![Box::new(baby), Box::new(duck), Box:
:new(note)];
```

La syntaxe `dyn Trait` s'appelle un Trait Object, c'est la manière en Rust de définir un polymorphisme d'héritage (Attention il n'y a pas d'héritage en Rust, seulement ce mécanisme de Trait). Cette syntaxe permet de changer, à l'exécution, quelle implémentation réelle sera utilisée. La variable `noisy_boxes` est donc un vecteur (une liste, une collection, un set ...) de Box (la structure chargée d'allouer/de déplacer ce qu'elle encapsule) de trait objects implémentant `Noisy`. Voici maintenant comment implémenter une fonction `print_a_noise` qui fera la même chose de manière dynamique :

```
fn print_a_noise_from_box(noisy: &Box<dyn Noisy>) {
  println!("{}", noisy.make_a_noise());
}
```

Et voici la fonction main complète faisant le tour de la question :

```
fn main() {
  let baby = Child { age: 2 };
  let duck = Duck {};
  let note = MusicNotes::C;

  print_a_noise(&baby);
  print_a_noise(&duck);
  print_a_noise(&note);

  let noisy_boxes: Vec<Box<dyn Noisy>> =
    vec![Box::new(baby), Box::new(duck), Box::new(note)];

  for noisy in noisy_boxes {
    print_a_noise_from_box(&noisy);
  }
}
```

Nous avons vu ici deux manières d'écrire la même chose alors que dans d'autres langages nous ne nous serions pas posé autant de questions. C'est nul !! Pourquoi ?? Parce que non, nous n'avons pas implémenté 2 fois la même chose. La première utilise les génériques et n'a aucun impact de performance à l'exécution. Elle sera donc à préférer. Rust reste ici fidèle à ses promesses : vous ne payez que ce que vous utilisez que ce soit en termes de facilité d'écriture ou de performances. La seconde fait appel au boxing (et donc à de l'allocation mémoire) et à un mécanisme de `dynamic dispatch` puisqu'il faudra "router" les appels vers la bonne implémentation de la fonction `make_a_noise` au moment de l'exécution. Ces différences rendent la deuxième version moins performante que la première. Cette distinction serait complètement ignorée dans bien d'autres langages mais Rust la rend explicite.

Rust fonctionnel

Rust n'est pas un langage purement fonctionnel tel qu'il serait défini par les standards de ce paradigme. Rust a des boucles (pas seulement des récursions), les variables sont mutables (pas par défaut mais quand même)… Mais Rust emprunte beaucoup aux langages fonctionnels. Les fonctions sont "citoyens de première classe" : regardez une méthode d'instance en Rust (donc un paradigme objet), c'est simplement une fonction qui prend `self` en premier paramètre. Tout est expression. Les traitements sont exprimés sous forme d'expressions qui peuvent être imbriquées à tout moment. C'est principalement pour contenir et séquencer explicitement les expressions que les "déclarations" sont utilisées. Les déclarations sont marquées par la terminaison en `;`.

Ci-dessous, un exemple de la différence entre déclarations (dans la fonction `main`) et expressions (dans la fonction `pure_expression_function`). Vous noterez que la fonction `pure_expression_function` ne contient tout simplement aucun point virgule, en effet, la dernière expression d'une fonction est automatiquement retournée.

```
fn main() {
  let result = pure_expression_function(1, 2);
```

```
  dbg!(result);
}

fn pure_expression_function(a: i8, b: i8) -> i8 {
  match a {
    0 => b + 1,
    1..=2 => pure_expression_function(3, 5),
    3 => {
      if b == 12 {
        b + a
      } else {
        a - b
      }
    }
    _ => panic!(),
  }
}
```

Dans le code suivant, vous pourrez voir un exemple de closure. Les closures sont des fonctions anonymes définies à la volée que vous pouvez stocker dans des variables, et qui peuvent capturer une partie de leur environnement.

```
fn main() {
  let name = String::from("closure");
  let closure = |s| format!("{} {}!", s, name);
  println!("{}", closure("Hello"));
  // println!("{}", closure(42));
  // error : expected `&str`, found integer
}
//> Hello closure!
```

Vous pouvez aussi noter dans le code ci-dessus l'inférence de type. En effet sur la deuxième ligne de la fonction `main()`, il n'y a aucun moyen de savoir à priori que le paramètre `s` de la closure sera "Hello", une chaîne de caractère, en 3ème ligne. C'est le premier usage de la closure qui va lui donner son type. Il serait ensuite possible de l'appeler plusieurs fois, toujours avec un paramètre de même type, mais il ne sera plus possible d'appeler la closure avec un paramètre d'un autre type. Ensuite voyons les itérateurs : le moyen de définir simplement de petits pipelines de traitement de séries d'éléments. Rust ne fait pas défaut sur ce point et permet comme bien d'autres langages de parcourir des listes simplement et avec une grande expressivité. Ci-dessous un exemple qui retourne en une ligne la somme des 3 premiers entiers supérieurs à 10 d'une liste donnée.

```
fn main() {
  let nums = vec![21, 2, 12834, 2, 12, 12847, 3, 5, 7];
  let sum: u32 =
    nums.into_iter().filter(|i| *i > 10).take(3).sum();
  dbg!(sum);
}
//> [examples\iterators.rs:7] sum = 12867
```

La gestion d'erreurs

Il n'y a pas d'exception en Rust, si une fonction peut générer une erreur, elle doit le déclarer dans son type de retour, ainsi les erreurs sont gérées par le système de types lui-même. Pour cela on utilise le type Result<T, E>, c'est un enum qui peut prendre une valeur `Ok` de type T ou `Err` de type E. L'opérateur `?` permet d'éviter de fastidieux tests dans les fonctions appelantes en récupérant facilement la valeur dans la variante Ok(), ou en retournant tout de suite l'erreur le cas échéant. Il y a aussi la possibilité de "faire planter" le thread (panic), mais vous voudrez sans doute l'éviter au maximum de sorte d'avoir le moins possible d'erreurs imprévues une fois en production ... Bien plus de détails dans la suite du dossier ;-).

Async / await

Dans ce chapitre nous allons implémenter une api rest qui utilisera Redis pour stocker et compter un nombre de visiteurs uniques (la question de visiteurs uniques sera ici implémentée de façon très naïve). Ce sera l'occasion de parler de programmation asynchrone.

La liste de crates que nous allons utiliser est la suivante :

- async-std : nous permet d'avoir une fonction `main` asynchrone et va fournir l'exécuteur de `Future`.
- tide : sera notre framework web pour implémenter l'api
- redis : sera notre client redis
- mobc : sera notre gestionnaire de pool de connections
- mobc-redis : fera le lien entre mobc et redis pour gérer un pool de connexions redis.

Voici les lignes à ajouter dans notre fichier `cargo.toml` :

```
[dependencies]
tide = "0.13.0"
async-std = { version = "1.6.3", features = ["attributes"] }
redis = { version =  "0.17.0", features=["async-std"] }
mobc-redis = "0.5.3"
mobc = { version = "0.5.12", features = ["async-std"] }
```

Vous pouvez observer ici un format d'import de dépendance un peu plus complexe et la notion de features dans les crates. Les features sont des conditions de compilation permettant d'activer ou désactiver des parties des crates et leurs dépendances. Ici, `tide` utilisant `async-std` comme exécuteur asynchrone, il est important de trouver des dépendances compatibles avec cet exécuteur, par défaut ou par features optionnelles.

Voici le code très commenté pour lancer cette api :

```
use mobc::Pool;
use mobc_redis::RedisConnectionManager;
use mobc_redis::{redis, Connection};
use tide::Request;
use tide::StatusCode;

#[derive(Clone)]
// implémentation automatique du trait Clone
// qui est nécessaire pour que tide puisse
// donner une copie de l'état de l'application
// à chaque requête
struct AppState {
   pool: Pool<RedisConnectionManager>,
}

#[async_std::main] // Expliqué dans le paragraphe suivant
async fn main() -> Result<(), std::io::Error> {
```

```
    let client =
        redis::Client::open("redis://127.0.0.1/").unwrap();
    let manager = RedisConnectionManager::new(client);
    // On crée ici un connection pool vers Redis
    let pool = Pool::builder()
        // On utilisera 5 connections concurrentes à Redis
        .max_open(5)
        // On évitera ici de produire un Timeout
        // à cause d'une attente sur Redis
        .get_timeout(None)
        .build(manager);

    // Ceci est l'état global de l'api,
    // on l'utilise pour stocker le connection pool
    // il sera cloné à chaque appel http
    let app_state = AppState { pool };

    // tide (le framework web utilisé ici)
    // est créé avec `app_state` pour état global,
    // Il a une route vers le handler `uniques`,
    // et écoutera sur le port 8080 de localhost
    let mut app = tide::with_state(app_state);
    app.at("/uniques/").get(uniques);
    app.at("/ping/").get(ping);
    app.listen("127.0.0.1:8080").await?;
    Ok(())
}
```

Avant de voir le code des handlers, parlons un peu des mots clés async/await et de l'asynchronisme en Rust. Parlons d'abord de la notion de Future, une promesse dans d'autres langages, une Task en dotnet. C'est une structure représentant une valeur ou un traitement qui n'est pas encore terminé. Cela peut être un téléchargement, la lecture d'un buffer, un traitement sur un autre thread, qu'importe. Il nous faut un moyen d'attendre le résultat et de synchroniser sur la fin de la Future, la continuation d'autres traitements (ou d'autres Futures). Les fonctions marquées `async` retournent en réalité des Future<de_leur_resultat>. Les mots-clés `await` dans une fonction `async` génèrent des instructions supplémentaires et un état interne à la fonction permettant la pause et la reprise de celle-ci. Contrairement à d'autres langages, Rust ne fournit pas d'exécuteur par défaut pour les Futures qui sont créées, vous devez fournir un exécuteur pour déjà démarrer son exécution puis la suivre et la rappeler lorsqu'elle signalera une avancée. Il en existe aujourd'hui plusieurs et toutes les librairies ne sont pas compatibles avec tous les exécuteurs, c'est un domaine dans lequel l'expérience de développement est encore un peu rêche en Rust.
La macro `async_std::main` qui précède notre fonction `main` nous permet d'être, dès le démarrage de notre programme, dans une Future, avec un exécuteur démarré. On peut alors considérer notre programme entier comme une grosse Future (et ses filles), et nous pouvons utiliser `await` sans nous soucier de l'exécuteur sous-jacent.
Maintenant voyons le code des handlers.

```
async fn uniques(
    req: Request<AppState>,
) -> tide::Result<String> {
    // ici on récupère un identificateur naïf de l'utilisateur
    let client_ip = req.peer_addr().unwrap_or_default();

    // La récupération d'une connection du pool peut générer
    // une erreur que nous allons gérer manuellement ici
    match req.state().pool.get().await {
        // si c'est une erreur
        Err(e) => {
            // on log le message
            println!("Could not get redis connection because : {}", e);

            // puis on retourne HTTP_500
            Err(tide::Error::from_str(
                StatusCode::InternalServerError,
                e,
            ))
        }
        // si on a bien récupéré la connection vers Redis
        Ok(mut con) => {
            // on l'utilise
            #[rustfmt::skip]
            let uniques : (u64,) = redis::pipe()
                    .cmd("pfadd").arg("visitors").arg(client_ip)
                    .ignore()
                    .cmd("pfcount").arg("visitors")
                .query_async(&mut con as &mut Connection)
                .await?;
            // ci dessus deux opérateurs se succèdent
            // - .await : pour attendre et récupérer le résultat
            //   de la Future retournée par `query_async`
            // - ?      : pour ensuite récupérer
            //   la valeur dans Ok(_)
            //   ou retourner directement Err(_)

            // puis on retourne HTTP_200
            // et un simple texte comme résultat
            Ok(format!(
                "There have been {} unique visitors",
                uniques.0 // la première valeur du tuple
            ))
        }
    }
}

// cette route va nous permettre de comparer les performances
// de cette fonction comparée à celle qui inclut l'appel
// a Redis
#[rustfmt::skip]
async fn ping(_: Request<AppState>) -> tide::Result<&'static str>
{ Ok("ok") }
```

Chatouillons un peu notre petit proxy vers Redis avec un test de charge, je démarre Redis dans docker et c'est parti. Mon test consiste ici en 500 utilisateurs concurrents, harcelant les 2 endpoints de l'api. Quelle que soit la longueur du test, la consommation mémoire de l'api culmine à 21 Mo. Sur la durée totale, l'api aura appelé Redis 1 380 fois par secondes (bravo Redis !), sans aucune erreur à reporter, avec un temps

de réponse inférieur à 44 ms pour ¾ des requêtes, tout en servant parallèlement 23 000 "Ok" par seconde sur la requête de ping. **Figure 8**

Macros

Depuis le début de cet article, si vous avez bien regardé les sources en exemple, il y a un petit détail qui pourrait vous avoir intrigué, auquel je n'ai pour l'instant pas donné l'importance qu'il mérite : pourquoi certaines fonctions sont nommées avec un point d'exclamation à la fin !! Réponse : ce ne sont pas des fonctions. Ce sont des macros, des appels permettant de réécrire l'expression que vous leur passez avant la compilation. Les différents types de macros en Rust ont leurs propres possibilités et limitations bien trop longues à détailler ici. Alors nous allons nous concentrer sur quelques exemples montrant leur plein potentiel.

```
// Version réellement écrite
#![feature(format_args_capture)]
fn main() {
  let vector = vec![1, 2, 3];
  let integer = 42;
  print!("vec = {vector:?}, integer = {integer}");
}
```

La fonction main a 3 lignes, chacune utilisant une macro :
déclaration d'un vecteur, avec les valeurs 1,2 et 3
déclaration d'un entier : 42
output de la représentation de Debug de ce vecteur (format `{:?}`), et de la représentation Display (format `{}`)

```
// Version réellement compilée
fn main() {
  let v = <[_]>::into_vec(box [1, 2, 3]);
  let integer = 42;
  ::std::io::_print(::core::fmt::Arguments::new_v1(
    &["vec = ", ", integer = "],
    &match (&v, &integer) {
      (arg0, arg1) => [
        ::core::fmt::ArgumentV1::new(
          arg0, ::core::fmt::Debug::fmt,
        ),
        ::core::fmt::ArgumentV1::new(
          arg1, ::core::fmt::Display::fmt,
        ),
      ],
    },
  ));
}
```

Ceci imprime le résultat suivant :

```
// vec = [1, 2, 3], integer = 42
```

C'est quand même bien plus concis à écrire (merci captain obvious). Mais les avantages à utiliser une macro ne sont pas seulement une question de factorisation/réutilisation du code, sinon une simple fonction aurait suffit. D'abord en Rust il n'y a aucune notion de paramètres optionnels ou de multiples signatures d'un même nom de fonction (Polymorphisme paramétrable). Hors la macro `print!` peut prendre un nombre dynamique de paramètres, en fonction du nombre de patterns `{}` présents dans la chaîne de format. Le nombre de ces paramètres est vérifié à la compilation. Dans l'exemple ci-dessus, je n'ai même pas utilisé de paramètres pour passer les valeurs puisque j'ai utilisé une fonctionnalité de capture des arguments (`format_args_capture`). On le voit dans l'exemple, la macro `print!` a été déployée en plusieurs lignes de code. Elle a littéralement écrit du code. Les patterns de formatage ont disparu et ont été transformés en utilisation directe de la fonction de formatage du trait Debug pour le vecteur, et de la fonction de formatage du trait Display pour l'entier. Il n'y aura donc aucun coût de lecture du format à l'exécution, c'est déjà fait. Un autre avantage découle de la réécriture : le type des paramètres demandés est vérifié. Le compilateur leur imposera des contraintes différentes, ici le vecteur devra implémenter Debug et Display pour l'entier. Il n'y a plus rien de dynamique à l'exécution, aucun comportement inattendu, si ça compile : ça marche.

Les macros permettent donc de générer du code à partir d'expressions et de n'importe quelle chaîne de caractère qui leur est passée, tout en laissant ensuite au compilateur le soin de vérifier que le code sera exécutable, c'est la possibilité offerte d'implémenter de véritables DSL (Domain Specific Language / Langage spécifique de domaine) embarqués dans votre source Rust.

La librairie SQLx (https://github.com/launchbadge/sqlx) possède justement une macro `query!` vérifiant à la compilation le bon usage du plus connu de tous les DSL : SQL. **Figure 9**

```
20:27:17 [ WARN] caught ctrl-c, stopping...
-------------------------------------------------------------------------------
 Name                    | # reqs          | # fails         | req/s  | fail/s
-------------------------------------------------------------------------------
 GET /ping/              | 8,258,035       | 0 (0%)          | 23,594 | 0
 GET /uniques/           | 483,178         | 0 (0%)          | 1,380  | 0
-------------------------+-----------------+-----------------+--------+--------
 Aggregated              | 8,741,213       | 0 (0%)          | 24,974 | 0
-------------------------------------------------------------------------------
 Name                    | Avg (ms)   | Min        | Max        | Median
-------------------------------------------------------------------------------
 GET /ping/              | 10         | 1          | 1361       | 9
 GET /uniques/           | 183        | 1          | 188413     | 14
-------------------------+------------+------------+------------+-------------
 Aggregated              | 19         | 1          | 188413     | 9
-------------------------------------------------------------------------------
 Slowest page load within specified percentile of requests (in ms):
-------------------------------------------------------------------------------
 Name                    | 50%    | 75%    | 98%    | 99%    | 99.9%  | 99.99%
-------------------------------------------------------------------------------
 GET /ping/              | 9      | 9      | 12     | 17     | 150    |    150
 GET /uniques/           | 14     | 44     | 1000   | 1000   | 4000   |   4000
-------------------------+--------+--------+--------+--------+--------+-------
 Aggregated              | 9      | 9      | 93     | 150    | 1000   |   2000
```

Figure 8

Compile-time verification

We can use the macro, `sqlx::query!` to achieve compile-time syntactic and semantic verification of the SQL, with an output to an anonymous record type where each SQL column is a Rust field (using raw identifiers where needed).

```
let countries = sqlx::query!(
        "
SELECT country, COUNT(*) as count
FROM users
GROUP BY country
WHERE organization = ?
        ",
        organization
    )
    .fetch_all(&pool) // -> Vec<{ country: String, count: i64 }>
    .await?;

// countries[0].country
// countries[0].count
```

Figure 9

Dans la même veine, Yew(https://github.com/yewstack/yew/) possède une macro `html!`. Et, là encore, impossible de compiler du html invalide. **Figure 10**

On pourra aussi citer serde_json et sa macro json!(). Cette macro transforme la syntaxe json en un code bien plus verbeux de création d'objet json ce qui lui permet d'abord de vérifier la validité de la syntaxe passée en paramètre, mais aussi de vérifier que toutes les variables passées dans l'expression implémentent les Traits nécessaires à leur utilisation dans un json. Encore une fois on en revient aux promesses de Rust : une grande partie du travail est faite à la compilation pour garantir fiabilité et rapidité à l'exécution. **Figure 11**

Regardez ensuite le code suivant :

```
#[derive(Debug)]
struct Test {
  i: u32,
  s: String,
}
```

`derive` est un attribut particulier : c'est une macro. Ici la macro va générer le code nécessaire à l'implémentation du trait Debug. Dans de nombreux langages, un système d'attributs similaire permet d'ajouter des comportements à des objets en se basant sur de la réflexion, un mécanisme lourd à l'exécution et susceptible de générer de nouvelles erreurs. Ici, pas de réflexion, pas de lenteur, pas de problème.
Ci-dessous, voici ce qui est réellement compilé une fois que la macro `derive` s'est déployée.

```
struct Test {
  i: u32,
  s: String,
}
#[automatically_derived]
#[allow(unused_qualifications)]
impl ::core::fmt::Debug for Test {
  fn fmt(
    &self,
    f: &mut ::core::fmt::Formatter,
  ) -> ::core::fmt::Result {
    match *self {
      Test {
        i: ref __self_0_0,
        s: ref __self_0_1,
      } => {
        let mut debug_trait_builder =
          f.debug_struct("Test");
        let _ = debug_trait_builder
          .field("i", &&(*__self_0_0));
        let _ = debug_trait_builder
          .field("s", &&(*__self_0_1));
        debug_trait_builder.finish()
      }
    }
  }
}
```

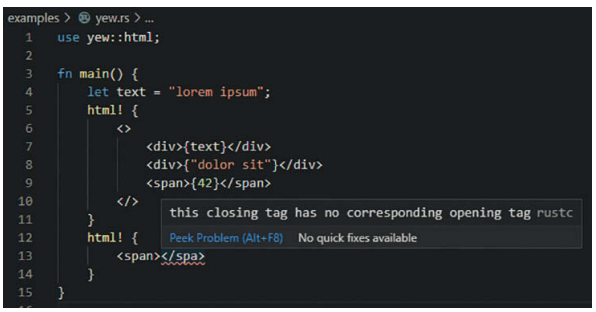


Figure 10

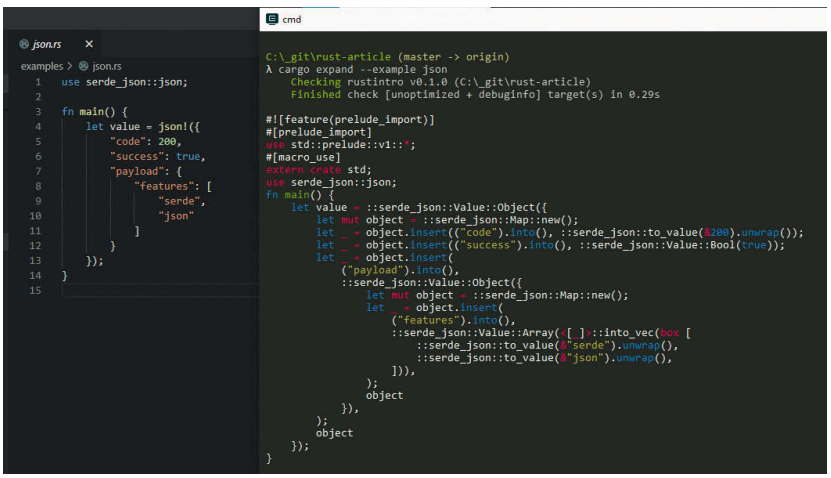


Figure 11

Les outils

La documentation

Avoir une documentation c'est bien, les commentaires de code en Rust utilisent un formatage markdown. Avoir des exemples de code dans la documentation, c'est encore mieux ! Mais en cas d'évolution du code, il faut penser à maintenir la documentation ! Eh bien vous ne pourrez plus jamais oublier. L'outil `rustdoc` et sa commande `--test` exécutent les blocs de code dans les commentaires. Il suffit donc d'y ajouter des assertions pour s'assurer que les exemples de codes ne deviennent jamais obsolètes. Et cela motive à en mettre. Résultat, on note un certain effort à documenter les crates avec des exemples de la part de la communauté et cela tombe bien car pour chaque version de chaque crate hébergée sur https://crates.io/, la documentation est générée et hébergée sur https://docs.rs.

RLS et rust-analyzer

Rust Language Server (RLS pour les intimes) est un `language server`. Une technologie proposée par Microsoft pour Visual Studio Code permettant d'offrir un backend supportant le `Language Server Protocol - LSP` pour un langage spécifique et ainsi de mieux supporter ce même langage sur n'importe quel IDE. C'est par exemple par ce biais que VSCode supporte TypeScript, Javascript et C#. Et c'est donc tout naturellement que la team Rust a implémenté un serveur LSP : Rust Language Server. C'est aussi RLS qui est utilisé par VIM, et donc l'expérience Rust est uniforme quel que soit votre éditeur de code.
RLS n'est que la première version du support pour IDE. Une nouvelle initiative nommée rust-analyser est déjà à l'œuvre. RLS utilise le compilateur, tandis que rust-analyser s'imprègne dans le compilateur. Rust-analyser travaille à la modu-

larisation du compilateur pour qu'il devienne incrémental et plus "à la demande", un effort qui permettra à terme de réduire les ressources demandées à la machine du développeur pour introspecter le code pendant sa rédaction.

Rustfmt

Rust**fmt**, comme son nom l'indique, est un formateur de code, en l'intégrant à votre IDE et/ou votre CI, vous garantissez un style de code homogène et standard à votre projet.

```
> rustup component add rustfm
```

Clippy

Vous vous rappelez de l'assistant Microsoft Office qui ressemblait à un trombone ? C'est en son hommage qu'a été nommé le linter officiel de Rust (honnêtement, je ne sais pas ce qui leur a pris). Clippy propose aujourd'hui 350 règles, comme tous les autres outils vous pouvez l'intégrer à votre CI et il a même une commande `--fix` pour appliquer ses suggestions à votre place. Si vous débutez, installez-le tout de suite, il va vous aider à écrire un code plus `idiomatique`, c'est-à-dire plus proche des standards de style de la communauté.

```
> rustup component add clippy
```

RIIR - REWRITE IT IN RUST

Rust étant au départ un langage système (ou marketé comme tel), vous ne serez pas étonné du nombre d'expérimentations de réécriture d'outils bien connus de la ligne de commande bash :

- **ripgrep** : une alternative a *grep*, plus rapide
- **bat** : alternative a *cat*, mais supportant la coloration syntaxique de nombreux langages
- **procs** : un *ps* multi platform, avec de la couleur
- **fd** : plus rapide que *find*
- **gitx** (gitoxide) : (work in progress) ré-implémentation de *git* en pure rust, plus ergonomique et aussi rapide et bien d'autres encore.

Il n'est pas question de tout réécrire en Rust, mais la forte compatibilité entre rust et C/C++ en fait un candidat de choix pour remplacer/expérimenter ce langage dans certains modules bien choisis d'une plus grosse base de code.

Ressources & communautés

Un langage aimé

"Most loved langage" de la communauté Stackoverflow pour la 4ème année consécutive, beaucoup de ceux qui ont goûté à Rust y ont pris goût. Pourquoi ? Pour la faire courte, parce que Rust résout de nombreux problèmes présents dans bien d'autres langages. En poursuivant la sécurité et la rapidité, en permettant d'être très bas niveau avec un langage de haut niveau, en ayant débarqué directement avec une suite d'outils impressionnante, et en ayant implémenté très rapidement un code de conduite sain autorisant tout un chacun à contribuer dans la bienveillance, Rust a agrégé une communauté avec un background hétérogène dont il bénéficie aujourd'hui. Ses caractéristiques font que même si sa courbe d'apprentissage est vraiment plus raide que celle de la plupart des langages, ce n'est rapidement plus un problème pour qui se lance dans un projet d'envergure puisque l'investissement de départ est "remboursé" par la qualité de ce qui est produit, et la réduction des coûts d'infrastructure et de maintenance qui en découle.

Tout n'est pas rose, on pourrait parler des temps de compilation qui sont un poil long, de l'immaturité de certaines librairies (quand d'autres crates sont littéralement au top : regex, actix, nom ...), de l'incompatibilité des exécuteurs de `Futures` (à la base de async/await) et de la difficulté intrinsèque que constitue l'apprentissage de nouveaux concepts tels que le borrow checker, les lifetimes (dont nous n'avons pas parlé), les différents types de chaînes de caractère. Les points douloureux sont connus. De brillantes personnes se penchent dessus mais certaines difficultés resteront probablement inéluctables : un grand pouvoir implique de grandes responsabilités ... et un peu d'effort.

Quelques ressources

Le livre officiel pour bien débuter avec Rust :
https://doc.rust-lang.org/book/
La newsletter indispensable pour se tenir au courant :
https://this-week-in-rust.org/
Autres ressources :
https://livebook.manning.com/book/rust-in-action/
https://fasterthanli.me/articles/a-half-hour-to-learn-rust
https://www.ssi.gouv.fr/guide/regles-de-programmation-pour-le-developpement-dapplications-securisees-en-rust/

De l'embarqué au webassembly

Le grand écart à la Van Damme, est-ce que ce ne serait pas de pouvoir utiliser la même librairie de sérialisation json sur un device IoT embarquant quelques kilos de RAM, jusqu'au navigateur en webassembly, en passant par une api web, des fonctions as a service et des plugins PostgreSQL ou Redis. C'est en substance ce que propose Rust.

D'un côté nous avons l'attribut `#![no_std]` qui indique qu'une librairie va pouvoir tourner en utilisant un sous ensemble de la `std` appelé `core`, allégé des abstractions et possibilités d'appel vers le système d'exploitation. De nombreuses librairies sont déjà compatibles `no_std` comme `serde` et `serde_json` (ser/de : serialisation/deserialisation). Sous quelques contraintes il est possible de les utiliser complètement ou partiellement, notamment pour déployer votre programme sur de l'embarqué (c'est a dire n'ayant pas de système d'exploitation). Si vous avez besoin de creuser encore un peu plus bas niveau, le mot-clé `unsafe` et la macro `asm!` vous permettront de toucher au matériel et à la mémoire au plus près. https://docs.rust-embedded.org/book/

D'un autre côté, il y a webassembly, la techno qui promet de révolutionner les performances et possibilités des clients web aussi bien que de l'hébergement en proposant une plateforme cible unifiée. Alors oui ça a déjà été imaginé avant (Java : Write once, run anywhere), mais disons seulement que cette fois c'est vraiment agnostique du langage puisque c'est "seulement" un set d'instructions que les compilateurs de tous poils vont cibler.

Rust a une longueur d'avance sur cette technologie et propose dès aujourd'hui tout ce dont vous aurez besoin pour compiler vers webassembly et générer le code permettant

d'interagir depuis et vers javascript ou typescript. A la différence d'autres langages, le runtime minimal de Rust lui permet de réduire la taille du fichier .wasm à envoyer au client, et donc le temps du premier chargement de la page.
https://rustwasm.github.io/
https://rustwasm.github.io/docs/book/
Que ce soit en embarqué ou sur webassembly, bien sûr, le langage reste le même, performances et borrow checker inclus : pas de garbage collector, pas d'erreur de gestion mémoire, de concurrence, ou de types.

Un langage ennuyeux ?

Certains des plus ardents défenseurs du langage Rust le qualifient de "boring technology". C'est très loin d'être une insulte, il faut même le prendre comme un compliment. En effet nous l'avons vu, il a une faculté assez unique à détecter les problèmes au moment de la compilation. C'est à dire que le plus souvent, vous allez détecter les problèmes quelques secondes ou minutes après l'avoir écrit, vous permettant plus souvent qu'avec d'autres langages, d'éviter de pousser du code nocif. Une fois que le compilateur Rust vous dit que c'est OK, vous avez plus de chance qu'avec d'autres langages d'avoir produit quelque chose qui n'a pas de bug. C'est pourquoi une fois en production, les applications Rust ont tendance à se faire oublier. Elles consomment peu de mémoire, peu de CPU, et sont stables.

Figure 12

Rust est parti pour rester en version 1 longtemps. Quand bien même une nouvelle version du compilateur, de la librairie standard et des outils est publiée toutes les 6 semaines, ce ne sont que des changements additifs. Le compilateur actuel est toujours capable de compiler le code de la version 1.0 (édition 2015). Les évolutions du langage portent le nom d'édition et demandent un opt-in du fichier `cargo.toml` pour les utiliser. Une version du compilateur Rust supporte toutes les éditions précédentes et permet d'utiliser ensemble des crates écrites dans des éditions différentes. Ces mécanismes font que les breaking changes sont extrêmement rares. Chaque mise à jour (`rustup update`) est aussi inoffensive que possible et rien ne vous empêche d'avoir une tâche de votre CI utilisant la version bêta et compilant votre projet régulièrement pour détecter les potentiels problèmes et ainsi les signaler quelques semaines avant qu'ils n'arrivent dans la version stable.
Un dernier exemple de l'attention particulière qui est donnée à la stabilité par l'équipe Rust : pour chaque pull request sur le compilateur, l'ensemble des librairies disponibles sur crates.io sont recompilées afin de détecter des régressions.

Conclusion

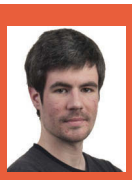
Le slogan de Rust en français **Figure 12** : tous les mots sont pertinemment choisis. "A tous" : vous l'aurez compris à la lecture des précédents chapitres, non seulement les champs d'application de ce langage sont très larges, mais il y a une réelle volonté inclusive de la part de la communauté. "Le pouvoir" : comprendre 'empowerment' : donner des moyens pour faire des logiciels ambitieux. "Fiables et efficaces" : les promesses de Rust.
Rust avance vite. C'est d'abord une lame de fond dans les domaines où C/C++ brillent depuis longtemps car, sans perte de performances, il amène un grand confort d'utilisation en éliminant toute une catégorie de bugs. Il a d'ores et déjà passé une première hype et s'impose comme un langage pour durer. Il est trop tôt pour savoir s'il va s'imposer aussi au-delà de ses premières cibles. Sa courbe d'apprentissage est encore rude à digérer et demande une certaine dose de motivation. Une fois que l'on a fini de se battre avec le compilateur et appris à l'utiliser, notamment pour se lancer dans des refacto dantesques, ce langage est vraiment productif. A mesure que le compilateur s'améliore et que les librairies les plus courantes gagnent en maturité, nous devrions voir apparaître des projets Rust qui d'ordinaire seraient développés dans d'autres langages et qui, en production, tout en étant moins buggés, tourneront avec une fraction des ressources actuelles. C'est à parier, et ce serait une bonne chose pour nos factures cloud, la consommation des datacenters et notre planète.

QUI UTILISE RUST ?

C'est une véritable lame de fond actuellement qui veut que les nouveaux projets qui devraient être écrits en C/C++ soient écrits en Rust. C et C++ ne sont pas près de disparaître, mais il y a une certaine mode incitant les nouveaux projets ambitieux à démarrer en Rust. Des noms ?
- **Microsoft** communique sur son usage de Rust et de ses avantages en termes de sécurité.
- **Mozilla** : c'est Rust qui a permis à Firefox de rattraper son retard de performance sur Chrome.
- **NPM** : le service d'authentification du package manager de l'écosystème JavaScript est écrit en Rust depuis 2018. Il n'a pas produit d'erreur depuis sa mise en production.
- Firecracker : une technologie de micro VM **Amazon** en relation avec leur service de functions as a service : Lambda.
- Fuchsia OS : un nouveau système d'exploitation de **Google**
- Diem : la crypto monnaie made in **Facebook**. Qui tente aussi de réécrire son contrôleur de source en Rust.
- **Dropbox** : déploie dans vos machines leur nouveau moteur de synchronisation écrit en Rust.
- **Discord** : utilise Rust à de nombreux niveaux de sa stack (SDK pour les jeux, live encoding, capture vidéo, services backend ...).

En France :
- **OVH et Clever Cloud** communiquent autour de Rust et en intègrent à leurs infrastructures.
- Sōzu : reverse proxy chez clever-cloud.com.
- Sonic : moteur de recherche léger de chez crisp.chat.
- MLA : un format d'archives chiffrées développé par l'**ANSSI** (Agence nationale de la sécurité des systèmes d'information)
- Enfin chez beezup.com, nous expérimentons une réécriture Rust d'un moteur d'exécution d'expression dynamique à disposition des clients finaux, gagnant 80% d'utilisation CPU sur cette partie gourmande du backend.

Rust, au-delà du buzz : un langage moderne, pour tous !



Jonathan Norblin

Software engineer (Cloud, DevOps, Data), je suis passionné par le cloud (GCP Cloud Architect) et la culture DevOps. À l'aise en Go, mon intérêt s'est porté depuis peu sur Rust.

https://www.linkedin.com/in/jonathan-norblin/

Au-delà du buzz suscité ces dernières années (désigné pour la 5e année consécutive comme langage préféré des développeurs dans la fameuse *StackOverflow Developer Survey*), Rust est un langage moderne fortement typé et compilé qui tire ses origines de différents langages, et qui tente d'en corriger les défauts en apportant des solutions ingénieuses.

Sa versatilité lui permet d'être utilisé pour de nombreux types de développements :

- pour des **applications systèmes**, comme alternative plus haut-niveau à C ou C++
- dans des **environnements embarqués** grâce à sa faible empreinte mémoire
- pour des **applications parallélisées**. Le modèle mémoire de Rust permet en effet de développer des applications parallélisées sans se soucier d'erreurs communes pouvant être rencontrées à l'exécution
- plus globalement, dans n'importe quelle autre application ! On notera particulièrement celles nécessitant des performances stables, rendu possible grâce à l'**absence de Garbage Collection** (contrairement à Java, Go,…)

L'objectif de Rust est de **tirer profit au maximum de sa machine**, comme C ou C++, mais sans se préoccuper de la gestion de la mémoire manuellement (`malloc`, `free`), ni des erreurs à l'exécution (*segmentation faults*), qui seront transformés en erreur de compilation. Un des gros avantages de Rust est aussi la clarté de ses messages d'erreur : tout est pensé pour que les développeurs, notamment les débutants, comprennent en un clin d'œil la cause du problème, détectée le plus tôt à la compilation.

Développé à l'origine par Mozilla pour son moteur de navigateur Servo (https://research.mozilla.org/servo-engines/), il est désormais utilisé en production par de nombreuses entreprises comme Dropbox, Canonical ou bien NPM (https://www.rust-lang.org/production/users). Pourquoi ne seriez-vous pas le prochain à l'utiliser ?

L'écosystème de Rust

Le langage Rust est accompagné d'un écosystème complet, de l'installation de sa *toolchain* (avec `rustup`, `cargo`), l'écriture et la compilation de programmes (compilateur `rustc`, linter `rustfmt`,…), jusqu'à la livraison (`crates.io`).

Installation

Toolchain

L'installation de Rust et de sa *toolchain* se fait à partir de `rustup`, un outil lui-même écrit en Rust, qui peut être installé en une ligne de commande sur une distribution GNU/Linux ou macOS :

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Pour les utilisateurs de Windows, l'installation nécessitera d'installer des composants supplémentaires, comme les C++ *Tools* de Visual Studio, mais l'installation se fera via un exécutable de la même manière. Dans la suite de cet article, nous supposerons être sous GNU/Linux.

Une fois `rustup` installé, la commande :

```
$ rustup toolchain install stable
```

permettra d'installer la dernière version stable de Rust. Notons aussi qu'il est possible d'avoir plusieurs versions de Rust en parallèle (versions plus anciennes, *nightly* builds) et de pouvoir changer facilement de version en lançant `rustup default <toolchain>`.

Pour la suite de cet article, nous utiliserons la dernière version stable de Rust disponible (1.48.0) en date d'écriture :

```
$ rustc --version
rustc 1.48.0 (7eac88abb 2020-11-16)
```

En plus du compilateur `rustc`, l'outil `cargo` (le gestionnaire de paquets de Rust, et bien plus…) est aussi installé

```
$ cargo --version
cargo 1.48.0 (65cbdd2dc 2020-10-14)
```

ainsi que d'autres outils : `rustfmt`, `rust-docs`, etc.

Sachez que grâce à sa communauté très active, le rythme de livraison du langage est assez soutenu (une nouvelle version mineure tous les mois environ), il convient donc de bien tenir à jour son installation pour bénéficier des nouvelles fonctionnalités du langage et des corrections de bugs. Pensez donc à exécuter assez souvent :

```
$ rustup update
```

pour maintenir sa *toolchain* à jour.

Les bons outils

IDE

Il n'existe pas aujourd'hui d'IDE spécialisé pour Rust. Mais, par chance, Rust peut être intégré dans les IDE suivants via des extensions ou plugins. Citons : IntelliJ IDEA, CLion, Visual Studio Code, etc. Vous pouvez aussi utiliser des éditeurs comme Vim, Emacs… tant qu'il y ait un support.

J'utilise IntelliJ IDEA mais votre choix doit d'abord se faire en fonction de vos habitudes. Si vous êtes débutants, je vous recommande quand même un des 3 premiers IDE. Ils ont un support complet du langage. Pour tester de petits morceaux de code, il existe aussi la playground de Rust disponible ici : http://play.rust-lang.org.

cargo

En ligne de commande, `cargo` sera l'outil le plus important

lorsque l'on utilise Rust. À la manière de Maven pour Java ou de npm pour Node, il permettra plusieurs choses, notamment :

- d'initialiser des projets : `cargo new`
- de les compiler : `cargo build`
- de les exécuter : `cargo run`
- de lancer des tests : `cargo test`
- de formater le code : `cargo fmt`
- d'installer des exécutables : `cargo install`

Notez que `cargo` est extensible et permettra, à la manière de plugins pour Maven, d'y ajouter de nouvelles fonctionnalités. On notera parmi ces extensions :

- `cargo-generate` (https://crates.io/crates/cargo-generate), qui permet d'initialiser des projets à partir de projets modèles, et qui peut s'avérer très pratique pour amorcer rapidement un projet :

```
$ cargo install cargo-generate
$ cargo generate --git https://github.com/githubusername/mytemplate.git
```

- `cargo-watch` (https://crates.io/crates/cargo-watch), qui permet, à la manière de `npm-watch`, d'exécuter des actions en continu (comme la compilation du projet ou l'exécution des tests) à chaque modification de code :

```
$ cargo install cargo-watch
$ cargo watch -x test # exécute les tests en continu
```

Hello World

Passons à l'incontournable *Hello World*. C'est simple et rapide !

Code source

Commençons par initialiser notre projet en utilisation cargo :

```
$ cargo new hello-world
```

`cargo` initialise pour nous la structure d'un projet :

```
$ tree
.
└── hello-world
    ├── Cargo.toml
    └── src
        └── main.rs
```

Jetons un coup d'œil à notre premier fichier source (`src/main.rs`) :

```
fn main() {
  println!("Hello, world!")
}
```

En une commande et grâce à `cargo`, le code source de notre premier *Hello World* en Rust est écrit ! En rentrant plus dans le détail du code, nous notons que le point d'entrée de notre programme est une fonction (`fn`) nommée `main`, **sans aucun argument**, résidant dans un fichier `main.rs` (`rs` pour Rust). L'affichage du texte `Hello, world!` est rendu possible grâce à l'appel de `println!`.

> **Tip** : `println!` n'est pas une fonction, mais une macro. Les macros sont facilement reconnaissables par leur point d'exclamation à la fin. Nous rentrerons plus dans le détail par la suite.

L'appel à la macro `println!` est possible, car celle-ci appartient au **prélude** de Rust, c'est-à-dire une liste d'imports automatiques dans n'importe quel programme écrit en Rust.

En plus du code source en Rust, `cargo` crée un manifeste `Cargo.toml`, dans lequel nous allons retrouver des informations sur notre application, les dépendances ou bien le type de package (binaire ou librairie) :

```
[package]
name = "hello-world"
version = "0.1.0"
authors = ["jonathan"]
edition = "2018"

# See more keys and their definitions at https://
doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Ici, notre application a pour cible un **binaire** ou **exécutable** (détecté automatiquement grâce à la présence du fichier `src/main.rs`).

Compilation

Plaçons-nous dans le dossier créé par `cargo` (`hello-world`). La commande :

```
$ cargo build
Compiling hello-world v0.1.0 (/tmp/tmp.u8EzRXGpbA/hello-world)
Finished dev [unoptimized + debuginfo] target(s) in 0.24s
```

permet de compiler notre projet. Par défaut, le projet est compilé en mode *debug* (non-optimisé). C'est l'option de compilation la plus rapide. L'option `--release` permettra d'optimiser le code, mais sera plus lente (sur notre petit exemple cependant, la différence ne sera pas flagrante). La compilation va générer un exécutable `target/debug/hello-world` :

```
$ tree
.
├── Cargo.toml
├── src
│   └── main.rs
└── target
    └── debug
        └── hello-world
```

> **Tip** : En mode debug, `cargo build` génère plus de fichiers (omis ici par souci de lisibilité), aidant principalement à la compilation incrémentale.

Exécution

Exécutons le binaire généré :

```
$ ./target/debug/hello-world
Hello, world!
```

Le binaire est exécutable sur n'importe quelle machine correspondant à la plateforme de compilation (*GNU/Linux amd64*, *Darwin amd64*,…). Des options de *cross-compilation* (par exemple, compilation d'un binaire pour Windows depuis GNU/Linux) sont possibles, mais ne seront pas abordées ici.
L'autre manière d'exécuter son programme est d'utiliser directement `cargo run`, ce qui est équivalent aux 2 étapes (compilation et exécution) effectuées plus tôt :

```
$ cargo run
  Compiling hello-world v0.1.0 (/tmp/tmp.u8EzRXGpbA/
hello-world)
   Finished dev [unoptimized + debuginfo] target(s) in 0.18s
    Running `target/debug/hello-world`
Hello, world!
```

Pendant le développement, c'est cette dernière option qui sera privilégiée pour sa simplicité et sa rapidité.

Rust 101

Le livre Rust (https://doc.rust-lang.org/book) contient tout ce qui est nécessaire à l'apprentissage de Rust. Nous allons tenter ici de résumer brièvement son contenu en se focalisant sur des exemples pratiques.

Types de base

Pour les entiers, Rust fait, à la manière de C ou Go, la distinction entre les entiers **signés** et les entiers **non-signés**. Les entiers non-signés (toujours positifs) sont préfixés par *u* (pour *unsigned*), alors que les entiers signés sont préfixés par *i*. Pour la taille (nombre de bits), Rust permet d'utiliser des entiers tenant sur 8, 16, 32, 64 ou 128 bits. Les types pour représenter les entiers sont donc les suivants : i8/u8, i16/u16, i32/u32, i64/u64, i128/u128. Il existe aussi un type d'entier particulier dont la taille correspond à l'architecture de la machine (32 ou 64 bits) sur laquelle le programme s'exécute (isize/usize), mais leur utilisation sera limitée à des cas très ponctuels, comme par exemple pour l'index d'un élément dans un tableau.
Il existe bien sûr d'autres types par défaut que les entiers :

- float : f32/f64 (f64 correspond au type double d'autres langages comme Java)
- booléen : bool
- character : char
- chaîne de caractères: String ou &str
- tuple
- tableau

Variables

Le mot-clé let permet de déclarer des variables :

```
let x = 1;
```

Lors d'une déclaration, il n'est pas nécessaire de déclarer le type de la variable. Rust étant fortement typé, le compilateur va inférer le type d'une variable lorsqu'il n'est pas spécifié explicitement. Dans l'exemple précédent, x sera par défaut de type i32, c'est-à-dire un entier signé sur 32 bits. La déclaration précédente est en tout point égal à let x: i32 = 1 (notez le type de la variable rajouté après:), ou bien let x = 1_i32. Cette dernière syntaxe sera aussi utilisée pour le ***cast*** d'une variable d'un type à un autre.

Variables immuables

Par défaut, les variables sont **immuables** (à l'exception de certains types très particuliers comme Cell). En suivant le principe de l'immutabilité, si l'on essaye de réassigner notre variable x précédemment créée une autre valeur, l'erreur suivante sera levée lors de la compilation :

```
x = 2;
^^^^^ cannot assign twice to immuable variable
cannot assign twice to immuable variable `x`
help: make this binding mutable
```

L'immutabilité permet de limiter les effets de bord, et rend le code plus lisible en évitant des surprises de modification de variables.

Variables mutables

Pour pouvoir réassigner une variable, il faut la rendre **mutable**, comme l'indique très clairement le message d'erreur précédent, et utiliser le mot-clé mut. Le code suivant :

```
let mut x = 1; // déclaration d'une variable mutable
x = 2; // ré-assignement possible, car x est mutable
println!("x vaut {}", x); // affichage de la variable
```

affiche : x vaut 2

> **Tip** : Notons au passage l'utilisation de println! pour formater facilement un message avec une variable ({}).

Il est aussi possible de déclarer plusieurs variables en même temps, immuables ou mutables, grâce à la syntaxe suivante :

```
// 2 variables immuables...
let (a, b) = (1, "un");
//... ou un mix entre variables mutable et immuable
let (mut mutable_var, immutable_var) = (2, "deux");
```

Constantes

Le mot-clé const permet aussi de déclarer des variables immuables. Les principales différences par rapport à let sont :

- le type de la constante doit être déclaré implicitement :

```
const MY_CONST: i32 = 1;
```

- une constante ne peut pas recevoir une valeur retournant d'une fonction : elle doit être **connue à la compilation**. Supposons que nous disposions d'une fonction return_some_i32() retournant une valeur de type i32 quelconque. Le code suivant renverra une erreur :

```
let x = return_some_i32(); // OK
const MY_CONST: i32 = return_some_i32();
                      ^^^^^^^^^^^^^^^^^^
calls in constants are limited to constant functions, tuple structs and tuple
 variants
```

Tuples

Le type de base tuple permet de définir un objet contenant différents éléments pouvant avoir un type différent :

```
let person = ("Doe", "John", 25)
```

L'accès aux éléments d'un tuple se fait via la syntaxe <tuple>.<index> (l'index commence à 0) :

```
println!("Name : {} {}, age : {}", person.1, person.0, person.2);
```

affiche : Name : John Doe, age : 25

Une des principales limitations des tuples est que les éléments ne sont pas nommés, et qu'il est parfois difficile de retrouver

une valeur juste avec l'index. L'autre limitation, sans rentrer dans le détail, est que pour des tuples ayant une arité (nombre d'éléments) supérieure à 12, ceux-ci perdent certaines de leurs propriétés (comme par exemple le fait d'être pouvoir affichés avec println!). Cependant, il est rare de voir des tuples avec 12 éléments : dans ce cas, on préférera les **structures**.

Tableaux

Instanciation

Contrairement aux tuples, un tableau est une collection d'objets **du même type**. La taille d'un tableau est définie à la compilation :

```
let array: [i32; 3] = [1, 2, 3]; // déclare un tableau d'i32 contenant 3 éléments
let first_element = array[0]; // 1, l'index du premier élément est 0 comme pour les tuples
let array_length = array.len(); // 3
```

Les valeurs d'un tableau comme instancié ci-dessus ne pourront pas être modifiées :

```
array[2] = 5;
^^^^^^^^^^ cannot assign
cannot assign to `array[_]`, as `array` is not declared as mutable
help: consider changing this to be mutable,
```

car le tableau est immuable. Pour pouvoir modifier son contenu, il faudra utiliser le mot-clé mut :

```
// redéclaration d'une variable ayant le même nom (shadowing)
let mut array: [i32; 2] = [1, 2];
array[1] = 10;
println!("array : {:?}", array);
```

affiche : array : [1, 10]

> **Tip** : Notons au passage l'utilisation de {:?} au sein de la macro println!, permettant d'afficher des structures plus complexes que les types de base (la structure devra implémenter std::fmt::Display pour que l'affichage soit possible).

Notons aussi l'existence d'un raccourci pour initialiser un tableau de taille *n* contenant le même élément :

```
let array: [u8; 256] = [1_u8; 256]; // tableau contenant 256 fois l'entier (u8) 1
```

Accès aux éléments

L'avantage de connaître la taille du tableau à la compilation est que Rust est capable de renvoyer une erreur **dès la compilation** en cas d'accès à une variable hors du tableau (contrairement à C, qui renverrait n'importe quelle valeur de la mémoire se trouvant contigüe au tableau, ou à Java, qui renverrait une ArrayIndexOutOfBoundsException à l'exécution) :

```
let array = [1, 2, 3];
println!("Fourth element : {}", array[3]);
                               ^^^^^^^^ index out of bounds: the len is 3 but the index is 3
this operation will panic at runtime
```

L'index utilisé pour accéder à un élément du tableau doit être de type usize :

```
let index: usize = 1;
println!("Element number {} : {}", index, array[index]);
```

affiche : Element number 1 : 2

Collections

Outre les tableaux, 2 types de collections sont principalement utilisés :

- Vec
- HashMap

D'autres collections comme LinkedList ou HashSet existent aussi, mais leur utilisation reste moindre par rapport à Vec ou HashMap.

Vec

Un vecteur est un **tableau redimensionnable**, dont la taille n'est pas connue à la compilation. Pour les développeurs Go, les vecteurs de Rust ressemblent aux *slices* : ils sont composés d'un pointeur sur le tableau sous-jacent, d'une taille et d'une capacité. Lorsque la capacité maximale est atteinte, le vecteur est agrandi automatiquement (sans opération manuelle à faire du côté du développeur). Les vecteurs sont utilisables de la manière suivante (il existe au moins 3 manières de déclarer des vecteurs) :

```
let vec_1 = vec![1, 2]; // création d'un vecteur grâce à la macro vec!

let vec_2 = Vec::from([1, 2]); // appel de la méthode statique Vec::from

let mut vec_3: Vec<i32> = Vec::new(); // instancie un vecteur vide
vec_3.push(1); // ajoute un élément
vec_3.push(2); // ajoute un autre élément

assert_eq!(vec_1, vec_2); // OK
assert_eq!(vec_2, vec_3); // OK
```

> **Tip** : assert_eq! permet de comparer 2 valeurs, et termine le programme en cas d'inégalité.

L'accès aux éléments se fait de la même manière qu'un tableau :

```
println!("Second element : {}", vec_1[1]);
```

affiche : Second element : 2

Attention, contrairement aux tableaux, l'accès à un élément en dehors d'un vecteur ne produira pas d'erreur de compilation mais *paniquera* à l'exécution. Le code :

```
println!("Fourth element : {}", vec_1[3]);
```

renvoie une erreur à l'exécution :

```
thread '<unnamed>' panicked at 'index out of bounds: the len is 2 but the index is 4'
```

HashMap

Une HashMap est une collection dont les éléments sont **indexés par des clés**, et non par un index comme les vecteurs :

```
// contrairement à Vec, HashMap ne fait pas partie du prélude et nécessite un import
use std::collections::HashMap;
```

```
let mut numbers = HashMap::new(); // initialisation avec la méthode statique new
numbers.insert("un", 1);
numbers.insert("deux", 2);
```

L'accès se fait donc via l'index que nous avons choisi (ici, une chaîne de caractères) :

```
println!("deux : {}", numbers["deux"]);
```

affiche : 2

Boucles

Rust dispose de 3 mots-clés pour déclarer des boucles :

- while
- loop
- for

while

while est le plus "classique", il permet d'itérer tant qu'une certaine condition est vraie :

```
let mut x = 3;
while x >= 0 {
 println!("x : {}", x);
 x = x - 1;
}
```

affiche :

```
x : 3
x : 2
x : 1
x : 0
```

loop

loop permet d'itérer de manière infinie, et le mot-clé break permet de sortir de la boucle. La boucle précédente (avec while) peut être réécrite de la manière suivante :

```
let mut x = 3;
loop {
 println!("x : {}", x);
 if x == 0 { break; }
 x = x - 1;
}
```

Le mot-clé continue permet, comme en C++, de passer à l'itération suivante sans exécuter le code après l'instruction :

```
let mut x = 0;
loop {
 x = x + 1;
 if x % 2 != 0 { continue; }
 if x > 10 { break; }
 println!("{}", x)
}
```

affiche : les nombres pairs de 2 à 10

Tout comme if, loop est une expression et il est possible d'assigner une variable à une valeur directement renvoyée par la boucle grâce à la syntaxe break <return_value> :

```
let mut x = 3;
let result = loop {
 if x == 0 { break x; }
 x = x - 1;
};
println!("result : {}", result);
```

affiche : result : 0

for

for permet de parcourir des itérateurs :

```
// 1..=10 correspond à l'intervalle [1; 10]
// 1..10 aurait signifié [1; 10[
for i in 1..=10 {
 if i % 2 == 0 {
  println!("{}", i);
 }
}
```

affiche : les nombres pairs de 2 à 10

Cela fonctionne donc sur des collections (via la méthode iter() pour les tableaux ou vecteurs) :

```
let array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
for i in array.iter() {
 if i % 2 == 0 {
  println!("{}", i);
 }
}
```

ou même des *maps* :

```
use std::collections::HashMap;

let mut numbers = HashMap::new();
numbers.insert("un", 1);
numbers.insert("deux", 2);
numbers.insert("trois", 3);

for (key, value) in numbers.iter() {
 println!("{}: {}", key, value);
}
```

affiche :

```
deux: 2
trois: 3
un: 1
```

> **Tip** : Attention ! Comme dans de nombreux langages, l'ordre n'est pas garanti lorsque l'on parcourt une HashMap. Un des moyens pour garantir l'ordre est de trier l'itérateur (iter().sorted()), ou bien d'utiliser une autre collection comme BTreeMap (trié par clé).

Conditionnelles

if - else if - else

Les mots-clés if, else if et else permettent de déclarer des expressions conditionnelles :

```
let x = 1;
if x == 1 {
 println!("x vaut 1");
```

```
} else if x == 2 {
 println!("x vaut 2");
} else {
 println!("x ne vaut ni 1 ni 2");
}
```

affiche : x vaut 1

Dans cet exemple simple, le texte x vaut 1 est affiché, car la valeur 1 a été assignée à la variable x. Plutôt que d'appeler println! dans chaque branche, il est possible de profiter du fait que la conditionnelle if - else if - else retourne une valeur (c'est une **expression**), à la manière de langages comme Scala ou OCaml. Il est donc possible d'assigner le retour à une variable, **puis** d'en afficher le contenu :

```
let x = 1;
let result = if x == 1 {
 "x vaut 1"
} else if x == 2 {
 "x vaut 2"
} else {
 "x ne vaut ni 1 ni 2"
};
println!("{}", result);
```

affiche le même résultat que précédemment. Attention ici à ne pas oublier le point-virgule à la fin (après l'accolade finale du else) !

Pattern matching

Un autre moyen d'écrire l'expression conditionnelle précédente est d'utiliser le *pattern matching* grâce à l'opérateur match :

```
let result = match x {
 1 => "x vaut 1",
 2 => "x vaut 2",
 _ => "x ne vaut ni 1 ni 2"
};
```

Les différents cas doivent être **exhaustifs** : dans l'exemple précédent, il faut gérer le cas par défaut (_), correspondant en quelque sorte au else. Le *pattern matching* permet de faire beaucoup plus de choses qu'un simple if-else, notamment lorsqu'il est utilisé conjointement avec des énumérations.

Énumérations

Le mot-clé enum permet de créer un type contenant un ensemble fini de valeurs :

```
enum Direction {
 Right,
 Left,
 Top,
 Bottom,
}
```

Les valeurs d'une énumération peuvent être simples, comme ci-dessus, ou bien contenir des valeurs. Prenons un exemple simple où l'on décrit les actions possibles d'un pion se déplaçant sur un damier en 2 dimensions :

```
enum Action {
 AvancerDeNCases { d: Direction, n: i32 },
 SeRendreSurLaCase { x: i32, y: i32 },
}
```

Comme évoqué plus tôt, on verra souvent les énumérations utilisées conjointement avec le *pattern matching* :

```
let position = (2, 3);
let action = Action::AvancerDeNCases {
 d: Direction::Right,
 n: 10,
};

let new_position = match action {
 Action::AvancerDeNCases { d, n } => match d {
  Direction::Right  => (position.0 + n, position.1),
  Direction::Left   => (position.0 - n, position.1),
  Direction::Top    => (position.0, position.1 - n),
  Direction::Bottom => (position.0, position.1 + n),
 },
 Action::SeRendreSurLaCase { x, y } => (x, y),
};

assert_eq!(new_position, (12, 3));
```

Fonctions et procédures

La déclaration d'une fonction ou d'une procédure se fait avec le mot-clé fn :

```
// fonction
fn sum(a: i32, b: i32) -> i32 {
 a + b
}

// procédure
fn say_hello(name: String) {
 println!("Hello {}", name);
}
```

Les paramètres des méthodes sont décrits avec la syntaxe suivante <nom>: <type>. Dans le cas des fonctions, le type de retour est spécifié par la syntaxe -> <type de retour>. Notons aussi que le mot-clé return n'est pas nécessaire : la dernière expression est la valeur retournée par la fonction. C'est la façon préférée de faire en Rust. Néanmoins, si vous souhaitez utiliser return, sachez que la syntaxe suivante est aussi possible :

```
/// fait la **somme** de deux entiers *signés*
fn sum(a: i32, b: i32) -> i32 {
 return a + b;
}
```

> **Tip** : Pour illustrer la documentation des fonctions ayant pour vocation d'être utilisées de manière publique, il est possible de laisser une description au format markdown commençant par /// pour générer la documentation automatiquement (l'équivalent de /** <description> */ en Java). Il s'agit juste d'une illustration ici : de grâce, ne laissez pas un commentaire comme cela sur une fonction aussi simple et dont le nom est suffisamment clair !

Dans le cas où l'on souhaite retourner plusieurs valeurs depuis une fonction, il est possible d'utiliser le type tuple pour encapsuler le résultat :

```rust
fn return_two_values() -> (i32, String) {
 (1, String::from("un"))
}

let (a, b) = return_two_values();
println!("a : {} et b : {}", a, b);
```

affiche : a : 1 et b : un

On peut aussi directement accéder aux éléments du tuple :

```rust
let result = return_two_values();
println!("result.0 : {} et result.1 : {}", result.0, result.1);
```

affiche :

```
result.0 : 1 et result.1 : un
```

Structures

Reprenons d'abord l'exemple de notre tuple défini précédemment :

```rust
let person = ("Doe", "John", 25)
```

À la place des tuples, il est possible de définir des structures. C'est une autre façon de déclarer des objets contenant différentes variables :

```rust
struct Person {
 last_name: String,
 first_name: String,
 age: u8
}
```

L'instanciation se fera de la manière suivante, en n'oubliant pas de rappeler chaque attribut de la structure. L'ordre n'est pas important :

```rust
let person = Person {
 first_name: String::from("John"),
 last_name: String::from("Doe"),
 age: 25
};
```

Pour accéder à un élément de la structure, nous pouvons spécifier **le nom de l'attribut**, contrairement à **l'index** lorsque nous manipulions des tuples :

```rust
println!("Name : {} {}, age : {}", person.first_name, person.last_name,
 person.age);
```

affiche : Name : John Doe, age : 25

Il est possible de définir des méthodes sur des structures, grâce au mot-clé impl. La définition de méthodes, statiques ou non-statiques, se fait à part de la déclaration de la structure :

```rust
impl Person {

 /// méthode statique renvoyant une personne correspondant à John Doe
 fn john_doe() -> Self { // Self ou Person peut être utilisé
  Person {
   last_name: String::from("Doe"),
   first_name: String::from("John"),
   age: 25
  }
 }

 /// méthode non-statique, affichant le prénom  et nom d'une personne
 fn present_himself(&self) { // self fait référence  à l'instance courante
  println!("Hello, my name is {} {}.", self
.first_name, self.last_name);
 }

}
```

Le code :

```rust
Person::john_doe().present_himself();
```

affiche :

```
Hello, my name is John Doe.
```

> **Tip** : L'appel de méthodes statiques se fait via ::, tandis que l'appel de méthodes non statiques se fait via ., comme dans la majorité des langages. Comme en Python, le paramètre &self est omis lors de l'appel de fonction.

Il est possible aussi de définir plusieurs blocs impl pour une même struct. Éclater la définition des méthodes sur une struct permet de séparer les responsabilités; cependant, attention à ne pas en abuser pour la lisibilité de votre code, sauf dans le cas d'implémentation de traits.

Traits

En Rust, il n'y a pas d'héritage au sens programmation orientée-objet du terme. Par contre, il est possible d'hériter de comportements grâce aux traits, qui sont semblables aux traits en Scala, dans une certaine mesure aux *typeclass* en Haskell, ou bien encore aux interfaces dans d'autres langages comme Java.
Un trait définit plusieurs comportements à implémenter :

```rust
trait Car {
 fn honk(&self, number_of_times: usize);

 fn roll(&self) {
  println!(r#"
     .-'--`-._
   ---'-0---0--'
  "#);
 }
}
```

À l'intérieur de la définition d'un Trait, il est possible d'implémenter :

- des méthodes statiques
- des méthodes non statiques (honk)
- des méthodes directement dans le Trait (roll), qui seront disponibles dans les structures implémentant le Trait

Implémentation d'un trait

Considérons la struct suivante :

```rust
struct RenaultFuego {
 plate: String
}
```

L'implémentation d'un Trait pour une struct se fait de la manière suivante (impl <Trait> for <struct>) :

```
impl Car for RenaultFuego {
 /// repeat <number_of_times> times honk sound
 fn honk(&self, number_of_times: usize) {
  println!("{}", "tut".repeat(number_of_times));
 }
}
```

La méthode peut ensuite être appelée sur une instance de la structure :

```
let fuego = RenaultFuego { plate: String::from("AA-111-AA") };
fuego.honk(2);
```

affiche :

```
tut tut
```

Une struct peut très bien implémenter plusieurs traits !

Utilisation de #[derive]

L'attribut #[derive] permet d'implémenter certains traits sans la définition de blocs impl, et d'hériter de comportements déjà prédéfinis. La structure suivante :

```
#[derive(Debug, PartialEq)]
struct Person {
 last_name: String,
 first_name: String,
 age: u8
}
```

implémente les traits Debug et PartialEq automatiquement. Debug permet de formater un objet via {:?} (dans println! par exemple), tandis que PartialEq définit automatiquement une fonction d'égalité entre 2 instances de type Person (sur ses attributs).

Génériques

Lorsque l'on essaye de factoriser du code, il est courant d'avoir besoin de recourir aux génériques, pour notamment écrire des fonctions pouvant prendre en paramètres différents types. Nous souhaitons disposer d'une fonction renvoyant le plus grand entier i32 :

```
fn largest(a: i32, b: i32) -> i32 {
 if a >= b {
  a
 } else {
  b
 }
}
```

Cette fonction ne prend en paramètre que des entiers signés sur 32 bits (i32), et il peut être pratique de la **généraliser** à tous les types nombres de base (par exemple des *floats*). Pour cela, plutôt que d'écrire une nouvelle fonction au contenu identique, mais prenant en entrée 2 *floats* et retournant le *float* le plus grand, nous pouvons user des génériques :

```
fn largest<T: std::cmp::PartialOrd>(a: T, b: T) -> T {
 if a >= b {
  a
 } else {
  b
 }
}
```

Le type générique T est introduit, avec une contrainte : T doit implémenter std::cmp::PartialOrd, qui définit la méthode >= utilisée dans le corps de la fonction. En effet, cela n'a pas de sens pour le compilateur d'appeler la fonction largest sur des éléments ne pouvant pas être comparés !
Grâce aux génériques, il est donc désormais possible d'appeler :

```
let l1 = largest(1, 2);
let l2 = largest(1.23, 4.56);
```

Un autre exemple d'utilisation des génériques est le type Option (il n'existe pas de valeur null en Rust), défini comme suit :

```
enum Option<T> {
 Some(T),
 None
}
```

Une instance de type Option peut contenir une valeur (Some(T)) ou aucune (None). Ici, T correspond à n'importe quel type. Cela s'avère pratique lorsqu'on dispose de fonctions pouvant retourner une valeur ou non, par exemple une fonction cherchant un élément dans un vecteur :

```
fn find_item_in_vec<T: std::cmp::PartialEq>(vec: Vec<T>, item: T) -
> Option<T> {
 for i in vec {
  if i == item { // T doit définir la méthode == (std::cmp::PartialEq)
   return Some(i);
  }
 }
 None // si l'élément n'est pas trouvé
}

find_item_in_vec(vec!(1, 2, 3), 2); // Some(2)
find_item_in_vec(vec!(1, 2, 3), 4); // None
```

Gestion des erreurs

Erreurs non récupérables

Les erreurs non récupérables surviennent lorsque le programme "panique". Il existe deux cas possibles :

- l'appel explicite à la macro panic!
- une erreur interne pendant l'exécution. Ce type d'erreur correspond aux erreurs *unchecked* de Java (IndexArrayOutOfBoundsException, ArithmeticException lors d'une division par zéro,…)

Il convient d'utiliser panic! avec parcimonie, car cela termine le programme. Cependant, il existe des cas où c'est justement ce comportement que l'on souhaite. La variable d'environnement RUST_BACKTRACE permet de débugger et d'avoir, en plus du message qui a causé la panique, une stacktrace pour aider à débugger.

Erreurs récupérables

Les erreurs récupérables sont souvent définies par la présence d'un objet Result, défini comme ceci :

```
enum Result<T, E> {
  Ok(T),
```

```
  Err(E),
}
```

Pour les développeurs familiers avec Scala, cela ressemble à s'y méprendre au type Try (avec Success = Ok et Failure = Err). L'énumération utilise les génériques (types T et E).
Les valeurs de cette énumération sont renvoyées par des méthodes qui sont souvent amenées à échouer, comme l'ouverture d'un fichier (si le fichier n'existe pas) ou la conversion d'une chaîne de caractères en entier.
Prenons comme exemple ce dernier cas :

```
let input_string = String::from("42");
let input_string_convert = input_string.parse::<i32>();
```

Contrairement à ce que l'on pourrait croire en lisant le code, input_string_convert n'est pas de type i32, mais bien Result<i32, std::num::ParseIntError>. En effet, si la chaîne en entrée ne contient pas un entier, Err correspondra à ParseIntError. Result étant une énumération, le *pattern matching* peut être utilisé pour gérer les 2 cas :

```
match input_string_convert {
 Ok(int) => println!("Conversion en entier réussie : {}", int),
 Err(_) => println!("La conversion en entier a échouée : {} n'est pas un
entier",
            input_string)
}
```

Propagation d'erreurs

Utiliser le *pattern matching* pour gérer le cas d'erreur à chaque fois qu'une fonction peut potentiellement retourner une erreur peut s'avérer lourd. Cela rappellera quelque chose aux développeurs familiers en Go (multiplication de blocs if err != nil dans le code). En Rust, depuis la version 1.13, il existe l'opérateur ? qui permet de propager les erreurs. Comparons les 2 fonctions suivantes qui lisent le contenu d'un fichier vers une chaîne de caractères :

```
use std::io::{Read, Error};
use std::fs::File;

// propagation des erreurs manuellement avec le pattern matching
fn read_file_content(filename: String) -> Result
<String, Error> {
  let f = File::open(filename);

  // si l'ouverture du fichier échoue, on renvoie une Err
  let mut f = match f {
    Ok(file) => file,
    Err(e) => return Err(e),
  };

  let mut s = String::new();
  // si la lecture du fichier échoue, on renvoie une Err
  match f.read_to_string(&mut s) {
    Ok(_) => Ok(s),
    Err(e) => Err(e),
  }
}
```

et :

```
// propagation des erreurs automatiquement
fn read_file_content(filename: String) -> Result
<String, Error> {
  let f = File::open(filename);
  let mut s = String::new();

  // la propagation d'erreur automatique se fait avec l'opérateur ?
  // en cas d'erreur, un result de type Err est directement renvoyé
  // et la méthode retourne automatiquement
  f?.read_to_string(&mut s)?;

  // en cas de succès de la ligne précédente, on renvoie un Result de type Ok
  Ok(s)
}
```

La second fonction est plus concise : on préférera donc l'utilisation de l'opérateur ? lors d'une propagation d'erreur simple. Le *pattern matching* sera préféré lorsque les cas d'erreurs seront gérés de manière particulière (par exemple, l'affichage simple d'un *log*).

> **Tip** : il existe aussi la macro try! qui est l'ancêtre de l'opérateur ? avant que ce-dernier ne devienne un élément de la syntaxe de Rust. N'utilisez pas try!, il risque d'être déprécié dans de futures versions.

Paniquer quand même en cas d'erreur récupérable

Appliqué sur un objet de type Result, la méthode expect permet au programme de paniquer avec un message d'erreur passé en paramètre dans le cas où le Result correspond à la valeur Err :

```
let input_string_to_int: i32 = input_string.parse::<i32>()
                               .expect("Un entier est attendu");
```

Il existe un équivalent sans message d'erreur, unwrap :

```
let input_string_to_int: i32 = input_string.parse::<i32>().unwrap();
```

Attention, dans le cas où la méthode parse renverrait une erreur, les 2 morceaux de code précédents feront paniquer le programme.
Ces méthodes sont particulièrement utilisées si l'erreur n'est vraiment pas récupérable et que celle-ci nécessite un arrêt "sans sommation" du programme.

Gestion de la mémoire

Rust dispose d'un modèle particulier de gestion de la mémoire :

- la gestion de la mémoire ne se fait pas manuellement par le développeur (malloc, free, …)
- Rust ne dispose pas d'un *Garbage Collector* déclenché périodiquement pour libérer la mémoire

Rust dispose en fait d'une gestion particulière :

- la mémoire associée à une variable est **directement libérée** lorsque la variable sort du *scope*. À la fin d'un scope, le **destructeur** de chaque variable initialisée dans ce *scope* est appelé et se charge de libérer la mémoire.
- la gestion de la mémoire en Rust et le fait que Rust soit *memory-safe* se base sur les principes d'*ownership* et de *borrowing*, que nous allons voir tout de suite.

Tip : Rust peut stocker des valeurs à deux endroits en mémoire : la pile (*stack*) ou le tas (*heap*). La *stack* est réservée aux valeurs de taille fixe (par exemple des types primitifs comme i32, f64, bool). tandis que la *heap* peut contenir des valeurs dont la taille n'est pas connue à la compilation, pouvant varier (*buffer* associé à une String, Hashmap), ou encore des variables déclarées explicitement sur celle-ci (type particulier Box).

Ownership et borrowing

En théorie, expliquer l'*ownership* et le *borrowing* est simple : une valeur ne peut avoir qu'un seul propriétaire (*owner*) à un instant donné. Néanmoins, la valeur peut être prêtée (*borrowed*) le temps d'un bloc de code, par exemple une fonction, puis retournée à son propriétaire. Le principe est le même que dans la vie réelle. Prenons l'exemple suivant :

- Je suis le propriétaire (*owner*) d'un livre. Je peux utiliser le livre tant que je veux
- Je peux prêter mon livre à un ami (*borrow*)
- À la fin de la lecture, mon ami me rend mon livre

Il existe aussi le cas où la propriété est donnée (**transfert d'*ownership***), sans prêt (*borrowing*) :

- Je suis le propriétaire (*owner*) d'un livre. Je peux utiliser le livre tant que je veux
- Je donne définitivement le livre à un ami : je ne pourrai plus jamais l'utiliser

Transfert d'ownership

Lorsque certaines valeurs (comme les String) sont assignées à une autre variable, la propriété est **transférée** : la variable initiale ne peut plus être utilisée. Le code suivant :

```
let a = String::from("Rust");
let b = a; // la propriété est transférée
println!("{}", a);
```

ne compile pas :

```
error[E0382]: borrow of moved value: `a`
  |
  |   let a = String::from("Rust");
  |       - move occurs because `a` has type `std::string::String`,
  |         which does not implement the `Copy` trait
  |   let b = a;
  |           - value moved here
  |   println!("{}", a);
  |                  ^ value borrowed here after move
```

Le message d'erreur est clair : la valeur de a a été déplacée dans la valeur b. Dans le cas du type String, la propriété est transférée, car le type String n'implémente pas le trait Copy.
Cela est aussi valable avec les fonctions :

```
fn print(s: String) {
 println!("{}", s);
} // la mémoire associée à s est libérée

fn main() {
 let s = String::from("Hello");
 print(s); // la propriété de s est transférée
 // ici, la mémoire associée à s a été libérée par la fonction print
 print(s); // ne compile pas, car la mémoire a été libérée par la fonction print
}
```

Dans le cas d'entiers ou d'autres types primitifs (qui eux, implémentent le trait Copy), la propriété ne sera pas transférée (dans notre analogie avec notre livre, une photocopie sera faite) :

```
let a = 1;
let b = a; // une copie est effectuée (le type primitif i32 implémente le trait Copy)
println!("{}", a); // compile parfaitement
```

Pourquoi une telle différence entre String et les autres types primitifs ? En fait, en interne, une instance de String est un vecteur de u8. Il est défini sur la *stack* comme 3 mots (un pointeur sur la *heap*, une longueur et une capacité). Assigner une nouvelle variable la valeur d'une instance déjà existante de type String ne nécessite pas d'allouer un nouveau bloc sur la *heap* pour copier le contenu de la chaîne de caractères : cela ne serait pas efficace en cas de longue chaîne de caractères ! Rust décide alors de déplacer (*move*) ou transférer la propriété plutôt que de copier (*copy*) l'instance. En effet, en cas de sortie du *scope* de la première variable, la mémoire associée à celle-ci serait libérée, laissant ainsi pointer la seconde variable sur une zone mémoire sur la *heap* déjà libérée. Je vous laisse imaginer ce qui se passerait en cas d'utilisation d'une variable pointant sur un morceau de mémoire considéré comme libéré !
Contrairement aux String, les types primitifs (i32, f64, bool) ont une taille fixe et sont instanciés sur la *stack*. Une copie sur la *stack* étant très peu coûteuse, la propriété n'est pas transférée : une copie est alors réalisée. Les autres types implémentant le trait Copy ont le même comportement.

Borrowing

Le code déjà présenté plus tôt et ne compilant pas :

```
fn print(s: String) {
 println!("{}", s);
} // la mémoire associée à s est libérée

fn main() {
 let s = String::from("Hello");
 print(s); // la propriété de s est transférée
 // ici, la mémoire associée à s a été libérée par la fonction print
 print(s); // ne compile pas, car la mémoire a été libérée par la fonction print
}
```

peut être corrigé pour que la propriété de s ne soit pas transférée, mais que s soit plutôt **empruntée** (*borrow*). Pour cela, il faut changer la signature de print pour qu'elle ne prenne pas en paramètre une String, mais plutôt une référence vers la String (&String) :

```
fn print(s: &String) {
 println!("{}", s);
}

fn main() {
```

```
let s = String::from("Hello");
print(&s); // s est prêtée à la fonction print
// ici, la propriété de s est rendue à la fonction main
// s peut de nouveau être utilisée !
print(&s); // s est prêtée à la fonction print
println!("{}", s) // s peut être utilisée !
} // la mémoire associée à s est libérée
```

Lors d'un emprunt tel qu'effectué plus haut, le contenu de la variable s ne peut pas être modifié, c'est pour cela que l'on peut avoir autant de références immutables que l'on souhaite, on parle de ***shared borrowing*** :

```
fn main() {
 let s = String::from("Hello");
 let ref1 = &s;
 let ref2 = &s;
 println!("{}", ref1); // ok
}
```

Il est possible de modifier une référence en utilisant le mot-clé `mut`. On parle alors de ***mutable borrowing***. Prenons l'exemple suivant :

```
fn modify_string(s: &mut String) {
 s.push_str(" world");
}

fn main() {
 let mut s = String::from("Hello");
 modify_string(&mut s); // possible de muter (mut) la valeur référencée
sans en prendre la propriété (&)
 println!("{}", s); // s peut ici être utilisée
}
```

Il existe néanmoins plusieurs restrictions en cas de *mutable borrowing* :
il ne peut y avoir plusieurs références mutables au même moment. Le code suivant :

```
fn main() {
 let mut s = String::from("Hello");
 let ref1 = &mut s;
 let ref2 = &mut s;
 ref1.push_str(" world");
}
```

renvoie une erreur :

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:4:14
  |
3 |  let ref1 = &mut s;
  |             ------ first mutable borrow occurs here
4 |  let ref2 = &mut s;
  |             ^^^^^^ second mutable borrow occurs here
5 |  ref1.push_str(" world");
  |  ---- first borrow later used here
```

Sans cette erreur, la valeur pointée `ref2` aurait pu être modifiée par effet de bord (modification de la valeur pointée par `ref1`), ce qui aurait pu être difficile à tracer et corriger à l'exécution. C'est pourquoi Rust interdit ce comportement à la compilation : si deux pointeurs accèdent à la même donnée, et qu'un pointeur est utilisé pour modifier le contenu, alors cette erreur est renvoyée et le code ne peut pas compiler : une référence mutable est un accès temporaire exclusif à une valeur.
Il ne peut pas y avoir une référence mutable et une référence immutable au même moment : la logique derrière cela étant que la référence immutable ne s'attend pas à ce que la valeur soit modifiée à travers la référence mutable ! Le compilateur interdit donc ce comportement :

```
fn main() {
 let mut s = String::from("Hello");
 let ref1 = &mut s;
 let ref2 = &s;
 ref1.push_str(" world");
}
```

en renvoyant l'erreur assez claire :

```
error[E0502]: cannot borrow `s` as immutable because it is also borrowed
 as mutable
 --> src/main.rs:4:14
  |
3 |  let ref1 = &mut s;
  |             ------ mutable borrow occurs here
4 |  let ref2 = &s;
  |             ^^ immutable borrow occurs here
5 |  ref1.push_str(" world");
  |  ---- mutable borrow later used here
```

Lifetimes

Lorsque l'on utilise des références, la valeur référencée ne doit pas être supprimée tant que la référence peut encore être utilisée. C'est le validateur d'emprunt (***borrow checker***), faisant partie du compilateur, qui se charge de vérifier cette assertion. Lorsque la durée de vie (*lifetime*) d'une valeur pointée par une référence est plus courte que la référence elle-même, alors le compilateur rejette le programme. Montrons ceci avec un tel programme, tiré du livre Rust :

```
let r;

{
 let x = 5;
 r = &x;
} // ici, la mémoire associée à x est libérée

println!("r: {}", r); // ne compile pas, car r pointerait sur une zone mémoire
 libérée
```

L'erreur est la suivante :

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:6:13
  |
6 |      r = &x;
  |          ^^ borrowed value does not live long enough
7 |  }
  |  - `x` dropped here while still borrowed
8 |
9 |  println!("r: {}", r);
  |                    - borrow later used here
```

Comment Rust (enfin, plutôt le *borrow checker*) arrive-t-il à détecter une telle erreur dès la compilation ? En fait, en dessous du capot, Rust associe à chaque variable une *lifetime* et vérifie le respect de la règle suivante : **la valeur doit avoir une durée de vie plus grande que les références sur cette valeur**. Les *lifetimes* associées sont la plupart du temps automatiquement assignées par le compilateur, mais il arrive parfois que l'on doive les spécifier nous-même pour pallier des limitations côté compilateur. Une *lifetime* est spécifiée par la syntaxe '<lifetime_name> (par exemple, 'a, 'b,…). Souvent, on a besoin d'annoter des entrées et sorties d'une fonction lorsque l'on utilise une référence (&) en entrée et en sortie. En général et de manière pratique, le développeur ne fera pas attention aux *lifetimes* et laissera le compilateur émettre une erreur si l'annotation est nécessaire. Reprenons un exemple du livre Rust :

```
fn longest(x: &str, y: &str) -> &str {
  if x.len() > y.len() {
    x
  } else {
    y
  }
}
```

ne compile pas :

```
error[E0106]: missing lifetime specifier
 --> src/lib.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |           ----     ----     ^ expected named lifetime parameter
  |
 = help: this function's return type contains a borrowed value, but the
 signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
1 | fn longest<'a>(x: &'a str, y: &'a str) ->
 &'a str {
  |       ^^^^  ^^^^^^^  ^^^^^^^  ^^^
```

En effet, dans notre fonction, x ou y (références) peuvent être renvoyées, et il convient d'associer la même *lifetime* à x, y et à la sortie pour en spécifier explicitement la durée de vie (celle de x ou y) : la référence en sortie doit vivre au moins aussi longtemps que les références en entrée.

Organisation de son code

Package et crates

cargo new <package> crée un nouveau **package**. Un package est caractérisé par la présence du fichier Cargo.toml. Par défaut, comme dans notre tout premier programme au début de cet article, ce package contient 1 *crate* exécutable (*binary crate*) grâce à la présence d'un fichier src/main.rs. Un *package* peut contenir une ou plusieurs *crates* : autant de *binary crates* que nécessaire (dans un dossier src/bin : src/bin/mybin1.rs, src/bin/mybin2.rs), mais seulement 0 ou 1 *crate* bibliothèque (*library crate*) : cargo détecte automatiquement la présence de ce type de *crate* par la présence d'un fichier src/lib.rs. Bien sûr, un package doit contenir au moins 1 *crate*, que ce soit une *crate* exécutable (*binary*) ou une bibliothèque (*library*) !

Modules

Lorsque le code commence à grossir, les **modules** sont une façon de regrouper des morceaux de code par fonctionnalités, comme le permettent dans une certaine mesure les packages dans d'autres langages comme Java (attention à ne pas confondre avec les packages Rust). Ils permettent aussi et surtout d'isoler ces morceaux de code en gérant la **visibilité** (privée/publique). Par défaut, tous les modules sont privés et il n'est donc pas possible d'accéder au contenu d'un module, dont les fonctions, depuis l'extérieur de celui-ci. Au sein d'un même module, toutes les fonctions, même privées, sont accessibles. Un module se déclare avec le mot-clé mod :

```
mod mymodule {
 fn function1() {
  //...
 }

 fn function2() {
  function1(); // équivalent à self::function1(), accessible, car au sein d'un
même module
 }
}
```

Pour rendre visible la fonction depuis un autre module (par exemple le module racine qui contient la fonction main), le mot-clé pub peut être utilisée pour rendre publique un module entier, ou seulement une fonction :

```
mod mymodule {
 fn function1() {
  //...
 }

 pub fn function2() {
  function1();
 }
}

fn main() {
 mymodule::function2(); // compile
 mymodule::function1(); // ne compile pas : error[E0603]: function `function1
` is private
}
```

Notons que mymodule::function2() peut être remplacé par le chemin absolu : crate::mymodule1::function2();, crate représentant la racine. Il est possible d'imbriquer des modules. Les modules fils peuvent appeler des méthodes du module parent grâce au mot-clé super :

```
mod mymodule {
 fn function1() {
  //...
 }

 pub mod childmodule {
  pub fn function1() {
   super::function1(); // appelle crate::mymodule::function1();
  }
 }
}
```

```rust
fn main() {
 mymodule::childmodule::function1();
}
```

Au niveau de l'arborescence et de l'organisation des fichiers, 3 choix sont possibles :

- déclarer le module dans le même fichier où il est utilisé (comme montré juste avant)
- déclarer le module dans un fichier à part mais dans le même dossier dans lequel il est utilisé
- déclarer le module dans un dossier et dans un fichier à part

Dans le 2d cas, l'arborescence ressemble à cela :

```
$ tree
.
└── hello-world
    ├── Cargo.toml
    └── src
        ├── main.rs
        └── mymodule.rs
```

Dans le 3e cas, il est courant de placer le code dans un fichier mod.rs, qui peut bien sûr être découpé en plusieurs autres petits fichiers si le code devient trop gros :

```
$ tree
.
└── hello-world
    ├── Cargo.toml
    └── src
        ├── main.rs
        └── mymodule
            └── mod.rs
```

Le contenu de mymodule.rs et mod.rs étant le suivant :

```rust
pub fn function1() {
 //...
}
```

, function1() peut être appelé depuis main.rs en déclarant l'utilisation du module mymodule :

```rust
mod mymodule; // même nom que le fichier ou dossier dans lequel se trouve function1

fn main() {
 mymodule::function1();
}
```

Tests

Les tests peuvent être lancés directement via cargo test.
Il existe 2 types de tests :

- les tests unitaires
- les tests d'intégration

Tests unitaires

Il est courant de retrouver ces tests unitaires dans le même fichier ou répertoire que le code testé, dans un sous-module dédié. L'idée est d'avoir les tests unitaires au plus près du code source. L'attribut #[cfg(test)] permet de marquer un module en tant que module de tests, et l'attribut #[test] permet à cargo de connaître les tests à exécuter.

```rust
mod calculator {
 fn add(a: i32, b: i32) -> i32 {
  a + b
 }

 #[cfg(test)]
 mod tests {
   use super::*; // permet d'importer les fonctions du module parent

   #[test]
   fn test_add() {
     assert_eq!(add(1, 2), 3);
   }
 }
}
```

Tests d'intégration

Il est courant de retrouver les tests d'intégration, qui comme leur nom l'indique testent plusieurs composants plutôt que des fonctions unitairement, dans un dossier tests/ au même niveau que src/ :

```
$ tree
.
└── hello-world
    ├── Cargo.toml
    ├── src
    │   ├── main.rs
    │   └── tests
    │       └── integration_tests.rs
```

Release de crate

Une fois son code terminé et testé, il est possible de publier sa *crate* sur https://crates.io (équivalent du Maven central ou du *repository* npm), afin de pouvoir être utilisé par d'autres développeurs. Pour cela, le pré-requis est d'avoir un compte sur crates.io. 3 actions sont nécessaires, et toutes réalisables avec cargo en 3 étapes :

- cargo login pour s'authentifier sur crates.io
- cargo package pour packager sa crate et pouvoir la déployer
- cargo publish pour déployer sa crate

C'est tout ! Avant de publier, vérifiez bien que votre manifeste Cargo.toml soit complet, et surtout respectez bien la gestion sémantique de version (https://semver.org) pour les futurs utilisateurs !

Fonctions avancées

Macros

Une macro est un genre de méthode, reconnaissable par le point d'exclamation à la fin du nom de celle-ci. Nous avons déjà utilisé des macros précédemment : println!, vec! et assert_eq!. Une macro permet de factoriser du code lorsque cela n'est pas possible par des moyens classiques. L'autre différence est que le code d'une macro est **étendu** avant la compilation du code, permettant une sorte de métaprogrammation (programmation de programmation).
Comme exemple pratique, écrivons une macro permettant d'initialiser une HashMap, de la même manière que vec! permet d'initialiser un Vec en une ligne. Sans macro, l'initialisation d'une HashMap avec des valeurs se fait de la manière suivante :

```rust
use std::collections::HashMap;
let mut map: HashMap<i32, &str> = HashMap::new();
map.insert(1, "un");
map.insert(2, "deux");
map.insert(3, "trois");
//...
```

Ce code est difficilement simplifiable. Nous aimerions disposer d'une macro `hashmap![1 => "un", 2 => "deux", 3 => "trois"];` qui permettrait en une ligne d'initialiser une telle `HashMap`. Eh bien, cela est possible :

```rust
macro_rules! hashmap {
  ($($key: expr => $value: expr),*) => {{
    use std::collections::HashMap;
    let mut map = HashMap::new();
    $(map.insert($key, $value);)*
    map
  }}
}
```

Sans rentrer dans le détail de la syntaxe des macros qui pourrait prendre un chapitre à elle seule, l'expression `macro_rules! hashmap` permet de définir une macro `hashmap!`. Cette macro prend en paramètre de 0 à *n* éléments (*), sous forme d'un tableau, comme `vec!`. La différence est que chacun des éléments doit être de la forme `key => value` (`$key: expr => $value: expr`), où `expr` désigne n'importe quelle expression, et que l'insertion dans la map `map` est répétée autant de fois que d'éléments du tableau (`$(map.insert($key, $value);)*`). Finalement, la map peuplée est renvoyée par la macro.
Le code précédemment écrit devient désormais :

```rust
let map = hashmap![1 => "un", 2 => "deux", 3 => "trois"];
println!("{:?}", map);
```

et affiche (l'ordre des éléments n'est pas garanti, cf la section sur les `HashMap`) :

```
{1: "un", 3: "trois", 2: "deux"}
```

Avant la compilation, la macro est **étendue** vers le code suivant (visible avec `cargo expand`, nécessite l'installation de `cargo-expand` : https://github.com/dtolnay/cargo-expand) :

```rust
let map =
 {
   use std::collections::HashMap;
   let mut map = HashMap::new();
   map.insert(1, "un");
   map.insert(2, "deux");
   map.insert(3, "trois");
   map
 };
```

Programmation fonctionnelle

En plus du *pattern matching* vu plus tôt, nous nous concentrerons ici sur 1 élément bien pratique : les méthodes définies sur les itérateurs.
Les itérateurs ont des propriétés particulières (ils sont *lazy*, c'est-à-dire qu'aucun calcul n'est effectué tant qu'une action comme `next`, `collect`, etc n'est appelée) disposent de méthodes intéressantes pour manipuler les éléments : `filter`, `map`, `take`, `for_each`, etc, qui seront familières aux développeurs d'autres langages où la programmation fonctionnelle occupe une part importante (Haskell, Scala, …).
Voici un exemple de leur utilisation. Notons la syntaxe concise :

```rust
let a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
a.iter()
 .filter(|&x| x % 2 == 0) // ne prend que les
nombres pairs, voir tip suivant
 .take(3) // ne prend que les 3 premiers éléments
 .map(|x| x * 2) // multiplie par 2
 .for_each(|x| println!("{}", x));
```

affiche :

```
4
8
12
```

> **Tip** : `filter` prend en paramètre une référence, d'où le `|&x|` présent dans le paramètre de la closure. `filter` ne prend pas l'*ownership* des valeurs mais juste une référence, car il se charge juste de décider quelles valeurs doivent passer à l'étape suivante.

Threads

Considérons la tâche suivante :

```rust
use std::thread;
use std::time;

fn long_task(task_number: i32) {
  println!("Executing task {}", task_number);
  thread::sleep(time::Duration::from_secs(3)); // pause de 3 secondes
pour illustrer
  println!("Task {} executed", task_number);
}
```

Il est possible d'exécuter les 3 fonctions `long_task` de manière simultanée à l'aide de **threads** :

```rust
fn execute_long_tasks_multiple_threads(n: i32) {
 let mut threads = Vec::new();

 for i in 1..=n {
  threads.push(
   thread::spawn(move || long_task(i)) // "move" force la closure à
prendre la propriété de i
  )
 }

 // on attend que les threads se terminent
 for t in threads {
  t.join();
 }
}
```

Si on remplace `execute_long_tasks_sequentially(3)` par `execute_long_tasks_multiple_threads(3)`, on obtient :

```
Executing task 1
Executing task 3
Executing task 2
```

```
Task 1 executed
Task 3 executed
Task 2 executed
Elapsed time : 3 seconds
```

Le temps total est bien de 3 secondes, contre 9 si les tâches avaient été exécutées séquentiellement.

Asynchronisme

Depuis la version 1.39 (novembre 2019), Rust dispose des mots-clés async et await (concepts qui existent déjà en JavaScript ou C#) qui permettent de faire de la **programmation asynchrone**. Utilisée surtout pour de l'IO (requêtes HTTP, lectures de fichiers), la programmation asynchrone permet de ne pas bloquer et d'éxécuter d'autres actions pendant qu'une *future* est bloquée (en attente). Illustrons cela avec un exemple. Nous utiliserons les dépendances suivantes :

```
[dependencies]
futures = "0.3.5"
isahc = "0.9.8" # make http requests
```

Le code suivant définit 2 fonctions asynchrones (get_web_page et say_hello). get_web_page appelle la méthode get_async, qui retourne une Future, et await qui permet de ne pas bloquer l'exécution tant que la réponse n'est pas retournée, puis affiche le statut HTTP de la requête. La méthode say_hello quant à elle affiche simplement Hello world !.

```
async fn get_web_page(url: &str) {
  let response = isahc::get_async(url).await;
  match response {
    Ok(r) => println!("{:?}", r.status()),
    Err(_) => println!("Error")
  };
}

async fn say_hello() {
  println!("Hello world !");
}

async fn async_main() {
  let f1 = get_web_page("http://example.com");
  let f2 = say_hello();

  futures::join!(f1, f2);
}
```

```
fn main() {
  futures::executor::block_on(async_main());
}
```

Définir une fonction avec le mot-clé async est juste un raccourci pour dire que la fonction retourne un objet de type Future. La fonction main ne pouvant pas être définie comme asynchrone (async), nous définissons une méthode async_main appelant futures::join! sur nos 2 objets Future, qui retourne un objet Future. À son tour, le main appelle notre async_main, bloquant le thread principal et exécutant les 2 objets Future jusqu'à leur complétion. Le résultat est le suivant :

```
Hello world !
200
```

Comme on peut le constater, lors de l'attente du retour de la méthode get_web_page (Future f1), la Future f2 a été exécutée (affichage de Hello world !). Le thread principal n'a pas été bloqué par l'appel HTTP pouvant prendre un certain temps.

Conclusion

À travers ce dossier, j'espère vous avoir donné envie de vous intéresser à Rust. En premier lieu alternative à C ou C++ pour la programmation système (pour exemple, il a été récemment évoqué l'éventualité d'écrire des parties du noyau Linux en Rust, rien que ça !), Rust est en réalité un langage complet pouvant être utilisé pour n'importe quelle application nécessitant des performances stables (absence de GC) ou *memory-safe*. Derrière Rust se cache aussi toute une partie *unsafe* dont nous n'avons pas discuté dans cet article, car trop éloigné des principes de base de Rust et pouvant même être considéré comme un langage à part entière.

Dans tous les cas, j'encourage vivement les développeurs à se pencher sur ce langage et à faire des erreurs : les messages d'erreurs explicites aideront à mieux apprivoiser le langage, et se demander d'où vient l'erreur permettra de mieux comprendre Rust et les éventuels problèmes évités grâce au compilateur. Même si à première vue il peut apparaître comme un ennemi, le compilateur doit être plutôt vu comme un allié vous évitant des soucis à l'exécution !

Rust reste un langage en plein développement, et la communauté autour s'efforce à construire un langage pour le futur. Même si la courbe d'apprentissage est raide - surtout lorsqu'on tombe sur des problèmes autour des concepts d'*ownership* et de *borrowing* - jouer avec Rust reste le meilleur moyen de s'en faire une opinion !

Jean-Christophe Riat

Passionné depuis le lycée par l'informatique, j'enseigne l'intelligence artificielle depuis plus de 20 ans à l'EPSI Paris, 1re école d'informatique créée en France par des professionnels. Comme développeur, je suis enthousiasmé par les progrès en apprentissage machine, d'autant qu'internet rend accessibles toutes les ressources pour mettre en œuvre les techniques les plus récentes, comme celle du « Deep Learning », ou réseaux de neurones profonds en français.

TensorFlow/Keras : accès facile au Deep Learning pour les développeurs Python

Dans l'univers de l'intelligence artificielle, le « machine learning » a connu ces dernières années un développement important, avec les progrès fulgurants des techniques de « deep learning ». Le déploiement d'applications a été largement favorisé par la disponibilité d'outils logiciels mis à disposition gratuitement par des géants de l'internet comme Google, Facebook ou Baïdu. Dans cet article, nous illustrons la facilité d'accès à la bibliothèque TensorFlow/Keras de Google avec l'écriture d'un programme Python capable, avec moins de 20 instructions, d'identifier des chiffres manuscrits sur des images.

Le but de l'intelligence artificielle est de reproduire, avec un ordinateur, tout ou partie des capacités de l'intelligence humaine ou animale. Une des premières idées a consisté à modéliser sous forme numérique les éléments connus sur le fonctionnement du cerveau, composé de milliards de milliards de neurones.

Modèle de « neurone formel »

Un neurone est une cellule élémentaire avec un fonctionnement autonome, composé d'un noyau, d'un axone et de milliers de dendrites **(cf. encadré n°1)**. À chaque instant le neurone produit un signal électrique sur son axone à partir des influx reçus sur ses dendrites. Le signal généré est à son tour, transmis vers d'autres neurones, via les connexions (appelées synapses) entre axone et dendrites, et ainsi de suite. Cette logique de fonctionnement en continu explique pourquoi notre cerveau reste toujours en effervescence !

En 1943, les neurobiologistes McCulloch & Pitts proposent la première modélisation mathématique du neurone. Comme son frère biologique, le neurone formel calcule une valeur en sortie à partir des valeurs reçues en entrées. La représentation et la logique de fonctionnement utilisées dans les modèles actuels sont décrites dans **l'encadré n°2** (il ne faut pas être effrayé par les notations mathématiques, car les seuls calculs utilisés sont des additions et des multiplications !).

Encadré n°1 : connexions des neurones biologiques

Un cerveau est constitué de milliards de milliards de neurones interconnectés. Ces cellules nerveuses échangent et traitent en continu des influx électriques au travers de connexions appelées synapses :

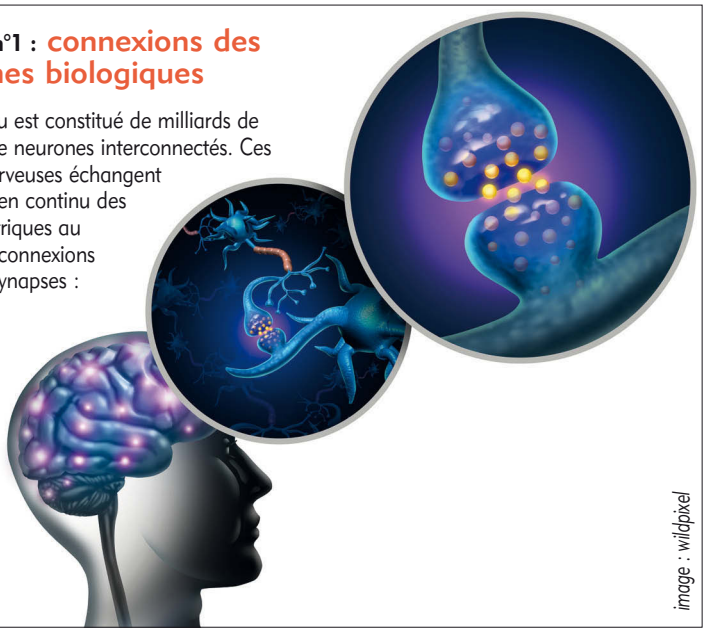
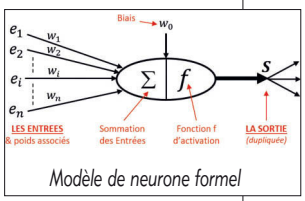


image : wildpixel

Encadré n°2 : description et fonctionnement du neurone formel

Modèle du neurone :



Modèle de neurone formel

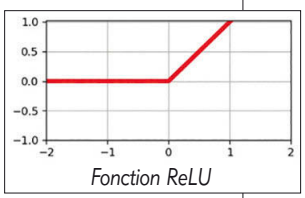
$e_1 \cdots e_n$ valeurs en entrée du neurone

S : valeur en sortie du neurone

$w_0, w_1, \cdots w_n$: paramètres du neurone (biais et poids), qui sont des valeurs constantes à fixer en fonction du comportement souhaité

f : fonction d'activation du neurone

De nombreuses formes de fonctions d'activation existent ; une des plus utilisées est ReLU(x) qui retourne x si x est positif et 0 si x est négatif.



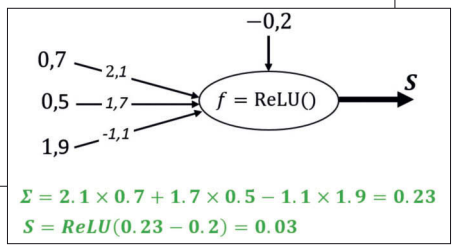
Fonction ReLU

Fonctionnement du neurone :

Le calcul se décompose en 2 étapes. Dans un premier temps, on calcule la « somme pondérée des entrées » qui est la somme des valeurs de chaque entrée multipliée par son poids. Ce résultat auquel, on ajoute le biais, est ensuite transmis en paramètre à la fonction d'activation du neurone pour calculer la sortie.

Soit en notation mathématique : $S = f(w_0 + \sum_{i=1}^{n} w_i \times e_i)$

Exemple de calcul :



Réseau de neurones de type perceptron multicouche

Construire un modèle de réseau de neurones artificiel consiste à assembler des neurones formels. Par analogie, les seules connexions possibles sont des liens orientés depuis les sorties des neurones vers les entrées d'autres neurones. Il existe de nombreuses topologies de réseaux, c'est-à-dire de manières d'organiser ces connexions. Nous avons choisi de développer dans cet article le modèle appelé perceptron multicouche (PMC en français ou MLP pour "Multi Layer Perceptron" en anglais). Ce réseau a été introduit par Rosenblatt en 1957/58 dans une version monocouche et a ensuite été utilisé pour des applications de « deep learning » avec des réseaux multicouche de grandes tailles.

Un réseau de type perceptron permet de représenter un comportement entrées/sorties, c'est-à-dire une logique de calculs entre des valeurs présentées en entrée et des résultats produits en sortie. Un modèle de ce type peut s'utiliser dans une très large variété de problèmes :

- prévision à 24h de la température à partir des températures et des degrés d'humidité mesurés les jours précédents,
- calcul de la valeur d'un appartement à partir du nombre de pièces, de sa surface et de sa situation géographique,
- diagnostic d'un risque de diabète à partir de données physiologiques (poids, âge, taux de glucose, indice de masse corporelle…),
- reconnaissance d'un visage sur une image à partir du codage de l'image avec les niveaux de couleur de chaque pixel,
- etc.

Ce réseau est organisé en couches et ne comporte que des connexions entre les neurones de couches adjacentes **(cf. encadré n°3)**. Cette architecture présente l'avantage de pouvoir facilement faire « fonctionner le réseau », c'est-à-dire calculer les valeurs en sorties à partir de valeurs présentées en entrées : chaque sortie de neurone est calculée successivement en respectant l'ordre des couches. Les entrées du réseau permettent de calculer les sorties des neurones de la 1re couche ; les valeurs de ces sorties sont utilisées en entrées de la deuxième couche et ainsi de suite jusqu'à la dernière couche où les sorties des neurones correspondent aux sorties du réseau.

Le choix du nombre de couches et du nombre de neurones par couche est libre. Souvent pour un problème donné on commence à expérimenter des réseaux de petite taille (1 ou 2 couches avec entre 5 et 15 neurones par couches) et on augmente la taille du réseau et le nombre de neurones jusqu'à obtenir les performances souhaitées.

Apprentissage pour un perceptron multicouche

Le fonctionnement d'un réseau dépend du nombre et de l'organisation des neurones, mais également des valeurs choisies pour les paramètres de chaque neurone (valeurs des poids et des biais). Tout l'intérêt du perceptron provient de l'existence d'un algorithme d'apprentissage pour calculer les valeurs de ces paramètres à partir d'exemples représentatifs du fonctionnement souhaité. La première version de cet algorithme a été présentée en 1986 par Rumelhart, Hinton & Williams et constitue une étape majeure dans le développement du deep learning (avec un effet retard, car il a fallu attendre plusieurs années pour que les puissances de calcul disponibles permettent l'exploitation de cet algorithme avec des réseaux et des bases d'exemples de grande taille !).

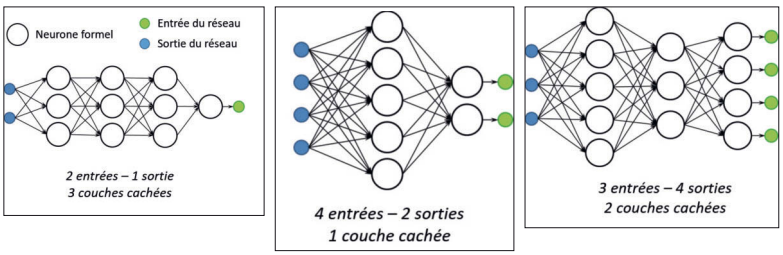
Détailler le fonctionnement de cet algorithme nécessite des connaissances mathématiques avancées et n'a pas grand intérêt pour développer de premières applications. En effet, les outils pour le deep learning intègrent des bibliothèques avec les méthodes d'apprentissage les plus récentes déjà programmées !

Encadré n°3: architecture du perceptron multicouche

Un perceptron multicouche est un réseau de neurones avec la topologie suivante :

- les neurones sont organisés en couches; chaque neurone dispose d'une fonction d'activation et d'un biais
- les valeurs des entrées sont reliées aux neurones de la première couche appelée ***couche d'entrée***,
- les valeurs de sorties sont les résultats des neurones de la dernière couche appelée ***couche de sortie***,
- les neurones de chaque couche sont totalement connectés à ceux de la couche suivante et il n'existe aucune autre connexion dans le réseau; un poids est associé à chaque connexion.

Un perceptron multicouche peut comporter un nombre de couches quelconque et un nombre de neurones par couche également quelconque. Les couches, autres que celles d'entrée et de sortie, sont appelées ***couches cachées***



Remarque: chaque "cercle" sur les figures correspond à un neurone tel que décrit dans l'encadré n°2; même si un neurone comporte plusieurs "flèches" en sortie, il n'a toujours qu'une seule sortie dont la valeur est dupliquée sur chaque flèche !

Étapes de mise en œuvre d'un perceptron multicouche

Après avoir introduit les notions théoriques de base sur le perceptron multicouche, il est nécessaire de bien identifier les étapes à suivre pour utiliser un tel réseau de neurones pour traiter un problème :

- Analyse du problème pour choisir les valeurs à utiliser en entrées du réseau et les résultats attendus en sorties,
- Identification d'une base d'exemples pour l'apprentissage, c'est-à-dire d'un ensemble de couples composés des valeurs d'entrées avec les sorties associées. Généralement cette base est divisée en deux groupes : les exemples effectivement utilisés pour l'apprentissage et ceux employés pour tester les performances du réseau. Le but est de comparer, après apprentissage, les résultats calculés avec ceux fournis par ces exemples de test (cf. dernier point).
- Choix de la topologie du réseau c'est-à-dire du nombre de couches et du nombre de neurones par couche,
- Phase d'apprentissage (également appelée entraînement) pour calculer des valeurs des poids et des biais à partir des exemples de la base d'apprentissage,
- Phase de test pour évaluer les performances du réseau sur d'autres exemples que ceux utilisés pour l'entraînement. Si les résultats sont satisfaisants le réseau pourra être exploité avec d'autres données. Dans le cas contraire il faut revenir à l'étape 2 ou 3 pour modifier, soit la base d'apprentissage, soit la topologie du réseau.

Après la théorie, passons à la pratique avec l'écriture d'un programme Python pour reconnaître, au moyen d'un perceptron multicouche, des chiffres manuscrits représentés sur des images. Pour développer cette application, il est nécessaire de traiter chacune des étapes de la démarche explicitée précédemment, en s'appuyant sur deux ressources disponibles sur internet :

- La base d'images de chiffres manuscrits MNIST **(cf. encadré n°4)**, qui contient des fichiers directement exploitables pour l'apprentissage et les tests,
- La bibliothèque TensorFlow/Keras développée par Google pour faciliter l'écriture d'applications de Deep Learning en langage Python **(cf. encadré n°5)**.

Analyse du problème pour choisir les entrées et les sorties

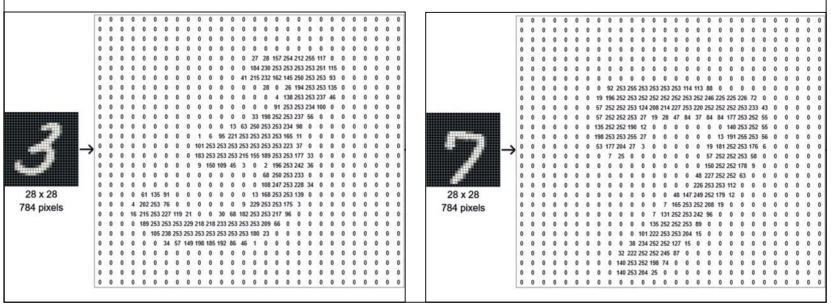
L'objectif est de construire un réseau de neurones pour reconnaître des chiffres manuscrits à partir d'images. Dans la base MNIST, les visuels sont représentés avec des matrices de 28 x 28 pixels, chaque pixel étant codé en niveaux de gris avec une valeur entre 0 et 255. Le réseau doit prendre en entrées les informations à traiter, donc une solution simple est d'utiliser 28 x 28 = 784 entrées, chaque valeur correspondant à un pixel de l'image. La reconnaissance de chiffres est un problème de classification. Pour chaque image dix réponses sont possibles, chacun des chiffres de 0 à 9. Il s'agit donc d'affecter une image parmi dix classes. Le plus simple est de définir autant de sorties pour le réseau que de classes possibles, ici dix. La valeur de chaque sortie s'interprète alors comme un niveau de probabilité de l'appartenance de l'image à la classe. Avec cette représentation, la conclusion du réseau correspond à la sortie avec la probabilité la plus élevée. Au final, le réseau de neurones comporte 784 entrées et 10 sorties **(cf. encadré n°6)**.

Encadré n°4 : base d'images MNIST de chiffres manuscrits

La base MNIST des chiffres manuscrits est détaillée sur le site : http://yann.lecun.com/exdb/mnist/
Elle est composée d'images de chiffres recueillis sur des formulaires administratifs renseignés manuellement aux États-Unis (le chiffre 7 est souvent représenté sans la barre horizontale, et le 1 avec uniquement un trait vertical).
De nombreux tutoriels utilisent cette base pour illustrer l'utilisation de réseaux de neurones pour des problèmes de classification, car elle est parfaitement conçue pour :

- Les images sont divisées en 2 groupes, avec dans chaque groupe une répartition homogène des chiffres de 0 à 9: un premier groupe de 60000 exemples pour l'apprentissage et un second groupe de 10000 images pour les tests.
- Les images sont "nettoyées" c'est-à-dire qu'elles ont toutes le même format sans contenu erroné,
- Les images sont "étiquetées" c'est-à-dire que la valeur du chiffre à reconnaître est renseignée.

Les images codées en niveau de gris ont un format 28x28 pixels; le site https://ml4a.github.io/demos/f_mnist_input/ permet de les visualiser, comme sur les deux exemples ci-dessous :



Identification d'exemples pour l'apprentissage

Un des intérêts des données de MNIST est d'être organisées avec une base de 60 000 images destinées à l'apprentissage et une base complémentaire de 10 000 images pour les tests. Chaque base est composée d'un fichier d'images avec les valeurs des 28 x 28 = 784 pixels, et d'un second fichier avec, pour chaque image, la valeur du chiffre à identifier (appelée étiquette ou label). La bibliothèque TensorFlow/Keras comporte un module spécifique, appelé '**.datasets**', avec des jeux de données d'apprentissage, parmi lesquels la base MNIST. Les premières instructions du programme utilisent ce module pour charger les exemples en mémoire :

```
# Importation des modules TensorFlow & Keras
import tensorflow as tf
# Importation du module graphique
import matplotlib.pyplot as plt
# Chargement en mémoire de la base de données des caractères MNIST
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

Après cette dernière instruction, la variable **x_train** (resp. **x_test**) contient les images pour l'apprentissage (resp. le test) et **y_train** (resp. **y_test**) les valeurs associées des chiffres à identifier.
x_train (resp. **x_test**) est un tableau de 60 000 (resp. 10 000) lignes. Chaque ligne correspond à une image codée avec une matrice de 28 x 28 valeurs entières entre 0 et 255. Pour un meilleur fonctionnement des neurones formels, il est préférable de réaliser un changement d'échelle pour obtenir des valeurs réelles entre 0 et 1 :

```
# Transformation d'entiers en valeurs réelles entre 0.0 et 1.0
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Les images sont mémorisées sous forme matricielle ; il faut les adapter à une représentation vectorielle pour coïncider avec le choix du format des entrées du réseau :

```
# Transformation de matrices 28x28 en vecteurs de 784 valeurs
x_train = x_train.reshape(60000, 784)
x_test  = x_test.reshape(10000, 784)
```

Encadré n°5 : développement Python avec TensorFlow/Keras

L'utilisation des bibliothèques TensorFlow/Keras est possible dans tous les environnements de développement Python (Windows, macOS ou Linux), sous réserve d'installer le module TensorFlow.
Pour expérimenter le programme de cet article, nous conseillons deux approches :

- Installer un environnement Python, par exemple celui disponible sur le site www.anaconda.com (distribution "Individual Edition" gratuite) et travailler en local sur son ordinateur
- Ou utiliser "le cloud" en se connectant sur le site « Colaboratory » avec un compte Google (https://colab.research.google.com/notebooks/intro.ipynb); l'avantage de cette solution est de ne nécessiter aucune installation locale puisque tout fonctionne en débarqué via l'utilisation d'un navigateur internet

y_train (resp. **y_test**) est un tableau de 60 000 (resp. 10 000) valeurs entières entre 0 et 9 qui correspondent aux chiffres à identifier pour les images du tableau **x_train** (resp. x_**test**). Étant donné que le réseau comporte 10 valeurs en sorties, interprétées comme un taux de reconnaissance pour chacun des chiffres de 0 à 9, il est nécessaire de modifier la représentation des tableaux **y_train** et **y_test** pour coïncider avec ce codage (par exemple l'identification du chiffre 3, produit le vecteur [0,0,0,1,0,0,0,0,0,0] en sortie du réseau) :

```
# Transformation des labels de sortie en vecteurs pour une classification en 10 valeurs
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

Choix de la topologie du réseau

Nous choisissons de compléter les entrées/sorties du réseau retenues précédemment, avec 2 couches cachées de 50 neurones. Ce choix est arbitraire, et pourra être modifié après la phase de tests si les performances sont jugées insuffisantes. Avec ces éléments, la topologie du perceptron est complètement définie (sa représentation est détaillée dans **l'encadré n°6**).

Pour utiliser un perceptron, le module Keras met à disposition une classe Python appelée '**.Sequential'.** La suite du programme consiste à instancier un objet de cette classe et à utiliser les méthodes disponibles pour décrire et exploiter le réseau. La méthode '**.add'**, qui permet d'ajouter une couche de neurones, est employée pour définir la 1ère couche composée de 50 neurones, la 2e couche avec également 50 neurones et la couche de sortie avec 10 neurones :

```
# Création d'un réseau multicouche
MonReseau = tf.keras.Sequential()
# Description de la 1re couche
MonReseau.add(tf.keras.layers.Dense(
    units=50,          # 50 neurones
    input_shape=(784,),   # nombre d'entrées (car c'est la 1re couche)
    activation='relu'))   # fonction d'activation
# Description de la 2e couche
MonReseau.add(tf.keras.layers.Dense(
    units=50,          # 50 neurones
    activation='relu'))   # fonction d'activation
# Description de la couche de sortie
MonReseau.add(tf.keras.layers.Dense(
    units=10,          # 10 neurones
    activation='softmax')) # fonction d'activation (sorties sur [0,1])
```

Les neurones des deux premières couches utilisent une fonction d'activation « ReLU » (cf. encadré n°2), alors que ceux de la couche de sortie sont activés avec la fonction « Softmax ». Cette dernière est mieux adaptée pour représenter des taux de reconnaissance, car elle transforme les sorties des neurones en niveaux de probabilité dont la somme vaut 1.

Phase d'entraînement

L'apprentissage fonctionne avec l'appel à deux méthodes, utilisées ici avec les paramètres les plus simples (ceux couramment employés).

'**.compile'** décrit les caractéristiques du fonctionnement de l'apprentissage avec l'algorithme mathématique à utiliser, la méthode de calcul d'erreurs pour la convergence de l'algorithme et celle pour l'évaluation des performances du réseau :

```
# Configuration de la procédure pour l'apprentissage
MonReseau.compile(optimizer='adam',          # algo d'apprentissage
        loss='categorical_crossentropy', # mesure de l'erreur
        metrics=['accuracy'])        # mesure du taux de succès
```

'**.fit'** lance l'apprentissage avec les données **x_train** et **y_train** ; étant donné que l'algorithme fonctionne de manière itérative, le paramètre **epoch** fixe le nombre d'itérations :

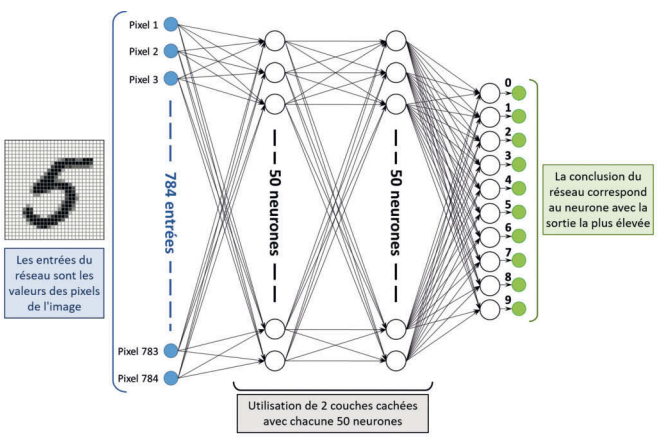
```
# Apprentissage du réseau
MonReseau.fit(x=x_train, # données d'entrée pour l'apprentissage
      y=y_train, # sorties désirées associées aux données d'entrée
      epochs=10) # nombre de cycles d'apprentissage
```

L'exécution produit à l'écran un résultat du type suivant :

```
Train on 60000 samples
Epoch 1/10
60000/60000 [==============] - 4s 63us/sample - loss: 0.2953 - accuracy: 0.9161
Epoch 2/10
60000/60000 [==============] - 3s 56us/sample - loss: 0.1399 - accuracy: 0.9590
Epoch 3/10
60000/60000 [==============] - 3s 56us/sample - loss: 0.1045 - accuracy: 0.9685
Epoch 4/10
60000/60000 [==============] - 3s 55us/sample - loss: 0.0840 - accuracy: 0.9742
Epoch 5/10
60000/60000 [==============] - 3s 56us/sample - loss: 0.0702 - accuracy: 0.9786
Epoch 6/10
60000/60000 [==============] - 3s 56us/sample - loss: 0.0586 - accuracy: 0.9816
Epoch 7/10
60000/60000 [==============] - 3s 56us/sample - loss: 0.0517 - accuracy: 0.9834
Epoch 8/10
60000/60000 [==============] - 3s 55us/sample - loss: 0.0441 - accuracy: 0.9856
Epoch 9/10
60000/60000 [==============] - 3s 58us/sample - loss: 0.0384 - accuracy: 0.9876
Epoch 10/10
60000/60000 [==============] - 3s 55us/sample - loss: 0.0326 - accuracy: 0.9895
```

Chaque ligne correspond à un cycle d'apprentissage, avec quatre informations : le nombre d'exemples traités (60 000), la durée des calculs, la valeur de la mesure d'erreur utilisée par l'algorithme (loss) et le taux de reconnaissance (accuracy) sur les images d'apprentissage. Durant l'entraînement, l'erreur « loss » diminue et la performance de reconnaissance « accuracy » s'améliore. À la fin du 10e cycle, les valeurs calculées

Encadré n°6 : perceptron utilisé dans le programme



pour les poids et les biais permettent de reconnaître correctement 98,95% des 60 000 images de la base d'apprentissage.

Phase de test

Obtenir de bonnes performances lors de l'entraînement est assez logique puisque les poids et les biais sont calculés par l'algorithme pour faire fonctionner au mieux le réseau sur les données d'apprentissage. Le plus important est la mesure de la performance de reconnaissance sur les images non apprises de la base de test. Cette évaluation s'effectue avec la méthode '**.evaluate**' :

```
# Test du réseau sur des exemples non utilisés pour l'apprentissage
perf=MonReseau.evaluate(x=x_test, # données d'entrée pour le test
                        y=y_test) # sorties désirées pour le test
print("Taux d'exactitude sur le jeu de test: {:.2f}%".format(perf[1]*100))
```

L'exécution de ces instructions produit à l'écran un résultat du type ci-dessous, qui montre une fiabilité de reconnaissance de plus de 97 %, ce qui constitue un bon score.

```
10000/10000 [==============] - 0s 49us/sample - loss: 0.0999 - accuracy: 0.9709
Taux d'exactitude sur le jeu de test: 97.09%
```

Exploitation du réseau

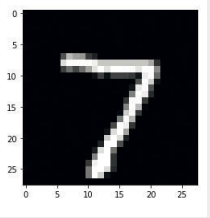
Ces résultats montrent que le réseau est capable de généraliser correctement, c'est-à-dire de reconnaître avec un taux élevé de succès des chiffres sur des images différentes de celles employées durant l'apprentissage. Il devient donc possible de l'utiliser avec un bon niveau de confiance pour des applications qui n'exigent pas une fiabilité à 100%, par exemple le tri automatique du courrier avec comparaison entre le code postal et la ville.

Pour terminer le programme, le code ci-dessous permet de choisir une image dans la base de test, et de l'afficher avec le chiffre identifié par le réseau de neurones :

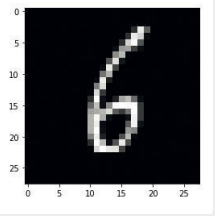
```
# Affichage de la conclusion du réseau sur des images unitaires
NoImg = 1
while NoImg != 0:
 # Saisie de la référence de l'image à identifier
 NoImg = int(input("Image à identifier (entre 1 et 10000 / 0 pour sortir): "))
 if NoImg != 0:
  # Affichage de l'image
  plt.imshow(x_test[NoImg-1].reshape(28,28), cmap='gray')
  plt.show()
  # Calcul et affichage de la conclusion du réseau
  print("==> chiffre:", MonReseau.predict_classes(x_test[NoImg-1:NoImg]))
```

Voici un exemple d'exécution :

```
Image à identifier (entre 1 et 10000 / 0 pour sortir): 1
```



```
==> chiffre: [7]
Image à identifier (entre 1 et 10000 / 0 pour sortir): 500
```



```
==> chiffre: [6]
Image à identifier (entre 1 et 10000 / 0 pour sortir): 0
```

Dans la mesure où elles utilisent le même format, ce programme peut fonctionner avec d'autres images de chiffres manuscrits en remplaçant le tableau **y_test**.

Conclusion

L'application détaillée dans cet article montre la facilité d'utilisation de la bibliothèque TensorFlow/Keras pour construire et exploiter des modèles de réseaux de neurones. Nous espérons que cet exemple vous aura donné envie d'approfondir l'utilisation de ces outils, soit avec la documentation disponible sur le site « tensorflow.org », soit avec les très nombreux tutoriels accessibles en ligne. Un premier exercice intéressant est de modifier dans le programme la topologie du perceptron pour améliorer sa performance. Nous avons vu qu'avec deux couches de 50 neurones, la performance en phase de test est légèrement supérieure à 97 % de classifications correctes, mais avec un réseau de taille plus importante il est possible d'atteindre les 99 %... à vos claviers !

Références

Documentations en ligne :

TensorFlow : https://www.tensorflow.org/

Keras : https://www.tensorflow.org/api_docs/python/tf/keras

Base d'images de chiffres manuscrits souvent utilisée pour montrer le principe de l'apprentissage machine :
http://yann.lecun.com/exdb/mnist/
https://ml4a.github.io/demos/f_mnist_input/

Environnement "Colaboratory" pour écrire et d'exécuter du code Python depuis un navigateur internet : https://colab.research.google.com/notebooks/intro.ipynb

Site de téléchargement de la version "Individual Édition" de la distribution anaconda
https://www.anaconda.com/products/individual

Site de l'école d'ingénierie informatique EPSI (membre de HEP Éducation) :
https://www.epsi.fr/

Publications historiques :

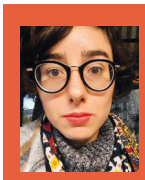
1943 "A logical calculus of the ideas immanent in nervous activity" Warren S. McCulloch & Walter Pitts:
https://web.csulb.edu/~cwallis/382/readings/482/mccolloch.logical.calculus.ideas.1943.pdf

1958 "The perceptron: a probabilistic model for information storage and organization in the brain" F. Rosenblatt:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf

1986 "Learning internal representation by error propagation" D.E. Rumelhart, G.E. Hinton & R.J. Williams:
https://web.stanford.edu/class/psych209a/ReadingsByDate/02_06/PDPVolIChapter8.pdf

Introduction au TALN (Traitement Automatique du Langage Naturel) avec spaCy

Le traitement automatique du langage naturel est une discipline de l'ingénierie informatique permettant d'analyser et d'interpréter le langage humain, écrit et oral. Le TALN ou NLP *(Natural Language Processing)* permet donc, par exemple, d'extraire la morphologie d'une phrase, de générer la traduction d'un texte, de créer une synthèse vocale ou de faire de l'extraction d'informations.



Constance-Louise Gauriau

Développeuse Python et React.js pour le compte de la société benext. Elle est actuellement en mission de développement chez Dior. Elle est diplômée du parcours Python d'OpenClassrooms.

Benoît Prieur

Développeur indépendant pour sa société Soarthec. Il est l'auteur de plusieurs livres à propos de développement logiciel publiés aux éditions ENI. Il est par ailleurs mentor Python pour OpenClassrooms.

Afin d'illustrer cela, on utilisera la bibliothèque Python spaCy. Cette librairie open source permet de réaliser les différents exemples précités. Dans le cadre de cet article, on s'intéressera à l'extraction d'informations issues d'un texte écrit en langue française.
L'extraction d'informations consiste à faire ressortir des éléments d'un document textuel ou sonore. Pour ce faire, les outils de TALN, tels que spaCy, permettent d'effectuer de la *tokenisation*. Il s'agit de détecter, découper et regrouper des segments d'un texte : des mots, des phrases ou des paragraphes.

La librairie spaCy incontournable du TALN

Nous utiliserons cette librairie Python développée depuis février 2015 par Matthew Honnibal et Ines Montani. Depuis sa création, une communauté active de plus de 400 contributeurs et contributrices s'est constituée. Le contenu de spaCy s'est considérablement enrichi pour apparaître aujourd'hui comme l'outil de référence en matière de traitement automatique du langage naturel. De nombreux sous-modules traitent d'une grande variété d'aspects, parmi lesquels :

- *Thinc* : bibliothèque d'apprentissage automatique ;
- *sense2vec* : bibliothèque de calcul de similitudes ;
- *deplaCy* : bibliothèque de visualisation d'arbre de dépendances d'un texte ou d'une phrase ;
- deplaCyEnt : bibliothèque de visualisation de reconnaissance d'entités nommées.

Une des dernières versions stables de spaCy est la 2.2.4. C'est celle qui est utilisée ici avec Python 3.8. Côté spaCy, les mises à jour fréquentes apportent corrections de bugs et améliorations de diverses composantes linguistiques. Les performances de la librairie spaCy lui ont permis de se placer parmi les parsers syntaxiques les plus rapides, ceci dès 2015 avec la version 1x. Ce classement fut confirmé en 2017 après la mise à disposition de la version 2x.
De nombreux projets utilisent spaCy : des projets de recherche dans le domaine du TALN, mais également des projets parties prenantes de solutions commerciales. On peut citer le projet Ludwig de la société Uber qui constitue un framework open source de deep learning. La bibliothèque spaCy est également massivement utilisée par Microsoft, ou encore par le laboratoire d'IA d'Intel par le biais du projet NLP Architect.

Il est impossible de présenter l'ensemble des possibilités de spaCy en quelques pages, le choix a donc été fait de se focaliser sur l'une des fonctionnalités fondamentales de Spacy : la tokenisation.

D'abord, nous commencerons par un exemple basique, permettant d'extraire des tokens à partir d'une simple phrase ; puis on effectuera un projet plus complet, avec spaCy et Flask, afin d'identifier des mots correspondant à des toponymes (noms propres de lieux) dans un texte fourni en entrée. On affichera ensuite ces toponymes sur une carte.

La tokenisation avec spaCy

On crée un répertoire pour notre projet, puis un environnement virtuel que l'on active :

```
python3 -m venv env
source env/bin/activate
```

Puis on installe spaCy tout en téléchargeant l'un de ses modèles (nommé "sm") dans sa version française :

```
pip install -U spacy
python -m spacy download fr_core_news_sm
```

Une fois cette étape réalisée, on crée le fichier Python ici nommé token_example.py en racine du projet :

```
touch token_example.py
```

Dans ce script Python, on importe spaCy puis on utilise la méthode load(), qui permet de charger le modèle ("sm") que l'on a téléchargé précédemment.

```
#! /usr/bin/env python
import spacy
nlp = spacy.load("fr_core_news_sm")
```

Désormais, on peut analyser une phrase et en extraire un certain nombre d'informations. Pour le moment, on crée juste une liste de tuples associant chaque token (c'est-à-dire chaque fragment) à ce que l'on appelle son étiquetage morpho-syntaxique.

```
sentence = "J'aime lire Programmez!"
doc = nlp(sentence)
```

En passant la phrase en argument de la fonction nlp() on appelle le fichier meta.json du modèle pré-entraîné de spaCy "fr_core_news_sm", ce qui lui permet d'initialiser le modèle

spacy.lang.fr.French et d'utiliser les composants par défaut du pipeline : **tagger, parser et entity recognizer.**

La valeur de retour, nommée ici doc est un objet **Doc** créé par spaCy. Il s'agit d'une séquence d'objets de type **Token,** qui correspondent aux différents segments de la phrase passée en argument de nlp().

Ces différents tokens sont générés de la manière suivante : d'abord le texte est découpé selon les espaces rencontrés (équivalent à un appel de fonction split(' ')). Ensuite, spaCy itère sur ces tokens et applique deux conditions :
- la première correspond à la recherche d'exceptions.
- la seconde aux préfixes et aux suffixes.

Cela a pour but de détecter si le token doit être de nouveau découpé ou non.
Affichons la liste de tuples de tokens avec leur étiquetage morpho-syntaxique suivi de la liste des entités détectées dans notre objet **Doc**.

```
list_of_token_with_pos = [(token.text, token.pos_) for token in doc]
list_of_entities = [token.text for token in doc.ents]
print(list_of_token_with_pos)
print(list_of_entities)
```

Affichons le résultat dans la console en tapant :
python token_example.py

On obtient :

```
[("J'", 'PRON'), ('aime', 'VERB'), ('lire', 'VERB'), ('Programmez', 'PROPN'), ('!', 'PUNCT')]
['Programmez!']
```

Chaque token est caractérisé grammaticalement : "j'" est un pronom, "aime" est un verbe et "Programmez!" est bien caractérisé comme un nom propre.

Extraction et affichage de lieux issus du dictionnaire Le Maitron avec spaCy, Flask, bs4 & geopy

Le Maitron et les noms de lieux

Le dictionnaire *Le Maitron en ligne* propose des milliers de biographies relatives au mouvement ouvrier, au mouvement social ou encore à la résistance française.

Chacune de ces biographies contient des lieux : le lieu de naissance, de mort, mais aussi les différentes villes où le personnage a milité, combattu, etc. Pour une biographie du Maitron donnée, le programme conçu recherchera, grâce à spaCy, les noms propres de lieux et les affichera sur une carte. Grâce à l'identifiant unique d'un article du *Maitron*, on peut récupérer son contenu HTML puis le parser de manière à obtenir le texte de l'article. Ensuite, on fera intervenir spaCy dans la constitution de la liste des noms de lieux de l'article *Maitron* considéré. La requête d'une base géographique permettra d'obtenir les coordonnées géographiques de chaque lieu et de les afficher sur une carte. Afin de les visualiser, on créera une application Flask avec un template bootstrap préexistant.

Création d'une application Flask

Dans l'interface Flask, un bouton enverra un formulaire, qui fera appel à l'identifiant de la biographie visée. Ceci permettra de lancer la recherche, et en retour, d'obtenir un résultat.

L'ensemble du code utilisé ici est disponible sur Github, notamment les dossiers **static** et **templates,** en racine, qui contiennent l'UI :
https://github.com/CoLoDot/spacyxflask/tree/1-create-flask-app/static/bootstrap
https://github.com/CoLoDot/spacyxflask/tree/1-create-flask-app/templates

On commence par installer Flask et créer deux routes :

```
source env/bin/activate
pip install Flask==1.1.2

app.py
```

```
#! /usr/bin/env python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
  return render_template('index.html')

@app.route('/response', methods=['POST'])
def response():
  try:
    if request.method == 'POST':
      return render_template('index.html', response="Hello World")
    else:
      return render_template('index.html', response="Error: Please try again")
  except:
    return render_template('index.html', response="Error: Please try again")

if __name__ == '__main__':
  app.run(debug=True)
```

Désormais, on dispose d'une app flask avec deux routes :
- la route '/' : qui permet de faire la requête.
- la route '/response' qui affiche le résultat.

Pour lancer l'application : flask run
Le code de cette partie est disponible dans cette branche (notamment l'UI que nous ne développerons pas ici) :
https://github.com/CoLoDot/spacyxflask/tree/1-create-flask-app

Récupération d'une biographie du Maitron avec requests et BeautifulSoup

L'application dispose d'une UI composée d'un formulaire HTML. Lors du clic sur le bouton, la route API response() reçoit une requête POST.

```
app.py / response()
```

```
  if request.method == 'POST':
    article = get_maitron_article(id=ARTICLE_ID)
```

```
    return render_template('index.html', article
=article)
  ...
```

On ajoute une ligne dans la condition de la route response. Il s'agit de la variable article, qui contient le retour de la fonction get_maitron_article() dont l'implémentation sera proposée un peu plus loin. Cette fonction prend un paramètre d'entrée : il s'agit de l'id de la biographie que l'on souhaite récupérer. Cet identifiant est nommé ARTICLE_ID dans notre code.

app.py
```
from utils.maitron import get_maitron_article
from utils.constants import ARTICLE_ID
```

Maintenant, on crée le dossier utils/ en racine du projet avec trois fichiers à l'intérieur :

- __init__.py, qui permet à Python d'interpréter ce dossier comme un package, ce fichier reste vide.
- constants.py, qui contient les constantes ARTICLE_ID & MAITRON_BASE_URL.
- maitron.py, qui contiendra la fonction get_maitron_article().

utils/constants.py
```
ARTICLE_ID = 16519 # Bertin Emilienne on Maitron,
full access with no user account
MAITRON_BASE_URL = "https://maitron.fr/spip.php?article"
```

Afin de récupérer le texte de la biographie du *Maitron*, on utilise le module *requests* pour effectuer une requête HTTP GET de l'url de l'article du *Maitron*. Le module BeautifulSoup permettra de parser le HTML récupéré.
On installe beautifulsoup4 (requests est au contraire disponible par défaut) :

```
pip install beautifulsoup4==4.9.1
```

utils/maitron.py
```
#! /usr/bin/env python
import requests
from bs4 import BeautifulSoup
from .constants import MAITRON_BASE_URL

def get_maitron_article(id: int) -> dict:
  maitron_getter = requests.get(MAITRON_BASE_URL +
str(id))
  maitron_response = maitron_getter.text
  soup = BeautifulSoup(maitron_response,
'html.parser')
  article_title = soup.find("h1", "notice-titre")
  article_intro = soup.find("div", "intro")
  article_content = soup.find("div", "notice-texte entry")

  result = {
    'article_title': article_title.get_text(),
    'article_intro': article_intro.get_text(),
    'article_content': article_content.get_text(),
  }

  return result
```

La fonction get_maitron_article() prend un identifiant en argument et retourne un dictionnaire, ce retour correspond au contenu de l'article nettoyé, de fait, on extrait le contenu textuel de chaque balise HTML grâce à beautifulsoup4

templates/response.html
```
{% block content %}
<div id="output">
 {% if article|safe %}
   <div class="row">
     <div class="alert col mx-auto text-md-left"
style="margin-top: 1rem; background: grey; color: #ffffff;">
       {% if article|safe %}
       <ol class="list-unstyled">
         <li>{{ article.article_title }}</li>
         <li>{{ article.article_intro }}</li>
         <li>{{ article.article_content }}</li>
       </ol>
       {% else %}
       <div>Try again...</div>
       {% endif %}
     </div>
   </div>
 {% else %}
   <div></div>
 {% endif %}
</div>
{% endblock content %}
```

À ce stade, on peut afficher une biographie depuis n'importe quel identifiant. On peut faire le test avec la biographie d'Émilienne Bertin (identifiant : 16519) qui s'affiche correctement.
Vous pouvez retrouver le code de cette partie sur cette branche github :
https://github.com/CoLoDot/spacyxflask/tree/2-retrieve-maitron-and-scrape-infos

Recherche de lieux avec spaCy

L'application permet pour le moment de récupérer du contenu HTML et de l'afficher. On peut alors utiliser spaCy pour identifier les noms de lieux relatifs à la biographie d'Émilienne Bertin, agente des renseignements de la résistance (sa biographie Maitron est à cette adresse : https://maitron.fr/spip.php?article16519)

Tout d'abord, on importe dans app.py la fonction spaCy_analysis() qui va être rédigée afin d'analyser le texte avec spaCy. Puis on modifie la route response() afin de pouvoir afficher le nombre de lieux détectés.

app.py
```
...
from utils.spaCy_analysis import spaCy_analysis
...
@app.route('/response', methods=['POST'])
def response():
  try:
    if request.method == 'POST':
      article = get_maitron_article(id=ARTICLE_ID)
      cities = spaCy_analysis()
      return render_template('index.html',
```

```
            cities=str(len(cities)) + ' places found.',
article=article)
    else:
        return render_template('index.html', cities="Error:
Please try again")
  except:
    return render_template('index.html', cities="Error:
Please try again")
```

On crée le fichier spaCy_analysis.py dans le dossier utils/. Ce fichier contient la fonction qui parcourt la collection des entités reconnues par spaCy ayant un label correspondant à une localisation.

utils/spaCy_analysis.py

```
#! /usr/bin/env python
import spacy
from .maitron import get_maitron_article
from .constants import ARTICLE_ID

nlp = spacy.load('fr_core_news_sm')

def spaCy_analysis(article_getter=get_maitron_article)
-> list:
  maitron = article_getter(id=ARTICLE_ID)
  treat = nlp(maitron.get('article_intro') + maitron
.get('article_content'))
  return list(set([t.text for t in treat.ents if t.label_
== 'LOC']))
```

On itère sur les entités reconnues par spaCy en faisant usage du modèle pré-entraîné et du composant du pipeline spaCy nommé EntityRecognizer. Ce composant est appliqué à l'appel de nlp(), ce qui nous nous permet d'accéder à la propriété ents de l'objet **Doc** retourné par nlp(). Il ne reste plus qu'à modifier le template afin de pouvoir afficher le nombre de lieux trouvés, en dessous de la biographie. On ajoute alors la nouvelle variable cities au template, à la suite de la liste contenant les éléments de la biographie.

templates/response.html

```
<ol class="list-unstyled">
        ...
        <li>{{ cities | safe }}</li>
    </ol>
```

Le code de cette partie sur cette branche :
https://github.com/CoLoDot/spacyxflask/tree/3-spaCy

Affichage des lieux sur une carte avec Nominatim et leafLet

Dans la partie précédente, le nombre de lieux détectés par spaCy a été affiché dans l'UI. Désormais, on souhaite les afficher sur une carte. Pour ce faire, on fera usage de la classe **Nominatim** de la librairie **geopy**, qui permet d'obtenir une latitude et une longitude à partir d'un nom de lieu. Ensuite on affichera une carte des lieux en utilisant **Leaflet** au sein d'un script JavaScript que l'on ajoutera au template response.html.

On installe geopy :

```
pip install geopy==1.22.0
```

Puis on crée un nouveau fichier coordinates.py dans le dossier utils/ qui contient la fonction permettant d'obtenir les coordonnées de chaque lieu.

utils/coordinates.py

```
#! /usr/bin/env python
from geopy.geocoders import Nominatim

def get_coordinates(entities: list) -> list:
  cities_found = entities
  coordinates = []
  geolocator = Nominatim(user_agent="spaCy_cities
_finder")

  for city in cities_found:
    location = geolocator.geocode(city)
    if location:
      coordinates.append([city, location.latitude,
location.longitude])
  return coordinates
```

Tout d'abord, on importe la classe **Nominatim** de **geopy**. Ensuite, on crée la fonction get_coordinates(), qui prend une liste en argument et retourne une liste de listes contenant le nom du lieu, sa latitude et sa longitude. Une fois que l'on dispose de ce résultat, on peut l'importer dans notre fichier app.py.

app.py

```
...
from utils.coordinates import get_coordinates
```

app.py / response()

```
...
if request.method == 'POST':
      article = get_maitron_article(id=ARTICLE_ID)
      cities = spaCy_analysis()
      coordinates = get_coordinates(entities=
cities)
      return render_template('index.html', coordinates
=coordinates, cities=str(len(cities)) + ' places found.'
, article=article)
    else:
      ...
```

templates/response.html

```
{% block content %}
<div id="output">
 {% if coordinates|safe %}
  <div class="row">
    <div class="alert col mx-auto text-md-left" style=
"margin-top: 1rem; background: grey; color: #ffffff;">
      {% if article|safe %}
        <ol class="list-unstyled">
          <li>{{ article.article_title }}</li>
          <li>{{ article.article_intro }}</li>
          <li>{{ article.article_content }}</li>
        </ol>
      {% else %}
      <div>Try again...</div>
      {% endif %}
    </div>
```

```
    <div class="alert col mx-auto text-md-center"
style="margin-top: 1rem; background: grey; color: #ffffff;">
        <div id="map" style="height: 450px;">
</div>
    </div>
  </div>
  {% else %}
  <div></div>
  {% endif %}
</div>
{% endblock content %}
```

On ajoute le code Javascript d'affichage de la carte :

templates/response.html

```
<script>
  var locations = {{ coordinates | safe}};
  var mymap = L.map('map').setView([locations[0]
[1], locations[0][2]], 6);

  L.tileLayer('https://api.tiles.mapbox.com/v4/{id}
/{z}/{x}/{y}.png?access_token=pk.eyJ1IjoibWFwYm94li
wiYSI6ImNpejY4NXVycTA2emYycXBndHRqcmZ3N3gifQ.rJcFIG
214AriISLbB6B5aw', {
    maxZoom: 18,
    attribution: 'Map data &copy; <a href="https://www.
openstreetmap.org/">OpenStreetMap</a> contributors, ' +
      '<a href="https://creativecommons.org/
licenses/by-sa/2.0/">CC-BY-SA</a>, ' +
      'Imagery © <a href="https://www.mapbox.
com/">Mapbox</a>',
    id: 'mapbox.streets'
  }).addTo(mymap);

  for (var i = 0; i < locations.length; i++) {
    marker = new L.marker([locations[i][1], locations
[i][2]])
      .addTo(mymap)
  }
</script>
```

Ce code de cette partie sur cette branche github :
https://github.com/CoLoDot/spacyxflask/tree/4-display-map
Le dépôt Github contient également une CI ainsi qu'un jeu de tests unitaires sur la branche master si vous souhaitez aller plus loin, ils ne seront pas évoqués ici.

Conclusion

L'objectif de cette introduction au traitement du langage naturel était de proposer un exemple de la puissance et des champs des possibles dans l'utilisation de spaCy. Dans ce cadre, on s'est borné à une utilisation de la tokenisation, qui permet en quelques lignes de code d'identifier les noms de lieux dans un texte donné. On pourrait aller plus loin en appliquant ce code sur l'ensemble du *Maitron en ligne* et ainsi proposer des statistiques à propos de la fréquence des lieux cités dans les biographies. Une prochaine étape serait d'identifier tous les lieux de naissance des fusillés de la Seconde Guerre mondiale répertoriés dans le *Maitron Les Fusillés (1940-1944) (sous-titré Dictionnaire biographique des fusillés et exécutés par condamnation et comme otages et guillotinés en France pendant l'Occupation)*.

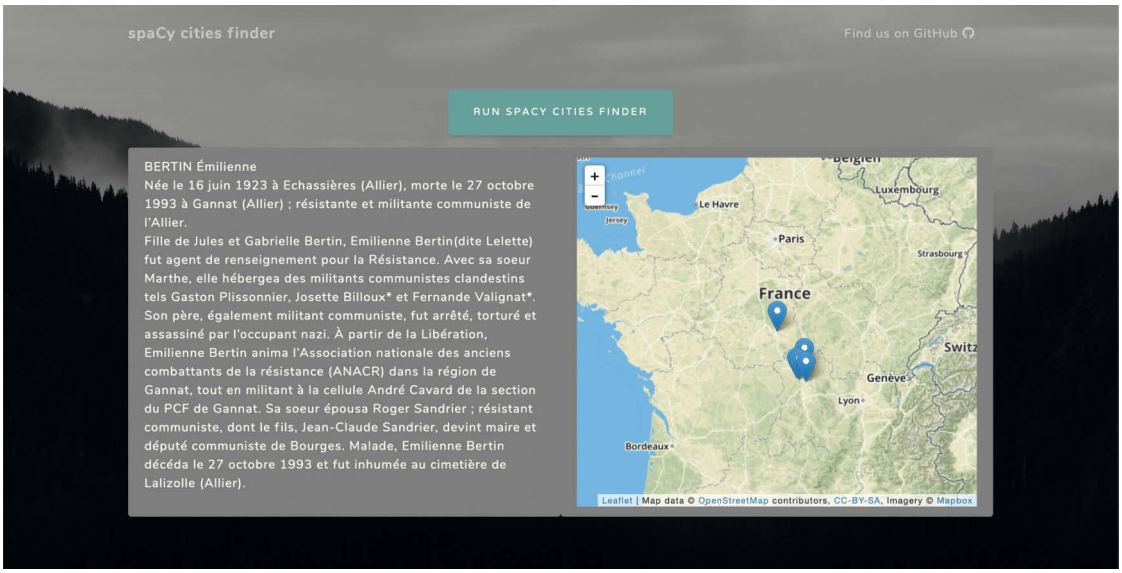
Pour aller plus loin avec spaCy, y compris en se limitant à de la tokenisation, il est également possible de personnaliser votre pipeline, d'ajouter vos propres entités ou d'entraîner vos propres modèles de données.

Ressources

Site officiel de spaCy avec sa documentation complète :
https://spacy.io/

Cours avancé sur le TALN par Ines Montani, co-créatrice de spaCy (en anglais) :
https://course.spacy.io/

Le Maitron :
https://maitron.fr/
https://fr.wikipedia.org/wiki/Le_Maitron



Philippe BOULANGER
Manager des expertises C/C++ et Python
www.invivoo.com

INVIVOO
BEYOND TECH

Gestion avancée de fichiers

Quand on travaille sur des tâches au niveau système, on doit régulièrement appliquer une même action à tous les fichiers d'un même type : archivage, compter le nombre de lignes de code, supprimer tous les fichiers log ayant plus de 30 jours, etc. Heureusement, il existe de puissants mécanismes dans Python.

PARCOURIR LES ENTRÉES D'UN RÉPERTOIRE

> listdir

Le package 'os' fournit la fonction 'listdir' qui, pour un nom de répertoire donné, retourne la liste de noms de fichiers/répertoires présents dans ce répertoire. Voici un exemple d'utilisation :

```
from os import listdir

for name in listdir( "C:\\Temp" ):
  print( name )
```

> walk

'walk' est un itérateur qui permet à l'utilisateur de s'affranchir du parcours récursif du répertoire.

```
from os import walk

for root, dirs, files in walk( 'C:\\Temp', topdown=True ):
    print( root )
    print( dirs )
    print( files )
    print( '--------------------------------' )
```

Cette fonction alloue de nombreuses listes de chaînes de caractères.

> scandir

Depuis Python 3.4, la méthode conseillée est 'scandir', car elle est beaucoup plus efficace. J'ai pu constater un gain de performance de x20 sur des cas complexes d'utilisation.

```
from os import scandir

with scandir( "C:\\Temp" ) as it:
  for entry in it:
    print( entry.name )
```

L'utilisation contextuelle avec 'with' est importante, elle garantit la libération des ressources ouvertes. L'itérateur obtenu avec 'scandir' permet de parcourir des instances d'objet de type os.DirEntry (https://docs.python.org/3/library/os.html#os.DirEntry) grâce auquel on peut accéder à différentes fonctions/propriétés permettant de connaître la nature de l'entrée (fichier, répertoire ou lien symbolique) ou des informations comme les dates et tailles.

RÉUTILISATION

L'idée consiste à créer une fonction 'filemap' qui va parcourir récursivement une arborescence et appliquer une action sur tous les fichiers qui sont une cible. On pourra ainsi réutiliser cette fonction dans différents contextes…

```
from os import scandir

def filemap( path, action, is_target ):
  with scandir( path ) as it:
    for entry in it:
      if entry.is_dir():
        filemap( entry.path, action, is_target )
      elif is_target( entry ):
        action( entry )
```

Le paramètre 'action' est utilisé pour passer une fonction callback qui prendra en entrée l'instance d'un objet de type os.DirEntry. Cette fonction appliquera une action (le supprimer, le déplacer, etc.) sur un fichier. Voici un exemple d'utilisation :

```
filemap( "c:\\Temp",
    lambda e : print( e.name ),
    lambda e : e.name.lower().endswith( ".py" ) )
```

On affiche le nom de chaque fichier source Python présent dans l'arborescence de « C:\Temp ».

COMPTER LES FICHIERS

Compter les fichiers est un besoin qui présente un intérêt d'un point de vue fonctionnel (valider que le nombre de fichiers générés par la production est identique à ce qui est attendu), mais aussi sur le plan informatique, car il nous amènera à mieux penser notre code. L'implémentation naïve serait :

```
count = 0
def count_file( entry ):
  global count
  count += 1

filemap( "c:\\Temp",
    count_file,
    lambda e : e.name.lower().endswith( ".py" ) )
print( count )
```

'count_file' permet effectivement de compter les fichiers, mais elle ne respecte pas le concept de pureté de la program-

mation fonctionnelle, car elle dépend d'une variable globale. Si j'oubliais de réinitialiser 'count' entre 2 appels à 'filemap' j'obtiendrais un résultat incorrect.
Pour corriger ce problème, Python dispose d'un outil très intéressant : on peut créer des objets qui se comportent comme des fonctions. Grâce à l'objet, on va pouvoir stocker l'état du compteur tout en préservant la pureté.

```
class CountFile:
  def __init__( self ):
    self.__count = 0

  @property
  def count( self ):
    return self.__count

  def __call__( self, entry ):
    self.__count += 1

  def __str__( self ):
    return F"{self.__count}"

c = CountFile()
filemap( "c:\\Temp",
      c,
      lambda e : e.name.lower().endswith( ".py" ) )
print( c )
```

La fonction '__call__' est l'action qui sera appelée et qui permet l'objet de se comporter comme une fonction.

LE CIBLAGE

> Ciblage par les extensions

Dans notre exemple précédent, nous avons ciblé les fichiers sources Python grâce à une fonction lambda :

```
lambda e : e.name.lower().endswith( ".py" )
```

C'est régulièrement que l'on va avoir besoin de cibler les fichiers via leur extension. Il nous faudrait avoir un moyen plus simple et plus rapide pour cibler un type d'extension ou un groupe d'extensions. La solution pythonique par excellence sera fournie par un générateur de fonction :

```
def endswith( *args ):
  extensions = [ ext.lower() for ext in args ]

  def _is_target( entry ):
    name = entry.name.lower()
    for ext in extensions:
      if name.endswith( ext ):
        return True
    return False

  return _is_target


c = CountFile()
filemap( "C:\\Tools\\Anaconda3",
      c,
      endswith( ".py", ".txt" ) )
print( c )
```

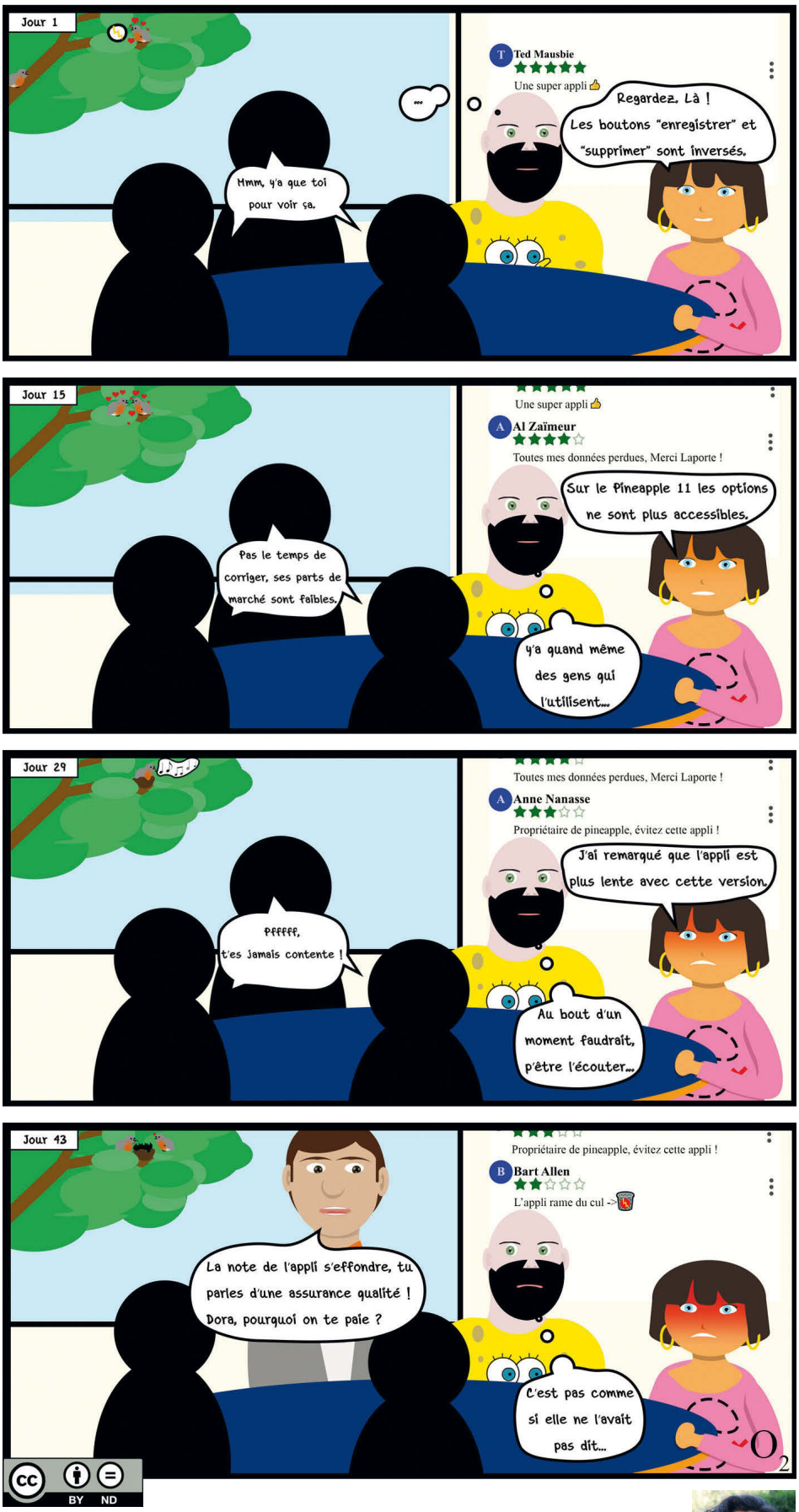
> Ciblage par la date de création

Lorsque l'on gère des arborescences de production, on est souvent amené à mettre en place des outils de suppression des fichiers en fonction de leurs dates de créations : on supprime les fichiers créés il y a plus de 30 jours.

```
def older_than( day_count ):
  from datetime import datetime as DT,
              timedelta, timezone
  today = DT.now()
  epoch = DT( 1970, 1, 1 )
  ref   = timedelta( hours = day_count * 24 )

  def _is_target( entry ):
    creation_date = epoch + timedelta( seconds = entry
.stat().st_ctime )
    return ( today - creation_date ) >= ref

  return _is_target


filemap( "C:\\Temp",
      lambda e: unlink( e.path ),
      older_than( 30 ) )
```

A vous de coder.

Cassandre



Aude Hage
Auteur, illustratrice, conceptrice de jeux de société.
@aude.hage

Directives de compilation

PROGRAMMEZ!
Programmez! n°244
Janvier – février 2021

Directeur de la publication & rédacteur en chef
François Tonic
ftonic@programmez.com

Secrétaire de rédaction
Olivier Pavie

Contacter la rédaction
redaction@programmez.com

Ont collaboré à ce numéro
ZDNet

Les contributeurs techniques
R-A Val, L. Julliard, S. Ducasse, M. Bojoly, T. Deman, J. Roubaix, J. Norblin, J-C Riat, C-L Gauriau, B. Prieur, P. Boulanger

Couverture
Logo RUST

Maquette
Pierre Sandré

Marketing – promotion des ventes
Agence BOCONSEIL - Analyse Media Etude
Directeur : Otto BORSCHA
oborscha@boconseilame.fr
Responsable titre : Terry MATTARD
Téléphone : 09 67 32 09 34

Publicité
Nefer-IT
Tél. : 09 86 73 61 08
ftonic@programmez.com

Impression
SIB Imprimerie, France

Dépôt légal
A parution

Commission paritaire
1220K78366

ISSN
1627-0908

Abonnement
Abonnement (tarifs France) : 49 € pour 1 an, 79 € pour 2 ans. Etudiants : 39 €. Europe et Suisse : 55,82 € - Algérie, Maroc, Tunisie : 59,89 € - Canada : 68,36 € - Tom : 83,65 € - Dom : 66,82 €.
Autres pays : consultez les tarifs sur www.programmez.com.
Pour toute question sur l'abonnement :
abonnements@programmez.com

Abonnement PDF
monde entier : 39 € pour 1 an.
Accès aux archives : 19 €.

Nefer-IT
57 rue de Gisors, 95300 Pontoise France
redaction@programmez.com
Tél. : 09 86 73 61 08

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 €** *

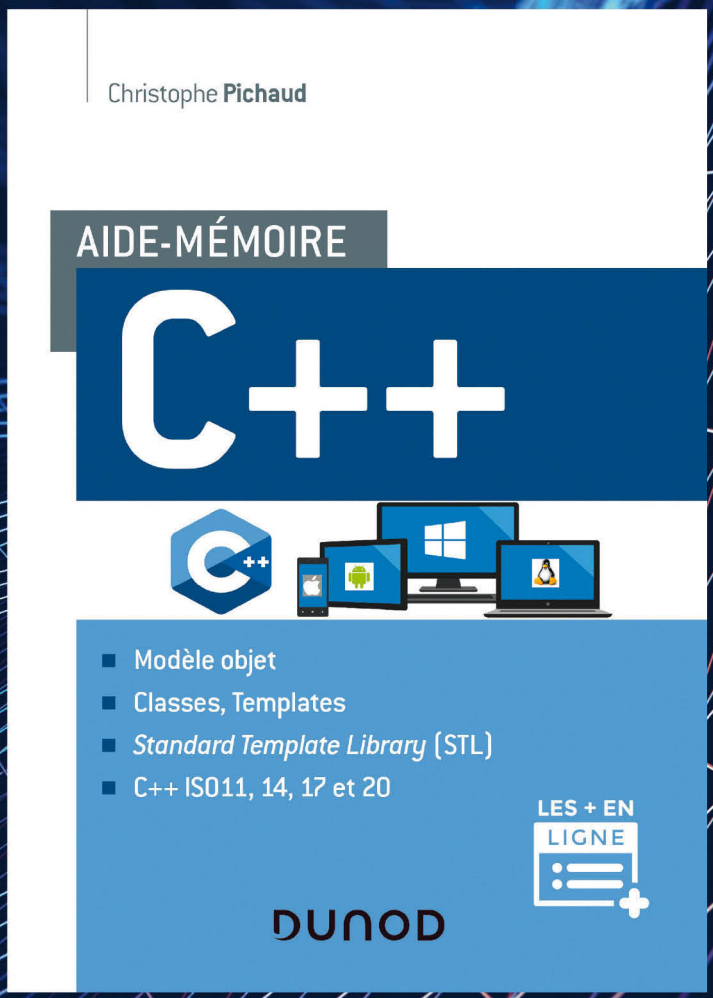
** Frais de port : 1,01 €*

Abonnez-vous pour ne rater aucun numéro.

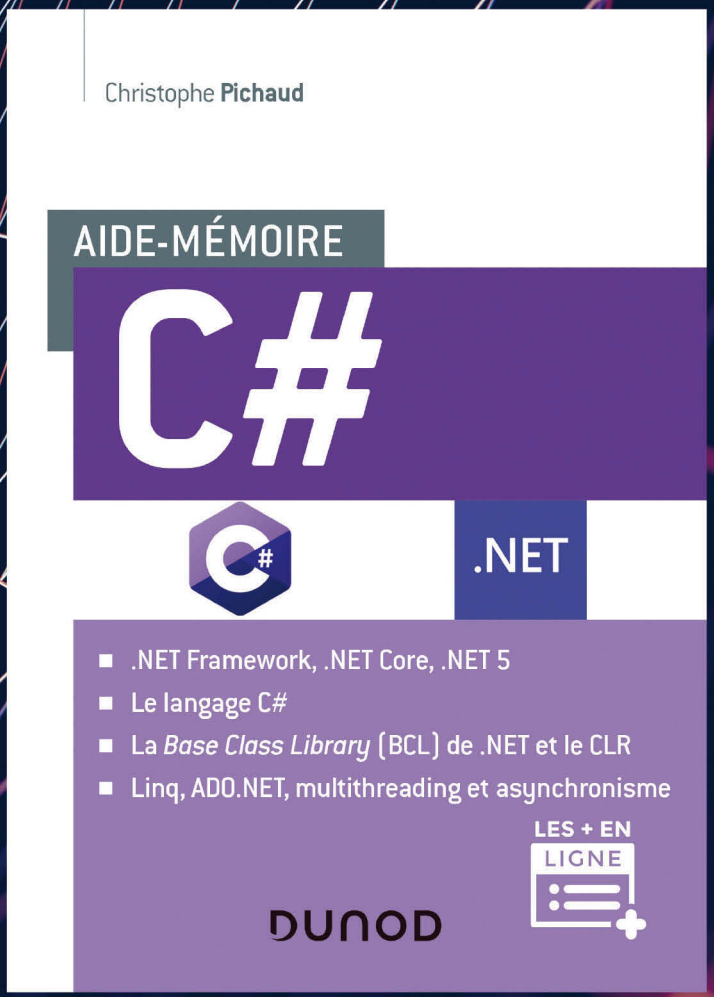
TECHNOSAURES

le magazine du **rétro-computing**

DÉVELOPPEURS, FORMEZ-VOUS À C++ ET C#



9782100807123 · 21,90€



9782100813223 · 23,90€

Pour les pros et les étudiants

DUNOD
une page d'avance