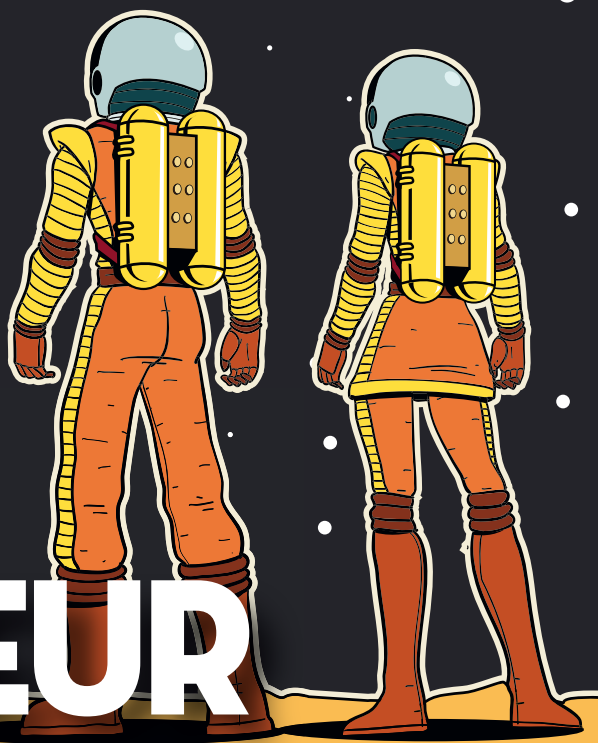
PROGRAMMEZ!

PROGRAMMEZ!

Le magazine des développeurs

.NET 6
JAVA 17

N°249
11/12
2021



LE DÉVELOPPEUR VA SAUVER LA TERRE !

Le seul magazine écrit par et pour les développeurs

M 04319 - 249 - F: 6,99 € - RD

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

OPÉRATION POUR 1 EURO DE PLUS

JUSQU'AU 03 DÉCEMBRE

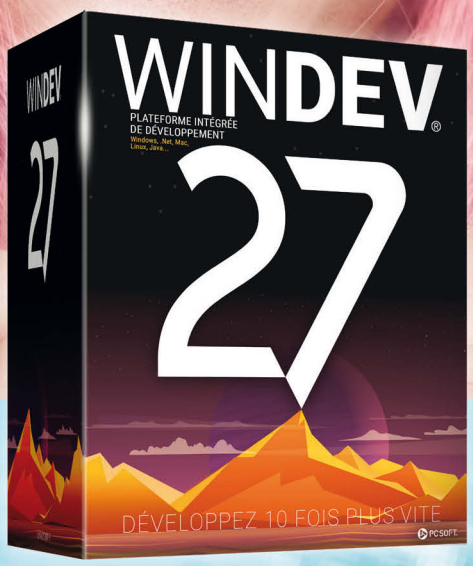
Aucun abonnement à souscrire.
Compatible tous opérateurs

COMMANDEZ WINDEV 27

OU WEBDEV 27 OU WINDEV MOBILE 27

ET RECEVEZ LE NOUVEAU

SAMSUNG Galaxy Z Flip3 5G



WINDEV 27
AGL DevOps
Cross-Plateformes
N°1 en France

CHOISISSEZ

- **Galaxy S21 5G** 128Go
ou
- **Smart TV* 4K 176cm**
ou
- **Smart TV* 4K QLED 14cm**
- **Moniteur* géant 48''**
ou
- **lot de 2 Galaxy Tab S6 lite**
ou
- **lot de 2 Galaxy A52S 5G**

Détails et autres matériels sur www.pcsoft.fr
* : une TV demande un supplément de 15€HT,
un moniteur un supplément de 5€HT

WWW.PCSOFT.FR

OPÉRATION POUR 1 EURO DE PLUS

Pour bénéficier de cette offre exceptionnelle, il suffit de commander WINDEV 27 (ou WINDEV Mobile 27, ou WEBDEV 27) chez PC SOFT au tarif catalogue avant le 03 Décembre 2021. Pour 1 Euro HT de plus*, vous recevrez alors le ou les magnifiques matériels que vous aurez choisis. Offre réservée aux sociétés, administrations, mairies, GIE et professions libérales, en France métropolitaine. L'offre s'applique sur le tarif catalogue uniquement. **Voir tous les détails sur : WWW.PCSOFT.FR ou appelez-nous au 04.67.032.032**

Le Logiciel et le matériel peuvent être acquis séparément. Tarif du Logiciel au prix catalogue de 1.650 Euros HT (1.980,00 TTC). Merci de vous connecter au site www.pcsoft.fr pour consulter la liste des prix des matériels. Tarifs modifiables sans préavis.

Contenus

6 Agenda
Les événements pour les développeurs
La rédaction

8 Quelques brèves du mois
Louis Adam (ZDnet.fr)

10 Dossier spécial .Net 6
En novembre, .Net 6 sera disponible pour tous les développeurs. Nos experts .Net vous proposent un tour d'horizon de la plateforme + Visual Studio 2022.
Pierre Gascoin, Jérémy Jeanson, Baptiste Bazureau, François Lefebvre, Sébastien Bernard, Florian Pouchelet, Clément Sannier, Gatien Montreuil

35 Java 17
Java 17 vient de sortir.
Quelles nouveautés pour les développeurs ?
Loïc Mathieu

37 Le développeur va sauver la Terre ! Partie 1 : posons les fondamentaux
L'éco-conception doit permettre de mieux utiliser les ressources technologiques, réduire l'impact de l'informatique et optimiser les codes, le binaire, le poids des pages web, etc; Dossier complet sur les bonnes pratiques et pourquoi les utiliser.
Mathieu Touchard, Pierre Lagarde, Raphaël Lemaire, Benoit Petit, Vincent Frattaroli, François Rézenthel

54 Refactorer le legacy intestable avec les Approval Tests
Quoi ? Vous ne connaissez les Approval Tests ?
Il est temps de combler vos lacunes.
Sepehr Namdar

58 Superviser votre Linux avec un écran LCD !
Un peu de code, des libs, un écran LCD. Ce projet permet monitorer son système Linux très simplement avec lcd4linux.
Sébastien Colas

62 Coder Fibonacci en Java avec des tests
Codons le calcul des termes de la suite de Fibonacci en Java grâce au TDD.
Thierry Leriche

69 Python et la tortue
Ce n'est pas une fable de la Fontaine mais de la programmation python en dessinant.
Malika More & Pascal Lafourcad

72 Programmation dynamique
Python est un langage surpuissant mais pas toujours simple à maîtriser quand il faut typer. La programmation dynamique est là pour nous aider.
Philippe Boulanger

76 Interopérabilité Kotlin et Java *partie 2*
Revenons sur l'interopérabilité entre Kotlin et Java.
Sallah Kokaina

81 Par les deux bouts de la lorgnette
Question sur l'observabilité !
Jean-Baptiste Bron

Divers

4 Edito
Le développeur est comme Chuck Norris, il va nous sauver (ou nous défoncer)

42 43 Abonnement & Boutique

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

SAISON 24 ÉPISODE 249 (OU 253)

Le développeur est comme Chuck Norris, il va nous sauver (ou nous défoncer)

© Excelsior, 17 février 2021. Tournage du Project X

Quel développeur n'a jamais rêvé d'être le Chuck Norris du code ? Notre Chuck est capable de maîtriser toutes les situations : injection de codes, attaques par DoS, erreurs de compilation, débordement mémoire, etc. Rien ne lui résiste.

Pour faire maigrir les applications, la solution Chuck n'est peut-être pas la plus appropriée. Nous vous proposons fitness et zumba tous les matins et terminé les pizzas ! Ouais, c'est dur. Fini aussi de reprendre du code ici et là ou de coder rapidement sans trop respecter les règles d'un bon code, car ce soir vous avez soirée battle royale.

Dans ce numéro, nous vous proposons de plonger au cœur de l'éco-conception et de la notion de responsabilité écologique de l'informatique et donc du développeur. Soyons clairs, le développeur doit jouer un rôle actif dans l'utilisation des ressources. L'utilisateur, que nous sommes tous, doit lui aussi jouer son rôle. Les constructeurs et éditeurs, les géants de la tech, les startups, etc. ont leur rôle à jouer.

L'informatique, le cloud computing, les smartphones, le web sont de plus en plus dénoncés par les écologistes de tout bord, sans forcément tout comprendre de quoi on parle. On entend tout et n'importe quoi sur le gaspillage de l'informatique en général et le % des infrastructures serveurs dans la consommation électrique mondiale. Un constat s'impose : oui la technologie consomme des ressources et pèse sur les infrastructures. Le recyclage n'est pas assez étendu. L'obsolescence est un autre problème qui est tout sauf simple à résoudre.

L'obésité logicielle est un réel problème. Au début du web, une page web pesait quelques dizaines de Ko, aujourd'hui, c'est près 2 Mo ! En 10 ans, le poids moyen d'une page web a été multiplié par 4. Une installation complète de Visual Studio, c'est 40 Go, XCode, 7 Go, etc. Une mise à jour de Pokémon Go ou Fortnite, 300-500 Mo, etc. Nous voyons qu'il y a là une marge énorme d'améliorations. Ne parlons même pas des outils de bureautique.

QUAND LES NOUVEAUTÉS SONT BONNES, FAUT EN PROFITER !

Soyons honnêtes, toute nouvelle version ne fait pas frémir les claviers. Même des adorateurs de Java reconnaissent que la JDK 17 apporte peu de nouveautés majeures mais des petites features sympathiques. Nous trouvons toujours le « plus » qui fait toujours plaisir. Pour notre part, une des fonctions qui nous a le plus intéressé est la détection automatique du langage utilisé dans l'éditeur de Visual Studio Code et l'IDE nous dit s'il faut, ou non, installer des plug-ins pour supporter le langage. Essayer c'est l'adopter.

Bref, cherchez le petit détail.

François Tonic
Chuck Norris à Programmez!

LES PROCHAINS NUMÉROS

HORS SÉRIE #5 AUTOMNE

100% Red Hat

Disponible
dès le 26 novembre 2021

PROGRAMMEZ! N°250

Disponible
le 3 janvier 2022


Disponible
dès le 26 novembre
PROGRAMMEZ!
Le magazine des développeurs
Compiler.
Déployer partout.
Red Hat
Developer
Développer et intégrer de l'IA
dans vos applications
Machine learning, deep learning, IoT :
du code, du code et du code !
Le seul magazine écrit par et pour les développeurs
Couverture
provisoire
Printed in EU - Imprimé en UE - BELGIQUE 7,50 € - Canada 10,55 $ CAN - SUISSE 14,10 FS - DOM Surf 8,10 € - TOM 1100 XPF - MAROC 59 DH
100 % Red Hat
100 % Développeur

novembre

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
1	2	3	4	5	6	7
	Meetup Programmez!					
8	9	10	11	12	13	14
Devoxx Belgique						
	Open Source Experience (Paris)					
	DevFest Strasbourg					
15	16	17	18	19	20	21
Hack in Paris (virtuel)			Codeurs en Seine (virtuel)	DevFest Lille		
			DevOps D-Day (Marseille)			
22	23	24	25	26	27	28
29	30					

décembre

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

janvier 2022

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

février 2022

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

Les événements Programmez!

Meetups Programmez!

2 novembre : spécial Clojure
7 décembre : sujet à venir !

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

DevCon

16 décembre : conférence Cybersécurité
+ Secure by Design. 6 sessions + 2 keynotes.

Où : campus EFREI Paris
A partir de 13h30

Informations & inscription : programmez.com

Directives de compilation

PROGRAMMEZ!

Programmez! n°249 - Novembre - Décembre 2021

Directeur de la publication & rédacteur en chef : **François Tonic** - ftonic@programmez.com
Contacter la rédaction : redaction@programmez.com

Expert en brèves : **Louis Adam (ZDnet.fr)**
Code review : **Dorra Bartaguiz**

Les contributeurs techniques

Pierre Gascoin, Jérémy Jeanson, Baptiste Bazureau, François Lefebvre, Sébastien Bernard, Florian Pouchelet, Clément Sannier, Gatien Montreuil, Loïc Mathieu, Mathieu Touchard, Pierre Lagarde, Raphael Lemaire, Benoit Petit, Vincent Frattaroli, François Rézenthel, Sepehr Nadmar, Sébastien Colas, Pascal Lafourcade, Maloika More, Sallah Kokaina, Jean-Baptiste Bron

Maquette : **Pierre Sandré**

Marketing – promotion des ventes : **Agence BOCONSEIL - Analyse Media Etude**
Directeur : Otto BORSCHA - oborscha@boconseilame.fr
Responsable titre : Terry MATTARD - Téléphone : 09 67 32 09 34

Publicité
Nefer-IT : Tél. : 09 86 73 61 08 - ftonic@programmez.com

Impression : **SIB Imprimerie, France**
Dépôt légal : **A parution**
Commission paritaire : **1225K78366**
ISSN : **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

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

N°7+8
60 pages

NOUVEAU !

Commandez directement sur technosaures.fr

14,99 € *(+frais de port)* **60 pages**

Abonnement 1 an : 29 €

Facebook, un serveur mal configuré et c'est le grand plongeon

C'est suffisamment rare pour être noté : pendant un peu plus de six heures lundi 4 octobre, l'ensemble des services de Facebook était complètement inaccessible. Ni Whatsapp, ni Instagram, ni Facebook, rien. Une cyberattaque ? Non : Facebook a expliqué par la suite que la panne était due à une erreur humaine à l'occasion d'une maintenance de routine. Résultat des courses : une infrastructure hors service, des sites hors ligne et des ingénieurs qui s'arrachent les cheveux parce que le système d'accès au datacenter a également été affecté, les empêchant de rentrer. Finalement tout est rentré dans l'ordre, mais Facebook promet de revoir ses procédures pour éviter que l'incident ne se reproduise.

Twitch, en open source… contre sa volonté

Une autre histoire de serveur mal configuré a eu des conséquences désagréables pour Twitch, la plateforme de streaming live détenue par Amazon. Dans la nuit du 6 octobre, un internaute a publié sur 4chan un lien menant vers une archive d'environ 128Go de données, apparemment dérobées sur les serveurs de Twitch. On y retrouvait notamment le code source de plusieurs applications développées par la plateforme, mais aussi des projets internes tels que Vapor, un potentiel concurrent de Steam développé par les équipes de Twitch. Et pour ajouter au drame, les attaquants ont également publié une liste des revenus perçus par les streamers les plus en vue de la plateforme. Rude semaine pour Twitch donc, qui continue ses investigations sur l'origine exacte de la fuite.

Cloud de confiance : le temps presse et les acteurs s'organisent

Les annonces de partenariats entre acteurs américains et européens se multiplient dans le cadre du label Cloud de Confiance. Petit rappel pour ceux qui s'y perdraient : afin de permettre aux entreprises de bénéficier de solutions sans s'exposer aux risques juridiques liés à des lois comme le Cloud Act, le gouvernement propose de labelliser des solutions étrangères opérées par des acteurs français. Depuis, les annonces se multiplient : Capgemini, Orange et Microsoft s'acoquinent sur « bleu », OvhCloud propose un partenariat avec Whaller et Thales avancera de son côté avec Google Cloud. Pour l'instant, ce ne sont que des annonces, mais les offres sont attendues avec une certaine impatience, surtout du côté de l'administration et des ministères, à qui on a gentiment rappelé que souscrire à des offres Office 365 hébergées sur Azure n'était pas vraiment en accord avec la politique de cloud de l'Etat.

Pénurie de semi-conducteurs : l'industrie automobile trinque

La pénurie de semi-conducteurs se poursuit et les fondeurs ont du mal à faire face à la demande croissante de microprocesseurs et de semi-conducteurs. Ils ont visiblement décidé de privilégier leurs clients historiques au détriment des nouveaux venus, notamment l'industrie automobile qui est devenue au cours des dernières années particulièrement gourmande en semi-conducteurs. Résultat, plusieurs constructeurs ont été contraints d'annoncer un ralentissement sur leurs usines de productions, à l'instar de Stellantis, issu de la fusion de PSA et FCA, qui a annoncé la fermeture de plusieurs de ses usines européennes jusqu'à la fin de l'année.

Let's Encrypt, un certificat vous manque et SPOF, tout est cassé

Quand vous devenez un acteur central de l'écosystème web, le moindre dysfonctionnement peut avoir des conséquences en cascades. C'est ce qui est arrivé aux utilisateurs de Let's Encrypt à la fin du mois de septembre : un certificat racine utilisé pour valider des certificats SSL fournis par l'ONG est arrivé à expiration.

Let's Encrypt avait pourtant averti les utilisateurs concernés et les avait invités à renouveler les certificats potentiellement affectés avant la date fatidique, mais ça n'a pas suffi. Le 30 septembre, de nombreux services ont donc connu des problèmes liés à leurs certificats : on peut ainsi citer Bluecoat, Ovhcloud, Shopify, Ledger ou encore Cloudflare parmi de nombreux autres. La rançon du succès pour Let's Encrypt, qui est devenu en l'espace de dix ans un acteur central de l'écosystème des autorités de certification et par la même un SPOF (Single Point of Failure, point de défaillance unique en français).

Braquage à l'APHP

Les hôpitaux de Paris ont annoncé début septembre avoir été victime d'un vol de données record : les données de test Covid appartenant à 1,4 million de Franciliens ont été dérobées dans le courant de l'été. L'attaquant a exploité une faille zero day dans un logiciel utilisé par l'AP-HP pour transférer ces données vers les serveurs de l'Assurance Maladie. Un suspect a rapidement été arrêté par les enquêteurs : il s'agit d'un étudiant en informatique de 22 ans, qui reconnaît être à l'origine du vol et clame avoir voulu prouver les failles de la sécurité du système de santé français. Des failles démontrées de manière un peu trop convaincante, qui lui valent donc aujourd'hui d'attendre un procès suite aux plaintes déposées par l'AP-HP et les différentes victimes du vol de données.

Les partenaires 2021 de

PROGRAMMEZ!

Le magazine des développeurs

Niveau maître Jedi

softeluent

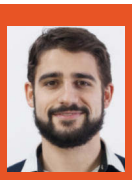
LA MANUFACTURE CACD2

Niveau padawan

OSAXiS

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

Contactez-nous dès maintenant :
ftonic@programmez.com



Pierre Gascoin
Expert technique, SQLI

SQLI DIGITAL EXPERIENCE

À la rencontre de .NET 6

Après 1 an d'attente, plusieurs previews et les multiples posts des équipes développeurs, Microsoft lance .Net 6. Pierre Gascoin (expert technique, SQLI) nous présente la nouvelle version. Bonne découverte. La Rédaction.

Vue d'ensemble

Lors de la planification de .NET 6 fin 2020, Microsoft s'est appuyé sur 8 thèmes : chaque thème regroupant des fonctionnalités correspondant à un concept cible auquel le framework doit répondre (cf. https://themesof.net/) :

- "***.NET doit être attrayant pour les nouveaux développeurs/étudiants***".
- "***.NET propose une bonne expérience de développement d'applications clientes***".
- "***.NET est reconnu comme un framework convaincant pour créer des applications cloud natives***".
- "***Entreprise et LTS***".
- "**Faire grossir l'écosystème .NET à l'aide d'une qualité, d'une confiance renforcée et d'un support amélioré**".
- "***Améliorer la productivité des développeurs***".
- "***Améliorer les performances en utilisant les informations d'exécution du runtime (PGO)***".
- "***Répondre au mieux aux attentes des développeurs***".

Le contenu de cette nouvelle release met l'accent sur plusieurs axes. Tout d'abord, la continuité de l'unification de la plateforme (entrepris avec l'arrivée de .NET 5) qui vise le développement multiplateforme mobile, bureau et web tout en minimisant l'effort de cibler plusieurs plateformes. Autre sujet très important pour Microsoft, le cloud ! .NET 6 apporte de nouvelles fonctionnalités native-cloud afin d'améliorer la valeur du framework dans les cas d'utilisation cloud. Les développeurs ne sont pas en reste : Microsoft prévoit d'améliorer la productivité du cycle de développement des applications .NET 6 ainsi qu'une meilleure prise en compte des retours utilisateurs afin que le framework réponde au mieux à leurs attentes. Pour finir, Microsoft continue de s'ouvrir à la communauté open source : c'est dans cette optique qu'une attention particulière est portée sur l'évolution de la communauté .NET. En effet, la firme de Redmond vise à élargir le nombre de développeurs qui utilisent .NET 6 en simplifiant/accélérant la mise en place de certains cas d'utilisation (qui pouvait rebuter certaines personnes venant d'autres langages) et en accueillant au mieux les "nouveaux arrivants" autour d'un écosystème simple et accessible. Cette nouvelle version du framework est prévue pour novembre 2021 avec un support long terme (Long Term Support) de 3 ans.

Nouveautés majeures

Dans les faits, .NET 6 apporte de nombreuses nouveautés :

- Développement multiplateforme
  - .NET MAUI (Multi-platform App UI) : décalé à 2022
  - Application Blazor desktop
  - Amélioration du support des processeurs Arm64
- Amélioration de la productivité
  - Nouvelle fonctionnalité de "Hot reload" ("rechargement à chaud")
  - Amélioration des performances du build
- Développement d'applications cloud-native
  - ASP.NET Core Minimal web API
- Évolution du SDK
  - Logique de workloads optionnels
- Amélioration des performances du runtime

.NET 6 sera accompagné de la nouvelle version de C# 10 de Visual Studio 2022. Cette dernière sera d'ailleurs requise pour espérer travailler avec .NET 6.

Plateformes supportées

Pour Windows, peu de changement : .NET 6 supporte exactement les mêmes versions que .NET 5, mais introduit le support de Windows Arm64 (spécifique à Windows Desktop). En ce qui concerne Linux, quelques modifications sont à noter : la disparition de Linux Mint de la liste des distributions supportées et la montée de version pour d'autres (Alpine Linux : 3.11+ > 3.13+ et support d'ARM32, Debian : 9+ > 10+, support d'Arm64 pour Red Hat). Au tour de macOS qui voit sa version passer de 10.13+ à la 10.14+, mais aussi (et surtout) le support de l'architecture Arm64 (et donc la prise en charge des processeurs Apple Silicon). Pour finir, on note bien sûr l'apparition des versions Android et iOS/tvOS. Plus d'informations ici :
https://github.com/dotnet/core/blob/main/release-notes/6.0/supported-os.md

.NET MAUI (write-one run-everywhere)

Nous reviendrons dessus dans les prochains mois. Sortie prévue : 1ere moitié de 2022.

.NET 6 & Arm64

L'architecture Arm64 reste un sujet très important pour la firme de Redmond. De nombreux efforts ont été accomplis sur la dernière release du framework afin d'améliorer la gestion de cette architecture. Dans cette nouvelle version, Microsoft continue d'investir sur les aspects performances et fonctionnalités. Sur Windows, .NET 6 ajoute le support pour Windows Forms et Windows Presentation Foundation (WPF) par-dessus les capacités Windows Arm64 poussées en .NET 5. Il est prévu que ces fonctionnalités soient redescendues sur .NET 5 par la suite.
Sur Mac, les puces Apple Silicon sont désormais supportées. Ces puces ont deux modes de fonctionnements : natif et émulé (x64). L'émulation est implémentée par l'intermédiaire du composant Rosetta 2. Cette émulation entraîne bien sûr une répercussion sur les performances par rapport à une exécution native.
.NET 6 fournira des builds Arm64 et x64 là où les anciennes versions de .NET et .NET Core ne fournissent que des builds x64 (qui se basent sur Rosetta 2 afin de tourner sur les machines équipées d'Apple Silicon). A priori, le support des puces Apple Silicon n'est pas prévu d'être porté sur .NET 5 ou les versions plus anciennes de .NET Core.

Hot reload

Un des grands piliers poussés par Microsoft est l'optimisation du cycle de développement des applications autour de son framework. C'est dans cette optique qu'une nouvelle fonctionnalité appelée « Hot reload » fait son apparition (ou « rechargement à chaud » pour les puristes de la langue française). Initialement mis en place par les équipes en charge de Xamarin pour le rechargement à chaud du XAML, il a été décidé de généraliser ce concept à l'ensemble du framework. Avec cette fonctionnalité, le développeur peut modifier le code de ses applications (code managé uniquement) pendant que celles-ci sont en cours d'exécution. Les modifications sont alors directement prises en compte et propagées sur l'application déjà lancée.

Pour cela, il suffit de réaliser une modification dans le code et de cliquer sur le nouveau bouton « Apply Code Changes » (« appliquer les changements ») dans Visual Studio afin d'appliquer ces modifications.

Microsoft mentionne que le Hot reload est supporté pour de nombreux types de projets tels que les applications WPF, ASP.NET Core (code-behind), console, WinUI3… Il est aussi disponible sur l'ensemble des applications qui sont basées sur les runtimes du Framework .NET et CoreCLR.

Avec .NET 6, le rechargement à chaud est disponible à partir de la ligne de commande dotnet watch. Pour rappel, l'utilitaire dotnet watch permet de surveiller les fichiers du projet pendant l'exécution afin de redémarrer celui-ci si une modification est apportée à un fichier. Désormais, les modifications seront rechargées à chaud sans redémarrage : à noter que dans le cadre de modifications nécessitant obligatoirement un redémarrage dotnet watch devrait demander une confirmation préalable du développeur.

Minimal web API & native-cloud

Microsoft souhaite faire du framework .NET un produit de premier choix lorsqu'il s'agit de produire une application cloud-native. Et cela passe forcément par une méthode de création simple et rapide de service. Jusqu'ici, la création des services nécessitait un certain nombre de codes d'initialisation : même pour la plus simple des API. Avec .NET 6, c'est désormais du passé avec l'arrivée des "minimal web API". Le concept est simple : avoir la possibilité de créer des services rapidement avec un minimum de code. Cela est particulièrement pratique pour les applications de type micro-services qui exposent de nombreuses API au scope limité.

Désormais la création d'un projet web :

```
dotnet new web
```

Génère le fichier unique suivant :

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/hello", () => "Hello, World!");

app.Run();
```

Cette nouvelle fonctionnalité permet d'éviter une initialisation fastidieuse grâce notamment :

- À la disparition du fichier Startup.cs
- Aux déclarations top-level dans le Program.cs (pas d'espace de nom, de classe ou de déclaration de méthode)
- Au nouveau modèle d'hébergement WebApplication.CreateBuilder
- Et à la mise en place des usings implicites

Ci-dessous, voici un exemple (provenant du devblogs de Microsoft) qui compare la quantité de code à écrire afin de mettre en place un service avec l'initialisation de Swagger/OpenAPI entre les versions ASP.NET Core web API en .NET 5 et ASP.NET Core web API en .NET 6 : **Figure1**

Initialisation en ASP.NET Core WebAPI en .NET 6

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddSwaggerGen(c =>
{
  c.SwaggerDoc("v1", new() { Title = "WebApplication22", Version = "v1" });
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
  app.UseSwagger();
  app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Web
Application22 v1"));
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

À noter que l'ancien modèle d'hébergement ainsi que le pattern Startup continueront d'être supportés.

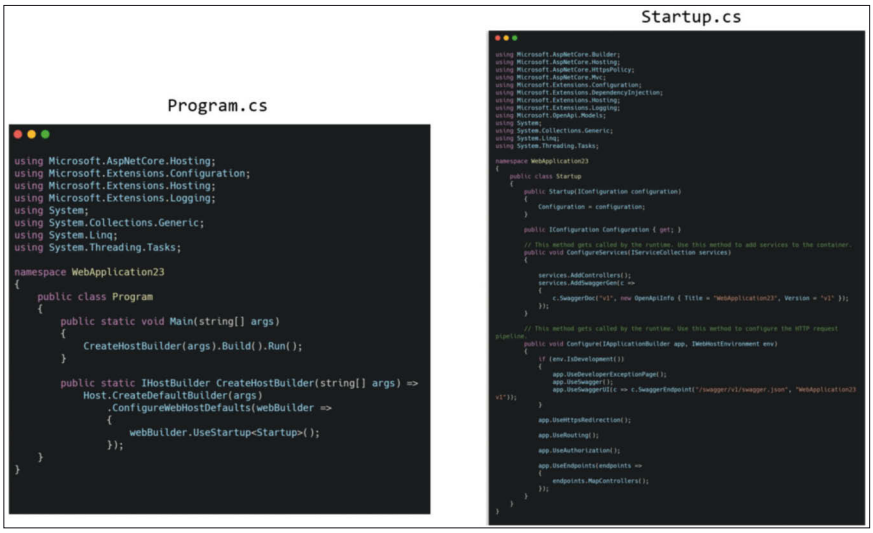


Figure 1

Initialisation en ASP.NET Core WebAPI .NET 5
Source : Microsoft

Bibliothèques

La bibliothèque de fonctionnalités du framework a été elle aussi enrichie pour cette nouvelle version. En voici un florilège :

Math

Un nouvel ensemble API mathématique fait son apparition dans System.Math (SinCos, ReciprocalEstimate...). Ces API sont orientées performance et seront accélérées par le hardware (si celui-ci le supporte). Le parsing des formats numériques standard et des BigInteger a, quant à lui, été amélioré.

JSON

.NET 6 agrémente la gestion du JSON de quelques fonctionnalités. Une nouvelle option ReferenceHandler.IgnoreCycles permet d'ignorer les références cycliques lors de la sérialisation. Une fois activée, les références cycliques seront remplacées par le token JSON null (contrairement à Newtonsoft.Json qui ignore la référence dans ce cas).

Il est désormais possible de (dé)sérialiser avec un IAsyncEnumerable<T> et d'utiliser des notifications exposées par le sérialiseur (au travers de l'implémentation des interfaces IJsonOnDeserialized, IJsonOnDeserializing, IJsonOnSerialized ou IJsonOnSerializing). Les propriétés d'un objet peuvent, dès à présent, être ordonnées lors de la sérialisation à l'aide de l'attribut System.Text.Json.Serialization.JsonPropertyOrderAttribute.

La nouvelle API « writable JSON DOM feature » offre la possibilité de modifier/requêter et transformer de grands arbres de données JSON et ce de manière performante. L'API propose aussi une syntaxe plus élégante et plus naturelle pour définir des objets JSON. À l'aide de celle-ci, il est aussi possible d'utiliser le mot clé dynamic qui permet l'utilisation de modèles faiblement couplés.

Pour finir, .NET 6 intègre une génération de source pour le JsonSerializer : l'idée est de générer le code source lors de la compilation afin d'éviter le processus d'analyse par réflexion qui a lieu au lancement de l'applicatif. Cela a pour effet de réduire le temps de lancement, d'améliorer les débits de sérialisation, de réduire l'empreinte mémoire et de supprimer l'utilisation de la réflexion au runtime.

LINQ

System.Linq introduit, lui aussi, des améliorations. Enumerable.ElementAt peut désormais prendre en paramètre un index partant de la fin de l'énumérable :

```
Enumerable.Range(1, 10).ElementAt(^2)
```

Dans la même veine, Enumerable.Take accepte un Range en entrée :

source.Take(..3) pour source.Take(3)
source.Take(3.) pour source.Skip(3)
source.Take(2. 7) pour source.Take(7).Skip(2)
source.Take(^3..) pour source.TakeLast(3)
source.Take(..^3) pour source.SkipLast(3)
source.Take(^7..^3) pour source.TakeLast(7).SkipLast(3)

La nouvelle méthode TryGetNonEnumeratedCount tente de récupérer le nombre d'éléments d'un énumérable sans réaliser d'énumération. Cette méthode vérifie si la source implémente ICollection/ICollection<T> ou tire avantage des optimisations internes propres à LINQ.

DistinctBy/UnionBy/IntersectBy/ExceptBy offrent de nouvelles variantes autorisant de spécifier une égalité en utilisant un sélecteur de clé et MinBy/MaxBy permettent de trouver le maximum/minimum à partir d'un même sélecteur de clé. La méthode Chunck permet de diviser un énumérable en tableau d'une taille fixe passée en paramètre :

```
IEnumerable<int[]> chunks = Enumerable.Range(0, 10).Chunk(size: 3);
// { {0,1,2}, {3,4,5}, {6,7,8}, {9} }
```

Jusqu'alors les méthodes FirstOrDefault/LastOrDefault/SingleOrDefault retournaient un default(T) si l'énumérable ne contenait aucune valeur. En .NET 6, il est possible de définir cette valeur de retour par défaut qui sera retourné si la source est vide. Pour terminer sur les Enumerable, Enumerable.Zip rend possible la fusion des trois énumérables en une série de tuple.

Autres

On note aussi pêle-mêle :

- Une nouvelle implémentation du FileStream
- Une meilleure gestion des Date, Time et TimeZone et une meilleure gestion des struct en tant que valeur dans les dictionnaires
- L'amélioration de la vérification et du cast des interfaces
- Le support des nint/uint pour Vector<T>
- Une nouvelle structure PriorityQueue
- L'apparition de nouveaux algorithmes de cryptage (ChaCha20 et Poly1305)
- Le support d'OpenSSL3 et d'OpenTelemetry Metrics API
- Une nouvelle implémentation du .NET thread pool

SDK

Avec l'arrivée de nouvelles plateformes supportées et l'ajout de nombreuses nouvelles fonctionnalités, on peut se poser la question de la taille du SDK qui ne cesse de grossir, notamment avec les nouveaux workloads mobiles.

Microsoft y a pensé et a introduit dans .NET 6 la notion de « workload optionnel ». L'idée est de passer d'un SDK monolithique vers de la composition de SDK. Il est ainsi possible d'ajouter de façon plus granulaire les éléments supplémentaires que chaque développeur souhaite installer dans son SDK. Cette fonctionnalité s'accompagne d'une ligne de commande dédiée :

```
dotnet workload
```

Cette ligne de commande permet d'installer, de désinstaller, de lister les workloads installés ou encore de les mettre à jour. Les workloads pourront aussi être manipulés par l'intermédiaire de l'installeur Visual Studio ou encore du gestionnaire de paquet de Linux. Le SDK devrait d'ailleurs à terme perdre du poids, car certains workloads déjà existants passeront dans un futur proche eux aussi optionnels. La firme de Redmond indique que cette nouvelle expérience "workloads SDK optionnels" sera présente dans .NET 6 et finalisée dans la prochaine version, .NET 7. Dans la même optique, un outil permettra désormais de vérifier si une nouvelle version du SDK et des runtimes est disponible. Pour cela, rien de plus simple, il suffit d'utiliser la commande suivante :

```
dotnet sdk check
```

Pour chacune des plages de fonctionnalités, il indique la dernière du SDK/runtime disponible et si les versions installées sont à jour.

Passer à .NET 6

Modification avec rupture

Avant de passer sur .NET 6, il est prudent d'analyser les changements qui créent une rupture avec les versions précédentes et qui nécessitent des modifications dans le code. Voici une sélection des modifications de ruptures marquantes (breaking changes) :

Réseau

Les classes WebRequest, WebClient et ServicePoint sont désormais dépréciées.

ASP.NET Core

- Les noms d'espace System.Security.Permissions et System.Windows.Extension ont été retirés des packs de ciblage et du runtime ASP.NET Core. Les noms d'espace Microsoft.Win32.SystemEvents et System.Drawing.Common ont, quant à eux, été supprimés du pack runtime d'ASP.NET Core, dans un but de réduire la taille du framework partagé d'ASP.NET Core.
- Le package permettant le support de MsgPack pour SignalR sur ASP.NET Core change de version. La dépendance du package npm @microsoft/signalr-protocol-msgpack passe de msgpack5 vers @msgpack/msgpack. De plus, l'interface MsgPackOptions voit ses champs disableTimestampEncoding et forceFloat64 supprimés.
- L'intergiciel de redirection HTTPS renvoie une exception s'il existe plusieurs ports définis dans la configuration : ceci pour éviter de possibles envois de données sensibles en HTTP lorsque HTTPS est disponible.
- Une nouvelle surcharge de UserExtensions.Use a été ajoutée à la release. Si le code de l'application appelle app.Use sans que l'intergiciel correspondant n'appelle l'intergiciel suivant (en utilisant await next()) alors une erreur de compilation sera levée. Dans ce cas, il sera nécessaire de le remplacer par app.Run. Exemple de code nécessitant une modification :

```
app.Use(async (context, next) =>
{
  await SomeAsyncWork();
  // next n'est pas appelé...
});
```

Razor

- Le compilateur Razor ne crée plus d'assembly séparée Views.dll contenant les vues CSHTML de l'application. Désormais, une seule assembly contenant les types de l'application ainsi que les vues est générée (par défaut les vues générées ont les modificateurs d'accès internal ou sealed et sont placées sous le nom d'espace AspNetCoreGeneratedDocument).
- Les types relatifs au type RazorEngine sont désormais obsolètes.

Blazor

- Modification d'une coquille résiduelle dans le nom du champ maxWith qui passe à maxWidth.
- L'interopérabilité avec les tableaux d'octet a été revue : il n'est désormais plus nécessaire de décoder l'entrée à partir de la Base64 :

```
function receivesByteArray (data) {
  // data est désormais un Uint8Array > plus besoin de décoder la Base64
}
```

- Le LongPolling n'est plus utilisé en solution de repli si les WebSockets ne sont pas disponibles (problèmes réseau ou non-compatibilité par exemple). Le client et le serveur supportent désormais uniquement les WebSockets : un message d'erreur est renvoyé dans les cas où au moins l'un des deux parties ne supporte pas les WebSocket.

Bibliothèque Core .NET

- L'équipe .NET abandonne le support des frameworks plus anciens que .NET Framework 4.6.1, .NET Core 3.1 et .NET Standard 2.0. Si un projet référence un paquet d'un framework antérieur, il ne sera plus possible de mettre à jour ce paquet vers une version ultérieure.
- Le comportement de la propriété Environment.ProcessorCount sur Windows (qui fournit le nombre de processeurs disponibles pour le processus courant) est désormais aligné avec le comportement sur Linux. Auparavant celui-ci retournait le nombre de processeurs logiques sans prendre en compte l'affinité des processus ni les limitations d'utilisation des CPUs. Le code dépendant de cette propriété sera donc peut-être à revoir.
- La propriété FileStream.Position est mise à jour une fois que les opérations de ReadAsync/WriteAsync sont terminées contrairement à avant où elle était mise à jour une fois les opérations démarrées.
- Microsoft informe qu'avec les surcharges ajoutées à la classe System.Linq.Queryable, il peut y avoir un impact sur du code utilisant la réflexion si celui-ci n'est pas assez robuste. Par exemple :

```
typeof(System.Linq.Queryable)
  .GetMethods(BindingFlags.Public | BindingFlags.Static)
  .Where(m => m.Name == "ElementAt")
  .Single();
```

Pour consulter la liste exhaustive des modifications avec ruptures, c'est par ici : https://docs.microsoft.com/en-us/dotnet/core/compatibility/6.0.

Migration vers .NET 6

Voir article dédié à la migration vers .Net 6.

Conclusion

.NET 6 s'inscrit dans la lignée directe du travail d'unification initié dans .NET 5. De nombreuses améliorations ont été apportées d'un point de vue fonctionnalités, mais aussi au niveau des performances. Des efforts ont été faits afin de rendre l'expérience autour du framework plus accessible. Microsoft souhaite attirer de nouveaux développeurs et ainsi étoffer la communauté .NET. Cela a pour objectif de placer .NET comme un concurrent légitime et polyvalent ! Reste à voir l'impact qu'auront les apports de cette nouvelle version sur la communauté. L'adoption sera-t-elle au rendez-vous ? .NET Conf 2021 du 9 au 11 novembre 2021 pour le lancement officiel de .NET 6 (https://www.dotnetconf.net/) !

Jérémy Jeanson
Ingénieur d'Études et Développement
Inetum Software

Préparer une application avant sa migration vers .net 6

Migrer une application vers la dernière version de .net n'est jamais une mince affaire. Heureusement, Microsoft a documenté bon nombre de procédures de migration, et breaking changes. Pour vous aider, vous pourrez aussi compter sur la communauté des développeurs et Programmez!.

Mais avant de partir pour un long périple :

- Votre application est-elle prête ?
- Votre dette technologique actuelle vous permet d'envisager sereinement une migration ?
- Votre roadmap prévoit une migration dès la publication de .net 6 ?

N'ayez pas peur de dire non. Vous ne serez pas seul dans ce cas. Cet article va vous aider à préparer votre application, et vous assurez que la migration vers .net 6 ne sera pas un cauchemar.

Anticiper les API dépréciées

Dès qu'une nouvelle version de .net pointe le bout de son nez, j'ai l'habitude d'aller sur docs.microsoft.com pour consulter la liste des breaking changes. Je ne peux que vous encourager à en faire de même afin d'identifier les classes et méthodes pour lesquelles il vous faudra trouver une alternative.

Je vous conseille de rechercher la page des nouveautés. Les API dépréciées y sont toujours indiquées.

Mais avant cela, il existe une solution plus rapide : il suffit de consulter la liste des warnings (vous savez : la liste que certains masquent après une première compilation). Parmi les warnings se cachent des messages liés aux API obsolètes. Bien souvent, les classes et méthodes sont marquées comme étant obsolètes par Microsoft pour anticiper des changements profonds à venir. Ceux-ci n'auront peut-être pas lieu lors de la prochaine itération de .net. Mais il ne faut pas se leurrer, ils auront lieu un jour.

Si vous prenez l'habitude de traiter les éléments obsolètes comme s'il s'agissait d'erreurs, vous ne devriez pas avoir trop de difficultés lors de la migration.

Solder sa dette technologique

L'apparition et la disparition d'API ont souvent pour vocation de maintenir .net dans l'air du temps. Si vous n'avez pas suivi les évolutions de .net de ces dernières années, ou que vous n'êtes pas passé à .net core, il est possible que vous mainteniez aujourd'hui des codes « datés ». Si en plus vous avez gardé du code créé pour .net 1 ou 2, vous avez certainement une dette technologique importante.

Dans les applications .net on peut trouver : des web services Asmx, du Remoting, WCF, Workflow Foundation, Authentication basic, BinaryFormatter, Code Access Security, AppDomain, SecureString, Edmx pour Entity Framework, Silverlight…

Ces éléments novateurs à leurs sorties sont aujourd'hui très compliqués et coûteux à maintenir. Même s'il peut sembler plus facile de rester sous .net 4.8, il ne peut s'agir que d'une situation temporaire. Plus cette situation durera et plus votre dette technologique augmentera.

Plutôt que de subir, il faut donc commencer à chercher des solutions de notre temps : Web API, Dapper, Open Auth, Entity Framework via code First (le tooling d'EF peut aussi permettre de créer un DataContext à partir d'une base de données existante).

Exemple : Si votre application expose un service SOAP. Celui-ci sera impossible à migrer tel qu'il est aujourd'hui. Plutôt que de chercher une nouvelle solution pour SOAP, il est préférable d'exposer dès à présent une version REST/XML ou JSON. Cette version deviendra la version recommandée. Si des tiers doivent utiliser vos services, il convient de les prévenir que la version SOAP est obsolète et ne sera plus supportée après votre migration.

Ne vous leurrez pas, si vous utilisez aujourd'hui .net core 1 votre situation n'est pas forcément plus enviable. Pour pallier aux manques de .Net core 1, nombre de développeurs ont été dans l'obligation de réimplémenter des fonctionnalités du .net Framework. Certains se sont retrouvés à utiliser des librairies du .net Framework dans leurs applications .net core. Ce code peut fragiliser votre application et creuser votre dette technologique. Il faut donc savoir prendre le temps d'identifier les portions de codes et les marquer avec l'attribut [Obsolete]. Quand la migration débutera, il sera alors plus facile de les remplacer.

Moderniser son architecture

En voulant solder sa dette technologique, il peut sembler plus aisé de réécrire certains composants. Pourquoi ne pas en profiter pour utiliser .net 6 dès à présent ?

Dans le cas où votre application est constituée de plusieurs tiers distincts, cela est tout à fait possible.

Exemple : Le frontend continue d'utiliser votre version actuelle de .net alors que votre backend exploite .net 6.

Dans le contexte d'une application monolithique, vous pourriez être tenté d'utiliser .net standard. Même si l'idée peut sembler séduisante, elle n'est pas adaptée à toutes les situations. En fonction de la version de .net standard exploitable avec votre version actuelle de .net vous ne disposez pas des mêmes API. Un projet .net standard est donc soumis à des contraintes qui peuvent compliquer les développements. Cette approche peut être très chronophage (surtout avec les versions de 1.0 à 1.6). Il est fortement recommandé de vérifier vos dépendances avant de vous lancer.

Identifier et sécuriser les chemins critiques

Aujourd'hui nos applications sont de plus en plus complexes et évoluées. Migrer celles-ci ne se limite donc plus simplement à effectuer une mise à jour, recompiler puis exécuter.

Plus une application à de fonctionnalités et plus il y aura d'éléments à vérifier. Avec le temps, il n'est pas rare que seuls quelques développeurs aient connaissance de l'ensemble des scénarios possibles. Même quand on travaille seul, il arrive que l'on ou-

blie l'existence de fonctionnalités développées quelques années auparavant.
Pour ne rien oublier, l'idéal serait d'avoir un cahier de recettes couvrant la totalité des cas.
Malheureusement, ce type de document évolue souvent moins vite que nos applications et rares sont les développeurs qui y prêtent attention. Ce qui se traduit souvent par de nombreux aller-retour entre les équipes de développement et les testeurs. Ceux-ci conduisent inévitablement à une perte de temps et d'énergie. Avec le temps, ce phénomène ne fera que s'accentuer et amènera vos équipes à résister face au changement. La migration pouvant impliquer de grands changements, il faut trouver une réponse au problème dès aujourd'hui.
Par chance, il existe une solution spécialement adaptée pour les développeurs. Il s'agit des tests unitaires. Mais là encore, il est rare qu'une application soit intégralement couverte par des tests unitaires.
Pour éviter des drames lors de la migration, il faut trouver le moyen d'optimiser l'usage du cahier de recette et des tests unitaires codés.
Présenter de la sorte, le chantier peut sembler colossal. Si vous partez de zéro, il ne faut pas s'inquiéter. L'approche la plus efficace consiste à procéder par itérations. Chaque itération consistant en :

- L'identification d'un petit nombre de fonctionnalités importantes pour les utilisateurs.
- La documentation des scénarios d'usages de ces fonctionnalités pour le cahier de recette.
- Le codage de tests unitaires relatifs à ces fonctionnalités.

Progressivement, vos applications seront davantage couvertes par des tests unitaires. Lors des publications pour les testeurs, ceux-ci vous remontrent moins de bugs. Ce que l'on appelle couramment le chemin critique finira par être entièrement sécurisé. Avant même d'avoir commencé la migration, votre application aura gagné en qualité. Quand le projet de migration débutera, vous pourrez rapidement vérifier son impact et éviter tout effet de bord avant même de publier vos premières versions en tests.
Par le passé, cette démarche m'a permis d'éviter de nombreux problèmes. Voici quelques cas concrets :

- Lors du passage de .net 1.0 à 1.1 j'ai été confronté à des erreurs du fait de Stream dont la méthode Flush() n'était pas appelée avant la méthode Close(). Ce problème était indétectable si je n'avais pas eu un test qui vérifiait le contenu du fichier généré.
- Lors d'une migration vers ASP .net core 3, il est devenu impossible de manipuler les Request et Response dans un Middlerware sans utiliser leurs méthodes asynchrones. Les méthodes synchrones existant toujours, la compilation se produisait sans problème. Les tests unitaires d'intégration étaient la seule solution pour anticiper les problèmes et leur trouver une solution.

Réduire la voilure

Parmi les nombreuses fonctionnalités de nos applications, il n'est pas rare que certaines ne soient plus utilisées. Il n'est pas rare non plus que l'on garde du code mort (des méthodes, ou classes qui ne servent plus). Ces situations peuvent sembler bénignes. Mais que se passera-t-il si après le passage à .net 6 on découvre qu'elles produisent une erreur de compilation. Faudra-t-il les corriger ou les abandonner ? Que se passera-t-il si l'on découvre au dernier moment que ces codes dépendent de librairies qui ne supportent pas .net 6 ?
Pour ne pas avoir à prendre des décisions à la hâte, et ne pas perdre de temps, il convient d'identifier très vite ces fonctionnalités et codes morts. S'ils ne sont plus utiles, autant les supprimer tout de suite.
Bien évidemment, je ne vous encourage pas à supprimer votre code de manière barbare. L'idée ici est de vous faire prendre conscience qu'il faut savoir réduire la voilure de son application plutôt que de subir. Parmi ces fonctionnalités, il est aussi probable que certaines n'auront plus de raison d'être après la migration. Si la suppression ne peut pas avoir lieu actuellement, il faut penser à parquer le code concerné avec l'attribut [Obsolete].
Exemple : je maintiens seul depuis 2011 une application qui a débuté sur Windows en WPF, puis Windows Phone 7 en SilverLight, et Windows 8, pour finir sur Windows 10 avec UWP. Dans le temps, j'ai été obligé de supprimer des fonctionnalités telles que le support de l'API de recherche de Windows 8. Pour supporter Windows 11, il est évident que je vais me séparer du code dédié à la gestion des tuiles dynamiques.

Gestion des dépendances

La gestion des dépendances est une étape critique de la migration. Il existe deux manières de référencer des librairies externes :

- L'approche manuelle, qui consiste à ajouter une référence en allant chercher une DLL sur son PC.
- L'approche automatisée via un gestionnaire de paquets comme NuGet.

La première approche est celle qui présente le plus d'inconvénients :

- Les DLL doivent être déployées sur le PC de développement (via le repository, XCopy, ou MSI)
- La mise à jour d'une même DLL pour plusieurs projets est plus laborieuse.
- Les dépendances indirectes ne sont pas facilement identifiables.
- Les dépendances indirectes ne sont pas référencées automatiquement.
- Il n'est pas possible de référencer les variantes x86 et x64 d'une même DLL dans un projet.

A contrario, NuGet :

- Télécharge, référence automatiquement les DLL, et leurs références indirectes.
- Permet la mise à jour en un clic de tous les projets d'une solution.
- Permet de gérer facilement des profils de compilation x86, x64 ou autres.

De plus, Nuget a un grand intérêt dans le contexte d'une migration : il permet de connaître les dépendances indirectes d'un paquet en fonction de la version de .net ciblée. La liste de dépendances d'un paquet est consultable via l'interface de gestion NuGet de Visual Studio.
Même si l'interface de Visual Studio est agréable, je ne la trouve pas adaptée dans le cadre d'une migration. Je vous conseille plutôt d'aller sur le site https://www.nuget.org/ et d'y rechercher les librairies que vous utilisez.
Exemple : Pour Open-XML-SDK **Figure 1**
Le grand intérêt du site Nuget par rapport à Visual Studio est qu'il permet de consulter la page du paquet d'une dépendance indirecte pour valider ses propres dépendances. De la sorte, il est possible de vérifier l'ensemble des dépendances directes ou indirectes, et de valider leur compatibilité avec .net 6. Dans le cas contraire, il faudra rechercher une librairie de substitution compatible.
En lisant ces quelques lignes, je pense que vous aurez compris que l'adoption de NuGet est indispensable. Continuer à référencer manuellement ses dépendances n'est pas raisonnable et s'avérera extrême-

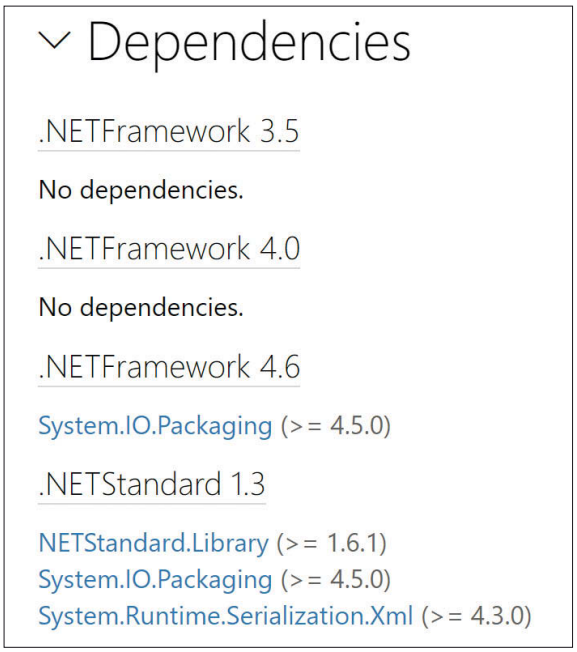


Figure 1

ment contraignant dans le futur (sans compter le nombre d'erreurs humaines qui peuvent en découler).

Si vous n'utilisez pas encore NuGet, commencez dès à présent. Ceci facilitera grandement votre migration.

NuGet a en plus l'avantage de fournir des informations chiffrées sur la popularité d'une librairie et sur le fait qu'elle soit activement maintenue ou non. Avant d'installer un paquet, prenez le temps de vérifier le nombre de téléchargements ainsi que la fréquence à laquelle celui-ci est mis à jour. Un paquet qui n'a pas eu de mise à jour depuis de nombreuses années est à éviter.

Pour les irréductibles des références manuelles

Si vous n'avez pas le choix et devez continuer à référencer manuellement vos dépendances, il y a cependant une approche qui peut vous permettre de préparer votre migration tout en limitant les risques de rencontrer un problème.

Celle-ci réside dans la procédure suivante :

- Stocker toutes vos dépendances directes et indirectes dans un répertoire unique. Ce répertoire doit être géré par votre gestionnaire de code source (Git, TFVC, Subversion…).
- Dans la liste des références de chaque projet, il ne faut pas que l'option « utiliser cette version spécifique » soit cochée.

Lors de la migration, les DLL présentes dans ce répertoire pourront être remplacées par leurs versions .net 6.

Gardez tout de même à l'esprit que cette solution ne résout qu'une partie du problème : l'identification des dépendances directes et indirectes. Ceci facilitera grandement la recherche de versions compatibles avec .net 6 et la mise à jour. Mais contrairement à NuGet et sa gestion via PackageReference, il n'est pas exclu que vous ayez à effectuer quelques opérations manuelles pour que Visual Studio prenne en charge vos dépendances.

Note : *Il est fréquent d'enregistrer manuellement les DLL des librairies que l'on achète auprès de fournisseurs tel que DevExpress, Infragistics, SyncFusion, etc. … Historiquement, il n'y avait pas d'autres solutions. Aujourd'hui, nombre de fournisseurs mettent à disposition des paquets pour NuGet ainsi que des serveurs Nuget pour faciliter les mises à jour. Si vous ne le saviez pas, je vous encourage vivement à consulter la documentation de votre fournisseur afin de migrer vos références vers Nuget avant de passer à .net 6.*

Lors de l'installation, Infragistics a ajouté un répertoire pour permettre aux développeurs d'utiliser Nuget plus facilement.

Les bienfaits de l'injection de dépendances

Le pattern d'injection de dépendance est l'alliée incontournable d'une migration réussie. Si vous ne l'avez jamais utilisé jusqu'ici, il est temps de changer d'habitude.

Dans le cadre d'une application qui doit migrer vers .net 6, l'intérêt est double :

- Le fait de découpler les différents éléments de l'application rend facilement remplaçables ceux-ci. L'objectif est de pouvoir disposer de classes qui n'ont pas besoin de savoir comment sont instanciées les classes dont elles dépendent. Dans le cas d'une migration, cela permet de limiter les modifications à opérer du fait d'un changement d'API (ou dans le cas où celui-ci disparaîtrait).
- Dans le cas d'une application ASP .net, il est impossible de ne pas l'utiliser. L'injection est à la base de .net MVC depuis .net core. L'utiliser dès aujourd'hui permet donc d'être paré pour ASP .net 6.

Si vous utilisez déjà .net core, je vous conseille vivement d'utiliser le Microsoft.Extensions.Hosting pour vos applications desktop et mobiles. Celui-ci offre de très bonnes performances. Il a aussi l'avantage de partager les mêmes interfaces que l'hôte d'injection d'ASP .net.

Cerise sur le gâteau, cette librairie peut être utilisée avec .net Framework 4.6.1. L'utiliser dès à présent dans tous vos développements vous fera gagner un temps d'avance.

Attention : L'injection sert aussi à lutter contre l'usage abusif d'objets intrinsèques et autres éléments statiques (singletons, et variables statiques). Ceux-ci rendent souvent la découverte des dépendances difficiles. Depuis .net core, nombre d'entre eux n'existent plus (exemple : HttContext.Current). Il est donc très important de considérer l'injection pour la gestion du cycle de vie de ses objets dès aujourd'hui, plutôt que de reposer sur des variables statiques.

Revoir sa configuration

Si vous utilisez toujours le .net Framework, il est fort probable que vous utilisiez des fichiers de configuration XML (app.config ou web.config) pour définir la configuration de vos applications. Depuis .net core, ceux-ci ont disparu au profit de fichiers *.json plus souples à manipuler.

Même si rien ne vous oblige vous pouvez déjà utiliser des fichiers de configuration *.json. Le .net Framework contient tout ce qu'il faut pour charger ce type de fichiers, et injecter des paramètres dans vos applications. Cerise sur le gâteau, vous pouvez dès à présent profiter de cette approche pour utiliser des classes fortement typées.

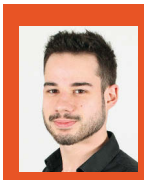
Fini les conversions effectuées manuellement, les tests sur le ConfigurationManager et les questions existentielles sur les noms des paramètres ou leurs types.

Il est aussi très important de noter que la notion de « section custom » disparaît avec les fichiers XML. Après votre passage à .net 6, la configuration se fera exclusivement via votre code. Si vous utilisez déjà l'injection de dépendance, il peut être intéressant de l'utiliser pour injecter les classes qui remplaceront les sections.

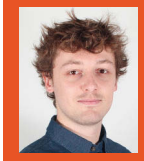
Si vous ne souhaitez pas vous passer tout de suite de vos fichiers de configuration XML, il faut cependant prendre le temps de les examiner avant la migration. Ceux-ci peuvent cacher des dépendances à des fonctionnalités du .net Framework (sécurité, authentification, log…). Ils peuvent aussi faire référence à des dépendances tiers qui seraient déployées dans le Global Assembly Cache (GAC) ou qui ne seraient pas référencées dans votre liste de package NuGet. Découvrir celles-ci au dernier moment pourrait être catastrophique et ruiner votre planning. J'ai déjà raté cette étape lors de l'étude « un peu trop rapide » d'une application à migrer. Je m'en suis mordu les doigts.

Les nouveautés de C# 10

Depuis la version .NET Core 3.1 en 2019, Microsoft a planifié de mettre à jour la version de son framework .NET chaque mois de novembre. 2020 a marqué la sortie de .NET 5, ainsi que de son langage phare C# en version 9. Cette année, Microsoft continue sur sa lancée et annonce la sortie de .NET 6 accompagnée de C# 10 dont nous vous proposons de décortiquer les nouveautés et améliorations.



Baptiste Bazureau
Expert technique
SQLI



François Lefebvre
Expert technique
SQLI

SQLI DIGITAL EXPERIENCE

Nous vous précisons qu'au moment où nous rédigeons ces lignes, C# 10 n'est pas encore sorti dans sa version définitive. Certaines fonctionnalités pourraient encore évoluer, voire ne pas être embarquées.

Les structures d'enregistrement

En 2020, C# 9 a introduit le mot clé **record** qui permet de définir un type référence fournissant des fonctionnalités intégrées pour l'encapsulation des données. Il était jusqu'ici possible de l'affecter uniquement à des classes, ce que C# 10 étend désormais aux structures. Les structures d'enregistrement sont de type valeur tout comme le sont les structures classiques. Cela signifie qu'elles répondent à des règles communes. À noter que C# 10 va lever certaines restrictions sur les structures. Il rend possible la création d'un constructeur sans paramètre, ainsi que l'initialisation des propriétés dès leur déclaration, comme le montre l'exemple suivant :

```
public struct Point
{
  public Point() // constructeur sans paramètre
  {
    X = 5;
  }

  public int X { get; set; }

  public int Y { get; set; } = 3; // initialisation à la déclaration
}
```

La déclaration d'une structure d'enregistrement

Il devient dès lors possible de déclarer une structure d'enregistrement sur une seule ligne :

```
public readonly record struct Point(int X, int Y);
```

Notons plusieurs points dans cette déclaration : tout d'abord la présence du mot clé **readonly** qui n'est pas disponible pour les classes d'enregistrement. Son ajout optionnel permet de spécifier que les propriétés de la structure d'enregistrement sont immuables. En effet, elles ne le sont pas par défaut, ce qui constitue l'une des différences majeures avec les classes d'enregistrement. De plus, les différents paramètres déclarés seront convertis en propriétés par le compilateur, et le constructeur ainsi généré les renseignera à l'initialisation d'une nouvelle instance. Cette syntaxe est donc équivalente à celle ci-dessous, qui utilise le mot clé **init** introduit en C# 9 rendant les propriétés immuables :

```
public record struct Point
{
  public Point(int x, int y)
  {
    X = x;
    Y = y;
  }

  public int X { get; init; }

  public int Y { get; init; }
}
```

Ensuite, nous remarquons la présence du mot clé **struct** qui spécifie qu'il s'agit bien d'une structure d'enregistrement, et non une classe d'enregistrement. En C# 9 les classes d'enregistrement étaient déclarées sans le mot clé **class**:

```
public record Person(string FirstName, string LastName);
```

Cela reste possible, mais C# 10 a rendu faisable l'ajout optionnel du mot clé **class** afin d'éviter les confusions.

```
public record class Person(string FirstName, string LastName);
```

Les expressions with

Tout comme avec les classes d'enregistrement, les structures d'enregistrement peuvent être instanciées à l'aide des expressions **with**. Celles-ci permettent de créer une copie d'une structure d'enregistrement tout en lui spécifiant une ou plusieurs données différentes. Prenons l'exemple suivant :

```
var firstPoint = new Point(1, 2);
var secondPoint = firstPoint with { Y = 3 };

Console.WriteLine(secondPoint);
// sortie : Point { X = 1, Y = 3 }
```

La propriété *X* de *secondPoint* est issue de l'instance de *firstPoint* tandis que la propriété *Y* est définie grâce à l'utilisation de l'expression **with**.

La comparaison d'égalité

L'égalité entre deux structures d'enregistrement s'effectue en fonction de leurs valeurs. Nous pouvons donc utiliser le mot clé **Equals** afin de les comparer comme nous le ferions pour les structures classiques, mais également les opérateurs == et != qui sont utilisables uniquement pour les structures d'enregistrement :

```
var firstPoint = new Point(1, 2);
var secondPoint = new Point(1, 2);

Console.WriteLine(firstPoint.Equals(secondPoint));
// sortie : true
```

```
Console.WriteLine(firstPoint == secondPoint);
// sortie : true
```

Afficher les membres avec la méthode ToString

C# 9 a introduit la possibilité d'afficher facilement les membres d'une classe d'enregistrement grâce à la substitution automatique de la méthode ToString. Cette fonctionnalité a également été introduite avec C# 10 pour les structures d'enregistrement :

```
public readonly struct record Point(int X, int Y);
public void Main()
{
  Point point = new (1, 2);
  Console.WriteLine(point);
  // sortie: Person { X = 1, Y = 2 }
}
```

La valeur de sortie est obtenue grâce à la substitution automatique de la méthode ToString du type *Point*. La méthode itère sur chaque membre de la structure et indique sa valeur grâce à une méthode interne nommée PrintMembers et implémentée automatiquement. Nous verrons dans la section suivante, un autre ajout de C# 10 qui concerne la possibilité de sceller cette méthode ToString pour le cas des classes d'enregistrement.

Les types d'enregistrements peuvent sceller ToString

Les classes d'enregistrement supportent l'héritage, ce qui est pris en compte par la substitution de la méthode **ToString**. Dans ce cas, la méthode itère sur l'ensemble des membres de la classe mère, puis de la classe fille :

```
public record Person(string FirstName, string LastName);
public record Employee(string FirstName, string LastName, int Grade)
  : Person(FirstName, LastName);

public void Main()
{
  Employee employee = new("John", "Doe", 1);
  Console.WriteLine(employee);
  // sortie: Employee { FirstName = John, LastName = Doe, Grade = 1 }
}
```

Considérons désormais que dans l'exemple précédent, nous souhaitions uniquement afficher la propriété *FirstName* des objets de type *Person* et des classes qui en héritent. Pour cela, nous allons modifier l'implémentation de la méthode **ToString** de la classe *Person* grâce au mot clé **override**. Cependant, cela n'aura pas d'impact sur les variables de type *Employee*. En effet, la méthode **ToString** de la classe fille sera tout de même automatiquement substituée et ne prendra pas en compte la modification apportée :

```
public record Person(string FirstName, string LastName)
{
  public override string ToString() => FirstName;
}

public record Employee(string FirstName, string LastName, int Grade)
  : Person(FirstName, LastName);

public void Main()
{
  Person person = new("John", "Doe");
  Console.WriteLine(person);
  // sortie: John

  Employee employee = new("John", "Doe", 1);
  Console.WriteLine(employee);
  // sortie: Employee { FirstName = John, LastName = Doe, Grade = 1 }
}
```

C# 10 résout cela en permettant de sceller la méthode **ToString** d'une classe mère grâce au mot clé **sealed**, afin de prévenir la substitution automatique de la méthode **ToString** pour les classes qui en héritent et ne pas avoir à modifier leur implémentation. Voici le résultat obtenu avec l'utilisation du mot clé **sealed** :

Code complet sur programmez.com & github

Assignation et déclaration dans une même déconstruction

Déconstruire un objet en C# permet d'assigner facilement les membres d'un objet à des variables. Pour cela, il faut tout d'abord implémenter une ou plusieurs méthodes **Deconstruct** dans une classe :

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public void Deconstruct(out string firstName, out string lastName)
  {
    firstName = FirstName;
    lastName = LastName;
  }
}
```

Sans C# 10, il faut choisir exclusivement entre deux syntaxes différentes pour déconstruire l'objet. La première syntaxe consiste à initialiser les variables puis déconstruire l'objet sur une seule et même ligne de code, ce qui a l'avantage d'être simple et concis :

```
(string firstName, string LastName) = person;
```

La seconde syntaxe permet d'instancier au préalable les variables assignées lors de la déconstruction. Néanmoins cette seconde syntaxe est plus longue que la première :

```
string firstName = "Jane", LastName;

(firstName, LastName) = person;
```

Il n'était jusqu'ici pas possible de mélanger ces deux syntaxes. Cela signifie que si nous avions besoin d'avoir la main sur la déclaration d'une seule des variables dédiées à la déconstruction d'une classe, nous étions tout de même obligés de déclarer chacune d'entre elles. C# 10 permet dorénavant de mixer ces deux syntaxes en fonction des besoins :

```
string firstName = "Jane";

(firstName, string LastName) = person;
```

Les directives global using

Dans un projet C#, il est fréquent d'utiliser les mêmes types issus des mêmes espaces de noms, ce qui provoque la multiplication de directives **using** similaires à travers les fichiers. Pour éviter cette répétition, C# 10 permet de rendre disponible des namespaces à l'ensemble des fichiers de code d'un projet. Pour cela, il suffit d'ajouter le mot clé **global** précédant une instruction **using** et l'espace de noms devient disponible globalement.

Par exemple, si l'on crée une application web à partir du modèle ASP.NET Core avec authentification, le *HomeController* généré ressemble à cela, sachant que le contenu de la classe est omis :

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Diagnostics;
using WebApplication.Models;

namespace WebApplication.Controllers;
[Authorize]
public class HomeController : Controller {}
// Le contenu de la classe est omis
```

Il y a de fortes chances pour que les autres contrôleurs du projet utilisent aussi les namespaces *Microsoft.AspNetCore.Mvc* et *Microsoft.AspNetCore.Authorization*.

Avec C# 10, nous pouvons créer un fichier GlobalUsings.cs à la racine du projet avec le code suivant :

```
global using Microsoft.AspNetCore.Authorization;
global using Microsoft.AspNetCore.Mvc;
```

Nous pouvons ainsi retirer ces **usings** dans le fichier HomeController.cs, qui devient donc :

```
using System.Diagnostics;
using WebApplication.Models;

namespace WebApplication.Controllers;
[Authorize]
public class HomeController : Controller {}
// Le contenu de la classe est omis
```

S'il est possible d'utiliser des **global using** à travers différents fichiers, peu importe l'endroit dans la structure du projet, la portée restera globale à l'ensemble du projet. Il est néanmoins préférable de regrouper tous les **global using** dans un même fichier pour en faciliter la lecture. *GlobalUsings.cs* n'est pas un nom obligatoire, mais la communauté semble avoir adopté en partie cette convention. Il est aussi fréquent de voir ce fichier se nommer *usings.cs* ou plus rarement *import.cs*. À noter que Visual Studio indiquera si une instruction devient facultative si l'espace de nom est déjà importé via un **global using**.

La déclaration d'espace de nom de portée de fichier

En C#, il est possible d'utiliser plusieurs namespaces dans un même fichier :

```
namespace MyApp.Accounts
{
    class Employee { }
}

namespace MyApp.Billing
{
    class Bill { }
}
```

Dans la plupart des projets, les fichiers de code ne contiennent qu'un seul namespace. Une nouvelle syntaxe introduite avec C# 10 permet de déclarer un espace de nom en haut de fichier, et dont la portée s'étend à l'ensemble du fichier. Pour cela il suffit d'utiliser le mot clé **namespace** sans les accolades :

```
namespace Acme.Software;
```

Cette syntaxe améliore la lisibilité horizontale du code en réduisant l'indentation d'un niveau.

Patterns de propriétés étendus

Historiquement, le mot clé **"is"** était un opérateur pour tester uniquement le type et convertir une donnée. C# 7 a introduit le pattern matching – se traduisant par « critères spéciaux » – qui permet de tester si une donnée correspond à un pattern. Le pattern matching a systématiquement continué à être amélioré depuis, et cette version ne déroge pas à la règle.

Jusqu'alors, pour faire un test de correspondance envers des propriétés imbriquées, il nous fallait utiliser un modèle avec des objets imbriqués. Prenons l'exemple suivant : une voiture est composée d'une roue, et la roue possède une taille. Nous devons effectuer une action spéciale si l'on traite une voiture avec une roue de 17 pouces. Voici l'utilisation d'un pattern matching en C# 9 :

```
record Car(Wheel Wheel);
record Wheel(int Size);
if (vehicule is Car { Wheel: { Size: 17 } })
    DoSomething();
```

Avec C# 10, la syntaxe pour accéder à une propriété dans un pattern a été simplifiée. Nous pouvons désormais référencer directement une propriété imbriquée.

```
if (vehicule is Car { Wheel.Size: 17 })
    DoSomething();
```

La concaténation de constantes avec des chaînes interpolées

Avec C# 9, s'il est possible de concaténer des constantes pour former une autre constante en utilisant l'opérateur +. A contrario, le compilateur ne permet pas d'utiliser une expression d'interpolation pour effectuer cette opération.

```
const string firstname = "toto";
const string helloPlus = "Hello " + firstname;

const string hello = $"hello {firstname}";
// error CS0133: The expression being assigned to 'hello' must be constant
// Cette erreur intervient, car une chaîne interpolée ne peut être assignée à une
constante

static readonly string helloStatic = $"Hello {firstname}";
// cette syntaxe est valide dans une variable – ici une propriété statique
```

C# 10 nous permet aujourd'hui d'assembler des chaînes de caractères via une chaîne interpolée et de stocker le résultat dans une constante. La valeur est définie à l'exécution, et non pas à la compilation. Mais il n'y a pas d'inquiétude à avoir pour l'influence de la culture, car seules des concaténations de chaînes sont autorisées.

```
const string hello = $"hello {firstname}"; // Valide avec C# 10

const float ratioMilesInKilometers = 1.60934f;
const string mile = $"One mile is {ratioMilesInKilometers} km";
// error CS0133: The expression being assigned to 'InterpolatedString.mile' must be constant
```

Nous obtenons ainsi un code plus lisible. L'autre avantage est de pouvoir utiliser l'interpolation de string à des emplacements où seules des constantes sont acceptées, comme dans un attribut.

```
const string PRODUCT_VERSION = "Hirsute Hippo";

[Osolete($"This method will be removed after version {PRODUCT_VERSION}")]
void Foo() { }
```

Les améliorations des fonctions lambda

C# 10 apporte quelques améliorations autour des expressions lambda. L'inférence de type a été améliorée. Il n'est dorénavant plus nécessaire de préciser le type d'une variable contenant une fonction lambda. Nous pouvons simplement la déclarer avec le mot clé **var**.

```
var returnOne = () => 1;
```

Jusqu'à C# 9, cela aurait produit une erreur de compilation (CS0815: Cannot assign 'expression' to an implicitly typed local). Pour faciliter le travail du compilateur ou modifier le type de retour d'une expression lambda, celui-ci peut être précisé. Pour se faire, il faut préfixer la fonction lambda par le type de retour souhaité.

```
Func<int, float> incrementAsFloat = float (int x) => x + 1;
```

Dans l'exemple précédent, la fonction lambda prend en paramètre un entier qu'elle additionne avec la valeur *1* qui est aussi un entier. Le résultat de la fonction lambda est donc lui-même un entier. Mais comme le type de retour est précisé, le résultat du calcul est retourné après avoir été converti en float. Il faut également noter que si la fonction lambda n'a qu'un seul paramètre, il est quand même obligatoire d'utiliser les parenthèses.

Les attributs peuvent désormais être ajoutés aux fonctions lambda et à leurs paramètres. Comme précédemment, l'utilisation d'attributs imposera l'utilisation des parenthèses pour les paramètres. Sans cela, il y aurait une confusion pour savoir si l'attribut s'applique sur la fonction ou sur un paramètre. L'exemple suivant illustre la création d'une fonction lambda avec l'attribut *CustomDiagnostic* appliqué au niveau de la fonction, et l'attribut *CustomFormatter* appliqué au paramètre msg.

```
var lambda = [CustomDiagnostic] ([CustomFormatter]string msg) => Console.WriteLine(msg);
```

Quel avenir pour C# ?

Si la fonctionnalité de vérification de paramètres *null*, qui était l'une des plus attendues par la communauté, a pu être évoquée comme faisant partie de C# 10 par de nombreux articles, elle est pour le moment toujours en cours de développement. Elle est actuellement identifiée comme une fonctionnalité faisant partie de la version suivante (voir C# vNext) et non pas dans C# 10. La prise en charge par le compilateur de cette fonctionnalité étant presque terminée, l'usage ne devrait plus évoluer. C'est pourquoi nous nous permettons de vous la présenter dès maintenant :

```
// Vérification de la nullité du paramètre en C# 9
void Foo(string bar)
{
  if (bar == null) throw new ArgumentNullException(nameof(bar));
}

// Equivalent avec la fonctionnalité de vérification de la nullité
void Foo(string bar!!) { }
```

L'opérateur **!!** permettra de s'assurer que le paramètre de la fonction est non *null*. Attention toutefois à ne pas confondre cette fonctionnalité avec la notion de types références nullables introduite avec C# 8. Pour rester concis, les types références nullables donnent des avertissements à la compilation sur des déréférencements qui pourraient potentiellement provoquer des NullRerefenceException. Ici l'opérateur **!!** n'a aucune incidence à la compilation, mais uniquement à l'exécution. Il simplifie la vérification de la nullité des paramètres, ce qui allège le code et facilite le travail des adeptes de la programmation défensive.

La fonctionnalité de propriétés obligatoires – propriétés préfixées par le mot clé **required** – a également pu être annoncée comme faisant partie des nouveautés de C#. Or, il n'en est rien. Cette fonctionnalité est toujours envisagée, mais il n'y a pas encore de spécifications abouties sur les usages de cette fonctionnalité.

Une autre fonctionnalité évoquée, et finalement absente est le mot clé **field** pour accéder au champ privé d'une propriété sans avoir besoin de le déclarer :

```
public Datetime Birthday { get; init => field = value.Date(); }
```

C# 10 nous apporte un lot de nouveautés intéressantes. Il continue en effet d'évoluer en intégrant de nouvelles fonctionnalités utiles comme l'extension des types d'enregistrements aux structures tout en s'efforçant d'améliorer continuellement sa syntaxe grâce notamment à la déclaration d'espace de nom de portée de fichier. Cela s'intègre dans la continuité des mises à jour .NET récentes qui diminuent la quantité de code nécessaire et améliorent sa lisibilité.

D'autres nouveautés n'ont pas été présentées dans cet article - comme l'amélioration de l'analyse de nullabilité et du déterminisme des assignations, la directive **#line**, ou les améliorations de générations de code - car nous trouvions qu'elles apportent trop peu de changements ou couvrent des cas d'utilisations trop spécifiques.

Microsoft continue donc d'enrichir son langage d'année en année et nous sommes enthousiastes à l'idée de pouvoir l'utiliser dans nos développements futurs.

Blazor .NET 6 : un bon lot de nouveautés nous attend

La feuille de route de Microsoft prévoit une sortie de version majeure de .NET tous les ans. .NET 6 est une version LTS (Long Term Support). Cependant, un des paramètres essentiels à anticiper concerne la nécessité d'utiliser Visual Studio 2022 pour coder avec .NET 6 puisququ'il n'y aura pas de rétrocompatibilité pour VS 2019. Cette nouvelle version possède son lot de nouveautés, celles dédiées à Blazor nous intéressent plus particulièrement dans cet article qui ambitionne d'en faire un tour d'horizon.

Sébastien Bernard
Concepteur développeur SQLI

Florian Pouchelet
Concepteur développeur SQLI

SQLI DIGITAL EXPERIENCE

Hot reload

Pour utiliser le Hot Relaod avec Blazor de manière efficace et transparente, il va falloir lancer le projet sans passer par le debugger de Visual Studio. Nous allons à la place utiliser une invite de commande et lancer le projet à l'aide de **dotnet watch**. **Figure 1**
Cette commande va démarrer le navigateur et afficher le projet Blazor en cours. On peut suivre les étapes de démarrage dans l'invite de commande. **Figure 2**
À partir de là, on peut modifier le code dans la solution.
Dès lors que le fichier modifié est sauvegardé, le changement est détecté et automatiquement appliqué, ce qui permet de visualiser presque en temps réel ses modifications (en effet cela prend environ 2 à 3 secondes). **Figure 3**
Il devient possible d'ajouter, par exemple, un nouveau composant comme le counter sur la page d'accueil et de visualiser ce changement dans le navigateur. Dans ce cas, l'application va se reconstruire et afficher la nouvelle page.
L'utilisation de dotnet watch pour le hot reload permet de modifier à la fois les pages Razor comme le code C#. Nous pouvons donc changer une fonction et constater en temps réel le nouveau fonctionnement.

Les performances .NET 6 dans Blazor

Dans les applications communes en .NET, on considère très généralement un seul runtime, le coreclr (parfois même sans se poser la question). Or, en utilisant Blazor, il devient intéressant de se pencher sur la question ; en effet, Blazor WebAssembly (Blazor wasm) utilise un runtime différent, nommé mono. Il est pertinent d'indiquer qu'avec Blazor wasm, le runtime est compilé dans l'application. Il faut savoir que mono n'effectue pas que des opérations en Just In Time (JIT), mais qu'il interprète également l'Intermediate Language. Cela apporte une certaine sécurité supplémentaire puisqu'il n'interprète plus le code à la volée. Alors, pourquoi parlons-nous de cela ? Parce qu'il s'agit du premier sujet à propos des performances de Blazor avec .NET 6.
L'interpréteur de mono a bénéficié d'une refonte, améliorant son « *ability to inline* » et notamment pour les méthodes qui sont marquées par le tag [MethodImpl(MethodImplOptions.AggressiveInlining)] qui est très utilisé dans les bibliothèques de bas niveau du runtime (pour rappel, le inlining consiste à ce que le compilateur remplace l'appel à une fonction par le corps de la fonction).
Pour mémoire, la gestion d'abandon des threads a disparu avec .NET 5 ; Blazor en bénéficie avec l'arrivée de .NET 6 puisque cette gestion est supprimée dans le traitement des blocs finally dans mono.
Une grosse avancée a été faite en ce qui concerne les Hardware intrinsics qui ont été introduits par .NET Core 3.0. Mono supporte désormais LLVM pour la génération du code et le support de l'architecture ARM64 est implémenté grâce aux API ARM64 AdvSimd. Dans le même registre, les supports de SHA1, SHA256 et AES sont complétés.
On notera l'apparition des Vector64 et Vector 128 ce qui seront nécessairement utiles pour les futures applications Blazor wasm qui utiliseront AOT.
Oh ? AOT !? Eh bien oui, sachez que les applications Blazor wasm pourront être entièrement compilées en AOT (ce qui éliminera l'utilisation du JIT).

Webassembly AOT

Blazor WebAssembly supporte maintenant la compilation « ahead-of-time » (AOT). Cela signifie que vous pouvez compiler votre code .NET directement en WebAssembly ce qui améliore grandement les performances d'exécution.
Actuellement, une application Blazor WebAssembly s'exécute en utilisant un interpréteur .NET IL implémenté dans le

Figure 1

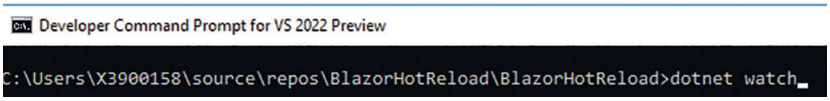


Figure 2

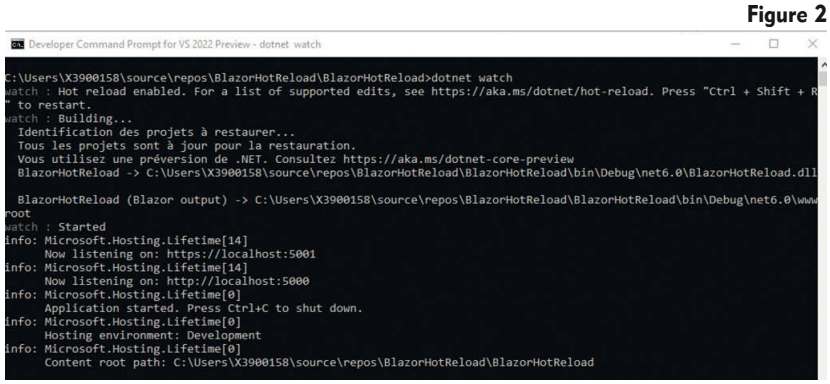
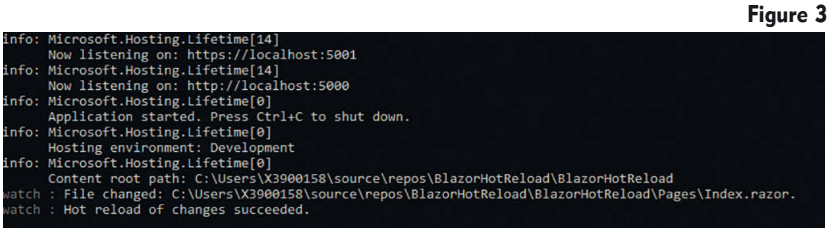


Figure 3



WebAssembly. Le code .NET étant interprété, cela signifie qu'il s'exécute plus lentement que lors d'une exécution .NET classique. La compilation .NET WebAssembly AOT corrige ces problèmes de performance en compilant directement le code .NET en WebAssembly.
Pour profiter de la compilation .NET WebAssembly AOT, il est nécessaire d'installer un outil de compilation additionnel disponible en option dans le SDK .NET. Pour l'installer, il suffit de taper dans une invite de commande :

```
dotnet workload install microsoft-net-sdk-blazorwebassembly-aot
```

Et dans le projet, il faut activer la compilation WebAssembly AOT en ajoutant la propriété :

```
<RunAOTCompilation>true</RunAOTCompilation>
```

La compilation WebAssembly AOT ne fonctionne que lorsque le projet est publié. Il faut donc publier le projet en mode release pour en profiter.

```
dotnet publish -c Release
```

Il faut noter malgré tout que l'utilisation de la compilation WebAssembly AOT a une conséquence sur le poids. L'application sera plus grosse et cela peut aller jusqu'à 2 fois le poids de l'application utilisant .NET IL.
Ainsi, l'utilisation de cette option dépendra aussi du type d'application déployée et sera surtout pertinente pour les applications faisant une utilisation intensive du CPU.

Error boundaries

Il s'agit d'un nouvel élément qui offre une façon pratique de gérer les exceptions au sein de la hiérarchie des composants. Pour l'utiliser, il suffit d'utiliser le nouveau composant ErrorBoundary pour envelopper le contenu pour lequel vous souhaitez avoir une personnalisation de l'exception.

```
<div class="content px-4">
    <ErrorBoundary>
      <MyComponent />
    </ErrorBoundary>
  </div>
```

Tant qu'aucune exception n'est levée, le composant affichera son contenu enfant. Mais dès qu'une exception arrive, il affiche une erreur spécifique. Par défaut, le composant affiche une div vide avec la classe CSS blazor-error-boundary.
Il est possible de personnaliser l'erreur en utilisant la propriété ErrorContent.

Library	Before	After	% I	.br
signalr.min.js	130 KB	39 KB	70%	10 KB
blazor.server.js	212 KB	116 KB	45%	28 KB

Figure 4

dotnet.wasm	Transfer size (kB)
.NET 5 default	884
.NET 6 default	780
.NET 6 relinked	756
.NET 6 invariant mode	393

Figure 5

```
<ErrorBoundary>
  <ChildContent>
    <MyComponent />
  </ChildContent>
  <ErrorContent>
    <p class="my-error">Voici une erreur personnalisée.</p>
  </ErrorContent>
</ErrorBoundary>
```

Il s'agit d'un moyen efficace pour gérer et personnaliser l'affichage des erreurs au sein de Blazor.

Persistance de l'état pendant le pré-rendu

Une application Blazor peut être pré-rendue à partir du serveur afin d'accélérer le temps de chargement perçu lors de la première utilisation. Le HTML est ainsi affiché directement pendant que le reste de la configuration se fait en arrière-plan. Malheureusement, l'état utilisé pendant ce pré-rendu est perdu et doit être recréé lorsque l'application est complètement chargée. Si un état est configuré de manière asynchrone, l'interface peut alors clignoter lorsque le pré-rendu est remplacé par l'affichage final.
Pour régler ce problème, une nouvelle balise <preserve-component-state /> a été ajoutée. Il faut ensuite ajouter le service ComponentApplicationState dans son composant pour préserver son état. L'évènement ComponentApplicationState.OnPersisting est déclenché quand un état doit être conservé dans la page pré-rendue.
Voici un exemple montrant comment le composant FetchData peut être persisté. Dans le fichier _Host.cshtml :

```
<body>
  <component type="typeof(App)" render-mode="ServerPrerendered" />
  ...
  @* Persist the component state after all component invocations *@
  <persist-component-state />
</body>
```

Et dans le fichier FetchData.razor :
Code complet sur programmez.com & github

Réduction de la taille de téléchargement

SignalR, MessagePack, et Blazor Server scripts sont significativement plus petits. **Figure 4**
Par ailleurs, la taille d'une application Blazor WebAssembly a été considérablement réduite. **Figure 5**

Paramètres requis

Il est possible de spécifier qu'un composant Blazor a des paramètres requis en utilisant l'attribut [EditorRequired].
Exemple :

```
[EditorRequired]
[Parameter]
public string Title { get; set; }
```

Si l'utilisateur ne précise pas le paramètre, il aura un avertissement et le composant sera souligné. Attention toutefois, il s'agit d'un attribut apportant une aide lors de la conception et il ne garantit pas à l'exécution que la valeur du paramètre sera non nulle.

Support du SVG

Il est maintenant possible d'utiliser la syntaxe Razor, dont les composants Blazor, dans un élément SVG foreignObject.

```
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
  <rect x="0" y="0" rx="10" ry="10" width="200" height="200" stroke="black" fill="none" />
  <foreignObject x="20" y="20" width="160" height="160">
    <p>@message</p>
  </foreignObject>
</svg>

@code {
  string message = "Wow, it's so nice that this text wraps like it's HTML...because that's what it is!";
}
```

Modifier le contenu <head> HTML

Blazor prend désormais en charge la modification de l'élément <head> depuis un composant. Il devient possible de modifier le titre et d'ajouter des éléments <meta>.
Pour modifier le titre d'une page, il faut utiliser le composant PageTitle. Le composant HeadContent permet quant à lui d'interagir avec les autres éléments de <head>.

```
<PageTitle>@title</PageTitle>

<HeadContent>
  <meta name="description" content="@description">
</HeadContent>

@code {
  private string description = "Description set by component";
  private string title = "Title set by component";
}
```

Pour activer cette fonctionnalité, il est nécessaire d'ajouter le composant HeadOutlet.

```
builder.RootComponents.Add<HeadOutlet>("head::after");
```

Diffusion de données de JavaScript vers .NET

Blazor peut maintenant diffuser des données directement depuis JavaScript vers .NET. Les flux sont appelés avec la nouvelle interface IJSStreamReference.
JavaScript :

```
function jsToDotNetStreamReturnValue() {
  return new Uint8Array(10000000);
}
```

C# :

```
var dataReference = await JSRuntime.InvokeAsync<IJSStreamReference>("jsToDotNetStreamReturnValue");
using var dataReferenceStream = await dataReference.OpenReadStreamAsync(maxAllowedSize: 10_000_000);

// Write JS Stream to disk
var outputPath = Path.Combine(Path.GetTempPath(), "file.txt");
using var outputFileStream = File.OpenWrite(outputPath);
await dataReferenceStream.CopyToAsync(outputFileStream);
```

Cela est utile pour notamment transférer des fichiers comme détaillé dans la section suivante.

Téléversement plus rapide de fichiers plus lourds

En utilisant la nouvelle façon de diffuser des données entre JavaScript et .NET, il est désormais possible de téléverser des fichiers d'une taille supérieure à 2GB avec le composant **InputFile**. Le transfert est également plus rapide grâce à l'utilisation d'un flux de byte[] directement sans utiliser un encodage Base64.

Arguments d'évènement personnalisés

En plus du support d'évènements personnalisés dans Blazor, il est aussi possible de passer des données au gestionnaire d'évènements pour les évènements personnalisés. Par exemple, il est possible de faire un évènement lorsque l'on colle le contenu du presse-papier avec le texte collé par l'utilisateur. Pour ce faire, il faut déclarer un évènement avec un nom personnalisé et une classe .NET qui contiendra les arguments pour cet évènement.

```
[EventHandler("oncustompaste", typeof(CustomPasteEventArgs), enableStopPropagation: true, enablePreventDefault: true)]
public static class EventHandlers
{
  // This static class doesn't need to contain any members. It's just a place where we can put
  // [EventHandler] attributes to configure event types on the Razor compiler. This affects the
  // compiler output as well as code completions in the editor.
}

public class CustomPasteEventArgs : EventArgs
{
  // Data for these properties will be supplied by custom JavaScript logic
  public DateTime EventTimestamp { get; set; }
  public string PastedData { get; set; }
}
```

Une fois cela fait, IntelliSense proposera un nouvel évènement appelé @oncustompaste.

```
@page "/"

<p>Try pasting into the following text box:</p>
<input @oncustompaste="HandleCustomPaste" />
<p>@message</p>

@code {
  string message;

  void HandleCustomPaste(CustomPasteEventArgs eventArgs)
  {
    message = $"At {eventArgs.EventTimestamp.ToShortTimeString()}, you pasted: {eventArgs.PastedData}";
  }
}
```

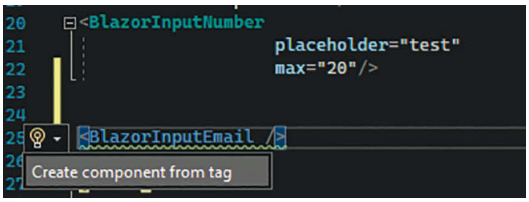
Cependant, cela ne suffit pas pour déclencher l'évènement, il faut également ajouter un peu de JavaScript dans le fichier index.html ou _Host.cshtml.

Code complet sur programmez.com & github

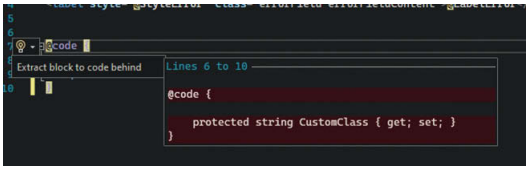
Cela dira au navigateur que chaque fois qu'un évènement "*coller*" a lieu, il doit aussi déclencher un évènement custom-paste et passer les arguments définis.

Génération automatique de composant Blazor

Il est possible de générer un composant Blazor en créant le tag dans une page puis faire un clic droit et générer le composant.



Il est également possible de créer automatiquement la classe partielle à partir de la balise code du fichier Razor.



Blazor peut désormais déduire des paramètres de type générique à partir de composants parents

Avec Blazor, il est possible d'utiliser des composants imbriqués. Nous avons alors un composant parent qui possède des composants enfants. Lors de l'utilisation d'un paramètre générique, il était jusqu'à présent obligatoire de préciser pour chaque composant ce paramètre générique. Par exemple lors de son utilisation dans une grid, il fallait spécifier Grid<TItem> puis spécifier à nouveau dans les colonnes Column<TItem>, ce qui donnait en termes de code :

Code complet sur programmez.com & github

Nous conviendrons que c'est un peu fastidieux de devoir spécifier pour chaque colonne les paramètres génériques de la grid. Il est maintenant possible de transmettre le paramètre générique aux composants enfants en le cascadant. Il faut que les paramètres parents et enfants portent le même nom.

```
@attribute [CascadingTypeParameter(nameof(TItem))]
```

Nous pouvons dès lors simplifier l'écriture de la grid en ne spécifiant plus le paramètre générique au niveau des enfants :

Code complet sur programmez.com & github

La version 5 de ma Datagrid pour Blazor (actuellement en bêta) utilise d'ailleurs ce nouveau mécanisme pour simplifier son écriture. (https://www.nuget.org/packages/BlazorDataGrid)

Contraintes de type générique

Dans une page Razor, il est possible de définir un paramètre générique avec la directive `@typeparam`. C'est très utile quand on doit recevoir un paramètre dans un composant Blazor dont on ne connaît pas le type. Par exemple, dans la datagrid, on reçoit en entrée une collection dont le contenue varie en fonction des usages ; Ça peut être une liste d'objets de météo, une liste de personnes, etc. Cependant, il peut arriver que l'on souhaite définir un type plus précis pour notre paramètre. Une contrainte sur un type d'objet permet à la classe générique d'utiliser des paramètres ou méthodes du type contraint.

Avec .NET 6, il est maintenant possible d'ajouter une contrainte en utilisant la syntaxe C# classique :

```
@typeparam TEntity where TEntity : IEntity
```

Rendu de composant dynamique

.NET6 permet de créer des composants Blazor dynamiques de la même manière que l'on utilise une variable de type dynamique en .NET.

```
<DynamicComponent Type="@someType" />
```

Les paramètres peuvent être transmis au composant en utilisant un tableau.

```
<DynamicComponent Type="@someType" Parameters="@myDictionaryOfParameters" />

@code {
  Type someType = ...
  IDictionary<string, object> myDictionaryOfParameters = ...
}
```

Récupération de paramètres depuis une query string

Depuis la preview 7 de .NET6, il est possible de spécifier que certains paramètres sont à récupérer dans la query string. Il faut ajouter l'attribut [SupplyParameterFromQuery] en plus de l'attribut classique [Parameter].

Exemple :

```
[Parameter]
[SupplyParameterFromQuery]
public int? Page { get; set; }
```

Avec l'url suivante https://localhost:5001/page=3, la valeur 3 sera alors affectée à Page.

Il est possible d'utiliser cet attribut avec les types String, bool, DateTime, decimal, double, float, Guid, int, long et leur variant nullable à l'exception de string. Les tableaux de ces différents types sont également autorisés.

Support de la sélection multiple dans l'élément select.

Il s'agit d'une fonctionnalité très appréciée et qui manquait lors des précédentes releases. Lors de l'utilisation d'un composant <select>, nous pouvons désormais spécifier l'attribut multiple. Cela permet de choisir plusieurs éléments dans la liste. Dans ces conditions, l'évènement onchange va alors fournir un tableau avec les éléments sélectionnés via ChangeEventArgs. En conséquence, il est possible d'utiliser un tableau comme valeur pour la liaison lorsque l'attribut multiple est spécifié.

Et lors de l'utilisation de <InputSelect>, l'attribut multiple est automatiquement déduit lorsque la valeur liée est un tableau.

MAUI & Blazor

Pour commencer, qu'est-ce qu'une hybrid app ? Il s'agit de faire une utilisation du code Blazor pour des applications natives :

- Réutilisation du développement web (code et compétences)
- Accès à toutes les fonctionnalités natives de l'appareil
- Mélange de d'interface native et de web
- Réduit le temps de développement des applications

Le code .Net est exécuté directement dans l'application native et exécute des composants Blazor localement. Le rendu du DOM est envoyé dans un contrôleur de vue web. Tous les évènements qui se déclenchent dans cette vue sont envoyés dans le code .NET puis les modifications sont retournées dans le contrôleur qui affiche la vue. Tout s'exécute dans l'application.

.NET Multi-platform App UI (MAUI) est une évolution de Xamarin Form, mais étendue à plus de plateformes (comme le desktop par exemple).

Ces principales caractéristiques sont :

- Cross platform
- Utilisation des interfaces natives de chaque plateforme
- Un seul projet système, une seule base de code
- Déploiement sur plusieurs appareils, mobiles et desktop
- Disponibilité avec .NET 6

C'est le principe des Blazor hybrid apps qui sont utilisées au sein de .NET MAUI.

Le contrôleur BlazorWebView est intégré dans MAUI. On retrouve donc :

- Réutilisation des composants UI entre native et web
- Mix & match web and native UI
- Accès direct aux fonctionnalités natives des appareils
- Applications cross-platform mobile & desktop (à la sortie de .NET 6, le focus sera mis sur le desktop)

On peut donc utiliser des composants Blazor tels quels au sein d'une application .NET MAUI. Tous les composants déjà développés dans le cadre d'autres projets peuvent donc facilement être réutilisés.

.NET MAUI possède le composant natif BlazorWebView qui permet de faire le rendu des composants Blazor. Cela représente un vrai gain de productivité.

BlazorWebView est également disponible pour Winforms & WPF, ce qui permettra d'utiliser aussi les composants Blazor avec ces technologies.

Mobile Blazor Bindings

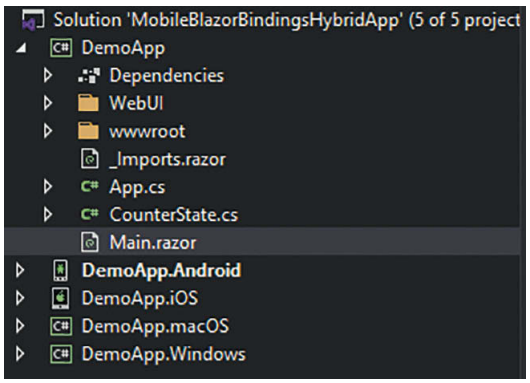
Vous connaissez certainement Blazor pour la création d'applications web, avec toute la puissance qui lui incombe. Mais Blazor ne s'arrête pas là ; en effet, il existe un projet expérimental dans le dépôt dotnet, nommé MobileBlazorBindings. Mais alors, késako ?

Il s'agit tout simplement de donner la possibilité aux développeurs de créer des applications lourdes ou mobiles en utilisant C# et .NET. Ça doit vous parler ça, C# et .NET pour des applications lourdes ou mobiles ? C'est normal puisque c'est ce que fait Xamarin, et c'est là toute la subtilité : Mobile Blazor Bindings se présente comme une abstraction de haut niveau de Xamarin. Il utilise la syntaxe des pages Razor pour définir les composants, autant sur le front, que sur le back-end.

On y retrouve ainsi tous les composants de Xamarin.Forms, que l'on peut utiliser à sa guise, le rendu étant lui toujours laissé à la charge de Xamarin.

En tant que projet expérimental, il n'est pas encore disponible en standalone de manière native et nécessite Edge Canary et le SDK .NET Core 3.1. De même, il faut installer le template grâce à une commande dotnet.

La solution se présente sous cette forme, on y retrouve bien les différents projets selon la plateforme visée.



Cependant, on constate la présence d'un projet Blazor. Voyons ce que contient la page principale : **Figure 6**

C'est bien un fichier Razor classique qui contient un counter !

Connaissant l'engouement et la montée en puissance de Blazor, ce projet reste donc à surveiller de près !

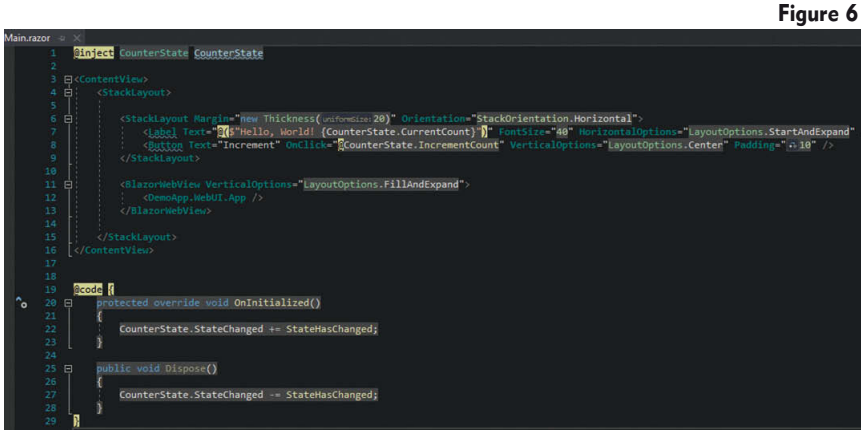
Micro FrontEnd avec Blazor

Tout d'abord, définissons ensemble ce que nous appellerons le **micro-frontend**. Le micro-frontend est une architecture dans laquelle plusieurs composants coopèrent tout en étant hautement indépendants. Il est facile de faire le parallèle avec l'architecture micro-services. Chacun des composants a sa propre logique, ils sont tous isolés les uns des autres et peuvent être développés par des équipes différentes.

Pour que cette architecture fonctionne, il faut envisager une application principale, nommée la "App Shell" qui appellera tous les composants indépendants. Ainsi, pour récupérer les composants faisant partie de l'application, il faut utiliser ce que l'on nomme un compositeur. En .NET, il est par exemple possible d'utiliser la bibliothèque Piral.

Dans notre exemple, nous utiliserons des composants Razor sous forme de Razor Class Library (RCL) qui représenteront les composantes du micro-frontend, l'app shell sera un projet Blazor wasm, et le compositeur sera fait grâce aux dépendances internes de C#. Puis nous mettrons en place le lazy loading, afin de rendre l'expérience plus réaliste.

Figure 6

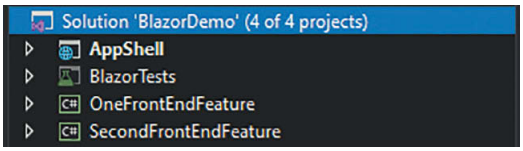


```razor
@using System.Reflection
@inject LazyAssemblyLoader AssemblyLoader

<Router AppAssembly="@typeof(Program).Assembly"
        AdditionalAssemblies="@LoadedAssemblies"
        OnNavigateAsync="@OnNavigateAsync"
        PreferExactMatches="@true">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>

@code
{
    private List<Assembly> LoadedAssemblies = new List<Assembly>()
    {
        typeof(FirstComponent).Assembly
    };

    private async Task OnNavigateAsync(NavigationContext context)
    {
        // Paths of the page that needs the lazy loaded assembly
        if (context.Path == "SecondComponent" || context.Path == "ThirdComponent")
        {
            var assembliesNeeded = await AssemblyLoader.LoadAssembliesAsync(new List<string> { "SecondFrontEndFeature.dll" });
            LoadedAssemblies.AddRange(assembliesNeeded);
        }
    }
}
```

Figure 7

Il est important de préciser que l'implémentation de la démo qui vous est faite servira d'introduction et ne représente pas une solution efficace en production.

Voici donc l'architecture de notre solution :



Chacune des features possède un ou plusieurs composants, qui seront utilisés dans l'app shell. Rien d'extravagant, seulement les composants basiques qui sont créés avec Blazor.

Intéressons-nous donc à l'import de ces composants dans l'app shell. Comme nous l'avons précisé, nous utilisons les références internes de la solution comme compositeur pour cet exemple. Vous devez donc ajouter les références de vos RCL dans l'app shell : ajoutez-les à la main dans le csproj, ou bien via la GUI. Une fois que vos références sont ajoutées, vous pourrez bénéficier des composants en les important à votre guise :

AppShell / _Imports.razor

```razor
@using OneFrontEndFeature
```

Mais souvenez-vous, nous allons utiliser le lazy loading pour cela, il faut commencer par indiquer quelle ressource sera à exécuter en lazy loading, pour cela, ajoutez la référence dans votre csproj.

```xml
<ItemGroup>
  <BlazorWebAssemblyLazyLoad Include="SecondFrontEndFeature.dll" />
</ItemGroup>
```

Importez ensuite le package de gestion des assemblies, et ajoutez donc "@using Microsoft.AspNetCore.Components.WebAssembly.Services" dans votre _Imports.razor.

```razor
@using Microsoft.AspNetCore.Components.WebAssembly.Services
```

Le Lazy Loading servira également de compositeur, en vous donnant la possibilité de ne charger vos composants qu'au moment où ils sont nécessaires, mais surtout il vous abroge de l'import de vos composants dans votre _Imports.razor !

Illustrons la solution dans l'App.razor : **Figure 7**

On ajoute deux nouvelles lignes à notre router :

- AdditionalAssemblies qui contiendra les assemblies des composants que l'on chargera à la volée.
- OnNavigateAsync qui exécutera une fonction lorsque l'on souhaite naviguer dans l'application.

Dans notre cas, on cherchera à vérifier si l'on veut naviguer vers nos composants et si tel est le cas, on ajoutera l'assembly correspondante.

Et voilà ! Vous avez une application qui respecte les principes du micro-frontend, et tout ça avec Blazor ! On constate bien que chacune des Features est strictement isolée, et qu'elle peut être modifiée indépendamment de l'application complète et sans impacter le reste.

Il existe bien entendu des manières bien plus efficaces et pragmatiques d'implémenter cette architecture, et vous en trouverez pléthore sur internet. Sachez toutefois que le micro-frontend ne se contente pas d'utiliser une seule technologie, vous pouvez tout à fait mixer les frameworks et utiliser chaque brique dans votre app shell, cependant cela requiert notamment un CI plus complexe.

Tester Blazor avec bUnit

Contrairement à ce que le célèbre adage nous dit, non, tester, ce n'est pas douter. Et plus encore, la certitude n'évite pas le danger. Alors, dans Blazor, comment fait-on ?

Comme Microsoft ne propose pas de framework de test officiel pour Blazor, il nous faut nous tourner vers des projets open source. Dans le cas de Blazor, on trouve assez vite bUnit qui est placée sous licence MIT.

BUnit fonctionne avec les frameworks de tests classiques, tels que nUnit ou bien xUnit (ne les citons pas tous, nous n'en finirons pas) et c'est pour cette raison qu'il faut envisager bUnit comme un fournisseur de contexte aux tests.

Allons un peu plus loin dans notre analyse : en utilisant bUnit pour le rendu des composants, vous pourrez aisément passer des paramètres, injecter un service et avoir accès simple au DOM du composant. Le rendu d'un composant se fait à partir de la classe TestContext de bUnit, il en résulte un objet IRenderedComponent qu'il sera facile de manipuler.

Pour écrire vos tests, vous aurez le choix entre le faire dans un fichier classique ".cs" ou bien dans un fichier Razor ".razor" ce qui facilite l'écriture du code HTML à tester ; cependant l'éditeur Razor de Visual Studio 2019 n'intègre pas encore toutes les fonctionnalités disponibles dans un fichier C# classique, et quelques bugs de formatage sont à prévoir.

Tout d'abord, il vous faut :

- Créer un projet de tests, avec le framework qui vous conviendra le mieux (pour ma part j'utilise xUnit)
- Installer le package NuGet bUnit. Attention, il faut penser à changer le SDK dans son nouveau projet, pointer vers Microsoft.NET.Sdk.Razor.
- Penser à ajouter la dépendance vers le RCL que vous souhaitez tester.

Vous devez obtenir un fichier csproj proche de cela : **Figure 8**

À noter qu'il est possible d'installer le projet de test via le template bUnit, mais cela fonctionne à l'heure actuelle exclusivement avec xUnit.

Prenons l'approche d'un test écrit dans un fichier C#, nous testons un composant qui n'intègre pas de logique : **Figure 9**

Le fait d'étendre la classe TestContext de bUnit nous donne accès au contexte de test de bUnit.

Maintenant, voyons si l'on ajoute un peu de logique à un composant. Pour cela nous utiliserons le composant Counter qui est automatiquement généré lors de la création d'un projet Blazor.

On constate que l'on a cliqué une fois sur le bouton et qu'en effet le composant affiche bien la bonne valeur.

Bien entendu, vous pourrez aller bien plus loin dans vos tests, comme précisé plus haut dans l'article grâce à notre introduction sur le sujet.

Vous pourrez par exemple donner une valeur précise à un paramètre, injecter des services, mocker vos données ou encore gérer les retours asynchrones.

En bref, cette bibliothèque est très complète, et son utilisation est sans appel bénéfique à nos développements !

JSInterop

Une fonctionnalité de .NET (arrivée avec .NET Core 3.1) très intéressante dans l'utilisation de Blazor est le JSInterop. JSinterop est le mécanisme permettant d'interagir entre C# et JavaScript. En effet, il offre la possibilité d'appeler des fonctions C# dans le JavaScript et vice-versa.

Attention toutefois à ne pas commettre deux erreurs importantes :

- Utiliser la balise <script> dans un composant Razor (.razor), car cette balise ne peut pas être mise à jour dynamiquement par Blazor.
- Modifier un composant dont le rendu a été fait par Blazor avec du JavaScript. Blazor garde en mémoire un arbre du DOM de ce qu'il a rendu, et si ce dernier est modifié sans passer par Blazor, cela peut poser des problèmes à l'usage.

En ce qui concerne l'isolation JavaScript, Blazor intègre le standard des modules JavaScript. On retrouve donc nos fichiers JavaScript dans la partie wwwroot du projet Blazor.

L'intérêt de cette fonctionnalité au sein de Blazor se retrouve essentiellement dans l'utilisation de bibliothèques JavaScript, ou bien de frameworks CSS tels que jQuery, Bootstrap ou encore Materialize. Mais on peut en retrouver une certaine utilité pour l'utilisation plus poussée de framework JavaScript, comme React.

Conclusion

.NET 6 apporte son lot de nouveautés en améliorant et simplifiant l'utilisation de Blazor. L'optimisation est toujours un fer de lance des nouvelles versions de .NET et nous constatons une nouvelle fois son efficacité tant par la réduction des données à télécharger que par l'amélioration du rendu et des vitesses de traitements. Ces nouveaux ajouts sont les bienvenus et raviront à la fois des utilisateurs finaux comme les développeurs. Si Blazor continue sur cette même lancée, il est à prévoir qu'il s'impose relativement vite au sein des technologies web, et en premier lieu chez les développeurs .NET. Si demain vous voyez Blazor truster le haut du classement des technos web, vous ne pourrez pas dire qu'on ne vous aura pas prévenus !

Figure 8

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>

    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="bunit" Version="1.2.49" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.9.4" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.3">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="coverlet.collector" Version="3.0.2">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\SecondFrontEndFeature\SecondFrontEndFeature.csproj" />
  </ItemGroup>

</Project>
```

FirstComponentTest.cs

BlazorTests | BlazorTests.FirstComponentTest | FirstComponentShouldRendersCorrectly()

```
using Bunit;
using OneFrontEndFeature;
using Xunit;

namespace BlazorTests
{
    0 references
    public class FirstComponentTest : TestContext
    {
        [Fact]
        0 references
        public void FirstComponentShouldRendersCorrectly()
        {
            // Act
            var theComponent = RenderComponent<FirstComponent>();

            // Assert
            theComponent.MarkupMatches("<div class=\"my-component\">This Blazor component is defined in the <strong>OneFrontEndFeature</strong> package.</div>");
        }
    }
}
```

Figure 9

Performance .Net 6 & EfCore 6, qu'en est-il ?

Clément Sannier
Leader Technique MS SQLI

SQLI DIGITAL EXPERIENCE

La sortie de .Net 5 en 2020 a mis en avant un certain nombre de gains de performance par rapport à .Net Core 3.1. La nouvelle version .Net 6 continue sur cette lancée, avec pas moins de 500 pull requests pouvant être associés à un gain de performance sur les 6500 de la release. Les ambitions sont également grandes pour EF Core 6.0 : rattraper les performances du Micro ORM qu'est Dapper. Je vous propose donc de décortiquer quelques fonctionnalités et d'essayer de percevoir comment les changements de certaines parties du code peuvent avoir un impact fort sur les performances du runtime.

La recherche de performance peut s'acquérir de deux manières. D'abord, il y a la recherche de la performance pure du code. On essaye de trouver des raisonnements et une logique différente permettant d'augmenter significativement les traitements. Puis, il y a les avancées du langage et les performances qu'elles peuvent apporter. Ce que j'apprécie avec cette version de .Net 6, c'est que les développeurs ont utilisé tantôt l'une et tantôt l'autre, et parfois même les deux.

Les tests de performance avec Benchmark .Net

Avant toute chose, l'utilisation d'une librairie de benchmark est nécessaire. Benchmark .Net est une librairie destinée aux benchmarks et à la mesure de performance. Sa modularité lui octroie la possibilité de mesurer différents frameworks à la fois du .Net Core ou du Framework. Ainsi c'est le partenaire idéal pour tester les gains de performance entre .Net 5 et .Net 6, et c'est ce que nous utiliserons tout au long de l'article.

JIT

Le code source écrit en C# est compilé dans un langage intermédiaire (il) conforme à la spécification CLI. Le code de langage intermédiaire et les ressources, telles que les bitmaps et les chaînes, sont stockés dans une assembly, en général, avec une extension .dll.

Lorsque le programme C# est exécuté, l'assembly est chargé dans le CLR. Le CLR effectue une compilation just-in-time (JIT) pour convertir le code de langage intermédiaire en instructions machine natives. C'est donc naturellement que si l'on veut augmenter les performances du runtime .Net, la gestion de la partie Jit est un bon candidat.

Figure 1

```
//
// Common worker for all signed integer TryFormat overloads
//
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static bool TryFormatInt64(long value, ulong mask, Span<byte> destination, out int bytesWritten, StandardFormat format)
```

Figure 2

Runtime	Mean	Ratio
.NET Core 3.1	21.881 ns	1.00
.NET 5.0	15.425 ns	0.70
.NET 6.0	9.003 ns	0.41

En .Net 6, de nombreuses améliorations ont été ajoutées par la communauté. L'une d'elles porte sur la gestion des méthodes dites inline. L'Inlinning est le processus d'optimisation d'un compilateur permettant de remplacer l'appel d'une fonction par son code. Bien que par définition, une méthode inline augmente la taille du programme, son principal avantage est d'offrir la possibilité d'améliorations qui ne peuvent être accessibles par l'appel de la fonction (code mort, optimisation d'invariant, élimination de variables d'induction, etc.). Pour certaines raisons, l'inlinning est à double tranchant côté performance : utilisé à mauvais escient, il peut réduire drastiquement les performances. Toutefois, utilisé correctement, il est peut-être extrêmement puissant.

Finalement, c'est un ensemble d'améliorations qui ont été effectuées sur la partie JIT afin de mieux comprendre le code faisant appel à la fonction inline. Pour tester les gains de performance, prenons l'exemple de la classe Utf8Formatter. Si nous regardons plus en détail le code, voici la signature de la méthode TryFormatInt64 : **Figure 1**

Nous pouvons observer que la méthode est taguée pour être en mode inline. En effectuant un test avec Benchmark .NET, nous remarquons d'après le ratio que les performances ont plus que doublé par rapport à .Net Core 3 .1, et que la différence est aussi significative avec .NET 5.0. **Figure 2**

La communauté a travaillé sur des améliorations concernant la dévirtualisation, le PGO Dynamique ou encore la vérification des limites.

Crossgen et AOT

Crossgen est arrivé très tôt dans le runtime de .Net Framework. Au fur et à mesure de l'avancée du framework .Net Core, le besoin s'est fait sentir de réécrire l'outil. Crossgen est un outil permettant la compilation AOT (ahead-of-time). Son but est de réduire les besoins de la compilation JIT au moment de l'exécution. La compilation AOT s'applique au moment de publication de l'application, Crossgen applique alors une pré-compilation JIT sur l'ensemble des dll et stocke le code ainsi créé dans une nouvelle section pouvant être récupérée rapidement par le runtime.

Crossgen 2 a été complètement réécrit avec une nouvelle architecture pour coller aux usages de .Net 6. En effet, ceux-ci diffèrent totalement suivant les besoins. Il peut être utilisé sur

du Windows ou du Linux, dans des web app Azure ou dans des « containers », pour du web ou du desktop. La nouvelle version de Crossgen prend en compte toutes les spécifiés de projet analyser et optimiser le code in fine.

System Types

System Types étant utilisé à tout moment et dans chaque application .NET, on ne peut augmenter les performances de .Net sans passer par l'optimisation des types de la librairie « System ». Des modifications ont été apportées à certains types moins habituels comme le type version, et d'autres retouches concernent des types plus « couramment utilisés », comme Random qui a été totalement refondu. L'histoire est simple, l'algorithme utilisé jusqu'à .Net 6 était le même depuis 20 ans. Il a donc été réécrit pour améliorer les performances de la génération d'une chaîne pseudo-aléatoire sans empiéter sur la qualité dont nous avons besoin en tant que développeur. Je pense que cette refonte de l'algorithme devrait sûrement avoir un article à lui tout seul ; c'est pour cela que j'ai préféré me pencher sur un autre type que l'on utilise tous les jours et qui au cœur des applications .NET : le Guid. Ce type est présent pour fournir un identifiant unique et universel dans une application.

Étant donné son caractère unique et universel, l'utilisation est répandue dans les applications et l'une de ses méthodes les plus utilisées reste le « parsing » (utilisé dans la désérialisation de Json par exemple). En .Net il existe plusieurs formats de Guid, avec ou sans parenthèse, accolade, séparée par un hypen. L'algorithme a été simplifié et l'utilisation de méthode inline permet de mieux gérer les cas d'extraction d'un Guid d'une chaîne de caractères.

Si on lance un test en comparant à partir de la version .Net Framework 4.8, voici ce que nous obtenons : **Figure 3**

Un bond en avant a été fait entre .Net Framework 4.8 et .Net Core 3.1, puis de petites avancées jusqu'à .Net 6. Toutefois, quand on pense que le parsing de string en Guid est inévitable dans la réception de Json par une API, nous nous rendons compte qu'après 1 million de parsing la maigre différence entre .Net 5 et 6 devient bien plus importante : **Figure 4**

String, Collections et Linq

Les tableaux, les collections et Linq ont une place importante au cœur de .Net. La communauté a donc travaillé à réduire l'impact des traitements sur les performances du code.

Prenons le cas d'une copie d'un dictionnaire. Étant utilisé à longueur de temps, rien de plus banal que de cloner un dictionnaire pour effectuer certains traitements. Si le dictionnaire source et le dictionnaire de destination partagent le même comparateur de clé, l'astuce a consisté à copier les objets sans les hacher par la suite :

Code complet sur programmez.com & github

En vérifiant par un test, nous remarquons un gain de performance depuis .Net Framework 4.8 : **Figure 5**

Sur les collections, d'autres changements ont été effectués pour améliorer un peu plus les performances.

Côté Linq, une amélioration m'a interpellé concernant les tests d'équivalence entre deux énumérables avec

Runtime	Mean	Ratio
.NET 6.0	82.23 ns	0.23
.NET 5.0	88.59 ns	0.25
.NET Core 3.1	109.17 ns	0.31
.NET Framework 4.8	352.44 ns	1.00

Figure 3

Runtime	Mean	Ratio
.NET Framework 4.8	366.69 ms	1.00
.NET Core 3.1	111.97 ms	0.31
.NET 5.0	103.45 ms	0.28
.NET 6.0	64.90 ms	0.18

Figure 4

Runtime	Toolchain	Mean	Error	StdDev	Ratio
.NET 5.0	net5.0	2.613 us	0.0376 us	0.0352 us	0.64
.NET 6.0	net6.0	2.488 us	0.0410 us	0.0383 us	0.61
.NET Core 3.1	netcoreapp3.1	2.940 us	0.0521 us	0.0487 us	0.72
.NET Framework 4.8	net48	4.074 us	0.0379 us	0.0354 us	1.00

Figure 5

Runtime	Mean	Ratio
.NET Framework 4.8	10,576.9 us	1.00
.NET Core 3.1	6,534.3 us	0.62
.NET 5.0	6,455.5 us	0.61
.NET 6.0	475.4 us	0.04

Figure 6

Enumerable.SequenceEqual. D'abord optimisé pour les byte[], une deuxième pull request est venue compléter le code pour n'importe quel type. Au lieu de comparer les séquences directement, celle-ci se base sur la « value type » System.Span<T>. Ainsi le traitement est délégué à la méthode span ajoutant une vectorisation de la comparaison sans utiliser plus de ressource. Le seul inconvénient est que le type T doit s'y prêter correctement :

Code complet sur programmez.com & github

Le résultat est bluffant et nous rappelle le bienfait de la librairie System.Span sur les performances lors des traitements : **Figure 6**

Les autres améliorations concernent la partie Distinct, Min, Max ainsi que l'ajout de nouvelles API telles que Enumerable.Zip (acceptation de 3 sources au lieu de 2).

Enfin, intéressons-nous au String. Une optimisation intéressante a été réalisée au niveau de la méthode « String.Replace(String, String) ». Certes, ce n'est pas la méthode en elle-même qui a été optimisée, mais plutôt le traitement des différents cas de Distinct. En effet, trois cas ont tiré leur épingle du jeu afin d'être optimisés dans cette version de .Net 6 :

Le cas le plus évident est lorsque nous souhaitons remplacer un caractère par un autre, souvent des caractères spéciaux comme "\n". Ainsi la rapidité d'un str.Replace("\n", " ") est accrue pour la simple et bonne raison qu'elle fait directement appel à la méthode String.Replace(char, char) :

Code complet sur programmez.com & github

Le deuxième cas réside lors du remplacement d'un caractère

unique par une valeur (unique ou non). Dans ce cas, c'est indexOf(char) qui est utilisée :

Code complet sur programmez.com & github

Le dernier cas survient lors du remplacement de plusieurs caractères avec l'utilisation d'un équivalent de IndexOf(string, StringComparison.Ordinal).

Si nous testons les performances de chaque cas, nous retrouvons une augmentation de performance significative dans le cas 1. L'augmentation reste tout de même honorable dans le cas 2 et 3 : **Figure 7**

Les autres modifications de la librairie String ont porté sur l'API String.join avec là aussi l'introduction de la fonctionnalité « ReadOnlySpan<string ?> », mais également string.format avec des changements surtout issues de C#10 et la string interpolation.

IO

La librairie FileStream est l'une des plus vieilles librairies de .Net. Elle a donc connu de nombreuses modifications année après année et était donc l'une des candidates les plus importantes à une mise à jour, sachant que l'accès au fichier est utilisé dans d'innombrables scénarios. Pour cette version .NET 6 la librairie FileStream a été complètement réécrite pour d'une part séparer les fonctionnalités en donnant plus de visibilité au code, et d'autre part faciliter les changements affectant la performance.

Deux méthodes ont été principalement revues pour offrir des gains de performance : FileStream.Seek et FileStream.Position. Le constat de départ, fait par la communauté, porte sur l'observation que les méthodes FileStream.ReadAsync et FileString.WriteAsync synchronisent l'offset du ficher de manière récurrente à chaque opération asynchrone. C'est d'ailleurs une problématique connue depuis de nombreuses années.

Les modifications apportées dans .Net 6 contourne ce problème en suivant l'offset plutôt en mémoire quand cela est possible ce qui accélère le traitement. Par la même occasion, les appels système ne sont effectués que lorsqu'ils sont explicitement requis.

Voici les résultats de ce long travail : **Figure 8**

D'autres améliorations de performance ont été introduites dans cette réécriture, nous avons par exemple un travail sur la fonction de Lenght, avec pour elle aussi un accès en mémoire tant qu'il n'y a pas d'accès Write sur le fichier. Les parties Aync file IO, comme nous l'avons vu, ont bénéficié des avancées de FileStream, mais également d'autres améliorations comme une meilleure gestion des allocations mémoire et de la libération.

Method	Runtime	Mean	Ratio
ReplaceCas1	.NET 6.0	15.47 ms	0.49
ReplaceCas1	.NET 5.0	31.43 ms	1.00
ReplaceCas2	.NET 6.0	22.77 ms	0.68
ReplaceCas2	.NET 5.0	33.73 ms	1.00
ReplaceCas3	.NET 6.0	25.08 ms	0.75
ReplaceCas3	.NET 5.0	33.57 ms	1.00

Figure 7

Figure 8

Method	Runtime	fileSize	options	Mean	Ratio
SeekForward	.NET 5.0	1024	None	426.0 us	1.00
SeekForward	.NET 6.0	1024	None	136.2 us	0.32
SeekBackward	.NET 5.0	1024	None	2,424.2 us	1.00
SeekBackward	.NET 6.0	1024	None	142.2 us	0.06
SeekForward	.NET 5.0	1024	Asynchronous	3,429.8 us	1.00
SeekForward	.NET 6.0	1024	Asynchronous	145.9 us	0.04
SeekBackward	.NET 5.0	1024	Asynchronous	8,324.4 us	1.00
SeekBackward	.NET 6.0	1024	Asynchronous	152.6 us	0.02

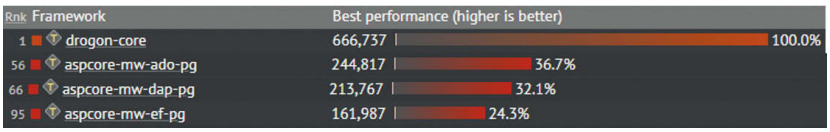


Figure 9

EF core 6.0 sur le banc des gains de performance

C'est au tour de EF Core 6.0 de passer sur le banc des gains de performance. Comme évoqué, l'ambition était grande pour cette nouvelle version, à savoir rattraper le « micro ORM » Dapper dans les scores de performance Tech Empower Fortunes. Les équipes d'EF Core ont annoncé être passées de 55% à environ 5% d'écart de performance. Si on considère les différences de feature entre Dapper et EF Core, c'est une avancée remarquable. Toutefois, il faut bien garder à l'esprit que les tests de performance concernent un scénario particulier utilisant du « no-tracking » et sans update. Il est donc possible que dans nos applications de production les performances soient différentes. Regardons, néanmoins ensemble, les résultats et quelques modifications qui ont été apportées à EF Core.

Tech Empower Benchmark

Les tests Tech Empower (https://www.techempower.com/benchmarks) comparent des performances de nombreux frameworks d'applications web exécutant des tâches fondamentales telles que la sérialisation JSON, l'accès aux bases de données et la composition de modèles côté serveur. Chaque framework fonctionne dans une configuration de production réaliste. Les résultats sont capturés sur des instances dans le cloud et sur du matériel physique. Les implémentations des tests sont réalisées par la communauté et toutes les sources sont disponibles sur le dépôt GitHub : https://github.com/TechEmpower/FrameworkBenchmarks.

Nous avons donc accès au code produit par chaque communauté sur les différents Frameworks et le C# (CSharp sur le repository) n'échappe pas à la règle.

Trois scénarios ont été retenus par l'équipe d'EF Core :

- Une implémentation qui utilise directement ADO.NET. Il s'agit de l'implémentation la plus rapide parmi les trois énumérées ici et se classe 12ième :
- Une implémentation qui utilise Dapper. Elle est plus lente que l'utilisation directe d'ADO.NET, mais reste rapide.
- Une implémentation qui utilise EF Core. C'est actuellement l'implémentation la plus lente des trois.

Pour l'heure voici les derniers résultats, malheureusement, il faudra attendre un nouveau round pour que les bénéfices

des optimisations soient pris en compte. **Figure 9**
Je vous invite donc à guetter les annonces du prochain round pour vérifier les gains.

Regroupement et recyclage des DbContext

Arrivé avec EF Core 2.0, le regroupement de contextes offre au développeur la possibilité de réutiliser un DbContext en le réinitialisant plutôt que de le supprimer purement et simplement. Dans cette gestion, un pool de contexte trop grand ou illimité aurait tendance à créer des objets DbContext au fur et à mesure des besoins, sans jamais les supprimer. La conséquence serait une gestion des ressources catastrophique. Ainsi, jusqu'à la version EF Core 6.0, le pool par défaut était de 128, ce qui est déjà un nombre important. Pour les besoins du benchmark Tech Empower, la valeur a été mise par défaut à 1024. D'après les tests réalisés par l'équipe EF Core, les performances seraient augmentées de 23%.
Il faut toutefois relativiser cette amélioration, d'une part car il était déjà possible de spécifier une valeur pour la taille du pool, et d'autre part, car les scénarios qui auraient besoin d'autant de Dbcontext sont limités.
Une autre amélioration touche quant à elle la manière dont EF Core interagit avec les objets ADO.NET (par exemple DbConnection, DBCommand, DBDatareader, etc.). De base, le profilage de la mémoire a révélé un nombre élevé d'instances de ces objets. L'une des améliorations a donc consisté à réécrire les interactions pour que chaque DbContext dispose de ses propres instances dédiées qu'il réutilisera à chaque fois.

Suppression de la journalisation

Les logs peuvent être importants pour comprendre d'éventuelles problématiques sur nos applications. Dans EF Core, il est possible de voir les instructions SQL avant leur exécution ainsi que leur temps d'exécution. L'utilisateur a la possibilité de s'appuyer sur les événements grâce à classe DiagnosticSource et la gestion des instructions avec un intercepteur. Bien que cela puisse être puissant, des pertes de performance peuvent apparaître, car une fois le « diagnostic listner » ou la journalisation activé(e), le code vérifie toujours à chaque instant l'activation ou non de ces derniers.
Afin d'offrir de meilleure performance et garder toujours cette grande flexibilité, l'astuce a été de vérifier si la journalisation ou l'interception n'était pas activé et dans ce cas supprimer la journalisation pendant 1 seconde. Par conséquent, l'activation de la journalisation peut prendre jusqu'à 1 seconde, là où elle était instantanée. L'équipe a calculé que ce processus permettait d'améliorer le débit de référence de 7%.

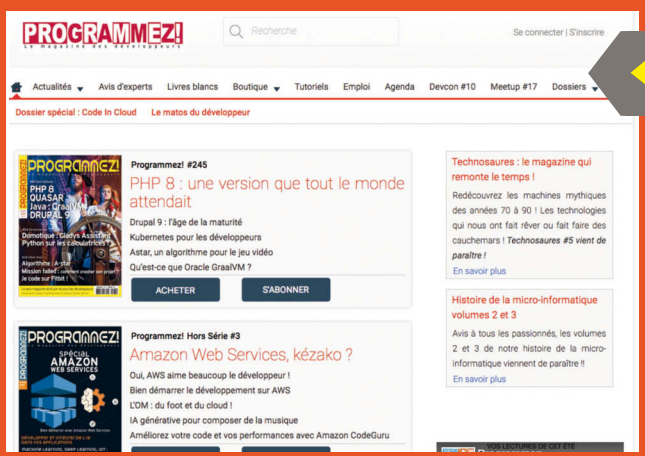
Désactivation des contrôles de sécurité des threads

En espérant ne rien vous apprendre, EF Core n'est pas « thread safe ». Une des raisons est qu'il encapsule une connexion à une base de données qui n'autorise presque jamais l'utilisation simultanée. Étant donné que les accès concurrents sont surtout dus à un problème de développement, EF Core inclut un mécanisme de sécurité interne essayant de détecter tant bien que mal (en mode best effort) les accès concurrents et lève une exception informative.
Il s'est avéré que ce mécanisme n'est pas aussi performant que l'équipe le souhaite notamment lors des requêtes asynchrones. La problématique réside dans le fait qu'il n'y a pas réellement d'amélioration à apporter à ce mécanisme à part le désactiver totalement. Étant donné les erreurs que cela provoquera, l'équipe a donc opté pour un indicateur de désactivation offrant à ceux qui le souhaitent la possibilité d'augmenter les performances (d'environ 7% d'après les tests) s'ils sont convaincus qu'aucun bug de concurrence n'existe.

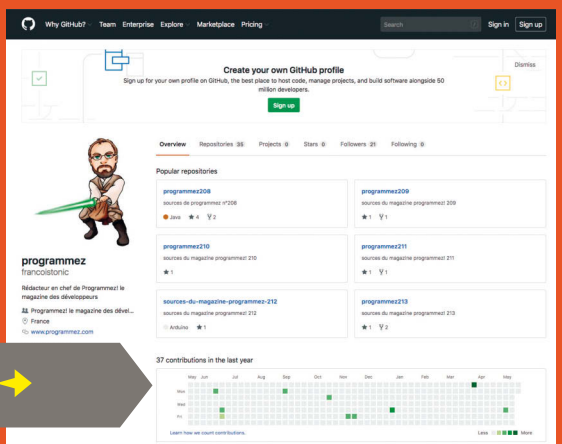
Que retenir ?

Que ce soit pour .Net 6 ou EF Core 6, les performances ont été au centre des actions de la communauté, et je dois dire que j'ai été séduit par celles-ci. Même si pour EfCore 6 certaines améliorations sont difficilement transposables pour une application en production, les gains de performance sont appréciables. Personnellement, j'ai hâte de pouvoir migrer mes applications et voir les gains que je peux tirer de .Net 6 !

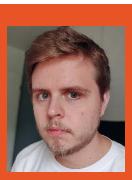
OÙ RÉCUPÉRER LES CODES SOURCES DES ARTICLES ?



programmez.com



github.com/francoistonic



Gatien Montreuil
Concepteur Développeur

SQLI DIGITAL EXPERIENCE

LE TOUR D'HORIZON DE VS 2022

Nous vous proposons de passer en revue tout ce que Microsoft a révélé en avant-première de Visual Studio 2022. Suivez le guide !

Pour commencer, Microsoft a porté son attention sur l'esthétique. Parmi les changements qui ne manqueront pas de vous sauter aux yeux, les équipes de Microsoft ont **redessiné certaines icônes**, les aplats de couleurs cédant leur place à des remplissages semi-transparents et des traits globalement affinés. **Figure 1**

La **police** utilisée sera ***Cascadia***. Cette police open-source de Microsoft, que l'on retrouve d'ailleurs dans le nouveau *Windows Terminal*, voit son intérêt au sein de *Visual Studio* grâce à sa déclinaison *Cascadia Code*, qui apporte les **ligatures** pour une bonne partie des opérateurs et décorations courantes dans du code. On gagne en **lisibilité** dans l'éditeur.

Le thème sombre a également subi quelques modifications dans la troisième prévisualisation pour limiter le risque de fatigue oculaire. En ce sens, la **couleur d'accent** bleu cyan tire sa révérence pour devenir **violette**, plus proche de l'identité visuelle de Visual Studio. **Figure 2**

Nouvelles technos et autres bonnes nouvelles

Avec cette nouvelle interface légèrement remaniée, les utilisateurs pourront s'essayer à de nouvelles technologies, en particulier .NET 6 dont la mise en production est prévue en novembre. **Blazor,** quant à lui, reste bien entendu de la partie, de même que le framework **.NET Multi-platform App UI (.NET MAUI)** qui remplacera Xamarin.

Les développeurs **C++** disposeront de la version **20** et de ses **outils de génération** en version **143**.

Les profils orientés frontend y trouveront également leur bonheur grâce à de nouveaux modèles de projets basés sur les frameworks front JavaScript / TypeScript les plus répandus, à savoir **Angular**, **React** et **Vue.js**. Ces nouveaux modèles sont déclinés dans une version autonome (standalone) et une version liée à un backend ASP.NET Core.

Enfin, un point d'honneur semble avoir été mis pour faciliter le **développement d'applications multiplateformes**, en témoignent la prise en charge de **projets Linux** basés sur *CMake* ou *MSBuild*, ainsi que la possibilité de **générer** voire **déboguer** des applications à travers une machine virtuelle *Windows Subsystem for Linux* (WSL), sans avoir recours à une connexion SSH.

Figure 1

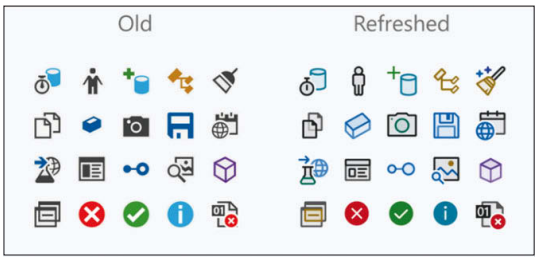
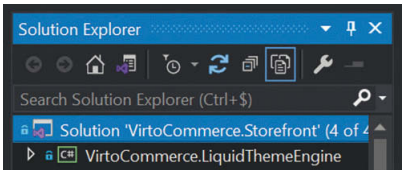
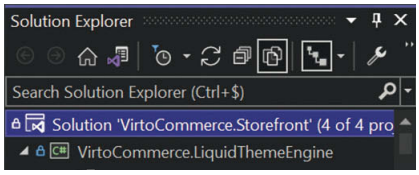


Figure 2



VS2019



VS2022 preview 3

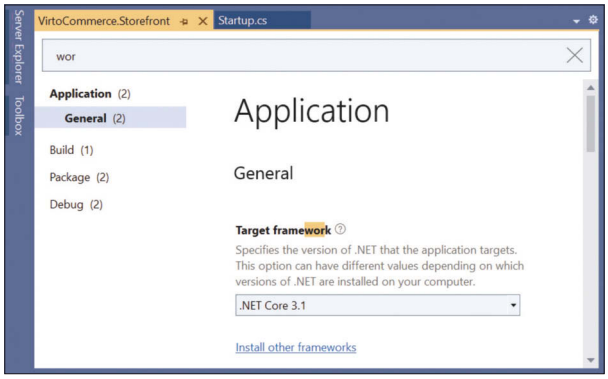


Figure 3

Le concepteur de projet fait peau neuve

Visual Studio a entièrement revu le **concepteur de projet**. Également accessible via les propriétés du projet, ce nouveau concepteur s'organise désormais autour d'**une seule colonne**, cette fois-ci subdivisée en **sections repliables** et le tout **intégré dans la charte graphique de Visual Studio**. De même, cette nouvelle mise en page accueille un **module de recherche**, utile pour modifier une propriété précise sans avoir à naviguer pour retrouver son emplacement. Enfin, les néophytes apprécieront la présence de **descriptions** autour de chaque propriété afin de les guider au mieux. **Figure 3**

Hot reload, everywhere !

Tel est le credo choisi lors de l'annonce de cette nouveauté. En effet, depuis la présentation de la deuxième préversion, le ***Hot Reload*** a fait son apparition dans Visual Studio. Ce mécanisme **simplifie les sessions de débogage en poussant du nouveau code à la volée lorsqu'une modification est détectée**. Dès à présent, nul besoin d'arrêter puis relancer une nouvelle session à chaque changement ! Ce mécanisme se révèle d'autant plus appréciable, sachant le temps que peut demander l'opération sur des projets de grande envergure.

A ce jour, le *Hot Reload* ne supporte que les projets MSBuild en C++, mais **le support devrait s'élargir** à minima aux applications Blazor, .NET MAUI, WebAssembly, et s'appliquer en CSS le cas échéant. Microsoft a également annoncé vouloir étendre ce principe de *Hot Reload* au processus de publication / déploiement du code, ce qui permettrait aux applications en production d'être mises à jour sans d'interruption.

Live Preview

Peut-être avez-vous déjà utilisé l'extension ***Web Live Preview*** de Microsoft ? Cette extension propose, pour les projets **ASP.NET,** de refléter le code source de la vue dans le HTML rendu **en temps réel,** sans même attendre le rafraîchissement de la page.

Avec Visual Studio 2022, une mise à jour de l'extension a déjà été déployée pour prendre en charge les champs liés à une source de données ; mais surtout, cette notion de rendu en direct est **étendue aux applications WPF** grâce à ***XAML Live Preview***.

Actuellement, *XAML Live Preview* propose déjà, durant une session de débogage, d'inspecter chaque fenêtre en défilant ou zoomant sur l'aperçu, mais également de positionner des règles afin de vérifier les espacements et / ou positions des différents contrôles. Ces derniers sont également sélectionnables, afin d'inspecter leurs propriétés. Pour les applications utilisant plusieurs fenêtres simultanément, une liste déroulante est disponible afin de forcer l'aperçu sur l'une d'elles (par défaut, le rendu sera celui de la fenêtre principale).

Du mieux pour l'accessibilité de vos développements

Pour aider les développeurs à rendre leurs applications accessibles, la firme de Redmond disposait d'un outil en interne nommé *Accessibility Insights*, dont le but est d'analyser des interfaces afin de **signaler les problèmes d'accessibilité**.

Après l'avoir rendu disponible à tous en tant que **standalone** en 2019 (extension Chromium pour les pages web, applicatif lourd pour interfaces desktop et Android, code source sur GitHub), ***Accessibility Insights* devrait être intégré dans *Visual Studio 2022***, même si Microsoft n'a pas encore apporté de détails sur cette intégration.

En intégrant le moteur d'Accessibility Insights au cœur de Visual Studio, vos différentes interfaces graphiques pourront être analysées en temps réel afin de signaler les problématiques d'accessibilité **de manière proactive** et ainsi éviter leur apparition en production. In fine, si les recommandations sont effectivement suivies et les erreurs corrigées, vos applications devraient être **éligibles au standard *WCAG*** proposé par le W3C.

Des applications hébergées dans le cloud plus attrayantes

Afin de promouvoir son offre cloud, Microsoft compte **promouvoir la mise en place d'applications basées sur Azure** dans Visual Studio. A destination des non-initiés, cela prendra la forme d'exemples validés par la communauté dont le code source est disponible, **exemples dont l'architecture servira de modèle**.

Ainsi, quiconque désire créer une application similaire aura à disposition une solution clé en main, **contenant tous les éléments infrastructure-as-code** permettant de provisionner toutes les ressources nécessaires dans Azure, la définition de la **stratégie d'intégration et de déploiement continus**. Ainsi, à la création du projet, celui-ci est déjà prêt à être déployé, et les développeurs peuvent se focaliser sur son développement.

Des points d'arrêts en tout genre **Figure 4**

Avec les points d'arrêts seront plus complets que jamais : comme sur *Rider* ; Visual Studio 2022 permettra de **créer des liens de dépendance entre différents points d'arrêts** (rendre un point d'arrêt B actif uniquement si le point d'arrêt A a été atteint), et ajoutera également la notion de **point d'arrêt temporaire** (supprimé après avoir été atteint).

Pour rendre plus intuitive la manipulation de ces points d'arrêts, les **lignes éligibles à les accueillir seront indiquées au survol de la souris** par l'apparition de l'icône de manière semi-transparente. Ensuite, un **clic droit** permettra de créer au choix : un point d'arrêt conditionnel, temporaire, ou un point de trace en un clic. Enfin, les points créés **pourront être déplacés** à la manière du curseur, en les glissant d'une ligne à une autre. **Figure 5**

Des petites nouveautés çà et là pour une meilleure expérience

Similaire à dotPeeks, un autre produit de JetBrains, l'EDI de Microsoft serait désormais capable de **décompiler des librairies en l'absence de code source** afin de donner un aperçu de l'implémentation, par exemple dans le cas où votre code reposerait sur une librairie externe. Plus simplement, si ladite libraire est open-source, Visual Studio prendra l'initiative de télécharger le code source afin de donner un aperçu sans passer par cette phase de décompilation. Ces sources externes seront regroupées dans l'explorateur de solution, directement sous la solution dans une section intitulée "Sources Externes".

Autre évolution, cette fois-ci concernant la recherche de code avec l'ajout d'une nouvelle portée. Celle-ci permet de **rechercher au-delà de la solution actuelle**, le but étant de retrouver l'extrait de code recherché, peu importe la solution dans laquelle il se trouve.

D'autres nouveautés visent à simplifier la surveillance de votre application et délimiter les parties sensibles de votre code, avec notamment le *flame chart*, un nouveau graphique dont le but est de **mettre en évidence les *chemins de code dits réactif* (ou *à chaud*)**, c'est-à-dire les plus régulièrement utilisés et / ou les plus demandeurs en ressources.

Visual Studio 2022 dote l'**explorateur de tests** d'une nouvelle liste déroulante afin de **choisir l'environnement d'exécution**. Ainsi les développeurs d'applications multiplateformes peuvent valider la bonne exécution de ces tests, non seulement dans leur **environnement local**, mais également dans des **containers Linux avec le SDK .NET** correspondant, dans des machines virtuelles Linux grâce à **WSL** ou dans toute autre machine connectée par **SSH**, couvrant ainsi plus de configurations différentes, et surtout bien plus simplement qu'auparavant. **Figure 6**

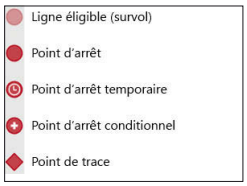


Figure 4

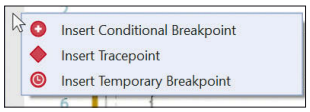
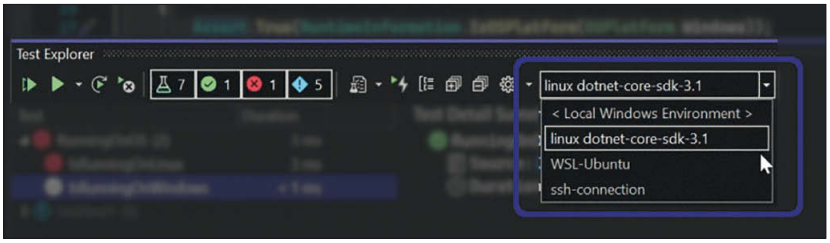


Figure 5

Figure 6



De son côté, *IntelliCode*, la version assistée par l'IA d'*IntelliSense*, s'est déjà démarquée depuis son intégration dans les précédentes éditions de Visual Studio par sa capacité à **prédire l'intention** des développeurs à travers son système d'auto-complétion. Pour la version 2022 de l'EDI made in Microsoft, *IntelliCode* s'est encore amélioré afin de proposer, à partir d'un simple mot-clé, de **construire le reste de la ligne de code**. A terme, l'objectif pour *IntelliCode* est de **suggérer**, à l'instar de *GitHub Copilot*, **des implémentations complètes**. La finalité pour Microsoft est multiple : proposer une expérience dans laquelle moins de temps est perdu sur des particularité syntaxiques propres à chaque langage de programmation au profit de la conception générale de vos algorithmes, et guider les profils les moins expérimentés en leur proposant des extraits de code "propre" pour *in fine* éradiquer les *code smells*. **Figure 7**

Si toutefois *IntelliCode* ne suffit pas et que vous cherchez à solliciter l'aide de quelqu'un, vous aurez sans doute recours à *Live Share*. Cet outil, intégré à Visual Studio, sert à créer des sessions de développement collaboratif en temps réel.

Avec Visual Studio 2022, *Live Share* se dote de nouvelles fonctionnalités parmi lesquelles un **chat textuel** afin d'échanger au sein de la session et éviter d'avoir à changer de contexte en basculant sur une application tierce, ou encore la possibilité de mettre en place des **sessions récurrentes** grâce à la génération de **liens réutilisables**.

Enfin, **les organisations pourront mettre en place leur propre politique de sécurité** sur les sessions de partage, notamment pour bloquer le partage du terminal avec accès d'écriture sur des machines dites sensibles.

Devenv passe au 64-bits

Visual Studio 2022 fait un bond en avant, puisque son processus principal (*devenv.exe*) devient un **processus 64-bits**. Ainsi, il pourra exploiter **plus de 4Go de RAM**, et éviter les erreurs Out-Of-Memory lors du chargement de solutions volumineuses.

De plus, les **performances** devraient s'accroître pour les ordinateurs possédant une architecture 64-bits puisque *VS 2022* n'aura plus besoin d'être exécuté à travers la couche de compatibilité *Windows on Windows64* (*WoW64*). Rassurez-vous, même si l'EDI fonctionne désormais sur 64-bits, vos applications resteront, malgré tout, compilables pour des plateformes 32-bits.

Microsoft a annoncé des temps de chargement 2,5 fois plus rapides. Pour valider le gain annoncé bien que la solution utilisée par l'éditeur pour sa propre démonstration ne soit pas disponible, nous avons réalisé quelques essais avec le code source du framework *aspnetcore* disponible sur GitHub.

Composé de 518 projets pour un poids total de 2Go, l'ampleur de la solution devrait nous permettre de mettre en exergue le gain potentiel. Pour cela, nous avons procédé ainsi : sur une même machine, nous avons tout d'abord lancé la dernière version en date de Visual Studio 2019, puis chronométré plusieurs actions comme le chargement complet de la solution et son déchargement. Après un redémarrage complet, nous avons répété la séquence cette fois-ci avec VS 2022 (*preview 3.1* au moment du test).

Quelques précisions sur l'environnement de test : il s'agit d'un ordinateur exécutant Windows 10 Pro en 64 bits qui est équipé de 16Go de RAM dont au moins 10 de libres, et les différentes versions de Visual Studio ainsi que la solution ont été installées sur un disque dur type NVMe pour limiter le risque de goulot d'étranglement.

Opération	Visual Studio	
	2019	2022 (pre 3.1)
Chargement	67s	30s
Déchargement	18s	7s
Compilation*	5.450s	5.650s*

**fera office de témoin, car géré par le SDK et non pas par VS*

A l'issue de ces quelques tests, nous validons d'ores et déjà la démonstration de Microsoft puisque lors du chargement de la solution, le processus devenv.exe a volontiers utilisé plus de 5.1Go de RAM, prouvant l'intérêt pour Visual Studio de passer au 64-bits. De plus, certains **chargements ont effectivement été réduits drastiquement** avec Visual Studio 2022 (~2.3x plus rapide pour le chargement, ~2.5x plus rapide au déchargement). **Figure 8**

Côté pomme

Enfin, les utilisateurs de ***Visual Studio for Mac*** ne sont pas en reste, puisque Microsoft a réservé plusieurs nouveautés à cette déclinaison. La nouveauté majeure est la **migration** d'une bonne partie du code de **l'interface** de *Visual Studio for Mac* sur du **code natif**. Ce changement permettra notamment de **prendre en compte les options d'accessibilité** sélectionnées au niveau du système d'exploitation, et globalement d'**améliorer les performances et la stabilité** de l'EDI. L'autre priorité est de réduire le delta entre l'expérience *Visual Studio* sur *Mac* par rapport à *Windows*. En ce sens, la **terminologie** et certains **menus** ont déjà été **unifiés**, et **l'intégration de *Git*** serait à terme entièrement disponible dans Visual Studio for Mac. Dans cette preview, nous pouvons d'ores et déjà utiliser la fenêtre « *Git Changes* » pour inclure / exclure des modifications avant de commit, et d'autres fonctionnalités devraient apparaître au fil des versions.

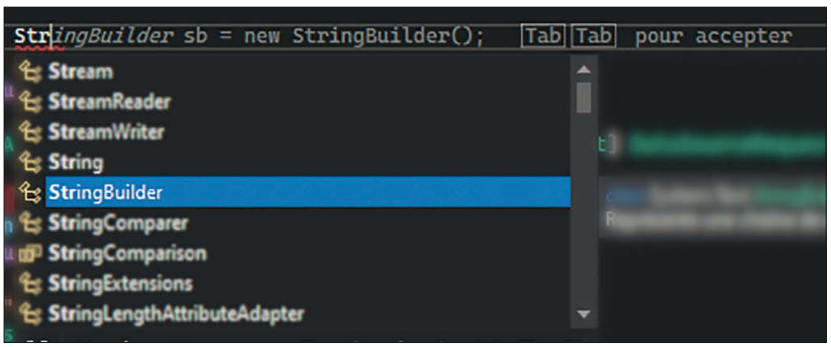
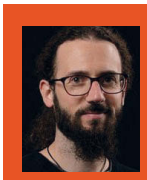


Figure 7

Figure 8

Java 17 : quoi de neuf ?

Java 17 est sortie le 14 Septembre dernier. Cette release ne contient pas beaucoup de JEP, et donc pas beaucoup de nouveautés importantes. Par contre, elle apporte quelques petites fonctionalités sympathiques, et, son statut de LTS (Long Term Support), en fait une release importante.



Loïc Mathieu
Consultant & formateur
Zenika Lille

zenika
<animés par la passion>

JEP 406: Pattern Matching for switch (Preview)

C'est sans doute la plus importante nouveauté de cette release, le pattern matching arrive dans les switchs, en preview. On peut désormais faire un switch sur le type d'une variable (y compris enum, record et tableau), et en extraire une variable locale au case qui sera du type correspondant. La JEP donne l'exemple suivant avec une chaîne de if/else :

```
static String formatter(Object o) {
  String formatted = "unknown";
  if (o instanceof Integer i) {
    formatted = String.format("int %d", i);
  } else if (o instanceof Long l) {
    formatted = String.format("long %d", l);
  } else if (o instanceof Double d) {
    formatted = String.format("double %f", d);
  } else if (o instanceof String s) {
    formatted = String.format("String %s", s);
  }
  return formatted;
}
```

Qui peut maintenant être ré-écrit avec une switch expression :

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

En plus de supporter du pattern matching, le switch permet maintenant de définir un case spécial null (dans ses deux formes : statement ou expression). Auparavant, une variable de switch nulle entraînait une NullPointerException. Dans sa nouvelle forme, on peut ajouter un case null pour gérer les nulls au sein du switch. Sans case null, l'ancien comportement est gardé, et une NullPointerException sera levée.
Si on reprend l'exemple précédent, cela nous donne :

```
static String formatterPatternSwitch(Object o) {
  return switch (o) {
    case null -> "null";
    case Integer i -> String.format("int %d", i);
    case Long l    -> String.format("long %d", l);
    case Double d  -> String.format("double %f", d);
    case String s  -> String.format("String %s", s);
    default        -> o.toString();
  };
}
```

Le grand intérêt est que cela évite de devoir faire un test défensif avant le switch, et permet d'inclure dans celui-ci la valeur nulle comme toute autre valeur possible de notre variable. Et ça ne s'arrête pas là, le switch a été enrichi de guards qui permettent d'inclure une condition au case. L'exemple suivant issu de la JEP montre son utilisation. En ajoutant un **guard** au case Triangle : case Triangle t && (t.calculateArea() > 100), on peut créer deux cases : un pour les grands triangles, et un autre pour les petits.

```
static void testTriangle(Shape s) {
  switch (s) {
    case Triangle t && (t.calculateArea() > 100) -> {
      System.out.println("Large triangle");
    }
    case Triangle t -> System.out.println("Small triangle");
    default -> System.out.println("Non-triangle");
  }
}
```

Plus d'informations dans la JEP-406 : https://openjdk.java.net/jeps/406

JEP 356: Enhanced Pseudo-Random Number Generators

La JEP-356 fournit une nouvelle interface RandomGenerator, et une factory RandomGeneratorFactory, qui permettent d'accéder à une implémentation de générateur de nombre aléatoire. Les générateurs existant : Random, SecureRandom, SplittableRandom et ThreadLocalRandom; implémentent maintenant cette interface qui ajoute, entre autre, l'accès à un stream de nombre aléatoire (RandomGenerator::doubles(), RandomGenerator::ints(), …). De nouveaux algorithmes de génération de nombre aléatoires ont été implémentés, plus sécurisés et plus performant (mais ils ne sont plus thread-safe), ils ont vocation à remplacer les anciens. Voici quelques exemples d'instanciation de générateur de nombres aléatoires :

```
// the old Random generator
RandomGenerator rng1 = RandomGeneratorFactory.of("Random").create(42);
// the default random generator, currently L32X64MixRandom
// but this can change
RandomGenerator rng2 = RandomGeneratorFactory.getDefault().create(42);
// shortcut for the default
RandomGenerator rng3 = RandomGenerator.getDefault();
// stream all available generators and display their names
RandomGeneratorFactory.all().forEach(generator ->
    System.out.println(generator.name());
```

Plus d'informations dans la JEP-356 : https://openjdk.java.net/jeps/356

Les fonctionnalités qui passent de preview à standard

Dans Java 17, une seule fonctionnalité passe de preview à standard : les Sealed Classes qui permettent de limiter le

nombre d'implémentations d'une classe ou d'une interface à une liste prédéfinie. La hiérarchie de cette classe/interface est donc close (scellée / sealed). Celles-ci existent depuis Java 15. Plus d'informations sur la JEP-409 : https://openjdk.java.net/jeps/409

Les fonctionnalités qui restent en preview

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

• Vector API : seconde incubation de la fonctionnalité. Vector API est une nouvelle API qui permet d'exprimer des calculs de vecteur (calcul matriciel entre autres), qui seront exécutés via des instructions machines optimales en fonction de la plateforme d'exécution.

Plus d'informations dans la JEP-414 : https://openjdk.java.net/jeps/414.

• Foreign Function & Memory API : nouvel incubator pour ces deux fonctionnalités qui sont maintenant liées (l'une utilisation l'autre) au sein d'un même incubator.

• Foreign Memory API permet de gérer des segments mémoire (on heap ou off heap) tandis que Foreign Function l'interconnexion de la JVM avec du code natif (en C par exemple) de façon facile et performante. Ces deux API sont les bases du projet Panama.

Plus d'informations dans la JEP-412 https://openjdk.java.net/jeps/412.

Un nouveau port de la JVM

Java 17 ajoute le support de l'architecture macOS/AArch64 (aka Apple Silicon). Plus d'informations dans la JEP-412 https://openjdk.java.net/jeps/391.

HexFormat

La class java.util.HexFormat permet la conversion de type primitif, tableau de byte, ou tableau de char en chaîne de caractère hexadécimal et vice versa.

```
HexFormat.of().toHexDigits(127); // "7f"
HexFormat.of().fromHexDigits("7f"); // 127
```

InstantSource

Tester du code contenant de la manipulation de date a toujours été un challenge, surtout si celui-ci use et abuse de System.currentTimeMillis(), LocalDateTime.now(), et autre initialisation de date avec la date en cours.

Pour faciliter la testabilité de ce genre de code, une nouvelle interface a été ajoutée au JDK : InstantSource, avec une seule implémentation : Clock. Le but de l'interface InstantSource est d'être une fabrique d'Instant. Au lieu de créer un Instant avec la date du jour, vous le créez depuis l'InstantSource. Un test pouvant alors utiliser une InstantSource à une date fixe au lieu de la date du jour. Imaginez le code suivant :

```
public class MyBean {
  private InstantSource source;  // dependency inject
  ...
  public void process(Instant endInstant) {
    if (source.instant().isAfter(endInstant) {
      ...
    }
  }
}
```

En fonctionnement normal, l'InstanSource est initialisé via InstantSource.system(), et en test via InstantSource.fixed(LocaDateTime.of(//the hardcoded date time)).

Divers

Divers ajouts au JDK :

- Map.Entry.copyOf(Map.Entry) : permet de créer une copie d'une entrée de Map qui ne soit pas connectée à la Map existante.
- Process.inputReader(), Process.outputWritter(), Process.errorReader() : permet d'accéder aux entrée/sortie standard et sortie erreur d'un process via un Reader ou un Writer.

Dépréciation et encapsulation

Java 17 voit la dépréciation pour suppression de l'API des Applets via la JEP-398, celle-ci n'étant plus utilisée depuis de nombreuses années, elle n'a pas fait couler beaucoup d'encre. Java 17 voit aussi la dépréciation du Security Manager pour suppression via la JEP-411. Il y a eu beaucoup de débat autour de cette annonce, avec, il faut l'avouer un peu de drama, comme quand Apache Netbeans a annoncé qu'il ne pourrait pas supporter Java 17 alors qu'il fallait juste changer quelques lignes de code…

La suppression du Security Manager a été annoncée, car celui-ci est complexe à maintenir, coûteux en termes de performance, et n'apporte pas la sécurité nécessaire face aux enjeux actuels. Il a été créé pour sécuriser les applets qui, par définition, exécutent du code untrusted, et n'a donc plus de sens dans une JVM qui ne contiendrait plus l'API Applet.

Pour finir, la JEP 403: Strongly Encapsulate JDK Internals a basculé le JDK vers une encapsulation stricte de ses classes internes. C'est la fin de ce qui avait été commencé avec Java 9 et la modularisation du JDK.

Concrètement, le mode d'encapsulation était passé de --illegal-access=permit en Java 15 à --illegal-access=deny en Java 16 avec la possibilité de changer l'option de configuration. Avec Java 17, --illegal-access disparaît et l'accès aux classes internes du JDK (hors Unsafe) n'est plus possible.

Conclusion

Même si cette version de Java ne comporte pas beaucoup de JEP, elle propose un grand nombre de nouveautés, et semble paver le chemin pour l'aboutissement du projet Panama dans une future version.

En dépréciant les Applets et le Security Manager, le JDK va aussi se séparer de pas mal de code historique plus utilisé, ou plus adapté au monde actuel. Même si cela induit quelques petits désagréments évidents, et complexifie le support de cette version dans de nombreux framework, c'est un mal pour un bien. Cela permettra de faciliter la maintenance du JDK, et aux équipes de celui-ci à se focaliser sur l'apport de nouveautés, le Security Manager étant une source d'importante complexité au sein du JDK.

Pour finir, le statut Long Term Support (LTS) de cette release va en faire la release de choix pour de nombreux développeurs, et il faut s'attendre à ce que beaucoup de développeurs migrent directement de Java 11 à Java 17 (plus que de migration de Java 16 à Java 17). De plus, l'annonce faite à Spring One de baser sur Java 17 minimum les futurs Spring Framework 6 et Spring Boot 3 (GA fin 2022) va définitivement entériner Java 17 comme version à privilégier en 2022.

LE DÉVELOPPEUR VA SAUVER LA TERRE !

PARTIE 1 : POSONS LES FONDAMENTAUX

Le développeur est le Chuck Norris de l'éco-conception. Sans lui, on ne peut pas réduire la taille des applications, réduire l'usage des ressources IT ou encore améliorer l'utilisation réseau. On peut appliquer l'éco-responsabilité dans les apps mobiles, le cloud, le desktop. Le développeur peut cibler le code, les piles techniques, tous les assets techniques et graphiques. Rien que sur les sites web, nous pourrions plusieurs Mo en optimisant les ressources et le poids de chaque page. Il faut revenir à un principe simple : 1 Ko est un 1 Ko utile. Dans les années 1980, chaque Ko était une ressource précieuse qu'il fallait utiliser avec rigueur. Nous avons perdu cette obsession des contraintes matérielles. Comme le sujet est énorme, nous avons décidé de publier un dossier sur deux numéros. La partie 2 sera publié dans le n°251.

Le Green Software dans tous ses états

Mathieu TOUCHARD
Pilote Innovation
Direction de l'Innovation
Groupe ALTEN

Au sein de la Direction de l'Innovation du Groupe ALTEN, nous sommes convaincus de l'importance des considérations environnementales et de l'opportunité qu'elles constituent.
En effet, l'impact environnemental, qu'on le mesure en W, en gCO2, en %CPU ou en ko est une métrique supplémentaire du système que l'on conçoit, et une métrique fortement liée à des considérations de performances.

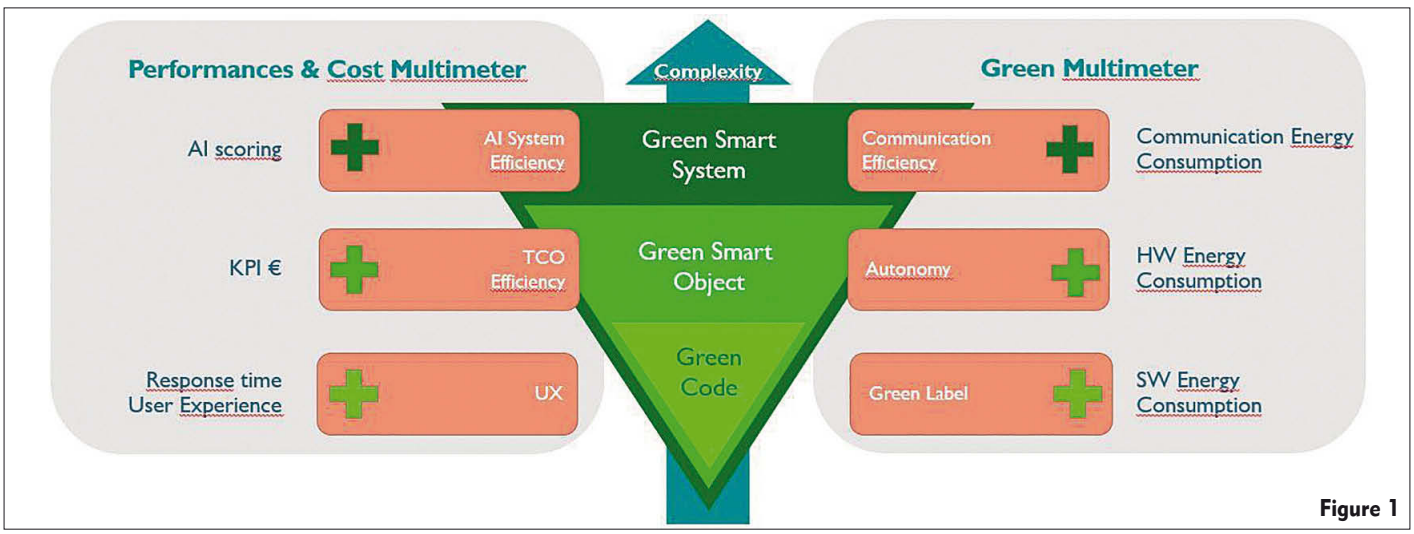


Figure 1

Dans un software, travailler l'obésité du code permet de gagner « facilement » 30% de performance, qui se traduisent souvent par des temps de chargement plus courts, et donc une expérience utilisateur améliorée.
Si ce logiciel est embarqué sur un smartphone, ces 30% sont autant d'autonomie, et donc de temps d'utilisation de l'application avant une recharge sur le secteur, ou avant un changement de batterie. Lors de la conception d'un capteur intelligent destiné à l'industrie, ces gains sont directement liés au coût total de possession du capteur, car la maintenance destinée à remplacer la batterie est souvent coûteuse.
Lorsqu'enfin on passe à l'échelle d'un système complexe assemblant de multiples logiciels & intelligences, le coût énergétique de la performance devient crucial. Ainsi qui voudrait d'une voiture électrique et autonome dont la batterie ne permettrait pas de parcourir plus de 13 km entre 2 bornes ?

Figure 1

Nous découpons nos travaux en trois projets, d'ampleur et de complexité croissante :

- Le projet Green Code porte sur le code, notamment dans les usages mobiles & web, et a pour objectifs d'en maîtriser les outils de mesure de l'impact environnemental et de la consommation énergétique, ainsi que les leviers d'actions. Ici le TRL est élevé, des outils existent, et notre effort se situe dans l'agrégation de ces outils et l'analyse des résultats pour en établir des leviers d'amélioration.
- Le projet Green Smart Objects s'intéresse aux objets intelligents, objets connectés et autres IOT & IIOT, dans lequel on doit mesurer et maîtriser à la fois la consommation du code, celle du hardware, et celle de la communication (wifi, Bluetooth, Lora,…). Dans ce projet, notre démarche de mesure de la performance « green » des objets intelligents et communicants devra nous permettre d'aller vers une modélisation green MBSE de ces systèmes en phase de conception.
- Enfin, le projet Green Smart System passe à l'échelle des systèmes complexes tels l'usine du futur, le véhicule autonome, ou encore l'avion de chasse. Dans ces systèmes, les intelligences sont en réseaux communicants, et la performance du système global repose non seulement sur les performances des sous-systèmes, mais aussi sur l'intelligence de la communication et les choix d'architectures (distribuée, centralisée, hétérarchique). L'enjeu « Green » devient alors un enjeu de performances de systèmes intelligents. Au travers de ce projet, notre objectif est de s'inspirer de la consommation énergétique du cerveau humain à 40 watts. En comparaison, le système d'intelligence artificielle Watson d'IBM qui a remporté le jeu « Jeopardy! » en 2011 avait besoin de 80.000 watts et AlphaGo de DeepMind qui a battu le meilleur joueur de go en 2016 faisait à peine mieux avec 20.000 watts. Nous nous attaquons ainsi à l'efficacité énergétique d'une intelligence artificielle en inventant une nouvelle échelle de mesure de l'efficacité d'une IA pour permettre de mieux les comparer et espérer ainsi s'approcher de l'efficacité énergétique de l'intelligence humaine…

Pierre LAGARDE
Principal Program Manager
Windows & Devices - Microsoft Corp

Qu'est-ce que la « Green Software Foundation » ?

La « Green Software Foundation » est une organisation à but non lucratif créée en 2021 basée sur la Linux Foundation. Elle est née d'un désir et d'un besoin mutuel de collaboration au sein des équipes de développement. Elle regroupe des organisations qui partagent un engagement envers le développement « green » autour de principes qui ont pris naissance il y a un an ici : https://principles.green/fr-fr/.

La mission de la « Green Software Foundation » est de construire un écosystème de confiance composé de personnes, de normes, d'outils et de meilleures pratiques pour la création et le développement d'applications « green » ou durables afin de contribuer à la réduction des émissions de carbone. Le choix intrinsèque qui a été défini par la fondation est bien de réduire les émissions et non de les compenser.
Elle a pour vision de changer la culture du développement logiciel dans l'ensemble de l'industrie technologique, afin que le développement « green » devienne une priorité pour les équipes projet, au même titre que la performance, la sécurité, ou les coûts aujourd'hui.
Tous les documents produits par la fondation sont en open source sur GitHub. https://github.com/Green-Software-Foundation
La fondation est constituée de 4 groupes de travail.

1 Les standards

Le périmètre de ce groupe de travail est de développer une série de spécifications de base pour les logiciels « green ». S'assurer que les spécifications peuvent être mises en œuvre de manière interopérable sur les différentes plates-formes les plus utilisées et dans toutes les régions.

2 Innovation

Orienter la recherche universitaire, les innovations sur le développement et sur les données pour faire progresser le développement « green ».
Sponsoriser les principaux projets open source autour du développement « green » afin de garantir leur pérennité.

3 Marque

S'assurer que les marques de la Fondation sont utilisées correctement et dans un contexte respectant les valeurs de la Fondation. Définir des directives concernant quand, comment et dans quel contexte les marques de la Fondation peuvent être utilisées.

4 Communauté

Faciliter une large adoption des normes et des meilleures pratiques en matière de développement « green » par la création de partenariats, de contenus, la participation à des événements. Assurer une participation diversifiée des contributeurs.

LES PRINCIPES DE DÉVELOPPEMENT « GREEN »

Écrire des principes de développement « green » est un exercice qui nécessite des connaissances scientifiques, sur le climat, sur la production électrique, le matériel électronique et bien sûr les principes de fonctionnement des data centers qui sont avec l'Intelligence Artificielle au centre des débats du développement « Green », car ils représentent une large partie des émissions de carbone pour le secteur du numérique.
Pour prendre ces 2 exemples, les premiers datacenters ont été conçus pour répondre à une forte demande en termes de ressources et de haute disponibilité, quant aux architectures d'IA, elles étaient basées sur de très grandes quantités de données et des algorithmes extrêmement gourmands en puissance de calcul. Dans l'élaboration de ces architectures, l'environnement n'entrait pas dans l'équation. Le défi qu'il faut relever maintenant c'est de redéfinir ces architectures logicielles en prenant en compte les différents aspects environnementaux liés au domaine du numérique.
C'est pourquoi une initiative lancée par Asim Hussain a vu le jour pour mettre par écrit ces « grands » principes de développement qui permettront de définir ce qu'est une application ou un service numérique « green ». Aussi bien au niveau du code source, mais aussi pendant l'exécution des applications et des services, et ce, indépendamment du domaine d'application, de l'industrie, de la taille ou du type de l'organisation, du fournisseur de cloud ou même du langage.

Ces principes de développement sont au nombre de 8 aujourd'hui, mais ils sont amenés à évoluer à travers la Green Software Foundation et ses contributions open source.
Voici un résumé de ces 8 principes que vous pouvez retrouver en français sur le site : https://principles.green/fr-fr/

1. Minimiser l'empreinte carbone : cela paraît simple, mais penser son application pour avoir une empreinte carbone faible doit être prise en compte dès la phase de conception et en tenant compte du maximum de paramètres : l'usage, le nombre d'utilisateurs, le temps d'utilisation, etc.
2. Économiser l'électricité : la consommation électrique des composants d'un ordinateur n'est pas égale et doit être prise en compte pour venir optimiser sa consommation électrique. Par exemple, un GPU pourra faire le même travail qu'un CPU, mais avec beaucoup moins de cycles. Par exemple, le décodage d'un codex vidéo. https://aka.ms/sse/blog/vlc
3. Prendre en compte l'intensité en CO2 : l'intensité carbone de l'électricité est une mesure de la quantité d'émissions de carbone (CO2éq) produite par kilowatt-heure d'électricité consommée. Chaque pays ou région va avoir une intensité carbone différente à un instant t en fonction de son type de production. Cette information peut être prise en compte dans une application pour venir réduire son émission de carbone et aussi éviter les émissions marginales en déplaçant la consommation électrique, par exemple, en différant une sauvegarde ou une indexation de données. **Figure A**

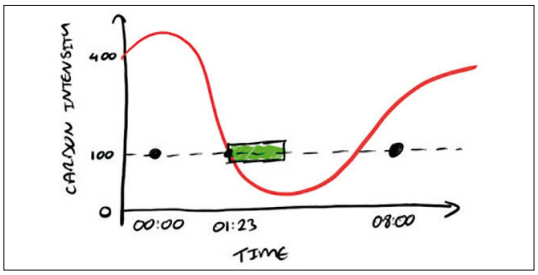


Figure A

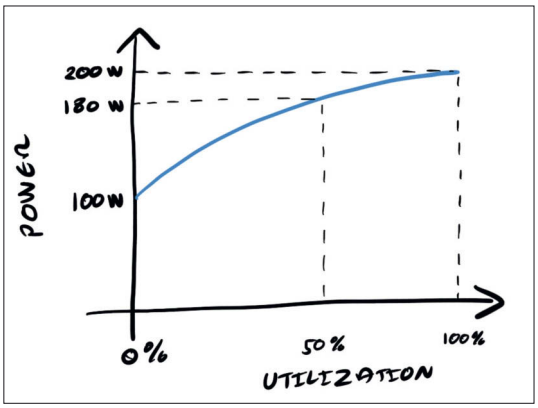


Figure B

4. Valoriser l'impact CO2 global : en créant des applications qui sont capables de s'exécuter sur du matériel plus ancien pour allonger la durée de vie du matériel, ordinateurs, tablettes, téléphones ou serveurs.
5. Proportionnalité énergétique : utiliser des serveurs avec un taux d'utilisation élevé, car la relation entre la puissance et l'utilisation n'est pas exactement proportionnelle, car un ordinateur inactif n'a pas une puissance à 0. **Figure B**
6. Mise en réseau : réduire la quantité de données et la distance à parcourir sur le réseau. Les différents facteurs d'émissions sont par exemple : la distance, le nombre d'appareils réseau traversés, ou encore le protocole réseau utilisé.
7. Formuler au mieux la demande : au lieu de façonner l'offre pour répondre à la demande, essayez de façonner la demande pour qu'elle corresponde à l'offre. Pour donner un exemple, les logiciels de vidéoconférence vont réduire la qualité vidéo pour donner la priorité à l'audio en cas de forte affluence.
8. Mesure et optimisation : concentrez-vous sur les optimisations de bout en bout qui augmente l'efficacité globale en empreinte carbone et utilisez des outils de mesure précis qui vous permettront de concentrer vos efforts au bon endroit.

- Sous Windows il y a la ligne de commande : powercfg.exe /srumutil qui vous donne le détail exact de la consommation électrique par process au format CSV ou XML si vous ajoutez le paramètre /XML.
- Il y a aussi un projet Open source https://github.com/powerapi-ng

Je suis convaincu que respecter ces 8 principes peut aussi bien sensibiliser les développeurs que les utilisateurs dans cette transition vers une meilleure sobriété numérique.

Raphaël Lemaire

Directeur technique chez Zenika, consultant et formateur, spécialisé dans le numérique responsable

Optimiser son site web pour réduire son impact environnemental

Les crises écologiques (climat, biodiversité, épuisement des ressources naturelles, …), sont de plus en plus présentes dans notre quotidien, et plus seulement comme des problèmes d'un futur plus ou moins lointain mais bien comme une actualité brûlante. Dans ce contexte, le numérique n'est plus vu comme neutre ou même automatiquement bénéfique mais comme ce qu'il est : un des secteurs économiques et industriels qui a un impact écologique, et qui comme les autres doit réduire cet impact.

Cet article expose des pistes d'actions possibles pour réduire l'impact environnemental du numérique dans le cadre du développement web. Il ne couvre pas l'ensemble de l'écoconception d'un service numérique qui est un travail qui implique toute l'équipe autour du projet. Nous y verrons comment évaluer l'impact d'une page, quelles pratiques mettre en œuvre, comment concevoir des applications web plus vertueuses.

Comment mesurer l'impact d'une page web d'un point de vue environnemental ?

Pour améliorer, il faut mesurer, analyser. Une chance : il existe une extension de navigateur, appelée GreenIT-Analysis, disponible pour Chrome[1] et Firefox[2], permettant d'analyser une page web d'un point de vue impact environnemental. Il s'agit d'un projet open source, dont le code est sur github[3].
Voyons ce qu'elle nous dit par exemple pour https://www.programmez.com/.

Pour l'utiliser il faut, après avoir installé l'extension, ouvrir les outils des développement du navigateur, aller dans l'onglet « GreenIT », ouvrir la page, puis lancer l'analyse. On peut optionnellement cocher la case « Activer l'analyse des bonnes pratiques » pour avoir plus d'informations. **Figure 1**

Analysons cette capture d'écran. On y trouve une note de E, déterminée à partir d'un score « EcoIndex » de 26,85. L'EcoIndex ici affiché, est inspiré de l'éco-index visible par exemple sur les appareils électroménagers. Les notes vont de A à G, et l'index de 0 à 100, 100 étant la meilleure note possible. La note est ici plutôt moyenne mais pas du tout surprenante pour une page d'accueil, qui est typiquement assez chargée.

(1) https://chrome.google.com/webstore/detail/greenit-analysis/mofbfhffeklkbebfclfaiifefjflcpad?hl=fr

(2) https://addons.mozilla.org/fr/firefox/addon/greenit-analysis/

(3) https://github.com/cnumr/GreenIT-Analysis

Figure 1

On note que l'extension fournit une estimation des émissions de gaz à effet de serre associés à la page, exprimée en grammes équivalent CO2 ainsi qu'une estimation de la consommation d'eau, exprimée en centilitre. D'où viennent ces chiffres ? Ils sont calculés à partir d'une analyse de cycle de vie d'un site sur lequel ont travaillé les concepteurs de l'extension. Une analyse de cycle de vie est une analyse des impacts environnementaux d'un produit, en tenant compte de toutes les étapes de sa vie (fabrication, utilisation, fin de vie) et en s'intéressant à plusieurs facteurs d'impacts, pas seulement les émissions de gaz à effet de serre comme pour un bilan carbone. L'extension GreenIT-analysis est donc d'abord un outil pédagogique permettant de sensibiliser à l'impact environnemental d'une page.

Individuellement, ces 2,46 centilitres et 3,69 grammes n'ont pas l'air très importants, mais si on multiplie par le nombre d'affichages, on peut arriver à des gros chiffres. Pour un site ayant des millions de vues, cela donne des tonnes équivalent CO2 par exemple. En tant que développeurs web, nous avons la chance, en pouvant optimiser ces pages, de réduire cet impact. Si on a un fort trafic, même un petit gain peut faire une grosse différence, par le pouvoir de la multiplication.

D'où viennent ces impacts environnementaux associés au web ?

Quand on pense à l'impact environnemental du numérique, on pense souvent aux data centers. On a l'image d'une salle serveur pleine de machines, qui consomment beaucoup d'électricité, qui nécessitent de la climatisation , etc… Or, contrairement à ce qu'on pense souvent, ce ne sont pas les centres de données qui sont responsables de l'essentiel de l'impact environnemental du numérique, mais les terminaux des utilisateurs, et en particulier leur fabrication. Les chiffres de l'étude de GreenIT.fr le montrent clairement.

Émissions de gaz à effet de serre du numérique mondial réparties par phase du cycle de vie et segment de terminaux

	Production	Utilisation	Total
Utilisateurs	40%	26%	66%
Réseaux	3%	16%	19%
Centres informatiques	1%	14%	15%
	44%	56%	

Utilisation des ressources naturelles du numérique mondial réparties par phase du cycle de vie et segment de terminaux

	Production	Utilisation	Total
Utilisateurs	76%	0%	76%
Réseaux	16%	0%	16%
Centres informatiques	8%	0%	8%
	100%	0%	

Source : Empreinte environnementale du numérique mondial, GreenIT.fr, 2019[4]

Cela s'explique simplement parce qu'il y a beaucoup plus de terminaux clients que de serveurs. 34 milliards contre 700 millions. Mais aussi parce que fabriquer un appareil high-tech est quelque chose de complexe et coûteux.

Un smartphone contient par exemple plusieurs dizaines de matériaux différents. Il faut extraire les minerais des métaux des mines, les concasser, les chauffer, les purifier avec des produits chimiques, pour pouvoir créer des alliages avec lesquels on peut créer des composants électroniques que l'on assemble, il faut aussi du sable pour le verre des écrans, du pétrole pour le plastique, et aussi transporter ces matières et ces composants, ainsi que les appareils jusqu'au magasin. 90% des émissions de gaz à effet de serre émises dans toute la vie d'un smartphone le sont à sa fabrication.

Sachant cela, on peut en déduire que **les actions clefs pour réduire l'impact du numérique sont d'abord d'utiliser moins de terminaux et de les faire durer le plus longtemps possible. Puis de réduire les ressources informatiques utilisées par les programmes.**

Construire des applications web qui se comportent mal sur les appareils des utilisateurs encourage ceux-ci à renouveler leurs appareils. « Je ne peux même plus surfer sur le web avec cette bécanne, il m'en faut une autre ». L'application contribue à pousser vers l'obsolescence le terminal de l'utilisateur alors que celui-ci fonctionne toujours très bien.

De plus, une application web peu performante utilisera plus les composants de la machine. La batterie se videra plus vite par exemple dans le cas d'un smartphone, ce qui l'amènera plus rapidement à la fin de sa vie, qui se compte en nombre de cycles décharge/recharge.

Enfin utiliser plus de bande passante que nécessaire, utiliser plus de ressources serveur que nécessaire, implique de dimensionner les architectures réseaux et serveur pour encaisser ce trafic. Sur une application à forte audience, une optimisation de quelques centaines de ko par page peut faire une grosse différence en volume si on multiplie par le nombre d'utilisateurs et de téléchargements.

Pour ne pas pousser au renouvellement des appareils et économiser des ressources, on va donc chercher à créer des sites légers et efficients.

Avant d'être performant il faut être compatible

Dans le cadre du développement web, cela veut dire être compatible avec les navigateurs dont les utilisateurs disposent, même s'ils sont un peu anciens ou n'implémentent pas les normes les plus récentes.

Figure 2

On a tous, c'est humain, envie de travailler avec les dernières évolutions du web, que ce soit des éléments html 5 ou des propriétés CSS. Mais les utilisateurs du site ne sont pas forcément maîtres du navigateur qu'ils utilisent, par exemple s'ils sont sur un ordinateur d'entreprise, ou sur une machine ancienne où ils ne peuvent pas installer de logiciels trop récents qui la ralentiraient. Ils peuvent même ne pas savoir qu'il est possible d'utiliser un autre navigateur que celui qu'ils possèdent déjà. On peut aussi prolonger la vie d'ordinateurs anciens en y installant une distribution linux légère et un navigateur minimal.

Le site https://caniuse.com/ permet de valider si une fonctionnalité particulière peut être utilisée : **Figure 2**

Comment optimiser son site ?

La note EcoIndex sur 100 donnée par l'extension est calculée à partir de trois indicateurs : le poids de la page (décompressée), le nombre de requêtes http et la taille du DOM, dont les valeurs sont affichées. L'idée est qu'une page avec beaucoup d'éléments et beaucoup de requêtes a des chances d'être difficile à afficher pour l'appareil, et que si elle est lourde en ko et en nombre de requêtes, elle utilise potentiellement trop de charge serveur et réseau.

Et si on y réfléchit un peu, nos pages web sont souvent très lourdes par rapport à ce qu'elles font.

Super Mario Bros, jeu iconique avec de nombreux niveaux et interactions ne pesait que 40 Ko, moins que beaucoup de fichiers servis par les sites webs aujourd'hui, que l'on parle de css, de js ou d'images. Depuis 2016, une page web est même en moyenne plus grosse que le jeu Doom, qui était un jeu en 3D avec plusieurs niveaux et des effets sonores. Quand on pense qu'un site se base sur le navigateur et sur ses API pour s'afficher et exécuter ses comportements et intéractions alors que ces programmes étaient autosuffisants et que parfois, quand on scrolle une page, on la voit ramer, la différence est encore plus impressionnante. Cela laisse une impression d'imparfait. On doit pouvoir mieux faire.

Il existe de nombreuses ressources pour améliorer la performance de nos sites web. On peut par exemple citer les checklist d'opquast[5], ou les outils de google comme Google PageSpeed Insights. En plus d'afficher un EcoIndex et une estimation d'impacts environnementaux, l'extension GreenIT

(4) https://www.greenit.fr/empreinte-environnementale-du-numerique-mondial/

(5) https://checklists.opquast.com/fr/assurance-qualite-web/

Les anciens numéros de PROGRAMMEZ!

Le magazine des développeurs

Tarif unitaire 6,5 € (frais postaux inclus)

TECHNOSAURES

Le magazine à remonter le temps !

Commandez directement sur www.technosaures.fr

☐ 235	: __ ex	☐ 240	: __ ex	☐ 246	: __ ex
☐ 236	: __ ex	☐ 241	: __ ex	☐ 247	: __ ex
☐ 238	: __ ex	☐ HS1 été 2020	: __ ex	☐ HS4 été 2021	: __ ex
☐ 239	: __ ex	☐ 242	: __ ex	☐ 248	: __ ex

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

soit __ exemplaires x 6,50 € = __ € ..... __ € soit au **TOTAL** = __ €

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

Prénom : __ Nom : __

Adresse : __

Code postal : __ Ville : __

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

Offres pouvant s'arrêter à tout moment, sans préavis PROG 249

Analysis propose des pistes d'améliorations à travers la fonction analyse des bonnes pratiques. Voyons par exemple la liste pour https://www.programmez.com/ : **Figure 3**

Ce type de checklist est utile. Car si tout le monde bien sûr veut des pages performantes et teste que celles-ci s'affichent bien, on ne parcourt pas systématiquement une liste d'idées d'améliorations.

Si on regarde dans le détail, on peut les classer selon trois axes, qui correspondent aux variables qui font la note.

Réduire le nombre de requêtes http

Il y a une ligne dédiée « Limiter le nombre de requêtes » qui s'affiche en rouge au-delà d'un certain seuil. Mais l'extension nous suggère également d'éviter les redirections, de limiter le nombre fichiers CSS, de ne pas télécharger des images non affichées ou d'utiliser des polices de caractères standard (déjà présentes sur la machine de l'utilisateur). L'utilisation des mécanismes de cache permet également d'éviter des requêtes, en fixant une date de validité du fichier par exemple.

On pourrait rétorquer qu'avec la généralisation d'http2, limiter le nombre de requêtes devient moins pertinent, car avec cette mise à jour, il n'y a pas de surcoût à faire une requête http supplémentaire plutôt que regrouper les fichiers. Mais un trop grand nombre de requêtes reste un indice qu'il y a peut être des améliorations possibles sur la page.

Réduire la bande passante utilisée

Réduire la quantité d'octets transmis permet de réduire la charge serveur et réseau. On pourrait se dire qu'un site web est très petit par rapport à d'autres choses qui transitent aujourd'hui sur internet, en particulier la vidéo ou les jeux en streaming. C'est vrai, mais comme on visite beaucoup de pages web, cela constitue tout de même une part importante du trafic et contribue à augmenter les besoins en infrastructures. De plus, les actions réduisant la bande passante utilisée vont aussi permettre d'avoir un site plus léger et rapide, ce qui réduira la pression au renouvellement des appareils. Les mécanismes de cache du navigateur permettent d'éviter de télécharger plusieurs fois la même chose. Dans cette optique, séparer les fichiers js et css du html permet de mettre ceux-ci en cache. Optimiser en taille les fichiers, images, pdf ou autres, et également minifier et compresser les fichiers qui peuvent l'être (css, js, html), permet d'avoir moins de données à transférer.

Pour les cookies : comme ils sont transférés à chaque requête, leur poids s'ajoute à chaque fois. Il faut donc faire attention à les conserver de petite taille. De plus les cookies, prévus pour conserver l'identité d'un utilisateur et gérer une session de connexion, ne sont pas utiles pour récupérer des fichiers statiques (images, css, js, documents PDF, …), il serait donc contre productif de les ajouter aux requêtes pour ceux-ci. On peut l'éviter en créant un sous-domaine différent, par exemple *static.monsite.fr*. En revanche, un grand nombre de noms de domaines est un signe que le site est probablement trop complexe.

Figure 3

Économiser du temps de calcul sur le navigateur

Enfin, certaines pratiques visent directement à limiter la charge de calcul imposée au navigateur pour l'affichage de la page. Des fichiers html, css ou js non valides nécessitent plus de calcul pour que le navigateur devine ce qu'il doit faire. Retraiter les images côté client nécessite également des ressources. Les plugins comme Flash ou Java sont aussi plutôt lents et gourmands, mais dans le web d'aujourd'hui, leur utilisation est devenue très rare.

Cette liste est très incomplète et ne liste pas tout ce qui est possible pour améliorer une page ou un site. Il s'agit d'un sous ensemble des 115 bonnes pratiques d'éco conception web[6] fournies par le collectif numérique responsable, parmi celles que l'on peut tester automatiquement. N'hésitez pas à aller lire le référentiel en entier, ainsi que d'autres comme la checklist d'opquast ou les conseils de Google PageSpeed.

Notez aussi que satisfaire l'extension pour tout avoir en vert n'est pas le but. Il s'agit d'un outil alliant un objectif de sensibilisation et la fourniture de quelques idées d'améliorations. Par exemple, si votre page ne sera jamais imprimée par personne, passer du temps sur une CSS print est sans doute inutile. Ce sera en revanche apprécié dans le cas d'un contenu textuel.

Mobile first

Il existe une stratégie de conception et de test permettant de valider qu'un site se comporte bien sur tous les appareils : mobile first. L'idée est de concevoir d'abord les pages pour des téléphones mobiles les plus contraignants en taille d'écran ou en puissance, et aussi de concevoir pour un réseau mobile faible, 3G voire 2G. Dans ces conditions, on est obligé de travailler sur le poids et les performances du site pour qu'il puisse être utilisable. On doit également se poser des questions difficiles sur les contenus qui sont importants à mettre en valeur et ceux qui le sont moins.

Une fois qu'on a une application qui fonctionne bien dans ces conditions extrêmes, elle sera forcément confortable à utiliser sur des machines plus puissantes, des écrans plus grands, des réseaux plus performants.

(6) https://collectif.greenit.fr/ecoconception-web/

Dans la pratique on peut tester avec des terminaux réels, des téléphones anciens conservés ou achetés d'occasion par exemple. On peut aussi utiliser des machines virtuelles, comme les simulateurs fournis pour les développements natif sur Android ou iOS.
Plus pratique, mais moins complet, on peut aussi utiliser le menu dédié des outils de développement navigateur permettant de redimensionner la page à la taille d'un écran de téléphone mobile et de simuler une bande passante limitée : **Figure 4**

Bien choisir les technologies en fonction du contexte

Les technologies aujourd'hui à la mode dans le web sont souvent basées sur une architecture particulière appelée Single Page Application (SPA). On sert au client une seule page, ainsi que du code javascript qui construira à la volée toute l'application web à partir de requêtes AJAX, en modifiant le DOM. Cela se fait avec des frameworks javascript connus comme Angular, React ou Vue.js.
Cette architecture a des avantages mais n'est pas forcément adaptée à tous les projets. Pour une application qui sera utilisée exclusivement sur des ordinateurs de bureau, qui nécessitera beaucoup d'interactions, affichera du contenu riche, c'est certainement un bon choix. Pour un site qui est surtout composé de texte, et qui est destiné à être visualisé partout dans le monde, sur tout type d'appareils et de réseaux, cela se discute. Dans la pratique, la limite entre un site web et une application web n'est pas forcément aussi claire, mais trancher est un choix structurant. Une application sera toujours plus lourde qu'un site.
L'utilisation de javascript a un prix : il faut du temps pour charger les scripts, et les interactions avec l'utilisateur ne sont possibles qu'une fois le fichier lu par le navigateur et les scripts de départ exécutés. Il y a même des pages qui nous expliquent comment se passer de bibliothèques populaires comme moment.js[7], underscore.js[8], jquery[9], voir même de javascript tout court[10].

Le low tech web, le charme du minimalisme

Le terme low-tech désigne des inventions et technologies sobres en matériaux et en énergie, faciles à construire et à réparer pour n'importe qui, par exemple un four solaire ou un frigo en terre cuite. Le terme a été construit par opposition à la high-tech, il peut donc sembler étrange de les associer. Pour construire n'importe quel appareil high-tech, il faut des alliages complexes de métaux purs, des salles blanches et beaucoup de savoir et de rigueur.
Pourtant, on peut voir certaines technologies numériques comme low-tech : elles paraissent désuètes ou obsolètes, mais elles fonctionnent partout depuis longtemps, et nécessitent peu de ressources. C'est le cas par exemple du SMS : tous les téléphones, même ceux fabriqués dans les années 90 peuvent recevoir des SMS, et ce protocole est très efficace en termes d'utilisation de ressources numériques.

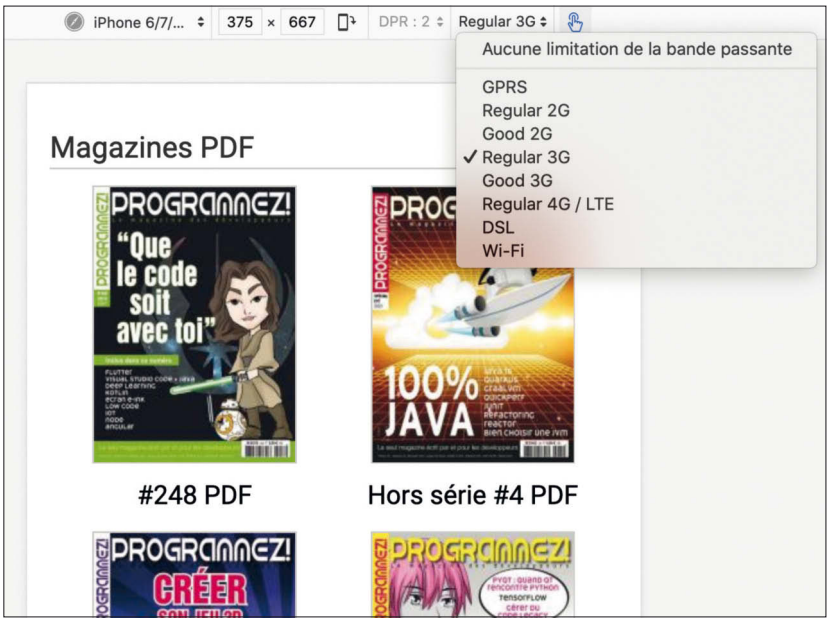


Figure 4

Le Low Tech Mag, magazine dédié au low-tech s'est posé la question de ce que pourrait être un site web low-tech[11]. Ils ont construit un site web statique, c'est à dire que les pages ne sont pas générées dynamiquement par le serveur à chaque requête comme le ferait des CMS comme Wordpress mais seulement une fois à la création ou mise à jour du site, et sont de simples fichiers html sur le disque du serveur. Ils ont choisi de ne pas inclure de police de caractère additionnelle, de ne pas afficher de logo, et d'optimiser agressivement leurs images avec une technique appelée *dithering*. De plus, le site n'a pas de tracking, pas de publicité, pas de cookies. Le résultat est une page minimaliste, plus sobre que beaucoup de pages web actuelles, mais où le contenu intéressant, l'article publié, est toujours parfaitement lisible. Et cela fonctionne sans latence sur tous les appareils.
Il y a d'autres exemples de sites internet minimalistes, qui le sont pour différentes raisons. On peut citer Craigslist[12], site de petites annonces américain, qui a gardé le minimalisme de ses premiers jours, à l'époque vital du fait des connections limitées, mais aussi Wikipédia[13], un des projets les plus cool de l'histoire du web et peut -être de l'humanité, dont l'interface a toujours été sobre et orientée sur le contenu, ou encore Hacker News[14], que beaucoup d'entre nous utilisent.
Figure 5

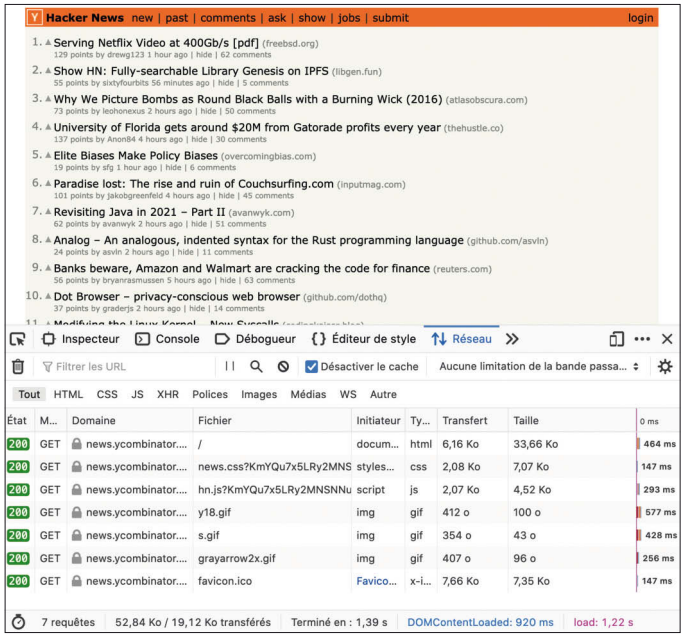
Offline first ?

Plus radical encore qu'un site minimaliste, on peut imaginer du contenu conçu pour être consommé d'abord hors ligne. L'utilisateur accède à une page, la télécharge, puis la lit, l'utilise une fois déconnecté, par exemple en situation de mobilité, dans un train, ou encore pendant un week-end de camping, comme un livre.
Ce modèle implique des pages plus grandes en termes de

(7) https://github.com/you-dont-need/You-Dont-Need-Momentjs
(8) https://github.com/you-dont-need/You-Dont-Need-Lodash-Underscore
(9) https://github.com/nefe/You-Dont-Need-jQuery
(10) https://github.com/you-dont-need/You-Dont-Need-JavaScript
(11) https://solar.lowtechmagazine.com/2018/09/how-to-build-a-lowtech-website/
(12) https://paris.craigslist.org/
(13) https://fr.wikipedia.org/
(14) https://news.ycombinator.com/

Figure 5

Hacker News - site de partage de liens centrés autour de la technologie peut être considéré comme minimaliste et low-tech avec ses 7 requêtes http et moins de 20ko transférés.



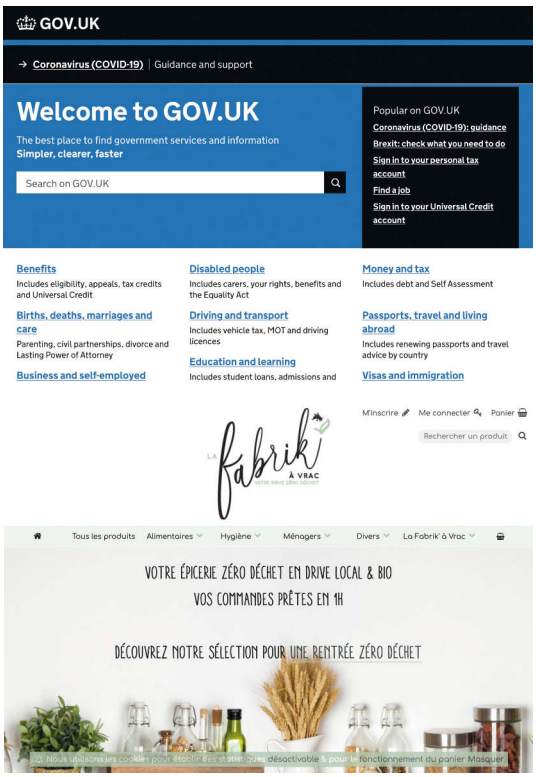


Figure 6

contenu, mais moins de pages et d'interactions, donc moins de requêtes, de charge serveur, et aussi une plus forte incitation à l'optimisation et à se limiter au contenu utile.

La nécessité d'un design adapté

Il est difficile, pour un développeur, de construire un site léger et performant, si ses collègues ont conçu une application web chargée d'un grand nombre d'images, d'animations, avec une expérience utilisateur laborieuse impliquant beaucoup d'écrans, avec beaucoup de fonctionnalités dont certaines sont peu utiles. Même très optimisée, une application web trop grosse sera toujours trop grosse.

L'écoconception web est d'abord la conception d'un service rendu à des utilisateurs via un site web, et ne s'arrête pas à appliquer des bonnes pratiques de performances web. Elle implique toute l'équipe : les designers qui vont penser les écrans et leurs enchaînements, les graphistes qui vont réaliser les maquettes et surtout les décideurs qui choisiront les fonctionnalités, lesquelles inclure, leur nature, les modes d'interactions.

N'hésitons pas à parler avec le reste de notre équipe de ces problématiques : l'impact du numérique sur le monde, l'intérêt de faire léger et simple.

Exemples de sites éco conçus

Avec la popularité croissante de l'écoconception web depuis quelques années, des développeurs et designers ont déjà construit des sites explicitement éco conçus. On y trouve le site du Low Tech Magazine déjà évoqué,ou encore le portail des services du gouvernement du royaume uni, https://www.gov.uk/ , volontairement minimaliste, mais aussi des sites plus esthétiques, où rien ne laisse penser qu'ils sont low tech, comme celui de la fabrik à vrac, épicerie bio. **Figure 6**

On peut donc parfaitement construire un site léger, beau et très fréquenté.

Conclusion : de l'écoconception web vous en faites déjà, et vous pouvez en faire plus !

Vous vous êtes peut-être fait la réflexion en lisant cet article que l'on énumère en fait beaucoup de bonnes pratiques que vous connaissez déjà. L'écoconception web n'est pas quelque chose de radicalement nouveau, mais une autre façon de voir et de promouvoir un web léger, largement compatible et accessible, centré sur l'utilisateur et ses besoins.

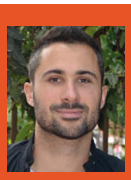
En fait **les bonnes pratiques convergent** :

- En construisant des applications plus légères, on fait des économies de ressources serveur et réseaux et donc des **économies d'argent**.
- Les **utilisateurs** seront **plus satisfaits** d'utiliser un site léger et performant, simple et ergonomique, et reviendrons volontiers sur le service.
- En se focalisant sur le contenu, en priorisant le texte, en faisant plus simple, on améliore l'**accessibilité**.
- La mise en avant du texte est aussi bonne pour le **SEO** (Search Engine Optimisation, l'optimisation de la position du site dans les moteurs de recherche). La performance du site fait également partie des aspects pris en compte par les moteurs pour choisir les sites qu'ils vont mettre en avant.

Toutes ces pratiques permettent aussi de **réduire les fractures numériques**.

En effet, tout le monde n'a pas accès au web dans les mêmes conditions, pour différentes raisons. Cela peut-être à cause de la géographie : il y a des endroits en France et dans le monde où les réseaux, mobiles ou fixes, ne sont pas très bons. Cela peut-être aussi une question de génération, les gens plus âgés étant souvent moins à l'aise avec la technologie, ou bien une question sociale, avec des personnes qui ont du mal avec des tâches complexes comme déclarer ses impôts et pour qui cela sera encore plus difficile en ligne. Typiquement, les gens peu à l'aise avec le numérique ne vont pas se procurer des appareils dernier cri, qui représentent un certain investissement. Donc pour eux ce sera encore plus lent. En construisant des sites plus légers et ergonomiques, nous contribuons à réduire ces fractures.

Mesurer la consommation d'énergie d'une application en développement avec Scaphandre



Benoit Petit

bpetit@hubblo.org

J'ai travaillé comme Ingénieur systèmes et cloud avant de créer Hubblo : pour aider les entreprises à mesurer, comprendre et réduire l'impact de leurs services numériques sur le climat. Hubblo est à la fois un éditeur de logiciels libres d'aide à la mesure d'impact et un intégrateur. Je suis membre de Boavizta, un groupe de professionnels qui œuvrent pour des méthodes et des données ouvertes pour aider à la mesure de l'impact des services IT sur l'environnement. Je suis également formateur pour la sobriété numérique des infrastructures chez The Green Compagnon.

Les conséquences du réchauffement climatique sont de plus en plus perceptibles (inondations en Allemagne, records de chaleurs…). Plusieurs secteurs de l'économie entament leur introspection afin de réduire leurs émissions de gaz à effet de serre. L'IT ne fait pas exception, ce qui est heureux. On s'entend assez facilement sur le fait qu'il faut mesurer son impact et le comprendre pour pouvoir le réduire. Mais comment mesurer l'impact de ses applications et services, quand on est impliqué dans une équipe de développement ? Comment être plus sobre dans nos métiers ? Nous allons vous donner des éléments de réponses avec Scaphandre.

Scaphandre est un agent de monitoring dédié aux métriques de consommation d'énergie. Il permet de mesurer la consommation d'électricité d'un serveur et des services qui y sont hébergés. La consommation de chaque processus système est mesurée, ce qui permet d'identifier des axes d'amélioration au niveau applicatif.

L'outil fonctionne pour le moment sur les plateformes suivantes :

- les serveurs bare metal sous GNU/Linux (le test pour cet article est lancé sur un laptop sous GNU/Linux)
- des machines virtuelles lorsque l'agent est également installé sur l'hyperviseur (Qemu/KVM pour le moment)
- Docker/Kubernetes

Figure 1

L'outil, sous licence Apache 2.0, est extensible. Son architecture permet d'y ajouter des modules pour s'adapter à de nouvelles solutions de monitoring ou de traitement de données. La philosophie de l'outil est de s'adapter à l'existant et non l'inverse. On peut aujourd'hui associer scaphandre à une base de données orientée temps de type Prometheus ou Warp10, ou une solution de stream comme Riemann. Il est également possible de récupérer les données dans des fichiers JSON pour un post-traitement ou pour utiliser les données de Scaphandre pour une application avancée (qualscan en est un exemple en permettant de mesurer la consommation d'énergie de l'installation des dépendances d'un projet NodeJS). **Figure 2**

Code : https://github.com/hubblo-org/scaphandre

Documentation : https://hubblo-org.github.io/scaphandre-documentation/

Site web d'Hubblo : https://hubblo.org

Projet Vegeta : https://github.com/tsenart/vegeta

Le cas d'usage

Prenons l'exemple d'une API qui est sollicitée par un grand nombre d'utilisateurs sur des périodes clés. Le code serveur de cette API sera vraisemblablement distribué sur plusieurs machines, voire dans des conteneurs répartis sur plusieurs machines à l'aide d'un scheduler, de manière à pouvoir déployer ou retirer des conteneurs et éteindre ou allumer des machines en fonction des pics de charge. On peut donc absorber des pics de charge et limiter la consommation de ressource lorsque les besoins sont moindres (ce qui est une excellente pratique). Cependant, la consommation de ressources peut être importante lors d'un pic de charge, ce qui n'est ni bon pour le porte-monnaie (surtout dans le cloud) ni pour la planète.

Quoi de mieux pour s'assurer que l'impact de l'application lors des pics de charge ne sera pas de plus en plus important de version en version, que de mesurer sa consommation d'énergie pendant un test de charge, avant le déploiement ?

Outillage

Nous allons prendre le cas d'une application Python/Flask (très basique), que nous allons solliciter à l'aide de l'outil de test de charge Vegeta. Nous mesurerons les effets de ces tirs de charge avec Scaphandre, stockerons les données dans Prometheus et afficherons les données avec Grafana pour

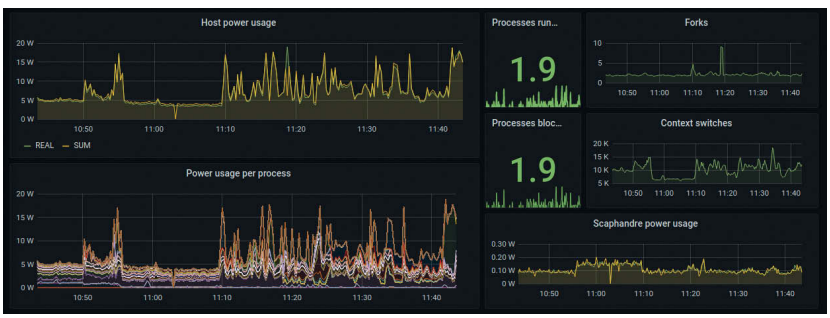


Figure 1

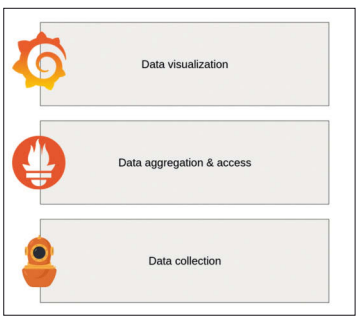


Figure 2

analyse. Pour une plus grande simplicité nous utiliserons docker-compose pour piloter ces outils et jouer le scénario.

Le setup

Voici à quoi ressemble le dossier courant :

```
├── app
│   ├── app.py
│   └── Dockerfile
├── docker-compose.yaml
├── grafana
│   ├── dashboards.yml
│   ├── datasource.yml
│   ├── Dockerfile
│   └── sample-dashboard.json
├── prom
│   ├── Dockerfile
│   └── prometheus.yml
├── results.bin
└── vegeta
        └── Dockerfile
```

Le dossier app contient l'application app.py

```
# save this as app.py
from flask import Flask

app = Flask(__name__)

@app.route("/")
def fibo():
    response = "\n la suite fibonacci est : "
    response += str(fibonacci(30))
    return response

def fibonacci(n):
    if(n <= 1):
    return n
    else:
    return (fibonacci(n-1) + fibonacci(n-2))
    response = ""
    for i in range(n):
    response += str(fibonacci(i))
    return response+", "
```

et un Dockerfile, que voici :

```
FROM python

RUN pip3 install -U Flask

COPY . .

CMD ["flask", "run", "-h", "0.0.0.0"]
```

On affiche donc la suite de Fibonacci à chaque nouvelle requête, jusqu'à 30. Rien de plus simple, on se contente d'une application web sans connexion à une base de données ou autre (mais l'exercice peut très bien se répéter avec quelque chose de plus complexe). Le dossier grafana contient un Dockerfile et des configurations de base pour installer le dashboard par défaut et se connecter à la datasource Prometheus.

Dashboards.yaml :

```
  apiVersion: 1

providers:
 # <string> an unique provider name. Required
 - name: 'Scaphandre examples'
   # <int> Org id. Default to 1
   orgId: 1
   # <string> name of the dashboard folder.
   folder: 'scaphandre'
   # <string> provider type. Default to 'file'
   type: file
   # <bool> disable dashboard deletion
   disableDeletion: true
   # <bool> allow updating provisioned dashboards from the UI
   allowUiUpdates: false
   options:
   # <string, required> path to dashboard files on disk. Required when
 using the 'file' type
   path: /var/lib/grafana/dashboards
   # <bool> use folder names from filesystem to create folders in Grafana
   foldersFromFilesStructure: true
```

Datasource.yaml

```
# config file version
apiVersion: 1


datasources:
 # <string, required> name of the datasource. Required
- name: Prometheus-scaph
 # <string, required> datasource type. Required
 type: prometheus
 # <string, required> access mode. direct or proxy. Required
 access: proxy
 # <int> org id. will default to orgId 1 if not specified
 orgId: 1
 # <string> url
 url: http://prometheus:9090
 isDefault: true
 version: 1
 # <bool> allows users to edit datasources from the UI.
 editable: true
```

Dockerfile :

```
FROM grafana/grafana


COPY /datasource.yml /etc/grafana/provisioning/datasources/
COPY /dashboards.yml /etc/grafana/provisioning/dashboards/
COPY /sample-dashboard.json /var/lib/grafana/dashboards/
```

Le dossier prom contient la configuration prometheus, prometheus.yml :

```
# my global config
global:
 scrape_interval:     10s # Set the scrape interval to every 15 seconds.
 Default is every 1 minute.
 evaluation_interval: 15s # Evaluate rules every 15 seconds. The default
```

```
is every 1 minute.
 # scrape_timeout is set to the global default (10s).

 # Attach these labels to any time series or alerts when communicating with
 # external systems (federation, remote storage, Alertmanager).
 external_labels:
    monitor: 'scaphandre-monitor'

# Load rules once and periodically evaluate them according to the global
'evaluation_interval'.
rule_files:
 # - "first.rules"
 # - "second.rules"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
 # The job name is added as a label `job=<job_name>` to any timeseries
 scraped from this config.
 - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
    - targets: ['localhost:9090']

 - job_name: 'scaphandre'
    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
    - targets: ['scaphandre:8080']
```

Nous voyons à la dernière ligne de la configuration que scaphandre fournit les données à prometheus en pull mode (c'est prometheus qui vient "scraper" les données pour les stocker en base), soit le mode classique. Dans le langage Prometheus, scaphandre est un exporter.
Le dossier prom contient aussi le Dockerfile suivant :

```
FROM prom/prometheus


COPY /prometheus.yml /etc/prometheus/prometheus.yml
```

Pour finir, voici le docker-compose qui nous permet de mettre tout ce petit monde à l'unisson :

Code complet sur programmez.com & github

Attardons-nous sur la ligne de commande du conteneur vegeta : /bin/bash -c 'sleep 10 && echo "GET http://app:5000/" | vegeta attack -duration=200s -rate=500 | tee results.bin | vegeta report'
On demande à "vegeta" de jouer la requête GET pendant 200 secondes, à un rythme de 500 requêtes par seconde, puis de nous afficher le résultat du test. La ligne de commande du conteneur scaphandre (["prometheus", "--containers"]) ressemblerait à celle-ci si l'on lançait scaphandre directement dans le terminal : scaphandre prometheus --containers
Le premier verbe est l'exporter choisi, soit le connecteur spécifique à la solution de stockage et d'analyse de données que l'on souhaite utiliser, ici prometheus (Une section dédiée aux *sensors* et aux *exporters* scaphandre est présente dans la documentation).
L'option --containers indique à scaphandre que l'on veut surveiller les conteneurs présents sur la machine et appliquer des labels en rapport sur les métriques de consommation d'énergie. Nous y reviendrons. Comme nous l'avons évoqué plus haut, scaphandre peut adopter un comportement différent en fonction de la base de données de série temporelle ou de la solution de monitoring que l'on adresse. Ici, on utilise l'exporter prometheus. Tous les conteneurs peuvent discuter à travers le réseau "lab-network". Tous les fichiers pour ce test sont disponibles sur github : https://github.com/hubblo-org/publications/tree/master/article-programmez-09-2021

Premier test

Grâce au docker-compose, nous allons pouvoir jouer le scénario de test très simplement :

```
docker-compose up -d
```

Une fois lancé nous devrions avoir les conteneurs suivants :

```
CONTAINER ID IMAGE                                  COMMAND
CREATED         STATUS              PORTS               NAMES
0de3ec310881 article-programmez-09-2021_vegeta
"/bin/bash -c 'sleep…"  2 seconds ago     Up Less than a second
                article-programmez-09-2021_vegeta_1
3afaa56604dc article-programmez-09-2021_app            "flask run -h
 0.0.0.0"   About an hour ago   Up About an hour      0.0.0.0:5000-
>5000/tcp   article-programmez-09-2021_app_1
71ada46d0bcf article-programmez-09-2021_prometheus
"/bin/prometheus --c…"  2 hours ago      Up 2 hours
0.0.0.0:9090->9090/tcp   article-programmez-09-2021_prometheus_1
57573fa6e388   hubblo/scaphandre:build-PR_84-docker-labels-9
 "/usr/local/bin/scap…"  2 hours ago      Up 2 hours
0.0.0.0:8080->8080/tcp   article-programmez-09-2021_scaphandre_1
1cf7a95368e2 article-programmez-09-2021_grafana          "/run.sh"
          8 hours ago      Up 8 hours          0.0.0.0:3000->3000/tcp
 article-programmez-09-2021_grafana_1
```

Grafana est alors accessible sur http://localhost:3000. Les identifiants sont admin / secret. **Figure 3**
Le premier graphe en haut à gauche représente la consommation d'énergie instantanée de puissance de la machine (en Watts), celui à sa droite représente la consommation d'énergie sur la période de temps sélectionnée (en Watts-heure). Le graphe du dessous est la consommation par socket CPU.

Figure 3

Puisque je n'ai qu'une socket CPU sur la machine qui me permet de lancer le test, c'est exactement le même graphe qu'au-dessus. Le dernier étage donne la consommation d'énergie par processus, avec à gauche, un top des processus les plus gourmands en énergie en ce moment et à droite l'historique de consommation de puissance par processus, sur la fenêtre de temps. Lorsque l'on arrive la première fois sur le dashboard, le graphe en bas à droite donne la consommation de tous les processus collectés. On peut alors filtrer pour obtenir uniquement la consommation de notre application en saisissant un mot clef dans la case "process_filter" en haut. En saisissant "flask", on va avoir la consommation des processus qui contiennent "flask" dans leur ligne de commande et ainsi isoler la consommation de notre application durant le test.
Une fois le test terminé on peut sélectionner la fenêtre de temps exact du test en cliquant et en faisant glisser le curseur au niveau du graphe de consommation. **Figure 4**
Grafana nous donne gentiment quelques stats sur cette fenêtre. L'application a consommé au maximum 5,16W et en moyenne 3,01W. Ces chiffres permettent un premier exemple de comparaison avec des versions ultérieures de l'application.

Mise à jour de l'application et second test

Vous aurez certainement reconnu dans le code de notre application une recursion :

```
# save this as app.py
from flask import Flask


app = Flask(__name__)

@app.route("/")
def fibo():
    response = "\n la suite fibonacci est : "
    response += str(fibonacci(30))
    return response

def fibonacci(n):
    if(n <= 1):
    return n
    else:
    return (fibonacci(n-1) + fibonacci(n-2))
    response = ""
    for i in range(n):
    response += str(fibonacci(i))
    return response+", "
```

Figure 4

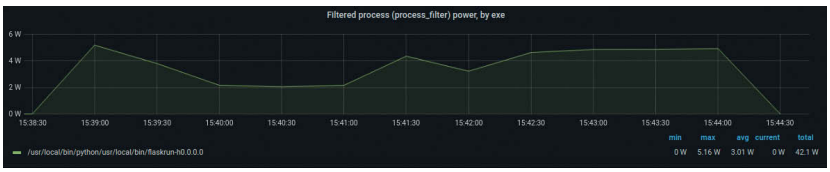
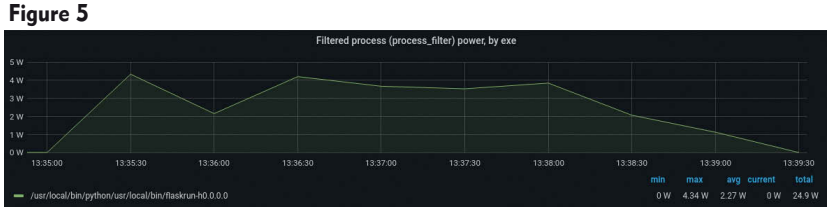


Figure 5



Voyons ce que change en termes de consommation d'énergie le fait de passer à une boucle simple :
Code complet sur programmez.com & github
Une fois le code remplacé, on peut reconstruire l'image du conteneur de l'app :

```
docker-compose build app
```

Puis relancer le test :

```
docker-compose up -d
```

Et constater l'effet de ce nouveau tir de charge : **Figure 5**
On relève maintenant comme maximum 4,34W et en valeur moyenne 2,27W. Cette version du code est donc *à priori* moins énergivore que la précédente. Notons tout de même que ce test est extrêmement rudimentaire et qu'il faudrait valider cette affirmation avec des scénarios de tir de charge différents (et valider ça avec un serveur d'application bon pour de la production, comme gunicorn).

Relever la consommation de conteneurs appartenant à un même service

Nous avons vu dans cet article l'option --containers. Voyons dans la pratique à quoi elle sert.
Lorsque notre application passera en production, nous aurons certainement un nombre important de conteneurs, derrière un load balancer, pour tenir la charge. Si l'on souhaite mesurer la consommation d'énergie de l'application dans ce contexte, il faudra accéder à cette information pour chaque conteneur participant au service. C'est là que l'option est utile, Scaphandre va alors appliquer des labels spécifiques aux métriques de consommation de chaque processus, lorsque celui-ci est conteneurisé. Voici à quoi ressemble la métrique de consommation de notre application vu par prometheus (et visible sur http://localhost:8080/metrics dans notre exemple) :

```
scaph_process_power_consumption_microwatts{container_docker_version="20.10.2",container_label_com_docker_compose_service="app",container_label_com_docker_compose_config_hash="a2cc6462a0ba2ab59f230d5ea16fe54c38666124dfc27f56f5789f0cf314a2bf",container_names="article-programmez-09-2021_app_1",container_label_com_docker_compose_oneoff="False",container_id="27f3159ee70707d74f5da09468c1076abba719ca3b97bb1f609071848569df9",container_label_com_docker_compose_version="1.25.0",exe="flask",container_scheduler="docker",pid="851283",container_label_com_docker_compose_container_number="1",container_label_com_docker_compose_project_working_dir="/home/bpetit/git/hubblo/formations/article-programmez-09-2021",container_label_com_docker_compose_project_config_files="docker-compose.yaml",container_label_com_docker_compose_project="article-programmez-09-2021",cmdline="/usr/local/bin/python/usr/local/bin/flaskrun-h0.0.0.0"} 0
```

Les éléments en gras sont les labels appliqués par Scaphandre à la métrique de manière à simplifier son exploitation dans un contexte plus complexe, comprenant par exemple plus de conteneurs pour le même service. **Container_label_com_docker_compose_service** nous permet de filtrer les données de cette série temporelle (scaph_process_power_consumption_microwatts, donc la consommation d'un processus) par nom de service docker-compose. Avec ce label on peut récupérer dans le même résultat la consommation de tous les conteneurs qui appartiennent à un service, ici "app".
Container_scheduler nous montre que Scaphandre a bien détec-

té le démon docker et donc est capable d'aller chercher des informations supplémentaires sur chaque processus associé à un conteneur. Kubernetes devrait être également supporté à la fin de l'été 2021. Ajoutons un panel dans le dashboard grafana pour illustrer, soit en cliquant sur le bouton "Add Panel", soit en dupliquant le panel qui montre la consommation de flask, avec "Duplicate" : **Figure 6**

Une fois fait, il faut changer la formule utilisée avec Edit, sur le nouveau panel. **Figure 7**

Notre requête est constituée de :

- scaph_process_power_consumption_microwatts : le nom de la métrique qui nous intéresse
- container_label_com_docker_compose_service : le label qui nous permet d'identifier les données qui nous intéressent, ici ce sont celles avec la valeur "app"

Notons que l'on divise le résultat par 1 000 000, car la métrique nous retourne des microwatts. Une valeur en microwatts n'est pas très explicite pour ma part, je préfère lire les données en watts pour mieux me représenter les choses.

Conclusion

Cet article est une première approche de l'utilisation de Scaphandre, pour mesurer la consommation d'énergie d'un service. Il est possible de décliner l'usage de cet outil à de nombreux autres cas. Les seuls pré-requis au moment où j'écris ces lignes, sont d' tre sur une machine physique, sous GNU/Linux, ou bien sur une machine virtuelle, si Scaphandre est également installé sur l'hyperviseur et que l'on utilise l'option --vm (voir la documentation).

Notre exemple est imparfait, car on lance les tirs de charge depuis la même machine que celle qui fait tourner l'application. Pour obtenir des métriques plus "propres", il serait mieux d'utiliser deux machines différentes. Il serait également intéressant d'intégrer cette suite dans la chaîne de CI, pour garantir un suivi de l'appétit du service en énergie et ça de version en version, systématiquement, pour chaque contribution de l'équipe de développement.

Enfin il faut prendre un peu de recul pour préciser que ces métriques de consommation d'énergie ne sont qu'une partie des informations nécessaires pour connaître l'impact d'un service numérique sur le climat. C'est certainement la partie la plus accessible pour un profil technique en termes de responsabilités, mais une analyse d'impact complète comprend d'autres éléments qui parfois pèsent encore plus lourd dans le bilan carbone. La consommation d'électricité est indirectement responsable d'émissions de Gaz à Effet de Serre (GES), car c'est une énergie dite secondaire, ou finale, par opposition aux énergies primaires (charbon, vent, rayons du soleil, pétrole, éléments radioactifs…) qui sont consommées pour la fabriquer. On peut effectuer la conversion en émissions de GES, grâce aux données présentées sur un service comme ElectricityMap. Ces émissions sont classées dans le "scope 2" des émissions, celui des émissions indirectes.

Le "scope 1" représente les émissions directes. Dans le cas d'un service IT, peu de choses sont classées dans le scope 1. Une exception, dans le cas d'un exploitant de datacenter, serait le fioul brûlé lors d'une coupure générale d'électricité malgré la redondance des arrivées électriques (ou lorsque ce fioul est considéré périmé et qu'il est brûlé pour être remplacé…).

Le troisième et dernier scope, le "scope 3", représente les autres émissions, notamment celles qui sont liées à la fabrication des équipements, à leur transport et à leur fin de vie. Les informations nécessaires pour calculer ce dernier scope sont souvent plus compliquées à obtenir et même parfois délibérément écartées (ce qui arrange bien les entreprises qui affirment être "neutre en carbone").

Mesurer la consommation d'énergie est donc une bonne chose pour un service plutôt gourmand sur sa phase de run (web à haute volumétrie, machine learning, recherche, média, etc.). C'est un indicateur qui permet d'orienter ses efforts de sobriété dans le développement d'une application ou l'évolution d'une infrastructure IT. Il reste cependant indispensable, si l'on souhaite réduire efficacement et durablement les émissions d'une entreprise, d'adopter une démarche plus globale et de prendre en compte le scope 3 des émissions dans la stratégie et dans les calculs.

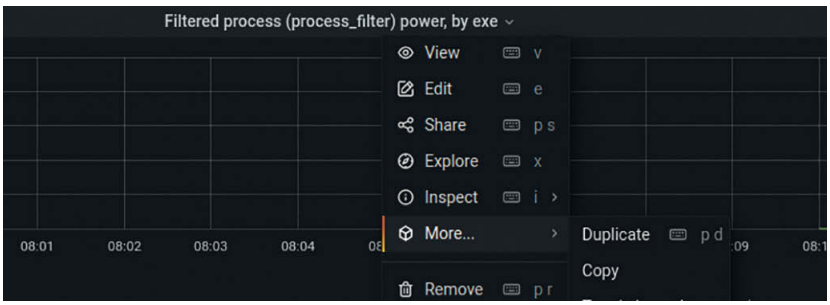


Figure 6

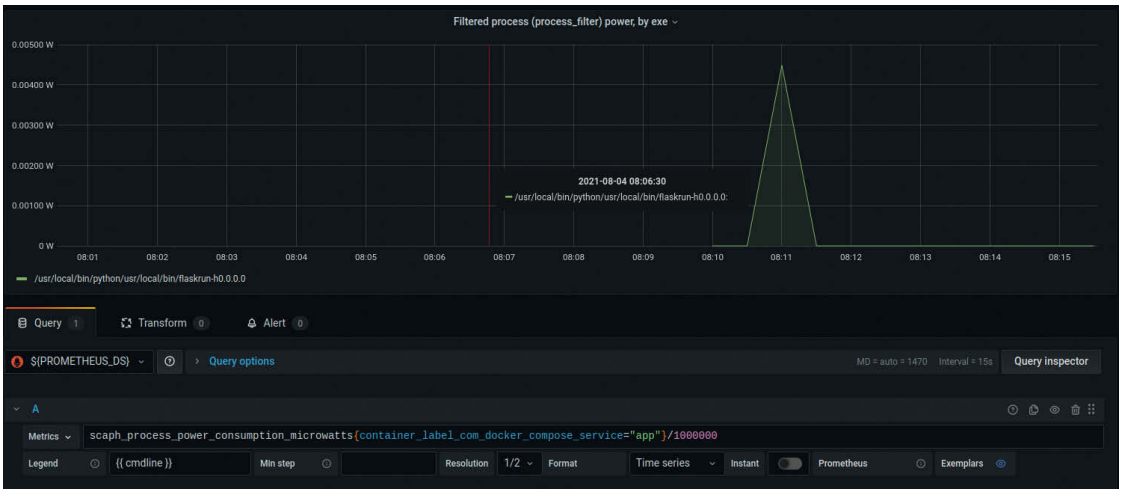


Figure 7

L'éco-conception au service d'un numérique plus responsable

L'objectif de cet article est de présenter les principaux enjeux et leviers théoriques d'une démarche d'éco-conception technique, que nous illustrerons avec un retour d'expérience concret mené chez Meetic. Nous allons faire un focus particulier sur les apps mobiles.

Vincent FRATTAROLI
CEO, Inside|app

François RÉZENTHEL
Directeur de production, Meetic

L'éco-conception technique consiste à concevoir son architecture technique et produire des codes les moins consommateurs d'énergie possibles, ce sur l'ensemble de la chaîne technique : Serveurs & datacenter - Réseau Télécom - Terminal de l'utilisateur. Pour des développeurs frontend (web / app), il s'agit de jouer sur trois leviers :

1 Optimisation de la chaîne complète

L'optimisation des échanges clients / serveurs va concerner aussi bien les applications web et les applications mobiles. Elle sera de plus bénéfique pour toute la chaîne technique (serveur - réseau - terminal).

Réduire les volumes de données et le nombre de requêtes échangées entre les clients et les serveurs est la principale source de réduction de l'empreinte carbone d'un service numérique.

Règle #1 : une conception des web services Front et Back

Une application sobre requiert une conception des web services par les Front et Back : découpage des appels, chargement des données utiles uniquement, règles de cache : la vision "Front" est primordiale pour optimiser l'API.

Règle #2 : Ne pas négliger les briques tierces

Dans le cas de Meetic, nous avons constaté que les partenaires tiers (publicité, tracking) généraient parfois plus de 30% des volumes de données. Il faut ainsi intégrer cette dimension de sobriété dans le choix d'un éditeur et prendre soin de l'intégration des leurs SDK/API.

Les autres actions possibles pour minimiser les échanges serveur - terminal

- utiliser graphQL qui permet de limiter les échanges client-serveurs couplé à une bonne gestion du cache HTTP
- utiliser des formats d'images ou vidéos optimisés pour chaque OS/plateforme (par exemple HEIC pour les iPhone, deux fois plus performants que JPEG)
- si votre app doit régulièrement télécharger des contenus volumineux, il est préférable que ceux-ci soient téléchargés lorsque le smartphone est connecté en WiFi et non 3/4/5 G (les réseaux mobiles sont plus énergivores que les réseaux fixes/Wifi)

2 Minimiser la consommation de CPU/mémoire sur le terminal

Cet aspect est important pour les smartphones : au-delà de la consommation d'électricité pour recharger une batterie trop sollicitée par une app, on va réduire l'espérance de vie de la batterie. Or la batterie est dans 40% des cas le premier motif pour changer de smartphone.

Règle #3 : limiter le fonctionnement en background

Le fonctionnement en arrière-plan peut présenter des bénéfices utilisateurs forts, mais aussi générer une surconsommation inutile. L'utilisation du GPS doit être limitée au strict nécessaire.

Règle #4 : utiliser de préférence les API et composants natifs de l'OS

Apple et Google font évoluer les API et composants UI fournis aux développeurs. ils sont moins consommateurs de CPU et mémoire que les librairies tierces ou composants développés spécifiquement.

Règle #5 : favoriser le dark mode

Il permet de réduire la consommation de batterie 5 à de 42% selon la luminosité paramétrée.

Règle #6 : préférez le développement natif

Les technologies Cross-Platform mobiles React Native, Flutter sont plus consommatrices d'énergies (surtout Cordova et React Native) que les développements natifs Kotlin et Swift. Sur ce sujet, vous pouvez lire ce post très inéressant : https://medium.com/swlh/flutter-vs-react-native-vs-native-deep-performance-comparison-990b90c11433

3 Réduire la taille du package des Applications

Les Apps mobiles (et dans une moindre mesure les Web App) fonctionnent avec des mises à jour régulières, qui génèrent des volumes de données sur le réseau.

Là encore il existe plusieurs leviers pour 1) réduire au maximum la taille des Apps et 2) réduire le volume des données téléchargées lors des mises à jour.

Règles #7 - Optimiser le packaging de son Application

Le téléchargement des mises à jour se fait en différentiel d'une version à l'autre. En organisant bien son App et en ne modifiant que les fichiers et assets graphiques nécessaires, on réduit sensiblement le volume téléchargé lors d'une mise à jour.

Règles #8 - Limiter les dépendances tierces

Les dépendances concentrent une part importante du poids d'une App. Il faut se demander si l'on tire pleinement partie de chaque dépendance ou si, tout simplement, on en a vraiment besoin. Exemple sur iOS : si votre application effectue peu d'appels réseaux GET, il n'est probablement pas utile d'intégrer une librairie réseau, iOS fournissant suffisamment de solutions.

Règles #9 - Utiliser le bitcode sur iOS

Le bitcode est une représentation intermédiaire d'un programme compilé. C'est un format à mi-chemin entre le code source et le code machine. Les applications iOS qui activent le bitcode vont être recompilées par l'App Store, une fois l'upload effectué.

À l'instar du **slicing,** qui va supprimer les assets qui ne sont pas destinés à notre appareil, le bitcode va permettre de supprimer le code compilé qui n'est pas utile pour l'architecture de notre appareil.

On peut ainsi espérer gagner jusqu'à 50%, sur la taille du binaire.

Rex : les services Meetic

Dans une démarche responsable entamée depuis plusieurs années, Meetic, le leader de la rencontre en ligne, s'est engagé à réduire son empreinte carbone de 10% tous les ans. Un plan d'actions est déjà lancé pour réduire l'impact écologique de son infrastructure informatique. Meetic souhaitait aller plus loin, en évaluant l'impact global de ses services web et apps sur son bilan carbone et optimiser celui-ci. **Meetic, accompagnée d'inside|app, a donc lancé un plan en quatre phases pour mesurer et optimiser le bilan carbone de ses services année après année.**

Phase 1 : Compter et évaluer

Nous avons tout d'abord établi le bilan carbone des services Meetic (site web, site web mobile, app iOS, app Android) en se basant sur les données d'usages.

Schématiquement, nous avons calculé l'ensemble des Go générés par les utilisateurs sur toute la chaîne (du terminal au serveur) et donné un équivalent en tonnes de CO_2 à ce trafic. **Nous avons ainsi constaté que 90% des impacts étaient liés sur la partie "Sessions utilisateurs", 10% sur la partie téléchargement des applications et mises à jour. Nos efforts se sont donc concentrés sur la partie "Sessions".**

Phase 2 : Amélioration des requêtes

Meetic appliquait déjà la majorité des bonnes pratiques listées précédemment pour des enjeux de qualité de services et d'optimisation des coûts d'infrastructure. Nous avons donc voulu valider l'impact de ces bonnes pratiques, et corriger les éventuels défauts d'implémentation. Nous avons observé les requêtes et données transitant sur le réseau sur les quatre applications (2 Web et 2 mobiles) avec Charles Proxy, et cherché à comparer les appels réseaux et volumes de données transitant sur trois parcours types. Notre idée était simple :

- iOS et Android devaient présenter des volumes de données et requêtes très proches
- le web mobile devait être moins gourmand que le web desktop
- nous voulions détecter des éventuels problèmes de surconsommation, évaluer leurs impacts et traiter les plus importants

Ce que nous avons trouvé

Piste #1 : Keep-alive sur iOS

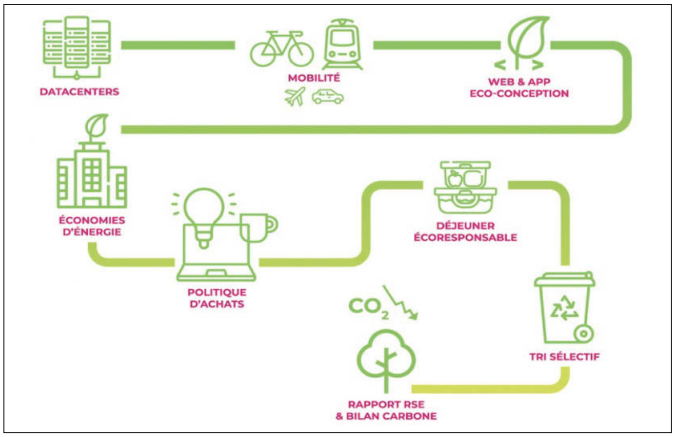
Le HTTP *keep-alive* (durée de la connexion avec le serveur) était trop court sur iOS, ce qui générait 1 Mo de surconsommation pour un type de session (overhead http). En augmentant le *keep-alive* sur iOS cette surconsommation était ainsi facilement supprimée.

Piste #2 : Optimiser les appels à des tiers

Le caching et l'intégration des services tiers (publicité et tracking) généraient trop de données, notamment sur le web mobile.

Piste #3 : Chargement en doublon

Sur une plateforme, une image haute résolution était chargée en doublon par erreur, ce qui augmentait de plus de 30% les données échangées sur un type de parcours. N'étant pas un bug en soi, cette erreur avait jusque là passé tous les tests qualité en interne. Ils ont depuis évolué afin de prendre en compte ce cas de figure.



Actions entreprises par Meetic afin de mesurer et améliorer le bilan carbone de l'entreprise

Piste #4 : Aligner certains appels WS

Nous avons constaté que deux applications utilisaient une API ancienne, plus verbeuse et génératrice d'appels que la version plus récente. En migrant sur la nouvelle API on pouvait ainsi réduire les volumes et faciliter la maintenance des applications.

Ces quatre premières pistes, pouvant être rapidement mises en œuvre, nous permettent une baisse de près de 20% de l'empreinte carbone de ces services.

Dix autres actions à plus ou moins long terme ont été identifiées et seront évaluées dans les mois à venir.

Phase 3 : Compensation carbone

Les collaborateurs volontaires ont pû participer en mai 2021 à une première action de reboisement en région parisienne, afin d'absorber une partie des émissions de carbone générées par l'activité de l'entreprise et de ses utilisateurs.

Phase 4 : Sensibilisation interne et plan à long terme

Une grande partie de notre travail a consisté à sensibiliser les équipes sur ces enjeux de sobriété numérique, via des présentations, un hackathon dédié sur le sujet et l'intégration de la sobriété numérique dans les indicateurs de la société et dans les processus projets.

Conclusion

En estimant l'impact de ses services sur toute la chaîne technique et pas uniquement sur ses serveurs et datacenter, Meetic s'engage dans une démarche volontariste et ambitieuse de réduction de son empreinte carbone.

Des premières actions permettront une baisse d'environ 20% de l'empreinte carbone des applications sur 2021. De plus, cette amélioration de l'impact environnemental se traduit souvent par une amélioration des performances applicatives, ce qui est un argument de poids pour convaincre les directions métiers et améliorer l'expérience perçue par les utilisateurs. Ces premiers résultats sont encourageants, et s'inscrivent dans un processus pluriannuel, sponsorisé par la direction de l'entreprise, avec notamment la préparation de la certification ISO14001, permettant de valider un processus d'amélioration continue dans le domaine de l'impact écologique.

Refactorer le Legacy intestable avec les Approval Tests

Sepehr Namdar

Développeur Techno-Agnostic chez Shodo

De la maintenance au suivi complet d'un projet, Sepehr a une forte expertise sur toutes les phases du développement logiciel. L'Agilité, le Pragmatic-Driven Design/Development et le Clean Code sont la clé de la réussite de la réalisation d'un produit logiciel.

Dans son livre, Working Effectively with Legacy Code, Michael Feathers définit un code legacy comme un code "non testé". Vous connaissez les conséquences que peut être avoir un code legacy. En tout bon crafter que vous êtes, vous souhaitez refactorer ce code legacy et lui appliquer une couche de tests.Sauf que vous êtes face à un code que vous ne connaissez pas, illisible, incompréhensible et vous n'avez même pas d'expert métier pour vous expliquer le domaine...

Les Approval Tests sont là pour vous aider ! Grâce à cette librairie multi-plate-forme, sécurisez la refonte de votre application grâce à un code couvert par des tests, plus fiable et plus sûr. La cerise sur le gâteau c'est que tout ça, c'est rapide, très facile à apprendre et à utiliser.

Refactorer, c'est changer la structure du code sans changer son comportement afin de l'améliorer et de le rendre plus performant et durable. Ceci peut passer par le fait de renommer un attribut, déplacer une ligne de code, extraire une méthode, etc.

Le Refactoring n'est pas une tâche dans votre "tableau de Scrum", mais une pratique de développement qu'il faut faire de manière systématique. Comme dit Uncle Bob, le Refactoring c'est comme se laver les mains. On le fait de manière récurrente sans le planifier ou sans y penser.

Refactorer un code n'a pas vraiment besoin d'outil en particulier, mais il y a des IDE qui nous proposent des outils afin de rendre cette tâche importante plus facile.

Mais refactorer le code nécessite un pré-requis qui est d'abord de le faire couvrir par des tests. Si on commence à refactorer un code sans le couvrir par les tests, on n'est pas à l'abri de le casser et de créer des bugs. C'est malheureusement une mauvaise pratique chez beaucoup de développeurs. Si vous voulez en savoir plus sur les différentes méthodes de Refactoring je vous conseille de lire le livre de Martin Fowler qui porte le même nom.

Écrire des tests avant de refactorer c'est bien beau, mais lorsqu'on ne comprend pas le code, c'est quasiment mission impossible. Pour atteindre cet objectif, on peut utiliser une technique qui s'appelle Golden Master.

Qu'est-ce que la technique du Golden Master ?

Le Golden Master consiste à envoyer des données en entrée (input) d'une méthode, peu importe sa taille, ses paramètres et ses dépendances, et de collecter les réponses qu'on reçoit (output).

Après avoir collecté assez de données (lorsque le code est totalement couvert par les tests), il est temps de prendre une image de ces résultats pour pouvoir commencer à "refactorer" le code.

Après chaque changement de structure dans le code, on relance les tests avec les mêmes inputs que nous avons collectés. Si les outputs sont identiques à l'image de départ, alors cela signifie que nos modifications n'ont rien changé à la logique de notre programme. Rien n'a été cassé dans notre code, on peut avancer !

Si le résultat diffère de l'image de départ, il faut alors revenir en arrière pour retomber sur des tests verts.

"Approval Tests" est un projet open source et une librairie qui facilite grandement la mise en place du Golden Master. Avec les Approval Tests vous pouvez générer plusieurs cas de tests afin de bien couvrir le code de production et éviter des mauvaises surprises et tout ça en une seule ligne de code. Allez, il est temps de passer aux choses concrètes, avec une démonstration !

Comment ajouter Approval Tests à notre projet ?

L'ajout d'Approval Test à votre projet est très simple, il suffit de choisir la plateforme qui vous intéresse parmi .Net, C++, Java, Lua, NodeJS, Objective-C, Perl ou Python en passant par un gestionnaire de dépendances ou en téléchargeant les packages nécessaires.

Par exemple pour un projet Java vous pouvez utiliser Maven ou Gradle pour ajouter la dépendance comme ci-après :

Maven

```
<dependency>
  <groupId>com.approvaltests</groupId>
  <artifactId>approvaltests</artifactId>
  <version>11.2.3</version>
</dependency>
```

Gradle

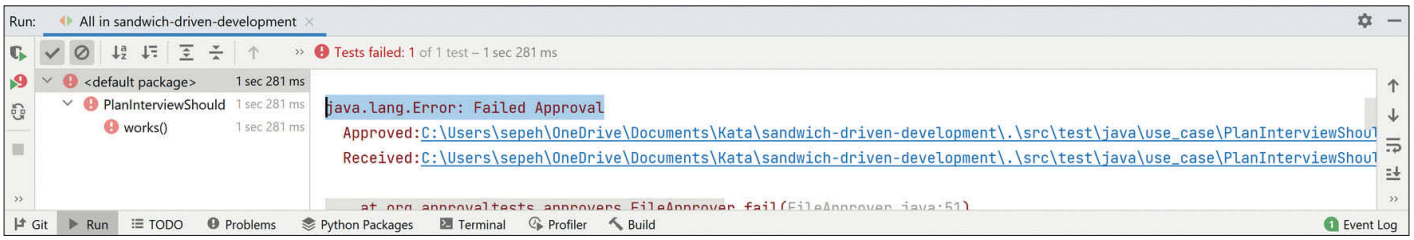
```
dependencies {
  testImplementation("com.approvaltests:approvaltests:11.2.3")
}
```

Mise en place du premier Approval Tests ?

Pour apprendre à utiliser Approval tests, essayons d'écrire un test classique en utilisant JUint et AssertJ :

```
@Test
void find_first_available_recruiter_who_can_test_candidate() {
  // Given
  var interview = new PlanInterview(
    candidates, recruiters, interviews);
```

Figure 1




```
    // When
    Interview plannedInterview = interview.plan("123", of(2021, 2, 21));

    // Then
    assertThat(plannedInterview).isEqualTo(expectedInterview());
}
```

La première étape sera pour nous de trouver les inputs et les séparer du code. C'est facile :

```
@Test
void find_first_available_recruiter_who_can_test_candidate() {
    String candidateId = "123";
    LocalDate interviewDate = of(2021, 2, 21);

    // Given
    var interview = new PlanInterview(
        candidates, recruiters, interviews);

    // When
    Interview plannedInterview =
            interview.plan(candidateId, interviewDate);

    // Then
    assertThat(plannedInterview).isEqualTo(expectedInterview());
}
```

Avec *Approval Tests* je n'ai pas besoin de la partie "Then". C'est lui qui fera ça pour moi. Je peux donc supprimer cette partie.
Ensuite, je vais extraire les parties "Given" et "When" pour créer une fonction à tester :

```
@Test
void find_first_available_recruiter_who_can_test_candidate() {
    String candidateId = "123";
    LocalDate interviewDate = of(2021, 2, 21);

    extracted(candidateId, interviewDate);
}

private Interview extracted(String candidateId, LocalDate interviewDate) {
    // Given
    var interview = new PlanInterview(
        candidates, recruiters, interviews);

    // When
    return interview.plan(candidateId, interviewDate);
}
```

Maintenant il suffit d'appeler la méthode "verifyAllCombinations" d'Approval Tests pour avoir l'équivalent du premier test classique que nous avons écrit plus haut.
Avant de faire ça, si on regarde la signature de cette méthode, on voit qu'elle prend comme attribut une fonction (extracted) et un ensemble d'inputs sous forme de tableau :

```
public static <IN1, IN2, OUT> void verifyAllCombinations(Function2<IN1, IN2, OUT>
call, IN1[] parameters1, IN2[] parameters2) {
    ...
}
```

Ce qui signifie que nous devons transformer nos inputs en tableau pour pouvoir les utiliser. Ce qui nous donne le code suivant :

```
@Test
void find_first_available_recruiter_who_can_test_candidate() {
    String[] candidateId = {"123"};
    LocalDate[] interviewDate = {of(2021, 2, 21)};

    verifyAllCombinations(this::extracted, candidateId, interviewDate);
}
```

Vous allez me dire " mais pourquoi se donner autant de mal pour écrire l'équivalent d'un test qui fonctionnait ? "
La réponse c'est qu'avec Approval Tests on peut lancer plein d'inputs sur un seul test. Je vous rappelle que nos inputs sont maintenant des tableaux de données et c'est exactement ce dont nous avons besoin pour utiliser la technique Golden Master.

Lancement d'Approval Tests :
Le première fois qu'on lance les Approvals Tests, c'est toujours rouge. C'est parce qu'on a besoin de lancer les tests au moins une fois pour récolter les outputs dont on a parlé pour le Golden Master.
En clair, pas d'image de comparaison = tests rouges
Figure 1
Si vous regardez dans l'arborescence de votre projet vous allez voir que 2 nouveaux fichiers ont été générés :
- Un fichier nommé "received"
- Un fichier nommé "approved"

Ce sont vos Golden Master : **Figure 2**
Maintenant il suffit de copier le contenu du fichier "received" dans le fichier "approved" pour passer les tests au vert. Vous pouvez aussi supprimer le fichier "approved" et renommer le fichier "received" pour "approved". **Figure 3**
Voyons voir à quel point ce premier test était efficace. Pour

cela, nous allons regarder la couverture des tests à l'aide d'IntelliJ. Il suffit de lancer les tests avec ce bouton :

Et voici le résultat : **Figure 4**

On peut constater que le seul test qu'on a écrit ne couvre pas toutes les lignes de notre code, car il faut plus d'inputs. Nous devons donc trouver plus de données. Pour ça, on peut demander l'aide d'un expert métier, ou regarder la base de données ou encore bien lire le code. Si ce n'est pas possible, essayez de mettre des données aléatoires qui vous semblent pertinentes par exemple "null", chaîne de caractère vide, des chiffres -1, 0, 1, etc.

Notre test va donc ressembler à ça :

```
@Test
void find_first_available_recruiter_who_can_test_candidate() {
  String[] candidateId = {"123", "456", "789", null, ""};
  LocalDate[] interviewDate = {
        of(2021, 2, 21),
        of(2021, 2, 20),
        of(2021, 2, 22),
        of(1300, 2, 21),
        of(3200, 2, 21),
        null
  };

  verifyAllCombinations(this::extracted, candidateId, interviewDate);
}
```

Lorsque vous lancez de nouveau les tests, ils apparaissent rouges. C'est tout à fait normal ! En effet, vous venez d'ajouter de nouveaux inputs.

On va alors copier les nouvelles données que nous avons reçues dans le fichier "received" et les coller dans "approved". Les tests passeront de nouveau au vert.

Regardons maintenant la couverture du code par nos nouveaux tests :

Parfait, le code est bien couvert par nos tests. On peut commencer à refactorer notre code !

Maintenant, regardons un peu ce qu'on a obtenu dans le fichier "approved" :

Code complet sur programmez.com & github

Trop fort cet outil ! Avec 1 test et 1 ligne de code j'ai obtenu 30 tests d'un coup. Cela correspond à la combinaison de toutes les données que je lui ai passée en input.

Prérequis de mise en place d'Approval Tests :

- Il est conseillé d'avoir l'outil *WinMerge* (ou un autre comme Kdiff) pour faciliter le passage entre les fichiers "received" et "approved" et aussi pour comparer la différence entre les données si jamais on fait une régression en faisant notre Refactoring : **Figure 5**
- Il faut implémenter la méthode *toString* dans tous les objets qui vont être testés par Approval Tests. Sinon on obtient des références à des objets et nos tests sont toujours faux :

Figure 2

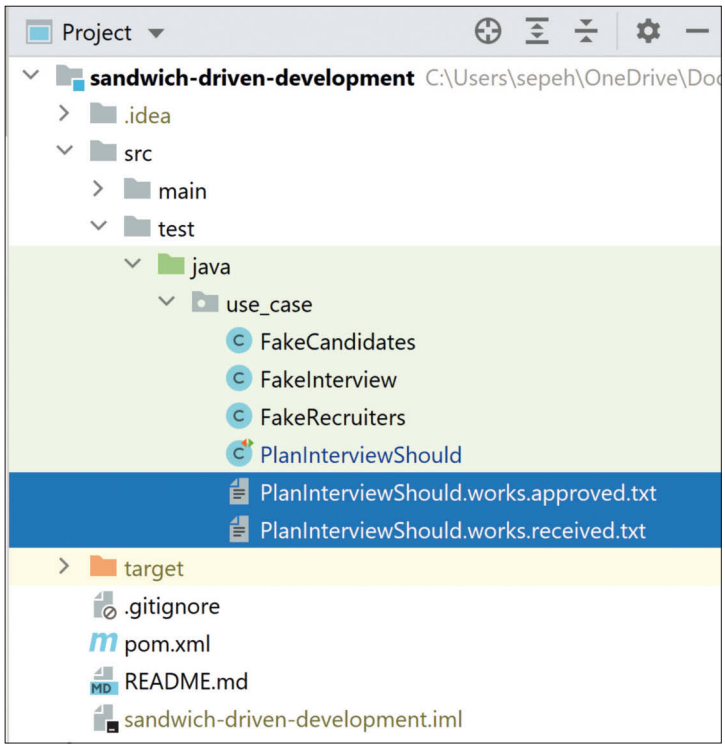
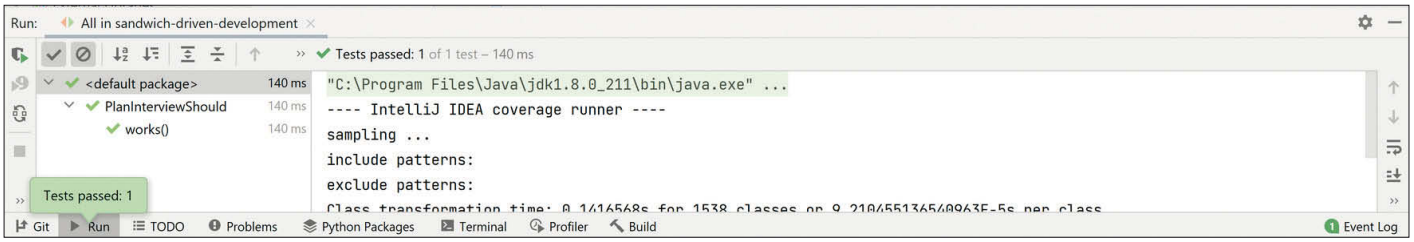


Figure 4

```
22    public Interview plan(String candidateId, LocalDate availability) {
23        Candidate candidate = candidates.findById(candidateId);
24        List<String> candidateSkills = candidate.getSkills();
25
26        List<Recruiter> availableRecruiters =
27                recruiters.findRecruiterByAvailability(availability);
28
29        Optional<Recruiter> recruiter = availableRecruiters.stream()
30                .filter(availableRecruiter ->
31                        availableRecruiter.getSkills().containsAll(candidateSkills))
32                .findFirst();
33
34        Recruiter appropriateRecruiter = recruiter.orElseThrow(AnyRecruiterFoundException::new);
35        recruiters.bookAvailability(appropriateRecruiter, availability);
36
37        Interview interview = new Interview();
38        interview.setCandidate(candidate);
39        interview.setRecruiter(appropriateRecruiter);
40        interview.setInterviewDate(availability);
41
42        interviews.save(interview);
43        return interview;
44    }
45 }
```

Figure 3



```
@Override
public String toString() {
    return "Candidate{" +
            "skills=" + skills +
            ", name='" + name + '\'' +
            '}';
}
```

REX :

Personnellement je ne m'amuse pas à refactorer du code pour me faire plaisir. Si une partie du code fonctionne correctement, même s'il est illisible, pas testé ou long, tant que je n'ai pas à le modifier, je ne vais pas le refactorer. Il faut toujours garder en mémoire que le refactoring a un coût et celui-ci n'est pas négligeable surtout lorsqu'on est face à une longue méthode, par exemple.

Par contre, si jamais je dois modifier ou améliorer une partie du code et que celle-ci n'est pas couverte (ou pas complètement) par des tests j'en profite pour utiliser les Approval Tests. Une fois, cette partie du code couverte par des tests, je peux ensuite commencer à le refactorer, à le casser en plus petits morceaux, afin de le rendre plus maîtrisable. Je vais aussi mettre en place des nouvelles classes, des méthodes et de nouveaux tests.

Mon refactoring est terminé. Grâce à mes Approval Tests, je sais que je n'ai rien cassé dans mon code. Je peux maintenant commencer à les supprimer, puisque j'ai couvert mon code par d'autres types de tests (unitaires, d'intégration, d'acceptance, etc.)

Cette procédure peut sembler coûteuse au niveau du temps. Cependant, il m'est arrivé plusieurs fois de refactorer un code non testé et de créer des bugs, voire des problèmes de prod. J'ai donc dû revenir en arrière, ce qui m'a parfois fait perdre une journée complète de travail. Les tests sont donc utiles et ne sont en aucun cas négligeables. Passer du temps à écrire des tests est un gain de temps pour le maintien de mes produits sur le long terme.

Conseils :

- Les Approvals Tests ne sont pas autosuffisants. Il faut créer des tests unitaires, d'intégration, etc. pour obtenir un résultat vraiment fiable.
- Il ne faut pas se fier à un code qui est couvert à 100 % par les tests. Pour cela, essayez d'ajouter du "Mutation Testing". Vous pouvez le faire à la main, en modifiant des conditions ou des valeurs dans votre code. Si après avoir modifié certaines logiques du code, les tests sont toujours verts, c'est que vous n'avez pas de tests fiables.
- Ne pas refactorer un code qui fonctionne correctement si on ne doit pas lui ajouter/supprimer une fonctionnalité.

FAQ :

- Peut-on tester une méthode qui renvoi *void* ? Oui, mais cela ne va pas vraiment nous aider à faire un Golden Master, car on n'obtient rien en output.
- Peut-on moquer les dépendances externes ? Oui vous pouvez très bien utiliser une librairie de bouchon par exemple Mockito pour Java.

Repos GIT :

- Java : https://github.com/SepehrNamdar/sandwich-driven-development/tree/solution-1/approval_tests
- C# : https://github.com/H-Ahmadi/DDDEU21_Sandwich_Driven_Development/tree/solution-1/approval_tests

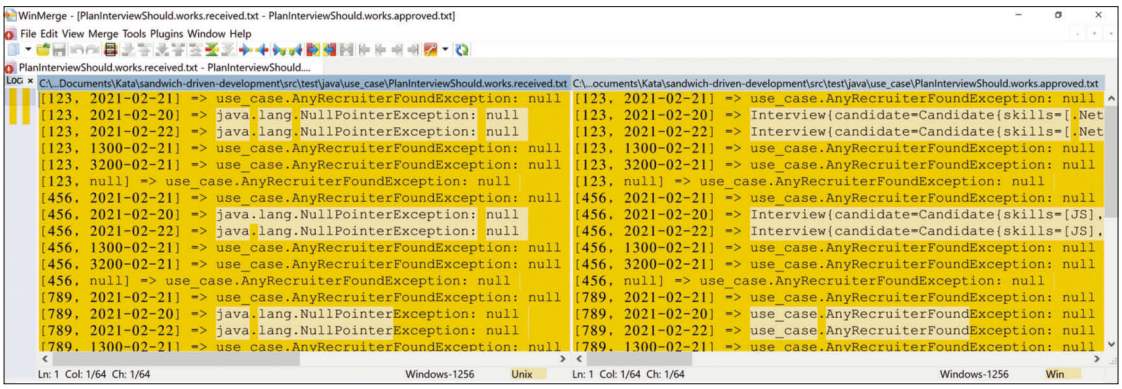


Figure 5

Sébastien Colas

Je suis formateur en informatique depuis bientôt 20 ans. Au cours de ma carrière j'ai pu dispenser des cours sur de nombreuses technologies : Serveur d'applications JavaEE, SOA, Services Web, Linux, Virtualisation, API led Connectivty et Application Network. Sans oublier les langages : Java, PHP, Python, JavaScript... Je suis aussi auteur pour « Linux Pratique » et « Hackable » autour des sujets Raspberry Pi, Arduino, IoT, électronique digitale et bien sûr Linux.

http://colas.sebastien.free.fr/

Superviser votre Linux à l'aide d'un écran LCD

Il peut s'avérer utile de pouvoir accéder à des informations de son Linux sans avoir à changer de fenêtre surtout si, par exemple, vous êtes dans une partie de jeu. Dans cet article je vous propose donc dans un premier temps de réaliser un montage électronique très simple permettant l'affichage de tout type d'information sur un écran LCD 16x2 caractères. Ensuite nous utiliserons lcd4linux pour afficher les données qui nous intéressent. **Figure 1**

Hardware

Voici les différents composants qui composeront notre projet :

- écran LCD 16x2 caractères **Figure 2**
- 1602 LCD USB Mini Drive Board **Figure 3**
- cordon USB vers micro-USB

L'assemblage de l'écran LCD vers le convertisseur USB est extrêmement simple, à chaque PIN de l'écran LCD correspond une PIN du convertisseur USB.

Le gros avantage de ce montage est que l'on pourra au choix :

- déporter l'affichage sur le bureau à l'aide du cordon USB
- Intégrer l'afficheur directement à la tour de son PC

On pourra aussi opter pour un LCD retro-éclairé couleur.

Software

Il existe différentes manières de contrôler notre LCD, ici comme nous voulons principalement afficher des informations systèmes en temps réel nous allons utiliser **lcd4linux**.

Figure 1

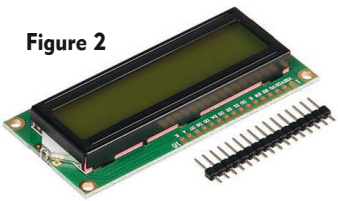


Figure 2

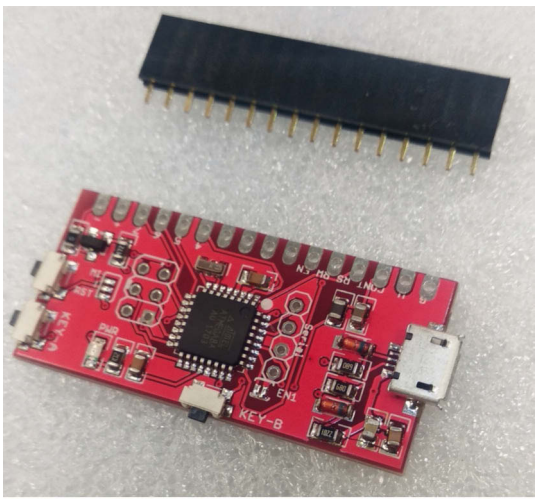


Figure 3

Par chance, le programme existe sur la plupart des distributions. Sous Ubuntu l'installation est très simple :

```
sudo apt-get update -y
sudo apt-get install -y lcd4linux
```

Lors de l'installation un fichier de configuration a été créé pour nous **/etc/lcd4linux.conf**. À des fins didactiques nous allons créer nous-mêmes le fichier à partir de zéro. Le but étant de couvrir les principales fonctionnalités de **lcd4linux**. Pour ce faire nous allons afficher :

- Un simple texte
- La date et l'heure
- L'utilisation du CPU
- La température extérieure stockée dans une base MySQL accessible dans notre réseau local
- La bande passante en upload et en download de notre Freebox à l'aide d'un script shell

Affichage d'un texte

Pour commencer créons quelques variables dans notre fichier /etc/lcd4linux.conf. Je rappelle que la modification du fichier doit se faire en tant que root :

```
Variables {
  seconde    1000
  five_secs 5000
  minute    60000
}
```

Comme on peut le constater l'unité de base est la milliseconde, d'où la définition de variables pour plus de simplicité. Définissons maintenant la configuration de notre afficheur :

```
Display LCD2USB {
   Driver   'LCD2USB'
   Size     '16x2'
   Backlight 0
   Icons    1
}
```

En l'occurrence nous avons un composant LCD2USB avec un afficheur LCD 16x2 caractères sans rétro-éclairage. On acti-

ve aussi la possibilité de se créer nos propres icônes.
Une fois défini il faut demander à LCD4Linux d'utiliser le display :

```
Display 'LCD2USB'
```

Maintenant nous allons devoir définir ce que nous voulons afficher, cela se fait à l'aide de composant appelé Widgets :

```
Widget Hello {
  class 'Text'
  expression 'Hello LCD'
  width 9
}
```

Ici notre Widget texte affichera « Hello LCD ». La longueur du texte est importante, elle servira lors de l'alignement du texte à droite par exemple.
Nous allons maintenant pouvoir positionner notre Widget au sein d'un Layout :

```
Layout MyLayout {
  Row1 {
    Col1 'Hello'
  }
  Row2 {
    Col2 'Hello'
  }
}
```

Un Layout se compose donc de lignes (row) et de colonnes (col) sur lesquelles on va pouvoir afficher des Widgets. Notre texte sera donc affiché sur la première colonne de la première ligne et la deuxième colonne de la deuxième ligne.
Notre fichier de configuration étant complété il ne nous reste plus qu'à tester le résultat en redémarrant le service :

```
sudo systemctl restart lcd4linux
```

N. B. : *Par la suite, il faudra exécuter à nouveau la commande pour tenir compte de chaque modification.*

Affichage de la date et l'heure

Nous allons afficher la date et l'heure sur les dernières colonnes de nos 2 lignes d'écran LCD. Il faut donc créer 2 Widgets : une pour la date et une pour l'heure :

```
Widget Date {
  class 'Text'
  expression strftime('%d/%m',time())
  width 5
  align 'L'
  update minute
}
```

```
Widget Time {
  class 'Text'
  expression strftime('%H:%M',time())
  width 5
  align 'L'
  update second
}
```

Nous définissons donc pour chaque Widget le fait que nous allons afficher du texte de longueur 5, qu'il faudra aligner le texte à gauche (L : Left) et qu'il faudra mettre à jour l'information toutes les minutes pour la date et toutes les secondes pour l'heure. Le plus important est ce que nous voulons afficher c'est-à-dire l'***expression***. LCD4Linux nous permet d'utiliser la fonction ***strftime()*** pour afficher la date courante ***time()*** avec un format spécifique. '***%d/%m***' pour la date et '***%H:%M***' pour l'heure. Mettons à jour notre Layout :

```
Layout MyLayout {
  Row1 {
    Col12 'Date'
  }
  Row2 {
    Col12 'Time'
  }
}
```

Affichage de l'utilisation du CPU

L'utilisation du CPU sur notre machine est aussi une information importante, nous allons afficher cette information au début de la première ligne. Comme précédemment il va nous falloir ajouter un Widget qui sera ensuite intégré à notre Layout :

```
Widget Busy {
  class 'Text'
  expression proc_stat::cpu('busy', second)
  postfix '%'
  width 5
  precision 0
  align 'R'
  update second
}
Layout MyLayout {
  Row1 {
    Col1 'Busy'
    Col12 'Date'
  }
  Row2 {
    Col12 'Time'
  }
}
```

Concentrons-nous sur les paramètres importants du Widget. Nous utilisons ***postfix*** pour afficher le caractère **%** après la consommation de notre CPU. Nous alignons notre texte à droite et nous ne voulons aucun chiffre après la virgule : ***précision 0***. L'expression nous permet de récupérer l'utilisation du CPU avec une périodicité de 1 seconde.

Affichage de la température extérieure

L'accès aux données du système local s'avère donc simple. Intéressons-nous maintenant à une source de données distante. Nous voulons connaître la température extérieure. Dans notre exemple, le relevé de température est stocké dans une base de données MySQL accessible depuis le réseau local. LCD4Linux propose la connexion à MySQL sous forme

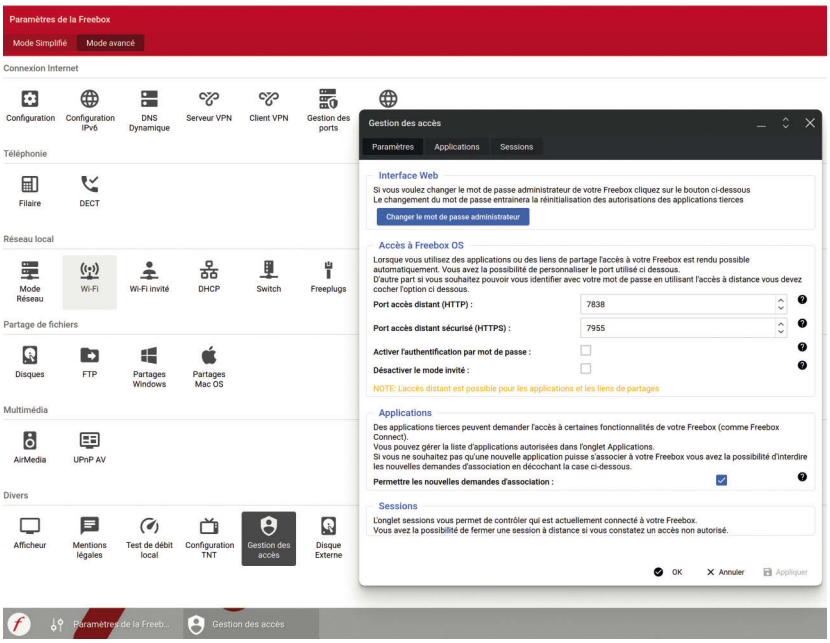


Figure 4

d'un plugin, il suffit donc de le configurer pour avoir l'accès à la base de données :

```
Plugin MySQL {
  server 'meteo.local'
  port 3306
  user 'user'
  password 'password'
  database 'meteo'
}
```

Une fois l'accès à la base configuré, nous pouvons créer un nouveau Widget qui contiendra la requête SQL d'accès aux données :

```
Widget MySQLTemp {
  class 'Text'
  expression MySQL::query('SELECT temperature FROM meteo.sensor_
data WHERE sensor_id=1 order by date DESC limit 1;')
  width 4
  update minute
}
```

Malheureusement le symbole degré n'est pas supporté par **lcd4Linux**, il est donc impossible d'utiliser **postfix** dans le **widget** comme nous l'avons fait précédemment. Nous allons donc créer à la main notre symbole degré dans un nouveau **widget**. Attention celui-ci aura pour classe **Icon** :

```
Widget Degree {
  class 'Icon'
  Bitmap {
    Row1 '.**..'
    Row2 '*..*.'
    Row3 '*..*.'
    Row4 '.**..'
    Row5 '.....'
    Row6 '.....'
    Row7 '.....'
    Row8 '.....'
  }
}
```

Il ne nous reste plus qu'à afficher nos deux widgets au début de la seconde ligne :

```
Layout MyLayout {
  Row1 {
    Col1 'Busy'
    Col12 'Date'
  }
  Row2 {
    Col1 'MySQLTemp'
    Col5 'Degree'
    Col12 'Time'
  }
}
```

Affichage d'informations réseau issues d'une Freebox

La dernière chose que nous voulons afficher sur notre écran est l'état de notre réseau. Il est bien sûr très facile de récupérer les informations de notre Linux en terme de débit réseau, mais sur un réseau local il est plus utile d'avoir une idée des débits réseau internet au niveau de la box internet elle-même. Ici nous allons récupérer ces informations d'une Freebox, car Free met à disposition une API de contrôle et supervision : https://dev.freebox.fr/sdk/os/#

Création des scripts de supervision

Inutile de ré-inventer la roue, on trouve facilement sur GitHub du code prêt à l'emploi pour simplifier l'accès à l'API Free. Ici nous allons récupérer les informations grâce à un script shell :

```
git clone https://github.com/JrCs/freeboxos-bash-api.git
```

Avant de pouvoir utiliser le script shell, il faut tout d'abord s'assurer que la Freebox accepte de nouvelles demandes d'association d'application. Il faut donc se connecter sur l'interface de la Freebox via l'URL http://mafreebox.freebox.fr.
Il faudra cliquer sur ***Paramètre de la Freebox***, ensuite sélectionner le ***mode avancé*** et cliquer sur ***Gestion des accès***. Ensuite dans le cadre ***Application***, il faudra bien vérifier que la case "***permettre les nouvelles demandes d'association***" est cochée. Pour des raisons de sécurité, une fois notre application enregistrée, il est fortement conseillé de décocher la case. **Figure 4**
Enregistrons notre application pour obtenir un jeton d'accès :

```
cd freeboxos-bash-api
source ./freeboxos_bash_api.sh
authorize_application 'NetworkMonitoring.app' 'Network Monitoring
Application' '1.0.0' 'Linux'
```

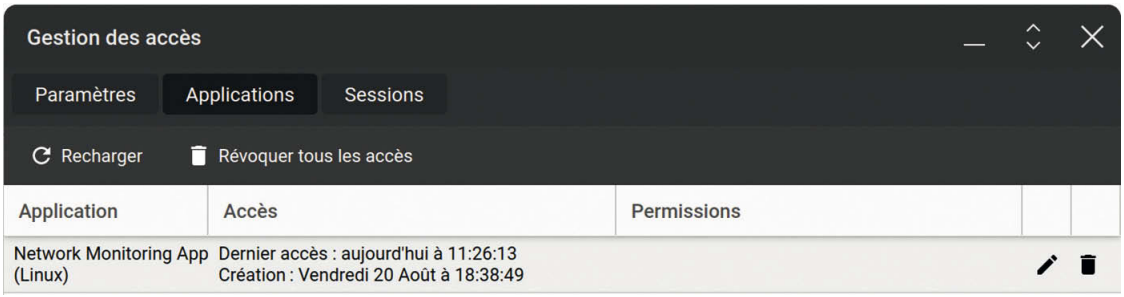


Figure 5

Il faudra ensuite confirmer la création du jeton en appuyant sur le bouton OK de la Freebox.
Récupérons le jeton que nous allons utiliser par la suite :

```
MY_APP_ID="NetworkMonitoring.app"
MY_APP_TOKEN="4uZTLMMwSyiPB42tSCWLpSSZbXIYi+d+F32tVMx2
j1p8oSUUk4Awr/OMZne4RRIY"
```

Dans le menu gestion des accès de la Freebox notre application doit apparaître. **Figure 5**
Nous pouvons désormais écrire un script pour accéder au débit internet en download. Ici l'unité sera exprimée en Mo/s : ***rate_down.sh***

```
#!/bin/bash
MY_APP_ID="NetworkMonitoring.app"
MY_APP_TOKEN="4uZTLMMwSyiPB42tSCWLpSSZbXIYi+d+F32tVMx2
j1p8oSUUk4Awr/OMZne4RRIY"
. /home/seb/freeboxos-bash-api/freeboxos_bash_api.sh
login_freebox "$MY_APP_ID" "$MY_APP_TOKEN"
answer=$(call_freebox_api '/connection/')
rate_down=$(get_json_value_for_key "$answer" 'result.rate_down')
echo "$rate_down 1048576" | awk '{printf "%.1f", $1 / $2}' | tr , .
```

De même le script pour l'updload, ici l'unité sera exprimée en ko/s : ***rate_up.sh***

```
#!/bin/bash
MY_APP_ID="NetworkMonitoring.app"
MY_APP_TOKEN="4uZTLMMwSyiPB42tSCWLpSSZbXIYi+d+F32tVMx2
j1p8oSUUk4Awr/OMZne4RRIY"
. /home/seb/freeboxos-bash-api/freeboxos_bash_api.sh
login_freebox "$MY_APP_ID" "$MY_APP_TOKEN"
answer=$(call_freebox_api '/connection/')
rate_up=$(get_json_value_for_key "$answer" 'result.rate_up')
echo "$rate_up 1024" | awk '{printf "%d", $1 / $2}' | tr , .
```

Affichage des données
Une fois les scripts prêts, il ne reste plus qu'à créer un Widget pour eux et de mettre à jour notre Layout.

```
Widget DownRate {
  class 'Text'
  expression exec('/home/seb/freeboxos-bash-api/rate_down.sh', five_secs)
  prefix 'D'
  width 4
  align 'R'
  update five_secs
}
```

```
Widget UpRate {
  class 'Text'
  expression exec('/home/seb/freeboxos-bash-api/rate_up.sh', five_secs)
  prefix 'U'
  width 4
  align 'R'
  update five_secs
}

Layout MyLayout {
  Row1 {
    Col1 'Busy'
    Col7 'UpRate'
    Col12 'Date'
  }
  Row2 {
    Col1 'MySQLTemp'
    Col5 'Degree'
    Col7 'DownRate'
    Col12 'Time'
  }
}
```

Conclusion

Nous avons vu, dans cet article, que grâce à LCD4Linux nous pouvons afficher facilement tout type de données sur un écran LCD connecté en USB.
Voici quelques idées pour aller plus loin :

- Remplacer LCD4Linux par LCDProc
- Créer un programme Python pour piloter l'écran LCD
- Déporter les boutons Key-A et Key-B pour une utilisation plus simple et bien sûr associer une fonction à chacun de ces boutons
- Câbler le Reset sur un bouton, en effet parfois le LCD2USB ne démarre pas correctement
- Démarrer/redémarrer LCD4Linux lorsque l'on branche ou rebranche l'écran LCD grâce à udev

A vous de jouer !

Thierry LERICHE
Architecte et Tech lead
@ThierryLeriche

Coder Fibonacci en Java avec des tests

Dans cet article, on se propose de coder le calcul des termes de la Suite de Fibonacci ainsi que de quelques variantes, comme Tribonacci, le tout en Java, et en écrivant des tests. L'idée est d'écrire les tests avant, pendant et après le processus d'implémentation des fonctionnalités.

Il existe de nombreuses façons de coder *Fibonacci*, dépendant du langage de programmation utilisé, mais également des concepts mis en œuvre. Dans ce billet, qui se veut assez simple, on va se limiter à de la programmation récursive. D'autres versions sont disponibles sur le Web. Et dans tous les cas, c'est un basique que les étudiants en informatique doivent connaître par cœur[1].

On va donc employer des tests tout au long du processus, en s'inspirant de *TDD* (Test Driven Development), sans en faire au sens pur, et de *3T* (Tests en Trois Temps). Les tests permettent de structurer la pensée, et donc l'architecture du code. On admet généralement qu'un code non testable est mal conçu, alors qu'un code testable est mécaniquement de meilleure qualité. Les tests vont aider à programmer, mais également à réfactorer le code, c'est-à-dire à revenir dessus pour l'améliorer (bugs, performances, beauté, etc.) en ayant une situation stable au départ, garantie par les tests, pour contrôler l'avancement.

Fibonacci

En mathématiques, la *suite de Fibonacci* est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle est très utile, par exemple pour calculer le nombre d'or, et on la retrouve largement dans la nature. Bla-bla-bla. On sait déjà ça et c'est très bien expliqué sur la page Wikipedia dédiée[2], vers laquelle je vais donc vous renvoyer pour la longue explication.

Dans les grandes lignes, la suite est définie par :

- f(0) = 0
- f(1) = 1
- f(n) = f(n-1) +f(n-2) si 1 < n

Les premiers termes de cette suite sont donc 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597…

(1) C'est le type d'exercice simple qu'on peut demander aux juniors en entretien.

(2) https://fr.wikipedia.org/wiki/Suite_de_Fibonacci

Écrire un bloc qui compile

Pour coder, l'idée va être de décomposer le travail en trois étapes : définition, test et implémentation. Et on boucle.

La plus grosse partie de l'intelligence est utilisée pour définir le contrat de service. On peut le faire dans une interface, comme ci-dessous, mais ce n'est pas obligatoire.

```
public interface Fibonacci {

 /**
  * Calcule la valeur du terme de la suite de Fibonacci pour le rang indiqué.
  *
  * REGLE #0 f(0) = 0
  * REGLE #1 f(1) = 1
  * REGLE #2 f(n) = f(n-1) + f(n-2) si 2 < n
  * REGLE #3 f(n) = Exception si n < 0
  *
  * Premiers termes : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597
  *
  * cf. https://fr.wikipedia.org/wiki/Suite_de_Fibonacci
  *
  * @param n le rang
  * @return la valeur du terme pour le rang indiqué
  * @throws IllegalArgumentException si le rang indiqué est négatif
  */
 int calculate(int n);
```

Si on prévoit d'écrire un peu de doc, l'interface est sans aucun doute l'endroit le plus adapté. On peut mettre la doc dans un fichier séparé, sur le réseau de l'entreprise par exemple, mais les développeurs préfèrent généralement avoir la doc sous les yeux et disponible dans l'éditeur.

À ce stade, on a juste besoin d'une implémentation qui compile. Les IDE, comme *Eclipse* ou *Intellij*, savent générer le code à partir de l'interface. Ils retournent habituellement *zéro* comme valeur par défaut. C'est ce qu'on appelle un *nombre magique* et c'est quelque chose qu'on veut absolument éviter. En effet, cette valeur pourrait correspondre à quelque chose de réel pour la fonctionnalité, ce qui est d'ailleurs le cas pour *Fibonacci*. Et choisir une autre valeur n'est pas forcément mieux. Cela porte à confusion. Ici, on va plutôt lancer une exception spécialisée, indiquant que le code n'a pas encore été écrit. Pour les développeurs, qui liront ce code, le message sera clair.

```
public class RecursiveFibonacci implements Fibonacci {

 @Override
```

```
public int calculate(int n) {
  throw new UnsopportedOperationException("bientôt");
}
```

Note : *En Kotlin, on aurait tout simplement utilisé la méthode TODO dont le message est encore plus clair. Il s'agit d'une méthode spéciale qui compile correctement, mais qui lance une exception indiquant que le code n'est pas prêt, quand on passe dessus. Ça revient au même, avec quelques petites améliorations.*

Écrire des tests

On prévoit d'avoir plusieurs implémentations pour le calcul de la suite. Et on prévoit donc d'avoir tout autant de versions à tester. Mais les valeurs des termes de la suite ne vont pas changer, tout comme les cas d'utilisation. On veut voir ce calcul comme une boîte noire. C'est pourquoi on peut écrire les tests dans une classe abstraite.

```
public abstract class AbstractFibonacciTest {

 protected Fibonacci fibonacci;

 @Test
 public void testCalculateIndex0() {
  // Arrange
  final int n = 0;
  final int expected = 0;

  // Act
  final int result = fibonacci.calculate(n);

  // Assert
  Assertions.assertEquals(expected, result);
 }
```

Il existe plusieurs systèmes de nommage pour les méthodes de test. Certains préféreront le *Snake-Case* : should_do_something_when_case. Certains préféreront le formalisme *Camel-Case*, plus classique en Java : testSomethingCase. Dans tous les cas, il n'y a rien d'obligatoire.

À l'intérieur de la méthode, j'aime utiliser l'organisation *AAA* (Arrange Act Assert) qui permet de bien séparer et identifier les blocs, et qui ressemble beaucoup à *GWT* (Given When Then). On veut ainsi séparer le cas de test avec ses données d'entrée et les résultats attendus, de l'appel à proprement parlé du calcul, et enfin des tests. On notera que la partie la plus intéressante est la première, les deux autres n'étant rien de plus que du code purement technique. Je trouve que cette organisation est beaucoup plus lisible que de tout écrire en un seul bloc, et permet d'identifier beaucoup plus facilement le cas de test, puisque les données fonctionnelles ne sont pas mélangées avec le code technique. Dites-moi ce que vous en pensez. En utilisant cette organisation, il sera d'autant plus simple de switcher sur des tests paramétrés (cf. plus bas).

Il faut bien commencer quelque part. Faute de mieux, on peut choisir les cas de test dans l'ordre où ils ont été décrits dans le cahier des charges (pour nous la page Wikipédia, ou l'interface). D'expérience, cet ordre est généralement pertinent.

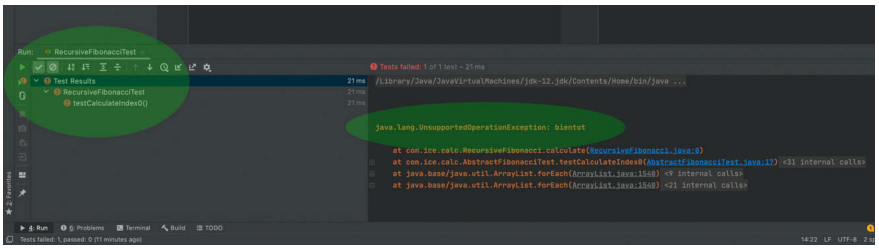


Figure 1

```
class RecursiveFibonacciTest extends AbstractFibonacciTest {

 public RecursiveFibonacciTest() {
  fibonacci = new RecursiveFibonacci();
 }
```

C'est la classe fille qui identifie l'implémentation cible. Elle est hyper simple.

On peut la lancer dès maintenant ; il n'y a qu'un test et il est bien rouge, ce qui est ce qu'on souhaite. En effet, l'idée est de commencer avec des tests qui échouent puis d'écrire le code nécessaire pour les valider, dans cet ordre. **Figure 1**

Pour le test suivant, c'est exactement la même chose, en changeant juste les valeurs en entrée et les valeurs attendues.

```
@Test
public void testCalculateIndex1() {
 // Arrange
 final int n = 1;
 final int expected = 0;

 // Act
 final int result = fibonacci.calculate(n);

 // Assert
 Assertions.assertEquals(expected, result);
}
```

En *TDD pur*, on aurait écrit un peu de test, puis un peu de code, puis un peu de test, puis un peu de code, etc. jusqu'à avoir tout traité. Dans la mesure où on a un cahier des charges bien fait, et un cerveau, l'écriture des tests n'est qu'une simple traduction du français vers du Java, et on peut écrire tous les cas triviaux d'un seul coup.

Et c'est aussi pareil pour toutes les autres valeurs. Les blocs *Act* et *Assert* sont toujours les mêmes, et à juste titre. Il suffit de les factoriser dans une méthode privée. Chaque méthode de test n'est alors plus qu'un cas de test.

```
@Test
public void testCalculateIndex2() {
 // Arrange
 final int n = 2;
 final int expected = 2;

 // Act and assert
 doTestCalculate(n, expected);
}

@Test
public void testCalculateIndex3() {
 // Arrange
```

```
final int n = 3;
final int expected = 3;

// Act and assert
doTestCalculate(n, expected);
}

@Test
public void testCalculateIndex4() {
// Arrange
final int n = 4;
final int expected = 5;

// Act and assert
doTestCalculate(n, expected);
}
```

Dans l'outil (Eclipse, Intellij, Maven, etc.), les tests sont rouges (ie. échouent), ce qui est encore ce qu'on veut à ce stade. Par contre, il y a encore de très nombreuses duplications de code. La classe de test semble énorme alors qu'il n'y a qu'une petite dizaine de cas. Pour simplifier, on peut itérer sur une liste de valeurs.

```
@Test
public void testCalculate() {
// Arrange
final int[] expecteds = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987};

// Act and assert
for (int n = 0; n < expected.length; n++) {
  doTestCalculate(n, expected[n]);
}
}
```

Le souci est que cela n'apparaît plus que comme un seul (gros) test dans l'IDE (ici Intellij) et que les tests s'arrêtent dès qu'une des valeurs est incorrecte.

Avec *JUnit 5 (Jupiter)*, il est possible de définir des tests paramétrés. Il y a plein de façons de le faire, la plus simple étant de passer les valeurs comme du *CSV* (Coma Separated Values) dans l'annotation.

```
@ParameterizedTest
@CsvSource(value = {"0:0", "1:1", "2:1", "3:2", "4:3", "5:5", "6:7", "7:13", "8:21"},
delimiter = ':')
public void testCalculate(final int n, final int expected) {
doTestCalculate(n, expected);
}
```

Dans l'IDE, les différents cas sont bien identifiés et séparés.

Figure 2

Le souci de cette approche est que ça devient illisible dès qu'on augmente le nombre de cas de test, car la taille du contenu de l'annotation explose.

Or, et on ouvre un nouveau paragraphe pour le dire, si quelqu'un, disons un expert, s'est embêté à fournir des cas fonctionnels dans le cahier des charges, il faut tous les avoir, à minima, dans les tests. Et cela n'empêche pas d'en ajouter et de compléter le cahier des charges. Tout en restant sur un format CSV, on peut utiliser un fichier, moins limitant en taille, dans lequel on peut mettre de très nombreux cas de test.

```
@ParameterizedTest
@CsvFileSource(resources = "/fibonaccicsv", delimiter = ':', numLinesTo
Skip = 1)
public void testCalculate(final int n, final int expected) {
 doTestCalculate(n, expected);
}
```

Reste encore le cas où le rang est négatif, qu'on n'avait pas encore traité.

```
@Test
public void testCalculateNegative() {
 // Arrange
 final int n = -1;

 // Act
 Assertions.assertThrows(IllegalArgumentException.class, () -> {
  fibonacci.calculate(n);
 }
}
```

Selon les IDE, les tests paramétrés pourront être regroupés et séparés des tests classiques.

Note : *avec JUnit 4, on pouvait indiquer l'exception attendue directement au niveau de l'annotation.*

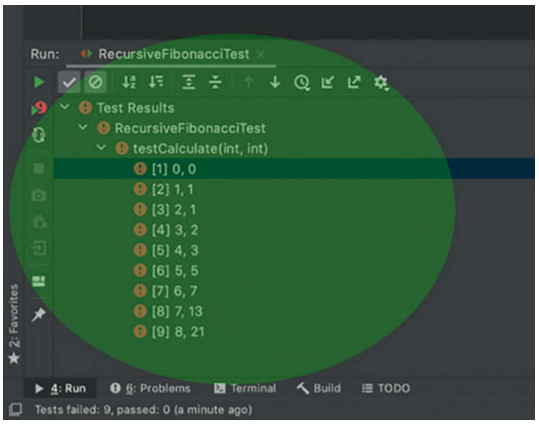
```
@Test(expected = IllegalArgumentException.class)
public void testCalculateNegative() {
 // Arrange
 final int n = -1;

 // Act
 fibonacci.calculate(n);
}
```

Pour l'instant, on en a fini avec les tests. Mais on y reviendra plus tard.

On peut envoyer le code des tests au client (ou à l'équipe) pour validation. Cela permet de valider qu'on est bien sur la même longueur d'onde.

Remarque : *Cela pose toutefois question. Si on commit/push dans Git, même dans une branche à part, on envoie du code qui compile, mais qui plante au niveau des tests (ce qui est d'ailleurs bien ce qu'on veut). En TDD pur, on aurait codé un test, puis le code pour le faire passer, puis un test, puis... La question ne se serait donc pas posée dans les mêmes termes (sans jeu de mot). Mais on n'aurait pas pu faire valider de la même manière. Un choix d'organisation devra donc être effectué en amont.*

Figure 2



Accessoirement, ces tests, en plus de l'interface, sont relativement simples à lire par un développeur, qui sera bien plus à l'aise avec du code qu'avec un long fichier Word. En ce sens, les tests permettent également de documenter le programme.

Écrire le code du calcul

Cette fois encore, si on n'a pas d'idée par où commencer, le plus simple est de commencer par le début en prenant les règles et les tests dans l'ordre d'où ils viennent. La première règle n'est qu'une formalité.

```
public class RecursiveFibonacci implements Fibonacci {

 @Override
 public int calculate(int n) {
  // REGLE 1
  if (n == 0) {
   return 0;
  }

  throw new UnsupportedOperationException("bientôt");
 }
```

On lance (tous) les tests dès qu'on est content d'un bloc de code. Ce n'est pas la peine d'attendre, car il n'y a aucune économie intéressante à espérer. **Figure 3**
Et puis on continue avec le deuxième test…

```
@Override
public int calculate(int n) {
 // REGLE 1
 if (n == 0) {
  return 0;
 }

 if (n == 1) {
  return 1;
 }

 throw new UnsupportedOperationException("bientôt");
}
```

Ce qui nous permet déjà d'avoir deux cas de test au vert (ie. passant). **Figure 4**
On a donc une (petite) situation stable. Et on peut en profiter pour se lancer dans un (petit) refactoring. L'idée est que ça marchait avant le refacto, avec rapport de test à l'appui, et que ça doit marcher (en mieux) après. Si des tests verts deviennent rouges, cela indique qu'on a raté une étape. Dans ce cas, on pose le stylo et on corrige. Ici le refactoring proposé est trivial, mais il illustre bien le processus.

```
@Override
public int calculate(int n) {
 // REGLE 1
 if (n == 0 || n == 1) {
  return n;
 }

 throw new UnsupportedOperationException("bientôt");
}
```

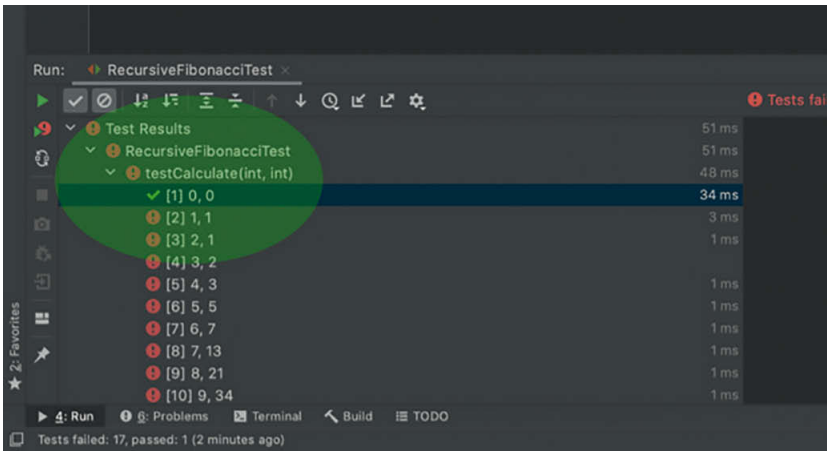


Figure 3

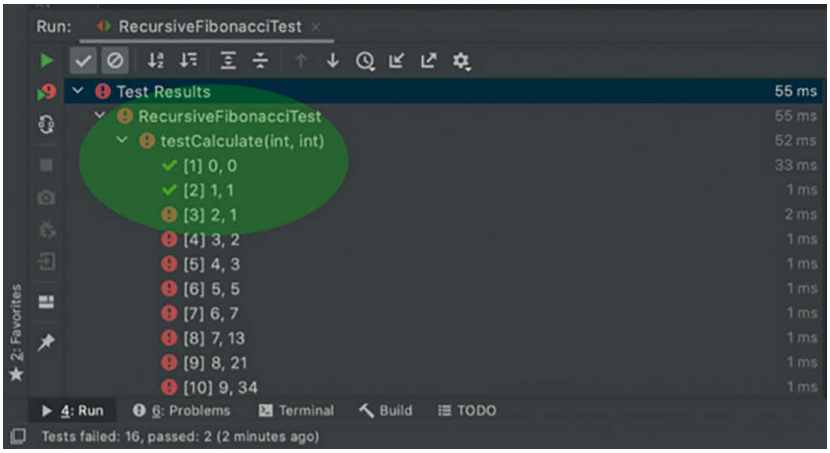


Figure 4

Le calcul final, quant à lui, n'est pas super compliqué. C'est un simple, enfin double, appel récursif.

```
@Override
public int calculate(int n) {
 ...

 return calculate(n – 1) + calculate(n – 2);
}
```

On voit que tous les cas des tests paramétrés deviennent verts d'un coup, sauf un... Flute de zut alors !... Hummm... c'est bizarre. Il faut analyser.
En y regardant bien, on voit que ce n'est pas le code qui plante, mais que c'est le jeu de test qui est mauvais. Ça arrive. Ici, on s'est trompé en recopiant les valeurs dans le fichier CSV. En effet, on attendait la valeur 7, mais on obtient bien 8 qui est la bonne valeur pour le rang 6.
Des fois, c'est le cahier des charges qui contient des erreurs. En travaillant en mode Agile, on accepte que ça arrive. Il suffit de passer un coup de fil au rédacteur pour clarifier la situation. Ce genre de chose est plus fréquent qu'on ne le pense.
Il ne reste plus qu'à vérifier le cas des rangs négatifs.

```
@Override
public int calculate(int n) {
 // REGLE 4
 if (n < 0) {
```

```
    throw new IllegalArgumentException("The parameter n can not be negative!");
  }

  ...
}
```

Tout est bon… On est légitimement en droit de penser qu'on a fini. On peut envoyer le produit au client.

Trop lent ?

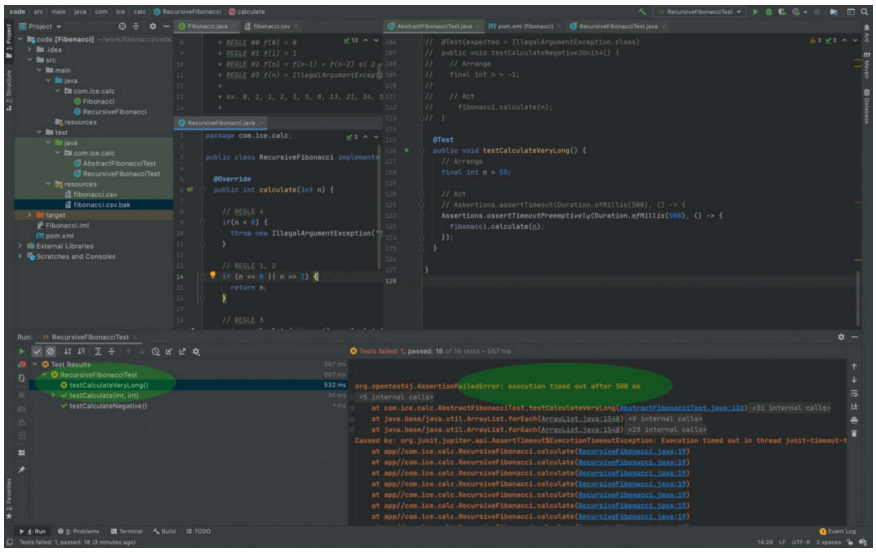
Mais quelques jours plus tard, après avoir bien tout vérifié, le client nous appelle. Il dit que c'est super long, en fournissant la valeur 50 comme référence. Dans ce genre de situation, le mieux est de revenir à nos tests et d'ajouter un nouveau cas. Ça va nous servir pour constater le problème, en conserver une trace, et enrichir la base de test.

```
@Test
public void testCalculateVeryLong() {
  // Arrange
  final int n = 50;

  // Act
  fibonacci.calculate(n);
}
```

Effectivement, quand on lance ce test, on a l'impression qu'il ne va jamais arriver jusqu'à la fin du calcul, et on est bien obligé de l'arrêter manuellement. Et pour cause, il peut durer très très longtemps… On doit donc introduire une gestion du timeout. Exercice : calculer précisément la complexité algorithmique de ce calcul.

```
@Test
public void testCalculateVeryLong() {
  // Arrange
  final int n = 50;

  // Act
  // Assertions.assertTimeout(Duration.ofMillis(500), () -> ...
  Assertions.assertTimeoutPreemptively(Duration.ofMillis(500), () -> {
    fibonacci.calculate(n);
  });
}
```

Figure 5



Note, la méthode de test *assertTimeout()* patiente jusqu'à la fin du calcul (et donc pour un temps infini) pour en mesurer la durée, ce qui n'est évidemment pas ce qu'on veut ici. C'est pourquoi on utilise plutôt *assertTimeoutPreemptively()* qui s'arrête dès que le timeout est atteint.
Le test échoue donc pour cause de timeout. **Figure 5**
Concrètement, effectuer le calcul de cette suite de manière récursive est hyper long. Et on revient sans cesse sur des termes pourtant déjà calculés. La solution la plus simple consiste donc à les stocker au fur et à mesure, ce qui permet d'améliorer les performances au niveau d'une complexité algorithmique linéaire. On parlera de *mémoïzation*. Le calcul devient quasi instantané.

```
private Map<Integer, Integer> memo = new HashMap<>();

@Override
public int calculate(int n) {
  ...

  final Integer memorized = memo.get(n);
  if (memorized != null) {
    return memorized;
  }

  // REGLE 3
  final int result = calculate(n – 1) + calculate(n – 2);
  memo.put(n, result);

  return result;
}
```

On envoie donc ce nouveau code au client.

Ça boucle ?

Le client appelle à nouveau pour dire que les valeurs sont mauvaises dès le rang 62. Zut alors[3]… Mais c'est l'opportunité d'enrichir à nouveau la base de test. Dans un premier temps, mettons un log temporaire (ou un point d'arrêt) pour comprendre ce qui se passe.

```
@Test
public void testCalculateBigCalue() {
  // Arrange
  final int n = 62;

  // Act
  final int result = fibonacci.calculate(n);
  System.out.println(result);
}
```

À l'exécution, on voit que ça donne une valeur négative (-1709589543). C'est ennuyeux pour une somme d'entiers positifs. Pour ce rang, le terme prend une valeur qui dépasse la capacité des ints. Du coup, ça boucle (binairement) et les valeurs positives finissent par devenir négatives. Bon, c'est un classique… Cela oblige à faire évoluer le contrat, ce qui est une grosse évolution. On pourrait utiliser des longs à la place des ints mais c'est reculer pour mieux sauter. On va plutôt utiliser un

(3) Kraftos ne va vraiment pas être content, cf. Programmez n°246

BigInteger dont le nom permet d'en comprendre l'utilité.

```
public interface Fibonacci {

 BigInteger calculate(int n);
```

Ça se change dans l'interface, et donc dans les tests en conséquence.

```
private void doTestCalculate(fina int n, final BigInteger expected) {
 ...
}

@ParameterizedTest
@CsvFileSource(...)
public void testCalculate(final int n, final BigInteger expected) {
 doTestCalculate(n, expected);
}

@Test
public void testCalculateBigValue() {
 // Arrange
 final int n = 62;

 // Act
 final BigInteger result = fibonacci.calculate(n);

 // Assert
 Assertions.assertTrue(0 < result.compareTo(BigInteger.ZERO));
}
```

Et donc aussi dans l'implémentation, où l'utilisation de BigInteger à la place du type primitif int est tout de même moins lisible.

```
private Map<Integer, BigInteger> memo = new HashMap<>();

@Override
public BigInteger calculate(int n) {
 // REGLE 4
 if (n < 0) {
  throw new IllegalArgumentException("The parameter n can not be negative!");
 }

 // REGLE 1, 2
 if (n == 0 || n == 1) {
  return BigInteger.valueOf(n);
 }

 final BigInteger memorized = memo.get(n);
 if (memorized != null) {
  return memorized;
 }

 // REGLE 3
 final BigInteger result = calculate(n – 1).add(calculate(n – 2));
 memo.put(n, result);

 return result;
}
```

Il n'y a pas de difficulté majeure. Et en toute logique, ça fonctionne du premier coup. On peut donc envoyer ce nouveau code au client. Et ainsi de suite…

Tribonacci

La suite de *Tribonacci* est une suite d'entiers dans laquelle chaque terme est la somme des trois termes qui le précèdent. Et, là aussi, c'est très bien expliqué sur la page Wikipedia dédiée[4]. Dans les grandes lignes, la suite est définie par :

- f(0) = 0
- f(1) = f(2) = 1
- f(n) = f(n-1) +f(n-2) + f(n-3) si 2 < n

Les premiers termes de cette suite sont donc 0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, 927, 1705, 3136, 5768, 10609, 19513, 35890, 66012, 121415, 223317, 410744, 755476, 1389537, 2555757…

K-bonacci

Ce sont des suites dont la relation de récurrence est d'ordre k. Un terme est la somme des k termes qui le précèdent
Dans les grandes lignes, la suite est définie par :

- f(0) = 0
- f(n) = 1 si 0 < n < p
- f(n) = f(n-1) + f(n-2) + … + f(n-p) si p <= n

Et donc, Fibonacci est de récurrence p=2 et Tribonacci est de récurrence p=3. D'une certaine manière, on pourra considérer, d'un point de vue programmation, que ce sont des cas particuliers de K-Bonacci. Après avoir codé Fibonacci, c'est donc une formalité de proposer une interface pour K-Bonacci. En plus du rang n, il suffit d'ajouter la récurrence p.

```
public interface KBonacci {

 /**
  * Calcule la valeur du terme de la suite de K-Bonacci pour le rang n et la récurrence p indiqués.
  *
  * REGLE #0 f(0) = 0
  * REGLE #1 f(n) = 1 si 0 < n < p
  * REGLE #2 f(n) = f(n-1) + f(n-2) + ... + f(n-p-1) si p < n
  * REGLE #3 f(n) = Exception si n < 0 ou si p < 0
  *
  * Premiers termes pour p=3 : 0, 1, 1, 2, 4, 7, 13, 24, 44, 81...
  *
  * cf. Tribonacci https://fr.wikipedia.org/wiki/Suite_de_Tribonacci
  *
  * @param n le rang
  * @param p la récurrence
  * @return la valeur du terme pour le rang et la récurrence indiqués
  * @throws IllegalArgumentException si le rang ou la récurrence indiqués sont négatifs
  */
 BigInteger calculate(int n, int p);
```

Comme toujours, on part d'une implémentation vide.

```
public class RecursiveKBonacci implements KBonacci {

 @Override
 public BigInteger calculate(int n, int p) {
  throw new UnsupportedOperationException("bientôt");
 }
```

(4) *https://fr.wikipedia.org/wiki/Suite_de_Tribonacci*

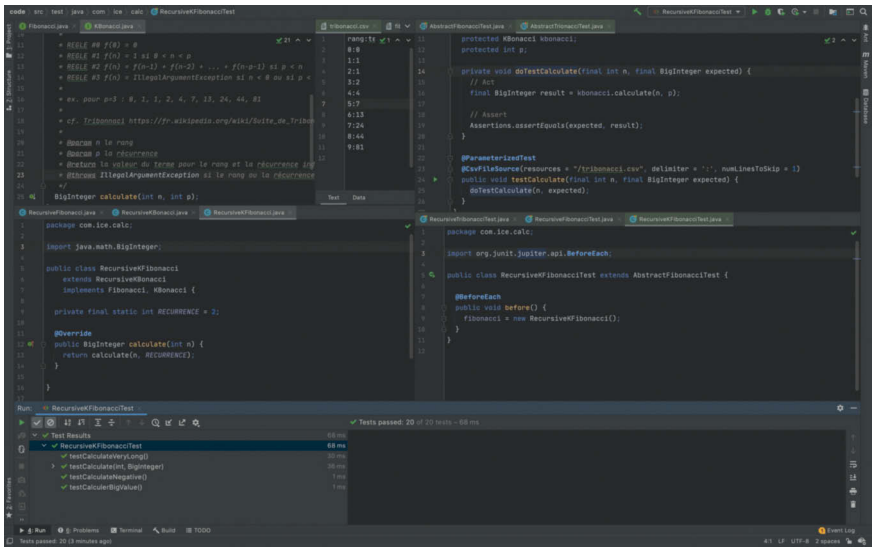


Figure 6

Et évidemment, on enchaîne directement sur les tests.

```
public abstract class AbstractTribonacciTest {

 protected KBonacci kbonacci;
 protected int p;

 private void doTestCalculate(final int n, final BigInteger expected) {
  // Act
  final BigInteger result = kbonacci.calculate(n, p);

  // Assert
  Assertions.assertEquals(expected, result);
 }

 @parameterizedTest
 @CsvFileSource(resources = "/Tribonacci.csv", delimiter = ':', numLinesTo
Skip = 1)
 public void testCalculate(final int n, final BigInteger expected) {
  doTestCalculate(n, expected);
 }
```

Il faut également écrire le CSV correspondant. C'est un simple copié-collé des valeurs.
Pour cet article, on s'épargne les autres méthodes de test qu'on avait écrites plus haut, mais il faudrait les écrire aussi, en adaptant les valeurs. On va de toute manière les réutiliser quand on fera une nouvelle implémentation de Fibonacci.

```
public class RecursiveTribonacciTest extends AbstractTribonacciTest {

 @BeforeEach
 public void before() {
  kbonacci = new RecursiveKbonacci();
  p = 3;
 }
```

Et sans surprise, tous les tests sont bien rouges à ce stade. On peut donc continuer.
Les deux premières règles sont triviales.

```
public class RecursiveKBonacci implements KBonacci {

 @Override
 public BigInteger calculate(int n, int p) {
  // REGLE 0
  if (n == 0) {
   return BigInteger.ZERO;
  }

  // REGLE 1
  if (n < p) {
   return BigInteger.ONE;
  }

  throw new UnsupportedOperationException("bientôt");
 }
```

Et elles passent au vert sans difficulté.
Pour le calcul, c'est un poil plus coton. On garde le principe de *mémoïzation* qu'on doit retranscrire sur une *multimap*[5]. Et il faut compulser la somme des termes. Il y a d'ailleurs très sûrement plus subtil que la version proposée ci-dessous, ce qui donnera l'occasion d'une nouvelle itération de refactoring. ***Code complet sur programmez.com & github***

Fibonacci via K

Cela nous donne l'occasion de coder une nouvelle version de Fibonacci en assemblant les briques déjà écrites, puisqu'on a redéfini Fibonacci comme une récurrence de 2 de K-Bonacci, pour plus simplement 2-Bonacci. Et oui, en fait, k = p.

```
public class RecursiveKFibonacci
    extends RecursiveKBonacci
    implements Fibonacci, KBonacci {

 private final static int RECURRENCE = 2;

 @Override
 public BigInteger calculate(int n) {
  return calculate(n, RECURRENCE);
 }
```

Avoir écrit les tests de manière abstraite prend donc tout son sens, puisqu'il suffit de l'étendre.

```
public class RecursiveKFibonacciTest extends AbstractFibonacciTest {

 @BeforeEach
 public void before() {
  fibonacci = new RecursiveKFibonacci();
 }
```

Et c'est magique ; il n'y a rien à faire. Ça marche tout seul.
Figure 6
On est parti d'un exemple finalement assez simple avec la suite de Fibonacci. Il n'y a pas de difficulté particulière. Mais cela permet de s'initier à quelques pratiques de test. Facile, non ? Pour s'amuser, on pourra coder d'autres implémentations, en réutilisant les tests déjà écrits, par exemple en itératif ou par calcul direct en profitant du fait que le ratio de deux membres consécutifs de la suite tend vers le nombre d'or, etc.

(5) Une map de collection

Python et la tortue : découvrir la programmation en dessinant

Deux activités de découverte de la programmation utilisant la bibliothèque turtle de python pour dessiner sur l'écran sont présentées ici. La première consiste à tracer des courbes de Bézier, et la seconde une courbe fractale. Bien qu'elles ne nécessitent pas de très nombreuses lignes de code, il ne s'agit cependant pas d'activités de niveau débutant, car elles ont pour but d'illustrer la notion de récursivité.

Pascal Lafourcade

Maître de conférences à l'IUT d'informatique de l'Université Clermont Auvergne et membre du Laboratoire d'Informatique, Modélisation et Optimisation des Systèmes. Il est spécialiste en sécurité informatique et cryptographie.

Malika More

Maîtresse de conférences à l'IUT d'informatique de l'Université Clermont Auvergne et membre du Laboratoire d'Informatique, Modélisation et Optimisation des Systèmes. Elle est responsable du groupe Informatique Sans Ordinateur de l'IREM de Clermont-Ferrand.

LA BIBLIOTHÈQUE TURTLE

Python possède une bibliothèque à vocation pédagogique appelée « turtle ». Elle offre la possibilité de créer des programmes à l'aide d'instructions proches du LOGO, un célèbre langage de programmation éducatif inventé par Wally Feurzig et Seymout Papert en 1967 au Massachusetts Institute of Technology.

La documentation de la bibliothèque se trouve ici :
https://docs.python.org/fr/3/library/turtle.html

Comme toute bibliothèque python, les lignes import turtle ou from turtle import * en début de programme permettent de l'utiliser. Elle fait appel au module tkinter pour sa partie graphique.

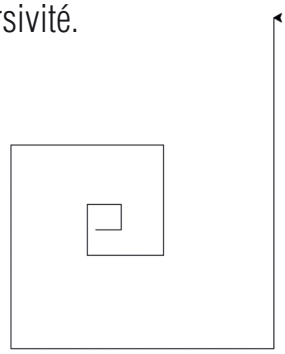
Les commandes ont pour objectif de contrôler sur l'écran un objet appelé « tortue », qui permet de dessiner avec un ensemble de commandes simples dans une fenêtre graphique. Le nom vient de la « tortue logo », un véritable robot piloté à l'aide du langage logo, utilisé dans certaines écoles dans les années 1970 et 1980. Il est à noter que le rôle de la tortue était aussi parfois joué par un enfant (ou un adulte) à quatre pattes, dans une démarche annonciatrice du courant de l'informatique débranchée, formalisé au début des années 2000 par Tim Bell et ses collègues en Nouvelle-Zélande. https://csunplugged.org/fr/

Pour déplacer la tortue et dessiner avec, nous pouvons utiliser les commandes suivantes :

- forward() permet d'avancer d'une distance donnée
- backward() permet de reculer d'une distance donnée
- right() permet de tourner à droite d'un angle donné (en degrés)
- left() permet de tourner à gauche d'un angle donné (en degrés)
- goto() permet d'aller à une position définie par deux coordonnées (à noter que le point (0,0) est au centre de la fenêtre)
- pendown() met le stylo en position d'écriture
- penup() lève le stylo
- clear() efface tout ce qui est dessiné

Il existe beaucoup d'autres commandes, mais les huit qui sont listées ci-dessus permettent de réaliser de nombreuses activités pour découvrir divers aspects de la programmation.

Voici un exemple permettant de tracer une spirale rectangulaire. Bien qu'il ne contienne que cinq lignes de code, il contient deux notions importantes pour les débutants : la boucle for et la réutilisation des variables.



```
from turtle import *

longueur = 30
for i in range(10):
    forward(longueur)
    left(90)
    longueur = longueur+10*i
```

COURBES DE BÉZIER

Les courbes de Bézier ont été inventées en 1962 par Pierre Bézier, un ingénieur travaillant chez Renault. Elles sont aujourd'hui à la base des images vectorielles, et très utilisées en CAO, en synthèse d'image et pour le rendu des polices de caractères.

La définition mathématique des courbes de Bézier fait appel à des polynômes. Les plus utilisées sont celles qui correspondent à des polynômes de degré 3, mais ce ne sont pas celles présentées ici, par souci de simplicité.

Les courbes de Bézier présentées ici sont dites « *quadratiques* » (elles sont définies par des polynômes de degré 2) et ont été proposées par Paul de Faget de Casteljau, un ingénieur travaillant chez Citroën). Elles utilisent trois paramètres : le point de départ A, le point d'arrivée C et un point de contrôle B. Pour comprendre le fonctionnement d'une telle courbe, il faut imaginer que notre segment [A;C] est un fil de métal souple, qui est attiré par l'aimant qu'est le point de contrôle B. Plus une partie du fil se trouve près de l'aimant, plus elle est attirée et se déforme.

La courbe de Bézier de point de départ A, de point d'arrivée C et de point de contrôle B, est la limite d'une suite de lignes brisées, obtenues à l'aide d'un algorithme également inventé par Paul de Casteljau.

La ligne initiale est formée des segments [AB] et [BC].

À l'étape suivante, les trois points suivants sont construits :

- D milieu de [AB]
- E milieu de [BC]

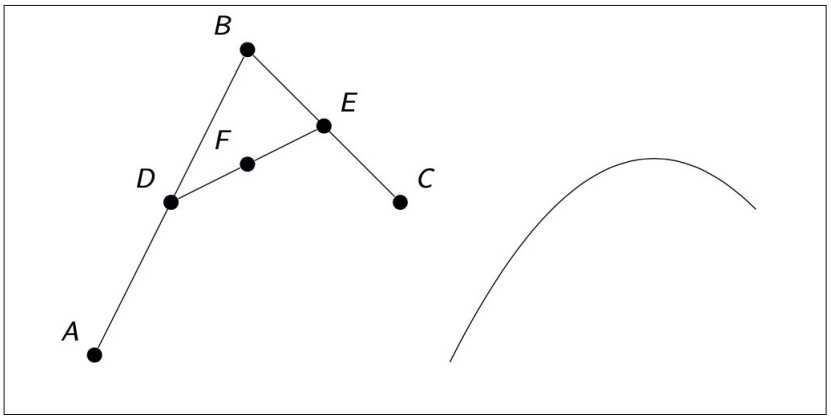


Figure 1

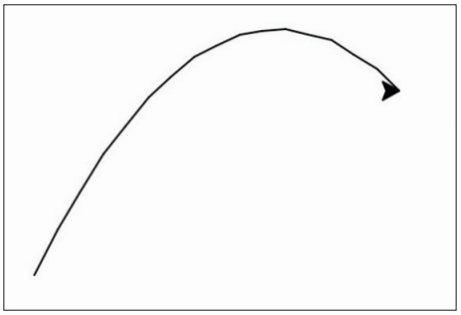


Figure 2

• F milieu de [DE]

Il faut noter que D et E sont des points auxiliaires, tandis que F est un point de la courbe finale.

Ensuite, il faut recommencer en remplaçant les points A, B, C par les deux triplets de points A, D, F et F, E, C. Après plusieurs itérations, le résultat est une courbe lisse, passant par A et C et tangente à [AB] et [BC]. Le résultat est déjà reconnaissable après 3 ou 4 itérations. **Figure 1**

Pour implémenter cet algorithme, une solution est de commencer par écrire une fonction nommée segment, qui trace un segment à partir des coordonnées de deux points.

```
from turtle import *

def segment(x1,y1,x2,y2) :
    up()
    goto(x1,y1)
    down()
    goto(x2,y2)
    up()
    return
```

Ensuite une autre fonction utile est le calcul des coordonnées du milieu de deux points à partir de deux points.

```
def milieu(x1,y1,x2,y2) :
    return ((x1+x2)/2.0,(y1+y2)/2.0)
```

À partir de ces deux fonctions, une des manières naturelles de coder cet algorithme est de réaliser une fonction récursive pour tracer la courbe de Bézier.

```
def bezier(x1,y1,x2,y2,x3,y3,n) :
    if n==0 :
        segment(x1,y1,x2,y2)
        segment(x2,y2,x3,y3)
    else :
        (a,b)=milieu(x1,y1,x2,y2)
        (c,d)=milieu(x2,y2,x3,y3)
        (e,f)=milieu(a,b,c,d)
        bezier(x1,y1,a,b,e,f,n-1)
        bezier(e,f,c,d,x3,y3,n-1)
```

Le résultat de la fonction bezier pour les points trois-points (0,0), (100,200) et (200,100) et une profondeur de 3 donne le graphique suivant (en appelant bezier(0,0,100,200, 200,100,3)) : **Figure 2**

L'algorithme de Casteljau est ici présenté pour construire des courbes de Bézier quadratiques, mais la construction se généralise assez facilement pour les courbes de Bézier de degré supérieur.

FLOCON DE VON KOCH

Le flocon de von Koch est une courbe fractale simple à tracer. Il s'agit d'un d'un polygone « creux » (non convexe). La figure de départ est un triangle équilatéral, et l'algorithme de construction s'applique à chaque côté du triangle. La construction est présentée étape par étape : pour passer d'une étape à la suivante, il faut diviser chaque segment en trois segments égaux. Ensuite, il faut remplacer le segment du milieu par les deux côtés d'un triangle équilatéral dont la base est ce segment du milieu, et orienté vers l'extérieur. À l'étape suivante, ce processus est réitéré sur les quatre nouveaux segments obtenus.

Cette construction correspond naturellement à un algorithme récursif, comme celui écrit ci-dessous.

```
from turtle import *

def Trait(n, L) :
    # Etat initial : plume baissée, direction trait, à gauche
    # Etat final :    plume baissée, direction trait, à droite
    if n == 1 :   forward(L)
    else :
        Trait(n-1, L / 3.0)
        left(60)
        Trait(n-1, L / 3.0)
        right(120)
        Trait(n-1, L / 3.0)
        left(60)
        Trait(n-1, L / 3.0)

N = int(input("Nombre de niveaux (entre 1 et 6) : "))
if (type(N) != int) or (N < 1) or (N > 6) :
    print ("Le nombre de niveaux ne correspond pas a la demande")
else :
    Long = 550    # Longueur du trait
    up()
    goto(-300, -180)
    down()
    width(2) # épaisseur du stylo
    left(60)
    for k in range(3) :
        Trait(N, Long)
```

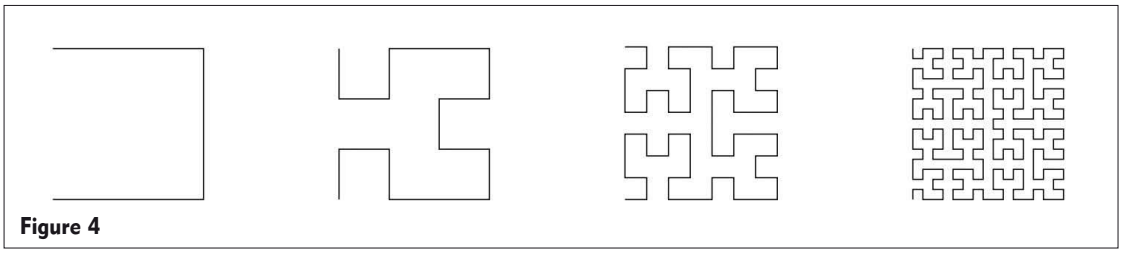


Figure 4

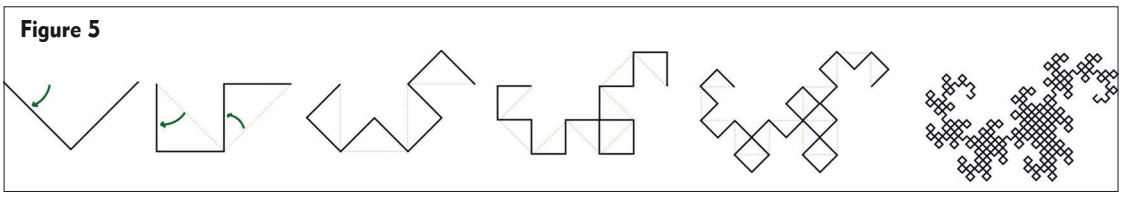
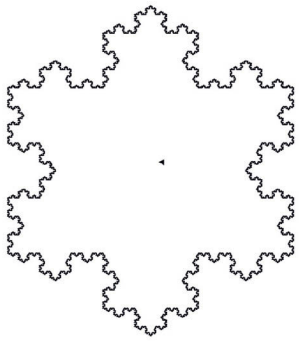


Figure 5



FLOCON DE VON KOCH

Figure 3

```
    right(120)
up()
goto(0, 0)
```

En théorie, la construction devrait être répétée indéfiniment pour obtenir une courbe fractale, car par définition, il s'agit de la courbe « limite », obtenue après une infinité d'étapes. Concrètement, on fixe préalablement la profondeur des appels récursifs. Dans le programme ci-dessous elle est limitée à 6, car d'une part le temps d'exécution est déjà long, et d'autre part, il n'y aurait plus beaucoup de différence visuelle en ajoutant des étapes. **Figure 3**

Il existe des variantes du flocon de von Koch, obtenues en modifiant les valeurs des angles du triangle ou bien la figure géométrique de base, et même en 2D (surfaces de von Koch).

POUR ALLER PLUS LOIN

Voici deux exemples supplémentaires de courbes fractales, la courbe de Hilbert et la courbe du dragon, dont il est possible de tracer des approximations à l'aide de la bibliothèque turtle.

Courbe de Hilbert

Cette courbe, inventée par le mathématicien allemand David Hilbert en 1891, est une ligne brisée, obtenue à partir de segments de droites orthogonaux. Le motif de base est composé de trois côtés d'un carré. À chaque étape, la longueur des côtés est divisée par 3. Chaque « coin » d'un motif de l'étape précédente est remplacé par le nouveau motif, selon la même orientation, et chaque « extrémité » est aussi remplacée par le nouveau motif, mais tourné d'un quart de tour, chacun dans un sens, de façon que les ouvertures soient vers l'extérieur. Il ne reste plus qu'à relier de proche en proche les extrémités des motifs pour obtenir la courbe suivante : **Figure 4**

La courbe fractale obtenue après une infinité d'itérations possède la propriété de passer par chacun des points du carré bâti à partir du motif initial.

Courbe du dragon

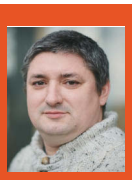
La courbe du dragon a été inventée par J.E. Heighway. Cette courbe fractale ne se recoupe jamais. En 1967, Martin Gardner l'a présentée dans sa chronique de jeux mathématiques du Scientific American, car sa construction est simple. À chaque étape, la courbe obtenue est composée de segments de droites qui suivent à angle droit. Au départ, il n'y a qu'un seul segment. Pour passer d'une étape à la suivante, en suivant la courbe, il s'agit de remplacer chaque segment rencontré par deux segments à angle droit en effectuant une rotation de 45° alternativement à droite puis à gauche. **Figure 5**

CONCLUSION

Nous espérons vous avoir convaincu de l'intérêt de la bibliothèque turtle de python pour la découverte de la programmation par le dessin, y compris pour des notions avancées comme la récursivité. Les codes Python pour les courbes de Hilbert et du dragon sont laissés en exercices au lecteur, pour qu'il puisse vérifier s'il a compris le fonctionnement de cette bibliothèque et la récursivité.

PROGRAMMATION DYNAMIQUE

Python est un langage fortement dynamique. Mais, dans les faits, qu'est-ce que cela signifie ? Qu'est-ce que cela peut nous apporter ? Grâce à ce dynamisme, Python permet de résoudre des problèmes de manière élégante et compacte là où les langages classiques nécessiteraient beaucoup de codes.



Philippe BOULANGER
Manager des expertises C/C++ et Python
www.invivoo.com

INVIVOO
BEYOND TECH

DÉFINITION

Si on se réfère à Wikipédia (https://fr.wikipedia.org/wiki/Langage_de_programmation_dynamique), la programmation dynamique est définie par :

> On utilise le terme **langage de programmation dynamique** en informatique pour décrire une classe de langage de haut niveau qui exécute au moment de l'exécution des actions que d'autres langages ne peuvent exécuter que durant la compilation. Ces actions peuvent inclure des extensions du programme, en ajoutant du code nouveau, en étendant des structures de données et en modifiant le système de types, cela pendant l'exécution du programme. Ces comportements peuvent être émulés dans pratiquement tous les langages de complexité suffisante, mais les langages dynamiques ne comportent pas de barrière, tel que le typage statique, empêchant d'obtenir directement ces comportements.

L'un des premiers langages dynamiques fut LISP. Basé sur la théorie du λ-calcul, ce langage a été un des outils prépondérants sur l'amélioration des techniques et langages de programmation : le premier langage-objet qui a eu du succès « Smalltalk » a été écrit en LISP. L'une des particularités du LISP est qu'un programme est une donnée et qu'une donnée peut devenir un programme : c'est une des raisons pour laquelle il a été à la base des recherches en intelligence artificielle. Nombre de langages sont aujourd'hui dynamiques à divers degrés ; voici certains parmi les plus connus et les plus utilisés :

- JavaScript : pour le web dynamique
- C# (en version 4.0 ou supérieure)
- Erlang
- Python : qui est utilisé aussi bien dans les classes de seconde que dans les entreprises de pointe comme Google ou Dropbox
- Lua : utilisé dans les moteurs de jeux vidéo

Si le LISP reste un langage que j'apprécie notamment grâce à sa forme préfixée pour le calcul formel ; Python l'a remplacé, dans les faits, pour les tâches quotidiennes nécessitant des solutions rapides à mettre en œuvre.

LES VARIABLES

Les variables sont des éléments essentiels d'un programme : elles servent à stocker les états. La notion de « dynamisme » pour les variables peut prendre différentes formes :

- Typage dynamique
- Création de variables à la volée

Typage dynamique vs typage statique

Les détracteurs de Python avancent souvent le fait que le typage statique est plus sûr : les problèmes de type sont détectés à la compilation plutôt qu'à l'exécution.

En effet dans les langages statiques (Java, C ou C++ par exemple), les variables sont déclarées et typées avant d'être utilisées permettant au compilateur de faire nombre de tests avant de générer le binaire. Par exemple si on prend le petit programme suivant :

```
double linear_interpolation( double a, double b, double x )
{
  double v = ( 1 - x ) * a + x * b;
  return v;
}


int main()
{
  // appel correct qui compilera
  double r1 = linear_interpolation( 10, 20, 0.5 ); // appel correct

  // ne compilera pas
  std::string s1( "toto" ), s2( "tata" );
  double r2 = linear_interpolation( s1, s2, 2 ); // erreur de type
  return 0;
}
```

Il générera comme erreur de compilation :

```
C:\personnel\private\tests\essai\main.cpp:19:39: error: cannot convert 'std::__cxx11
::string' {aka 'std::__cxx11::basic_string<char>'} to 'double'
   double r2 = linear_interpolation( s1, s2, 2 ); // erreur de type
                                     ^~
```

En Python, par défaut, le type des variables n'est défini qu'au moment de l'affectation. Et de ce fait les problèmes ne sont vus qu'à l'exécution… En reprenant l'exemple développé en C++ et en le portant en Python on obtient :

```
def linear_interpolation( a, b, x ):
  r = ( 1 - x ) * a + x * b
  return r

r1 = linear_interpolation( 10, 20, 0.5 )
print( type(r1), r1, sep=' : ' )
r2 = linear_interpolation( "toto", "tata", 2 )
print( type(r2), r2, sep=' : ' )
```

Le code s'exécute sans erreur et affiche les valeurs suivantes :

- <class 'float'> : 15.0
- <class 'str'> : tatatata

En Python on peut multiplier un entier par une chaîne de caractères… Le code s'exécute, mais, dans le cas présent, le résultat n'est pas celui attendu !

L'autre avantage du typage statique est la performance : les

variables sont associées à un espace mémoire par une adresse et une taille dépendant du type. L'accès à la donnée est donc direct. Dans le cadre du typage dynamique, on doit passer par un « dictionnaire » qui lie le nom de la variable à un espace mémoire alloué (dynamiquement) : l'accès est donc le coût d'une recherche dans cette structure de donnée.

Création dynamique de variables

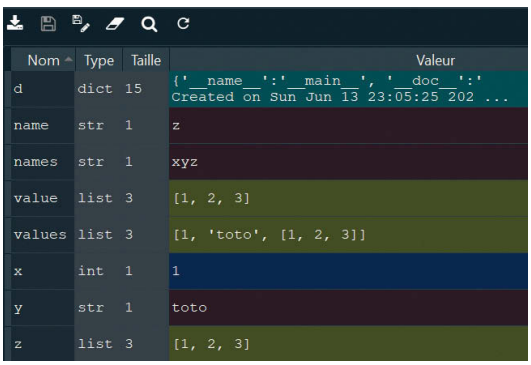
Python ne dispose pas que d'un typage dynamique, il permet aussi de créer des variables à la volée. En effet, nous avons un accès direct aux dictionnaires des variables globales ou locales :

- globals() : retourne le dictionnaire des variables globales
- locals() : retourne le dictionnaire des variables locales

Avec le code suivant :

```
names = "xyz"
values = [ 1, "toto", [ 1, 2, 3 ] ]
d    = globals()

for name, value in zip( names, values ):
  d[ name ] = value
```

on obtient la liste des variables suivante dans Spyder :



Nom	Type	Taille	Valeur
d	dict	15	{'__name__':'__main__', '__doc__':' Created on Sun Jun 13 23:05:25 202 ...
name	str	1	z
names	str	1	xyz
value	list	3	[1, 2, 3]
values	list	3	[1, 'toto', [1, 2, 3]]
x	int	1	1
y	str	1	toto
z	list	3	[1, 2, 3]

Ajout de variables à un objet

En python, tout est objet. Et les objets peuvent être étendu par ajout d'attributs. Nous disposons des fonctions suivantes pour manipuler la structure interne des objets Python :

- **def** *hasattr*(*obj*, *attr_name*)
  Le résultat est **True** si la chaîne *attr_name* est le nom d'un des attributs de l'objet, sinon **False**. L'implémentation appelle **getattr**(object, name) et regarde si une exception **AttributeError** a été levée.
- **def** *getattr*(*obj*, *attr_name* [, *default*])
  Retourne la valeur de l'attribut associé au nom dans *attr_name* de l'objet *obj*. *attr_name* doit être une chaîne de caractères. Si la chaîne est le nom d'un des attributs de l'objet, le résultat est la valeur de cet attribut. Par exemple, *getattr*(x, 'foobar') est équivalent à x.foobar. Si l'attribut n'existe pas, et que *default* est fourni, il est renvoyé, sinon l'exception **AttributeError** est levée.
- **def** *setattr*(obj, attr_name, attr_value)
  C'est la fonction complémentaire de getattr. Les arguments sont : un objet Python, une chaîne de caractères, et une valeur à associer à l'attribut. La chaîne peut nommer un attribut existant ou un nouvel attribut. La fonction assigne la valeur à l'attribut, si l'objet l'autorise. Par exemple, *setattr*(x, 'foobar', 123) équivaut à x.foobar = 123.

Cela peut sembler peu de chose, mais c'est extrêmement utile : le module argparse en fait usage pour retourner args :

```
import argparse

parser = argparse.ArgumentParser( description = 'Process some integers.' )
parser.add_argument( 'integers',
                     metavar = 'N',
                     type    = int,
                     nargs   = '+',
                     help    = 'an integer for the accumulator' )
parser.add_argument( '--sum',
                     dest    = 'accumulate',
                     action  = 'store_const',
                     const   = sum,
                     default = max,
                     help    = 'sum the integers (default: find the max)' )

args = parser.parse_args()
```

'args' est initialisé en utilisant setattr à l'intérieur de la fonction parse_args. Cette façon de programmer permet de créer à la volée des variables.

Ce mécanisme permet de mettre en place un système qui chargerait une configuration à partir d'un fichier et créerait des variables. Nous avons une application qui s'exécute sur plusieurs environnements (production, intégration, développement). Pour chaque environnement nous avons un fichier « config.txt » contenant les paramètres utiles (connexion au serveur, à la base de données, les répertoires utiles, etc.) :

```
# connexion
connection.server.hostname=192.77.32.1
connection.server.port=8080
connection.server.account.user=zeus
connection.server.account.password=jupiter
connection.db.name=MY_DB
connection.db.account.user=aphrodite
connection.db.account.password=venus

# directories
directory.log=C\Temp\log
directory.input=C:\Temp\input
directory.output=C:\Temp\output
```

Avec le programme suivant, nous allons pouvoir lire ce fichier et nous servir des valeurs lues :

Code complet sur programmez.com & github

La fonction « load_file » lit le fichier ligne par ligne et la fonction « add_variable » crée les variables et les attributs à la volée.

EVALUATION DYNAMIQUE

Nous avons appris à créer des variables et des attributs dynamiquement. Mais est-on capable d'évaluer des formules ou du code dynamiquement ? Si j'ai une formule récupérée dans un formulaire, puis-je en évaluer la valeur en fonction ?

Évaluation de formule

C'est en 2000 que j'ai eu la première fois besoin d'évaluer dynamiquement des formules. Je travaillais sur un logiciel de CAO et je créais un plug-in en C++ qui allait permettre de

créer des objets 3D à partir d'équations paramétriques saisies par un utilisateur dans une fenêtre de l'application. Créer une telle fonctionnalité m'avait demandé beaucoup de codes, mais en Python cela s'avère beaucoup plus facile ; en effet, il existe une fonction « eval » qui facilite la tâche :

```
from math import *

formula = input( "formule?" )
a     = float( input( "a?" ) )
b     = float( input( "b?" ) )
nb    = int( input( "nb?" ) )

h = ( b - a ) / nb
for i in range( nb + 1 ):
  x = a + i * h
  print( "f(", x, ") =", eval( formula ) )
```

En l'exécutant, on obtient :

```
formule?x*x
a?0
b?1
nb?10
f( 0.0 ) = 0.0
f( 0.1 ) = 0.010000000000000002
f( 0.2 ) = 0.04000000000000001
f( 0.30000000000000004 ) = 0.09000000000000002
f( 0.4 ) = 0.16000000000000003
f( 0.5 ) = 0.25
f( 0.6000000000000001 ) = 0.3600000000000001
f( 0.7000000000000001 ) = 0.4900000000000001
f( 0.8 ) = 0.6400000000000001
f( 0.9 ) = 0.81
f( 1.0 ) = 1.0
```

Les fonctions paramétriques

En Python, une fonction est un objet comme un autre : nous pouvons créer des alias à des fonctions et nous pouvons retourner une fonction d'une fonction. Cette fonctionnalité peut sembler obscure, au premier abord, et peu utile, mais c'est, en fait, une fonctionnalité extrêmement utile de Python : elle est notamment à la base des décorateurs.

Prenons l'exemple suivant :

```
def f( a ):
  def _f( x ):
    return x + a
  return _f

h = f(3)
print( h, end='\n\n' )

for i in range(5):
  print( h(i) )
```

Lorsque l'on fait « h=f(3) », h est une fonction : <function f.<locals>._f at 0x0000023C1B386DC0>. Et dans la boucle, le programme affiche bien la valeur i+3 à chaque itération. « f » est une fonction qui crée une nouvelle fonction (que l'on stocke dans « h ») qui dépend des paramètres de « f ». Certes c'est moins dynamique que la fonction « eval », mais cela permet d'adapter du code en fonction de conditions. Cela permet de construire dynamiquement des fonctions spécialisées ou des fonctions permettant de modifier des fonctions existantes via le mécanisme des décorateurs. Supposons que nous souhaitions rajouter une fonctionnalité de logging :

```
def log( func ):
  def wrapper( *args, **kwargs ):
    print( F"enter in {func.__name__}" )
    res = func( *args, **kwargs )
    print( F"exit from {func.__name__}" )
    return res

  return wrapper

@log
def compute( x ):
  return x*x*x

print( compute(3) )
```

Dans ce cas-là, « wrapper » dépend de la fonction passé en paramètre de la fonction « log » et la fonction paramétrique est créée en appliquant « @log » à la fonction « compute ».

Création de module

Un module peut être un simple fichier Python. Et la commande « import » permet de charger un module. La question que l'on peut se poser est : peut-on créer dynamiquement un module et le charger ?

À quoi pourrait servir ce type de fonctionnalité me direz-vous ? Plaçons-nous dans un contexte embarqué avec des ressources limitées comme un petit robot piloté par un programme Python. Notre seul moyen de communication avec notre robot est via un réseau non-filaire (un wifi par exemple). La tâche que doit accomplir notre robot est programmée dans un module appelé « mission ». Sa tâche terminée le robot a encore de l'autonomie. Il nous faut donc mettre à jour sa mission :

- envoyer un fichier texte contenant la nouvelle mission
- recharger le module via un appel à importlib.reload
- exécuter la fonction principale du module « mission »

Essayons le code suivant :

```
import importlib

while True:
  formula = input( "f(x)=" )
  if len(formula) == 0:
    break

  texte = F"""
from math import *

def f(x):
  return {formula}
  """

  with open( "my_module.py", "w" ) as file:
    file.write( texte )
```

```
try:
  importlib.reload( my_module )
except:
  import my_module
print( my_module.f(3) )
```

Si je l'exécute et, qu'à la question « f(x)= » je rentre « x*x » comme formule, j'obtiens l'affichage de 9 ce qui est la bonne réponse. Si, à l'itération suivante, je rentre « x*sin(x) » comme formule j'obtiens 0.4233600241796016 comme résultat (ce qui est bien 3*sin(3)).

Compilation : convertir un texte en code exécutable

Python fournit différentes fonctions permettant d'accéder directement à l'exécution ou à la compilation (en byte-code) d'un répertoire, d'un fichier ou d'une chaîne de caractères. Nous allons nous concentrer uniquement sur la fonction « exec ».

```
from math import *

fct = input( "g(x)? " )
code = F"""
def g(x):
  return {fct}
"""

exec( code )
for i in range( 10 ):
  print( "g(", i, ")=", g( i ) )
```

En tapant « 5*x*x » à la question « g(x)? », nous créons dynamiquement la fonction g grâce au code contenu par la chaîne de caractères « code », puis nous évaluons la fonction g avec des arguments comme si c'était une fonction Python normale. La fonction « **exec** » se base sur le contenu de « ***globals***() » et « ***locals***() ». Il existe une fonction « ***compile*** » qui permet de compiler du code avec des options d'optimisation (voir https://docs.python.org/3/library/functions.html#compile pour plus de détails). Le résultat étant un objet « code » qui peut ensuite être exécuté via « **exec** »…

SÉRIALISATION/DESERIALISATION

Sauvegarder et restaurer les données est une nécessité pour toute application. Mais cela devient aussi un impératif si l'on doit échanger des données via un réseau. Convertir une donnée en un flux de bytes est appelé marshalling en anglais (sérialisation) et l'action inverse est appelé unmarshalling (désérialisation). Avoir ce type de fonctionnalité disponible participe aux créations dynamiques.

pickle

Python propose le module pickle qui est le plus simple à mettre en œuvre tout en restant efficace pour sauvegarder les combinaisons des types simples suivants :

- **NoneType**
- **bool** : les booléens à valeur **True** ou **False**
- **int** : les entiers longs
- **float** : nombres flottants en double précision
- **complexe** : les nombres complexes
- **str** : chaînes de caractères
- **list** : listes
- **dict** : dictionnaires
- **tuple** : tuple
- **set** : ensemble

Il permet de sauvegarder et/ou restaurer des données.

dill

« dill » est une extension que l'on peut télécharger sur PyPI via la commande : « *pip install -U dill* ». C'est une version améliorée de pickle car elle permet de sérialiser/désérialiser des types de données supplémentaires :

- des instances de classes
- fonctions avec yield
- fonctions lambda
- nested functions : fonctions paramétriques
- un objet « code »

Voici un exemple de sérialisation de fonction :

```
import dill as pickle
from math import *

def f(x):
  return x*sin(x)
with open( "c:/temp/objet.pickle", "wb" ) as file:
  pickle.dump( f, file )
```

Et voici la désérialisation associée :

```
import dill as pickle

with open( "c:/temp/objet.pickle", "rb" ) as file:
  f = pickle.load( file )

for i in range( 11 ):
  x = 0.1 * i
  print( F"f({x}) = {f(x)}" )
```

Cette méthode est intéressante car elle permet de transmettre du code sous un format binaire entre un client et un serveur. Prenons le cas d'une application client lourd de type CAO (modélisation mécanique 3D) qui contient des fonctionnalités gratuites et d'autres payantes. Les fonctionnalités sont trop lourdes en données à transférer ou en calculs pour les exécuter du côté serveur : on souhaite les exécuter côté client. Afin de réduire le risque de piratage des fonctions payantes, nous pouvons mettre en place la mécanique suivante :

- clic sur la fonctionnalité payante
- connexion au serveur
- vérification de la licence ou du paiement
- téléchargement de la fonctionnalité via un fichier « dill »
- exécution de la fonctionnalité
- suppression du fichier téléchargé

CONCLUSION

Python se rapproche du LISP pour sa capacité à faire évoluer son code et ses données. Cela permet de développer en Python des fonctionnalités avancées qui nécessiterait d'interfacer un interpréteur dans des programmes écrits dans d'autres langages.

Sallah Kokaina

Ingénieur en informatique et auteur du livre « Software Craftsmanship : L'art du code et de l'agilité technique en entreprise » (Éditions ENI 2019). Sur ces 12 dernières années, il a tenu différents rôles (Développeur, CTO, Qualiticien, Manager Technique…) au sein de différentes structures et y a mis en place différentes solutions basées notamment sur Scala, Kotlin et Java. Twitter (@koksbox).

Zoom sur l'interopérabilité Kotlin et Java

PARTIE 2

Depuis l'officialisation de Kotlin en tant que langage de développement pour les apps Android en 2019, ce langage, créé en 2011 par Jetbrains, a gagné en popularité et en adoption au sein de la communauté des développeurs pro-JVM.

Conçu pour être totalement interopérable avec Java, il apporte un ensemble d'avantages en productivité de développement. Ainsi, il est possible d'appeler autant du code Java depuis Kotlin que du code Kotlin depuis Java. Cela en fait donc un atout pour la programmation fonctionnelle et orientée objet en entreprise. On y retrouve une gestion avancée de null-safety, l'intégration d'extension de classes ou encore des optimisations en temps de compilation avantageux par rapport à des langages JVM similaires tels que Scala. Aujourd'hui, à sa version 1.5+, Kotlin fournit une interopérabilité totale avec Java 6+, toutefois la compilation en 1.6 est désormais obsolète, privilégiant la version 1.8 par défaut. Sans plus attendre, explorons-en les caractéristiques qui expliquent l'intérêt de migrer de Java vers Kotlin tout en gardant le meilleur des deux mondes.

Du code Java depuis du code Kotlin

C'est le sens le plus naturel et qui permet de bénéficier de la modernité syntaxique de Kotlin tout en réutilisant l'ensemble du capital de librairies et fonctions écrites en Java (JUnit, Guava, Spring ou encore la SDK Android). Nous allons illustrer différentes propriétés mettant en lumière l'invocation de Java au sein d'un code Kotlin.

Invoquation de librairies

Il est possible d'appeler des librairies écrites en Java depuis Kotlin. Prenez par exemple le code ci-dessous.

```
import com.google.common.collect.Lists
import java.util.ArrayList

fun main(args: Array<String>) {

  // 1. Libraries standard
  val months = ArrayList<String>()  // java.util.ArrayList
  months.addAll(arrayOf("Janvier", "Février", "Mars"))

  //2. Altération & Itération sur ArrayList via iterator()
  arrayList[2] = arrayList[0]  // Equivalent en Java à arrayList.set(2, array
List.get(0))
  for (item in arrayList) { println(item) } // ("Janvier", "Février", "Janvier")

  //3. Librairie tierce (Google Guava)
  val numbers = Lists.newArrayList(4, 2, 22, 43)
  println(numbers.sum())  // 71
}
```

On retrouve les illustrations suivantes:

- Intégration Librairie Standard Java: On peut voir qu'il est aisé d'appeler la librairie standard Java depuis Kotlin. On peut relever que la création d'une nouvelle instance de java.util.ArrayList ne nécessite pas l'opérateur new. En effet, Kotlin ne dispose pas de l'opérateur new. Ainsi, pour créer une instance de classe, il suffit d'appeler le constructeur comme on le ferait avec toute autre fonction.
- Intégration syntaxique: Les "syntaxics sugar" de Kotlin, tels que l'opérateur d'indexation [], s'intègrent de façon naturelle avec les classes de Java. Ainsi, l'opérateur d'indexation encapsule respectivement les méthodes set et get quand il s'agit d'affecter et d'accéder aux valeurs de la liste.
- Intégration Librairies Java tierces: Les libraires tierces, telles que Google Guava n'échappent pas à la règle. Elles s'intègrent et peuvent être invoquées à l'image de la librairie Standard Java. Dans l'exemple ci-dessus, on a pu instancier une liste avec la classe Lists de Guava et en enrichir les fonctionnalités avec la fonction sum. Ceci est possible grâce à la notion d'extension que l'on abordera plus tard dans l'article. À ce stade, on notera que la fonction sum est un rajout de Kotlin à l'interface Iterable, implémentée par la classe ArrayList. Magique, non ?

Gestion de setters et getters

Une autre force de Kotlin est la gestion implicite des asseseurs (getters) et mutateurs (setters). En effet, au sein de Kotlin, il n'est pas nécessaire d'appeler le getter ou setter pour modifier une propriété telle que la bonne pratique d'encapsulation le recommande en Java. Et en passant, c'est l'une des pratiques qui rend le code java particulièrement verbeux.
Ainsi, la modification ou la lecture de la valeur d'un attribut de classe mutable (var) se fait directement en manipulant la propriété en question, hello dans l'exemple ci-dessous.

```
//Kotlin
class GetSet(var hello: String)

fun main(args: Array<String>) {
  // Kotlin
  val gs = GetSet("Hello !")
  print(gs.hello) // hello !

  gs.hello = "toto"
  print(gs.hello) // toto
}

// Equivalent à GetSet(var hello: String)
class GetSetEq(hello: String) {
  var hello : String = hello  // hello représente un backing field
        get() { return field }      // méthode redondante
        set(value) { field = value }  // méthode redondante
}
```

En effet, pas besoin de déclarer les setteurs et getteurs, ils sont générés automatiquement grâce aux mot-clés val et var. Et comme illustré précédemment, les propriétés de classe sont publiques par défaut.

```
/ Java
public class GetSet {
  private String readOnly = "Only getter defined";
  private String writeOnly = "Only setter defined";
  private String readWrite = "Both defined";

  public String getReadOnly() { return readOnly; }

  public void setWriteOnly(String writeOnly) { this.writeOnly = writeOnly; }

  public String getReadWrite() { return readWrite; }
  public void setReadWrite(String readWrite) { this.readWrite = readWrite; }
}

// Kotlin
private val gs = GetSet()
println(gs.readOnly)        // Read-only attribute acts like a val property

gs.readWrite = "I have both"  // Read-write attribute acts like a var propertyprintln
(gs.readWrite)

gs.setWriteOnly("No getter")  // Write-only properties not supported in Kotlin
```

Cette conversion automatique vers une propriété de classe Kotlin fonctionne du moment que la classe Java fournit une méthode sans paramètres commençant par get et éventuellement une méthode à paramètre unique commençant par set. Cela fonctionne également pour les expressions Boolean où le getter commence par it au lieu de get, mais il n'y a actuellement aucun mécanisme de ce type si elles commencent par has ou d'autres préfixes. Les propriétés de type WriteOnly (écriture seule) ne sont actuellement pas supportées par Kotlin, raison pour laquelle setWriteOnly ne peut être appelée avec une propriété de syntaxe comme étant un WriteOnly.

Gestion de null

Kotlin a pour avantage une gestion avancée des valeurs nulles. Ceci offre aux programmes implémentés dans ce langage la caractéristique de null-safety. Cette caractéristique est l'une des raisons principales pour laquelle Kotlin est devenu à ce jour le langage préféré pour la réalisation de solutions mobiles, offrant par ailleurs le même avantage qu'en programmation iOS via Swift. Vous l'aurez compris, moins de NullPointerException, moins de crashs !
Par exemple, l'opérateur ? permet d'indiquer qu'une valeur peut être éventuellement nulle et ainsi en sécuriser l'accès.

```
// Java
public static String test() { return "Je peux être null"; }
public static String testNull() { return null; }

//Kotlin

//1. Nullsafe: Gestion explicit de null,
val test: String? = test()      // Explicit type: String?
println(nullable?.length)       // 15

val testNull: String? = testNull()   // Explicit type: String?
println(nullable2?.length)        // null

//2. Pas Nullsafe: Infération avec NullPointerException (NPE)
val infTest = test()          // Inferred type: String (by default)
println(str.length)           // 15

val infTestNull = testNull()    // Inferred type: String (by default)
println(str.length)           // NullPointerException
```

Toutefois, l'interopérabilité entre les deux langages introduit des problèmes de mapping et de gestion de null pour les objets provenant de code Java.
En règle générale, les types primitifs (int, boolean, long,..) sont associés par le compilateur au type Kotlin non null, en l'occurrence (kotlin.Int, kotlin.Boolean, kotlin.Long,...), définis au sein de la librairie standard de Kotlin. Mais les types non-primitifs (Integer, Boolean, Long, String..), seront traduits, sans déclaration explicite, en leur équivalent Kotlin non-null. Chose qui peut créer un NPE tel qu'illustré par l'infération par le compilateur du type String, au lieu du type null-safe : String?. Il est ainsi fortement recommandé d'expliciter le type retourné à l'invocation d'une méthode Java. Cependant, avoir à expliciter le type ou encore traiter toutes les valeurs comme éventuellement nullable avec l'opérateur ? peut rapidement encombrer le code avec des vérifications inutiles. Pour réduire ce besoin, Kotlin supporte les annotations @Nullable et @NotNull prévenant de la JSR 305, ou encore @NonNull de la JSR 308 pour les types génériques.

```
// Java
public static @Nullable String nullable() { return null; }
public static @NotNull String nonNull() { return "Could be null, but with warning"; }

// Kotlin
val s1 = nullable()  // Inferred type: String?
val s2 = nonNull()   // Inferred type: String
```

Point important: Les objets annotés @NotNull en Java et qui contiennent une référence nulle par erreur pourront toujours causer des NullPointerException en Kotlin. C'est pour cette raison qu'il est nécessaire de redoubler de vigilance sur les inférences de type en intégrant les deux langages. Pour finir, on notera que les IDE tels qu'Android Studio et IntelliJ peuvent afficher des warnings dès lors que les annotations JSR 305/308 sont utilisées dans le code Java intégré. Cela permet d'anticiper les risques, voire les besoins de gestion de null-safety.

Conflits d'identifiant syntaxiques

Les mot-clés réservés sont un autre point clé à garder dans le viseur. Comme on a pu le voir dans les précédents exemples, Kotlin possède un ensemble de keywords syntaxiques: val, var, when, fun... Ces mot-clés déclaratifs en Kotlin ne le sont pas en Java, du moins pas dans certaines versions. Il est ainsi possible en Java, par exemple dans la version 8, de déclarer une variable portant le même nom. L'exemple ci-dessous l'illustre :

```
// Java
class KeywordsAsIdentifiers {
   public int val = 100;
   public Object object = new Object();
   public boolean in(List<Integer> list) { return true; }
   public void fun() { System.out.println("This is fun."); }
}
```

Pour les invoquer en Kotlin, il est nécessaire d'utiliser des backticks (`).

```
// Kotlin
val kai = KeywordsAsIdentifiers()
println(kai.`val`) // 100
println(kai.`object`) // java.lang.Object
println(kai.`in`(listOf(1, 2, 3))) // true
println(kai.`fun`()) // This is fun.
```

Pas de panique, dans la mesure où ce cas de figure devrait se produire, certains IDEs, tels qu'Intellij, intègrent une gestion de ces mots-clés. Ainsi, les backsticks seront rajoutés automatiquement.

Gestion de SAM

En Java, on définit depuis la version 8, une interface fonctionnelle comme une interface contenant une seule méthode abstraite. Les interfaces fonctionnelles sont la base des expressions lambda. On peut énumérer différentes interfaces dans la librairie standard de Java : Comparator, Consumer, Collector.

```
// Java
interface Producer<T> {  // SAM interface (single abstract method)
 T produce();
}

// Kotlin
private val creator = Producer { BigDecimal(9000) }
// Inferred type: Producer<BigDecimal>
```

L'instrumentalisation de ce type d'interface depuis un code Kotlin, ne nécessite pas la création d'une inner-classe ou encore d'un objet. Il suffit simplement d'invoquer une Lambda telle qu'illustrée dans l'exemple ci-dessus. Et comme on le constate, la syntaxe est plus légère avec l'inférence de type en prime.

Operators functions

Par conception, la surcharge d'opérateurs arithmétiques en Java n'est pas supportée. Il n'est pas possible de soustraire ou bien d'additionner des Objets. Mais dans les langages JVM modernes tels que Scala ou Kotlin, c'est une pratique courante. Le plus amusant dans tout ça, est que dans le cadre de l'interopérabilité entre Java et Kotlin, on peut conserver la même richesse syntaxique en manipulant des Objets Java depuis Kotlin. Par exemple, Il suffit de définir au sein de la classe Java les méthodes dites operator telles que plus, minus, inc ou div. Cela permet ainsi d'appliquer depuis Kotlin des opérateurs arithmétiques aux instances de l'objet Java.

```
// Java
public class Box {
   private final int value;

   public Box(int value) { this.value = value; }
   public Box plus(Box other) { return new Box(this.value + other.value); }
   public Box minus(Box other) { return new Box(this.value - other.value); }
   public int getValue() { return value; }
}

// Kotlin
val value1 = Box(19)
val value2 = Box(37)
val value3 = Box(14)
val result = value1 + value2 - value3  // Uses 'plus' and 'minus' as operators
println(result.value)  // 42
```

Du code Kotlin depuis du code Java

Il est tout aussi possible d'embarquer une librairie de code Kotlin au sein d'un projet Java. Bien que ce sens soit moins conventionnel, il n'en est cependant pas dénué d'intérêt. Vous pouvez par exemple avoir un projet Legacy Android écrit en Java dans lequel vous voulez réutiliser un module écrit pour un projet plus récent réalisé en Kotlin, chose qui vous permet de capitaliser sur une base de code existante.

Le compilateur de Kotlin traduit toutes les instructions en Java avant de les compiler en bytecode JVM. Au sein de l'IDE Intellij IDEA, Il est possible de voir comment un fichier Kotlin sera traduit en Java. Pour le faire, il suffit d'ouvrir le raccourci de commande (Shift+Shift) et de saisir '**skb**' (Show Kotlin Bytecode), puis de cliquer sur "Décomplier".

Code complet sur programmez.com & github

Une simple déclaration data class Personne(...) en Kotlin va ainsi être traduite en une classe Java intégrant:

- Les getters et setters des propriétés de classe : On y verra la création de getters et setters pour les propriétés en var (i.e age), et que des getters pour les propriétés déclarées en val. En plus, le compilateur rajoute le modificateur final pour les propriétés immuables nom et adresse.
- La gestion de hashcode, equals, copy et toString, offrant ainsi une implémentation par défaut de ces fonctions.
- Les annotations de la **JSR 305** illustrant la gestion des nulls, tel que décrit dans la section précédente.
- Les appels à **Intrinsics**, insérés par le compilateur Kotlin et permettant de faire des vérifications supplémentaires au runtime.
- La création de méthodes dites synthétiques. Celles-ci permettent de gérer les invocations depuis Kotlin, tel que l'appel du constructeur Personne sans fournir l'âge qui prend 10 comme valeur par défaut, chose qui n'est pas supportée par Java.

Ceci est un aperçu du nombre de lignes de code que l'on peut économiser à l'aide de Kotlin. On peut désormais comprendre que certains concepts propres à Kotlin n'ont pas nativement leur pareil en Java (i.e extensions, file-level functions, file-level class, reified inline functions...), que le code Java généré peut parfois s'avérer faiblement optimisé, et malheureusement engendrer des collisions de noms.

Dans ce dernier cas, quand on passe d'un univers Kotlin où il est possible de créer plusieurs classes dans un même fichier, à un univers Java dénué de cette flexibilité, on constatera en fonction des cas que des classes sont générées avec des noms par défaut.

```
// sampleName.kt
package com.example

class FileLevelClass                // génère la classe FileLevelClass.java
object FileLevelObject              // génère FileLevelObject.java en tant
 que Singleton
fun fileLevelFunction() {}          // s'insère dans la classe générée
 SampleNameKt
val fileLevelVariable = "Usable from Java"  // s'insère dans la classe
 générée SampleNameKt
```

Mais pour notre plus grand bonheur, Kotlin vient avec un ensemble de moyens et d'annotations pour contribuer à influencer le code Java généré.

Annotations

Kotlin fournit un ensemble d'annotations pour ajuster le comportement du compilateur et ainsi maîtriser le bytecode JVM qui sera généré. Dans cette section, nous allons en illustrer quelques-unes:

@JvmField

Par défaut, Kotlin expose les propriétés de classe via des getters et setters publics, ajustant à private la visibilité des propriétés concernées.

Code complet sur programmez.com & github

Comme l'exemple ci-dessus le montre, l'annotation **JvmField** permet d'exposer la propriété en tant que champ public. La génération d'assesseurs d'encapsulation est ainsi évitée pour la propriété exposed.

@file:JvmName & @file:JvmMultifileClass

Par défaut, tout fichier Kotlin est transcrit en au moins une classe Java contenant le suffixe Kt. Un fichier Exemple.kt donnera lieu à la création d'un fichier Java nommé ExampleKt.java. Si le fichier Exemple.kt contient des classes et des objets avec un nom distinct, Kotlin produira autant que fichier de classe que nécessaire. (ref. section intro). Mais dans le cas où le projet Java contient déjà une classe portant le même nom (Exemple), l'annotation **file:JvmName** offre la possibilité de contrôler le nom final que portera la classe, réduisant ainsi le risque de collision.

```
// sampleName.kt
@file:JvmName('Utils')

package com.example

class FileLevelClass                    // Génère la classe FileLevelClass.java
object FileLevelObject                  // Génère FileLevelObject.java en
 Singleton
fun fileLevelFunction() {}              // Membre de la classe Utils
val fileLevelVariable = "Usable from Java"  // Membre de la classe Utils
```

En complément, l'annotation **file:JvmMultifileClass** permet de donner le même nom Java à plusieurs fichiers Kotlin. Cette approche n'est toutefois pas recommandée, dans la mesure où elle ouvre la porte à des collisions de noms.

Code complet sur programmez.com & github

@JvmStatic

Cette annotation est spécifique aux instances de type object. Au sein de Kotlin, ceux-ci peuvent être de type top-level ou companion.

```
// Kotlin
object Cache { // Top level
  fun cache(key: String, obj: Any) { … }
}

class Coin {
  companion object Factory { // Companion level
    fun produceCoin() { … }
  }
}
```

Un objet, censé contenir les membres statiques, sera transcrit en un singleton contenant une propriété INSTANCE au sein de son équivalent Java. Et contrairement aux fonctions déclarées au niveau file-level, les méthodes déclarées au sein d'un object ne seront pas par défaut converties en méthodes static. Par conséquent, il sera nécessaire d'accéder à l'instance pour les invoquer.

```
// Au sein d'une méthode Java
Cache.INSTANCE.cache("car", new Coin());  // Pas joli
(new Coin()).produceCoin();               // Pas joli

Coin.Factory.produceCoin();               // Pas possible
Cache.cache("supercar", new Coin());      // Pas possible
```

En intégrant l'annotation **JvmStatic**, on peut instruire au compilateur de permettre un accès dit static à ces fonctions, comme on s'attendait à pouvoir le faire en Java. L'exemple ci-dessous en est une illustration:

```
// Kotlin
class Coin {
  companion object Factory { // Companion level
    @JvmStatic fun produceCoin() { … }
  }
}

// Inside a Java method
Coin.Factory.produceCoin();               // désormais possible
Coin.produceCoin();                       // possible aussi
```

@JvmName

Comme on a pu le voir précédemment, il n'est pas nécessaire de créer les getters et setters en Kotlin. Ceux-ci sont automatiquement générés. En plus, ils deviennent de facto le point d'entrée principal pour modifier un attribut de classe depuis Java. Par effet de bord, les attributs de classe de type Boolean génèrent aussi des assesseurs avec le préfixe get. Toutefois, si le préfixe d'un champ est is, le getter généré garde is comme préfixe, que ce soit pour un type Boolean ou autre. Concernant le setter, le préfixe is sera remplacé par un set.

Code complet sur programmez.com & github

Cependant, cette génération automatique de setters et getters peut avoir des effets indésirables. Elle pousse à adapter les conventions de nommage côté Java et dans le pire des cas, à briser la compilation dans les projets en dépendance. Pour réduire cet effet, il est possible d'utiliser l'annotation **JvmName.**

```
// Kotlin
class KotlinClass {
  var mutable: Boolean = false
    @JvmName("isMutable") get  // Spécifie le nom du getter côté Java bytecode
    @JvmName("mutable") set   // Spécifie le nom du setter côté Java bytecode
}

// Java
boolean value = kotlinClass.isMutable();    // Le getter est accessible en
 tant que 'isMutable'
boolean newValue = kotlinClass.mutable(true);  // Le setter est accessible
 en tant 'mutable'
```

Comme illustré dans l'exemple ci-dessus, grâce à l'annotation JvmName, il est possible de maîtriser le nom des setters et getters qui seront générés. Ainsi, même si la variable mutable est amenée à changer de nom dans le code Kotlin, il n'y aura pas d'impact sur le code Java vu que les getters (i.e isMutable) et setters (i.e mutable) garderont le même nom.

@Throws

En Kotlin, les exceptions rejetées par une méthode ne sont pas déclarables au niveau de la signature. Contrairement à java, il n'y pas de clause Throws. Le bytecode JVM généré ne contient pas d'information permettant de gérer les exceptions côté Java de façon pro-active. Ce qui pourrait créer quelques effets indésirables pour le code appelant côté Java. C'est là que l'annotation **Throws** entre en jeu.

Code complet sur programmez.com & github

Comme illustré dans l'exemple ci-dessus, l'intégration de l'annotation Throws, permet de rajouter la clause **Throws** au niveau de la signature de la méthode dans le bytecode Java. Ainsi, côté java, il devient obligatoire de gérer l'exception déclarée. Dans l'exemple ci-dessus, nous avons déclaré une seule exception, toutefois l'annotation peut recevoir une liste d'exceptions si nécessaire.

Inline Functions

Il est possible d'appeler des inline functions à partir de Java comme n'importe quelle autre fonction, mais bien sûr, elles ne sont pas réellement intégrées - une telle fonctionnalité n'existe pas dans Java. L'exemple ci-dessous montre l'inlining lorsqu'il est utilisé depuis Kotlin. Il faut savoir que les inline functions avec des paramètres de type réifié ne peuvent pas du tout être appelées à partir de Java, car elles ne prennent pas en charge l'inlining. Ainsi, il n'est pas possible d'utiliser de paramètres de type réifié dans des méthodes qui devraient être utilisables à partir de Java.

Code complet sur programmez.com & github

KClass Kotlin

KClass est la représentation Kotlin de l'objet Class Java. Elle fournit des capacités de réflexion. Pour une fonction Kotlin que l'on souhaiterait appeler depuis Java et acceptant une Kclass, il sera possible d'utiliser la classe prédéfinie kotlin.jvm.JvmClassMappingKt comme illustré dans l'exemple ci-dessous. Depuis Kotlin, on peut accéder plus facilement tant à l'objet Kclass que Class.

```
// Java
import kotlin.jvm.JvmClassMappingKt;
import kotlin.reflect.KClass;

KClass<A> clazz = JvmClassMappingKt.getKotlinClass(A.class);

// Kotlin
import kotlin.reflect.KClass

private val kclass: KClass<A> = A::class
private val jclass: Class<A> = A::class.java
```

Visibilité

Les déclarations de visibilité (private, protected, public) ne correspondent pas exactement entre Kotlin et Java, et en plus il existe des déclarations de niveau supérieur au sein Kotlin. Voyons donc comment les visibilités sont présentées sur Java. Premièrement, certaines visibilités peuvent être mappées comme ci-dessous :

- Les membres privés restent privés et accessibles qu'au niveau de la classe qui les déclare
- Les membres protégés restent protégés, au-delà de la classe déclarante, ne sont accessibles que par les classes héritants et présentes dans le même package.
- Les éléments à visibilité publique restent publiques sont accessibles sans restriction.
- Les autres visibilités ne peuvent pas être mappées directement, mais sont plutôt compilées selon la correspondance la plus proche :
- Les déclarations privées de haut niveau restent également privées. Cependant, pour autoriser les appels depuis le même fichier Kotlin (qui peut être une classe différente en Java), des méthodes synthétiques sont générées sur la JVM. De telles méthodes ne peuvent pas être appelées directement, mais sont générées pour transférer des appels qui ne seraient pas autrement possibles.

Toutes les déclarations **internal** de Kotlin deviennent publiques, car package-private serait trop restrictif. Celles qui sont déclarées à l'intérieur d'une classe subissent une modification des noms pour éviter les appels accidentels de Java. Par exemple, une méthode interne C.hello apparaîtra comme c.hello$module() dans le bytecode, mais elles ne sont pas invocables en tant que telles. Il est nécessaire d'utiliser @JvmName pour modifier le nom dans le bytecode Java et ainsi rendre l'appel possible.

Considérations supplémentaires

Les méthodes qui émettent une exception vérifiée en Java peuvent être appelées depuis Kotlin sans avoir à gérer l'exception. Ce concept (throws) n'existe pas en Kotlin et nécessite de redoubler de vigilance lors de l'intégration d'une librairie Java où certaines méthodes déclarent dans leur signature des exceptions faisant partie ainsi du contrat d'API.

Vous pouvez récupérer la classe Java d'un objet comme ceci UnObjet::**class.java** ou encore instanceObjet.**javaClass**

Vous pouvez utiliser la réflexion sur les classes Java depuis Kotlin et utiliser une référence à la classe Java comme point d'entrée. Par exemple, UnObjet::**class**.java.declaredMethods renvoie les méthodes déclarées de la classe UnObjet.

L'héritage fonctionne de façon similaire entre Kotlin et Java; les deux ne prennent en charge qu'une seule superclass (abstraite ou non), mais n'importe quel nombre d'interfaces à implémenter.

Nothing: il n'y a pas d'équivalent au type Nothing de Kotlin en Java, car même java.lang.Void accepte null comme valeur. Comme il s'agit toujours de la représentation la plus proche de Nothing disponible en Java, les types et paramètres de retour Nothing sont mappés à Void. L'utilisation de Nothing comme argument de type générique génère un type brut en Java pour au moins provoquer des avertissements d'appel non contrôlés. Par exemple, List<Nothing> devient une liste brute (List) en Java.

Kotlin mappe non seulement les tableaux primitifs à leurs types mappés correspondants (IntArray, LongArray, CharArray, etc.) et vice versa, il n'encourt également aucun coût de performance par rapport à Java lorsque vous les utilisez. Certains des exemples de code référencés dans cet article ont été récupérés de l'excellent livre de Peter Sommerhoff, Kotlin for Android App Development. Un livre qui aidera les lecteurs plus avancés à explorer plus de détails sur l'interopérabilité entre ces deux langages, qui à ce jour, est la plus grande force de Kotlin par rapport aux autres langages JVM.

Références

https://developer.android.com/kotlin/interop

https://www.baeldung.com/kotlin/java-interoperability

https://www.informit.com/articles/article.aspx?p=2952625

https://www.javatpoint.com/kotlin-java-interoperability-calling-java-from-kotlin

https://www.raywenderlich.com/books/kotlin-apprentice/v2.0/chapters/19-kotlin-java-interoperability

https://medium.com/kayvan-kaseb/calling-java-codes-from-kotlin-b74890fb4a78

Par les deux bouts de la lorgnette

Les apps changent, pas seulement parce qu'elles migrent vers le cloud avec ses services de nouvelles générations, mais aussi par la manière dont elles sont créées et déployées. Les pipelines CI/CD permettent aux développeurs de déployer de nouvelles applications ou de nouvelles fonctionnalités plusieurs fois par jour. Mais quelle est la place des ops dans cette chaîne ? Certains parlent même de « no ops ».

Mais qui sont les ops ?

Selon que l'on s'adresse à un développeur ou à un responsable d'exploitation, la réponse peut changer.

Le responsable d'exploitation répondra instantanément que les ops sont le travail de l'IT et des architectes et gestionnaires de l'infrastructure (une petite voix dissonante me murmure que le responsable des services managés n'est pas d'accord à 100 %).

Le développeur me répondra lui, que les ops c'est le travail de son PO (Product Owner) qui discute avec les métiers des fonctionnalités livrées et à livrer et qui finalement, alimente son backlog. On a donc en face de nous une chaîne de partie prenante qui se répartit le travail autour d'un outillage de déploiement automatique. Notons que nous n'avons encore interrogé que deux personnes.

La chaîne de responsabilité semble parcourir le chemin, de l'utilisateur de l'application jusqu'à l'infrastructure sous-jacente. Ce faisant, elle implique les « product owners » (PO), les développeurs, les concepteurs du pipeline de déploiement et les concepteurs de la plateforme. (En lisant cette chaîne, l'urbaniste ou Architecte d'entreprise me signifie qu'il ajouterait quelques éléments).

Finalement, cette chaîne peut être perçue différemment suivant notre interlocuteur et on se rend compte que les responsabilités sont partagées au sein d'un groupe d'acteurs qui doivent communiquer plus que jamais.

Les développeurs souhaitent plus que jamais être autonomes dans le choix des technologies à mettre en œuvre. Les exploitants souhaitent conserver la maîtrise des plateformes pour répondre aux demandes qui leur sont faites. Les RSSI souhaitent plus de contrôle sur la nature des déploiements effectués et assurer la protection des données et des services de leur entreprise. Les utilisateurs métier, ne demandent rien, si ce n'est des applications et des plateformes qui leur permettent de pratiquer leur métier. Ils ne souhaitent en aucun cas choisir entre le cloud ou le Datacenter, entre OpenShift et Docker, entre Oracle et MongoDB.

Le seul souhait des métiers vis-à-vis de l'IT, c'est d'obtenir une plateforme applicative fonctionnelle, performante et sûre. Bref de pouvoir l'oublier une bonne fois pour toutes !

L'observabilité

Jusqu'à présent, les différents interlocuteurs impliqués dans la gestion des systèmes informatiques utilisaient la supervision. Très vite, il s'est avéré qu'en fonction de ce qui devait être géré (applications, systèmes, stockage, réseau…) certains outils spécialisés mettaient en évidence les caractéristiques propres de l'objet supervisé. Les différentes parties prenantes se sont donc mises à utiliser des outils spécialisés pour superviser leur partie de la chaîne de l'information. Il y avait des manques, certaines informations pouvaient provenir de sources différentes, la consistance de l'information pouvait varier en fonction du point de vue. On a alors inventé les hyperviseurs ! Ceux pour viser, pas pour virtualiser. Le concept de « single source of truth » naissait.

Avec l'arrivée des applications modernes (new apps ici), les hyperviseurs ont montré leurs limites et le public intéressé par les différents états de l'IT s'est diversifié. Les outils ont évolué avec les besoins, les applications (et le vocable) d'observabilité sont apparues. Observabilité… Observabilité, subst. fém. : caractère de ce qui est observable.

On observe quoi ?

Une réponse partagée par l'ensemble des éditeurs se résume en trois points :

- Des logs
- Des traces
- Des métriques

Mais il peut en aller autrement si on s'intéresse aux motivations des observateurs. On peut par exemple observer les performances d'une application, d'un service, d'une infrastructure. On peut observer la disponibilité d'un serveur, d'un cluster, d'une base de données, d'un container.

On peut observer aussi les coûts d'une plateforme, d'un service cloud. Ce coût est souvent corrélé à l'utilisation réelle des ressources réservées, on parle alors de FinOps.

On peut observer la consommation électrique (la consommation d'énergie en général) d'un datacenter, d'une plateforme, d'un serveur, d'une application.

On peut observer l'accès aux plateformes et aux données pour s'assurer de la confidentialité et du respect des réglementations (PCI DSS, Export Control, GDPR…).

On peut observer l'expérience utilisateur en temps réel (RUM), la durée d'exécution d'un processus, le nombre de transactions d'un même type, le nombre d'exemples dans le présent article…

Bref ! On peut tout observer.

Encore faut-il le consolider et le présenter aux parties prenantes intéressées.

Qui observe ?

La liste des indicateurs pouvant être observés donne le tournis. Aujourd'hui, les gestionnaires de service ne sont plus les seuls intéressés et les besoins des uns sont intimement liés, voire subordonnés aux besoins des autres.

Essayons ensemble de remonter la chaîne alimentaire ou de la descendre, après tout, on n'est pas des saumons.

Les métiers sont intéressés par les indicateurs relatifs à leurs

Jean-Baptiste Bron

Chief Architect pour l'avant vente "Cloud Infrastructures Services" France chez Capgemini, je conçois des solutions d'infrastructure en relation avec mes collègues des apps, du testing et de la Cyber sécurité. Ces solutions se composent de blocs techniques mais aussi de processus de support innovants et agiles. Les solutions se doivent d'être globales et non isolées par silos. Elles se doivent avant tout d'être opérables par l'ensemble des parties prenantes autour de l'IT.

Architect | Presales

www.capgemini.com

services. Un responsable de secteur souhaitera savoir le nombre de produits vendus durant une opération promotionnelle. Un gestionnaire de services back office bancaire voudra être informé de la durée de ses traitements et du respect de ses cut off. Un logisticien souhaitera connaître les volumes transportés et les éventuels retards.

Tous ces indicateurs sont disponibles dans les résultats d'observation des applications. En parlant d'applications, les développeurs voudront connaître le bon fonctionnement de leur code, l'adéquation des technologies fournies à l'utilisation qu'il en fait. Les performances réelles de l'application qu'il a développée.

Les administrateurs seront intéressés par les performances de la plateforme en général, par la disponibilité des ressources afin d'établir un plan de capacité, par les incidents et problèmes générés par le fonctionnement du système informatique. Un responsable de la sécurité sera lui plus absorbé par l'observation des flux et des accès, par les fuites de données et les usurpations. Tous ces interlocuteurs ont de bonnes raisons d'observer et ils ont besoin d'informations, voire, d'informations en temps réel. Les prestataires ont besoin de justifier de la qualité de leur service.

En prenant en compte l'ensemble de ces besoins, on se rend compte que le format des rapports équivalents est difficile à produire, parfois même impossible à lire. Il faut donc diviser les lecteurs de ces rapports en différents rôles et leur proposer une vue du rapport focalisée sur leur profil. On peut parler de Role Based Reporting.

La mise en œuvre des moyens d'observabilité assure l'alimentation par les sources de données, le reporting, leur consommation par les intéressés.

On observe avec quoi ?

En général, on observe avec les moyens habituels pour la supervision des objets traditionnels, complétés par des outils capables d'observer le comportement des applications modernes. Les éditeurs de telles applications sont présents depuis l'apparition de ces applications cloud natives.

Tous ces outils génèrent une quantité importante de données qu'il faut canaliser dans un bus de données adapté à sa source. Chaque source de données est plus ou moins compatible avec les logiciels d'observabilité, mais une mise en forme de leur structure, à commencer par la référence temporelle, est souvent nécessaire. Afin de préserver la possibilité de faire évoluer les différents composants sans mettre en péril la stabilité de l'ensemble de l'édifice, le bus de données est segmenté en différents conteneurs indépendants qui ont pour but de canaliser les données d'une source vers un datalake. Eh oui, les outils d'observabilité des applications modernes sont modernes !

Ce bus de données composite est appelé IPaaS. Le datalake utilisé pour stocker l'ensemble de ces données est adapté au contexte du client, de nombreux éditeurs proposent des outils de captation, de transfert ou de stockage des données d'observation, mais aussi des solutions d'analyse et de reporting adaptées. Ce qui était confidentiel et réservé aux aficionados de l'open source est désormais convoité et promu par les éditeurs de solutions d'entreprise.

On fait quoi de ces informations ?

Comme nous l'avons vu précédemment, ces informations sont stockées dans un datalake afin d'être analysées et exploitées. La chaîne d'exploitation de ces données pour les applications et les plateformes permettent aux développeurs de vérifier le comportement de leur code et aux Ops de vérifier le fonctionnement de leur service métier.

En ce qui concerne le traitement des informations, les données recueillies sont analysées par une intelligence artificielle (AIOps) afin de réduire la durée de résolution des incidents et problèmes. La chaîne de traitement des données liées à l'infrastructure permet d'optimiser le fonctionnement des services gérés.

Ces deux chaînes doivent être compatibles, et être conçues parallèlement. Elles utilisent des composants hébergés sur site et d'autres services hébergés dans le cloud. On peut imaginer une telle plateforme totalement hébergée dans le cloud. En effet, la volumétrie de systèmes à mettre en œuvre et à gérer pour réaliser des analyses en temps réel est bien souvent disproportionnée face à la volumétrie du parc IT du client. De telles solutions sont proposées en SaaS par les différents éditeurs et permettent de ne pas avoir à investir dans une plateforme dédiée ni de materner depuis la phase embryonnaire une intelligence qui reste artificielle, quoi qu'on en dise. Ces outils ne sont en général pas une fin en soi, mais des moyens dédiés à l'amélioration des services proposés aux clients. Les incidents sont automatiquement regroupés en fonction de leur source, la résolution automatique de certains d'entre eux est appliquée et les autres préanalysés et transmis aux équipes les plus qualifiées. C'est le processus d'optimisation des services gérés. Cette optimisation est proposée au client final pour améliorer la compétitivité du service. Le but est d'anticiper les appels des utilisateurs et de réduire le nombre d'incidents à résoudre manuellement. Il s'agit enfin d'éviter les incidents prévisibles sans intervention humaine et sans émission de ticket de résolution.

Les prestataires peuvent aussi aider leurs clients à se doter de leur propre chaîne d'observabilité et de résolution. Il s'agit alors de travailler en bonne intelligence et de coordonner les besoins des différentes parties prenantes en fonction de leur maturité et de leurs besoins. Une telle chaîne d'observabilité n'est pas un ensemble fini. Elle va croître avec le temps et les besoins émergents. Il est important de la concevoir évolutive et de s'assurer que ses nouvelles capacités ne perturbent pas celles qui rendent le service existant.

Les outils sont une chose, les processus, une autre. Une coordination de bout en bout, depuis les métiers jusqu'aux infrastructures est nécessaire à l'obtention de résultats tangibles. C'est un effort quotidien qui permet de faire bénéficier aux clients de méthodes agiles. Les frameworks (Scaled Agile Framework (SAFe), Large Scale Scrum, Disciplined Agile, Nexus…) permettent, pour peu qu'on en choisisse (ou adapte) un, de travailler ensemble à l'amélioration du service d'information dans sa globalité. Un nouveau genre de spécialiste a même vu le jour : le SRE. Le SRE (voir la définition sur votre référentiel préféré) est le mortier entre les briques de responsabilités au sein de l'entreprise, le médiateur de la fiabilité, bref le spécialiste qui va vous sauver la vie ! Oui, le DevOps est compatible avec la maîtrise des infrastructures ! Oui, la gestion des infrastructures peut être un peu plus agile qu'une poutre en titane !

softfluent

Rejoignez **l'ESN** créée par des **développeurs** pour les Développeurs

Vous possédez une ou plusieurs de ces compétences ?

- Azure DevOps
- Cloud
- SQL Server
- Angular
- React
- C#
- CI/CD
- .NET Core
- ASP.NET MVC
- Microservices

Contactez nous !

Retrouvez-nous sur notre page carrière : **softfluent.fr/nous-rejoindre**

Ou contactez Marine, en charge du recrutement chez SoftFluent :

06 69 26 27 87

marine.genetay@softfluent.com

- Conseil
- Expertise
- Partage

softfluent.fr

OPÉRATION POUR 1 EURO DE PLUS

JUSQU'AU 03 DÉCEMBRE

Aucun abonnement à souscrire.
Compatible tous opérateurs

COMMANDEZ WINDEV 27

OU WEBDEV 27 OU WINDEV MOBILE 27

ET RECEVEZ LE NOUVEL

iPhone 13 Pro

WINDEV 27
AGL DevOps
Cross-Plateformes
N°1 en France

CHOISISSEZ

- **iPhone 13 Pro** 128Go
  ou
- **iPad Pro** 256Go
  ou
- **MacBook Air** 13,3" 256Go

- **lot de 2 iPad** 10,2" 64Go
  ou
- **lot de 2 iPhone SE** 64Go
  ou
- **lot de 2 Apple Watch**

(Détails et autres matériels sur www.pcsoft.fr)

WWW.PCSOFT.FR

OPÉRATION POUR 1 EURO DE PLUS

Pour bénéficier de cette offre exceptionnelle, il suffit de commander WINDEV 27 (ou WINDEV Mobile 27, ou WEBDEV 27) chez PC SOFT au tarif catalogue avant le 03 Décembre 2021. Pour 1 Euro HT de plus, vous recevrez alors le ou les magnifiques matériels que vous aurez choisis. Offre réservée aux sociétés, administrations, mairies, GIE et professions libérales, en France métropolitaine. L'offre s'applique sur le tarif catalogue uniquement. **Voir tous les détails sur : WWW.PCSOFT.FR ou appelez-nous au 04.67.032.032**

Le Logiciel et le matériel peuvent être acquis séparément. Tarif du Logiciel au prix catalogue de 1.650 Euros HT (1.980,00 TTC). Merci de vous connecter au site www.pcsoft.fr pour consulter la liste des prix des matériels. Tarifs modifiables sans préavis.