Micro-Services Réactifs avec Quarkus

ALI KOUDRI – ALI.KOUDRI@GMAIL.COM

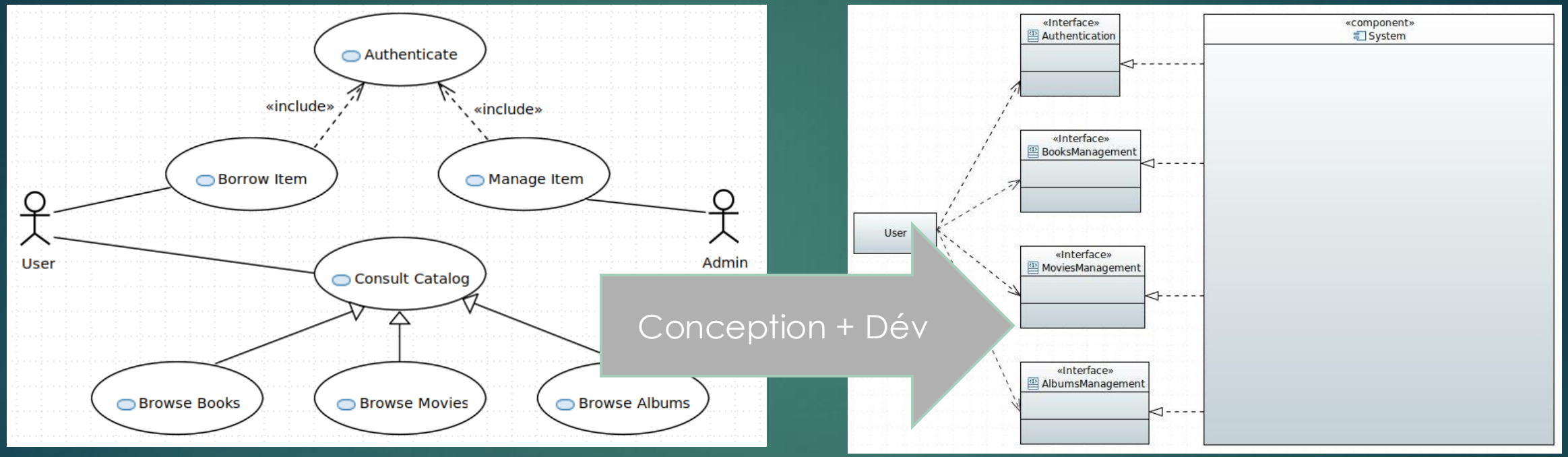
Agenda

- Introduction
  - Contexte et motivations des microservices
- Principes
  - Patterns et réactivité
- Bases
  - Annotations
  - IoC et AOP
  - Programmation fonctionnelle et réactive
- Quarkus
  - Présentation
  - Configuration
  - Cycle de vie
  - Gestion des événements
  - R2DBC
  - Advices communs
  - Intégration avec les autres technologies

Introduction

Exemple d'application

- Application de gestion de contenus multimédia

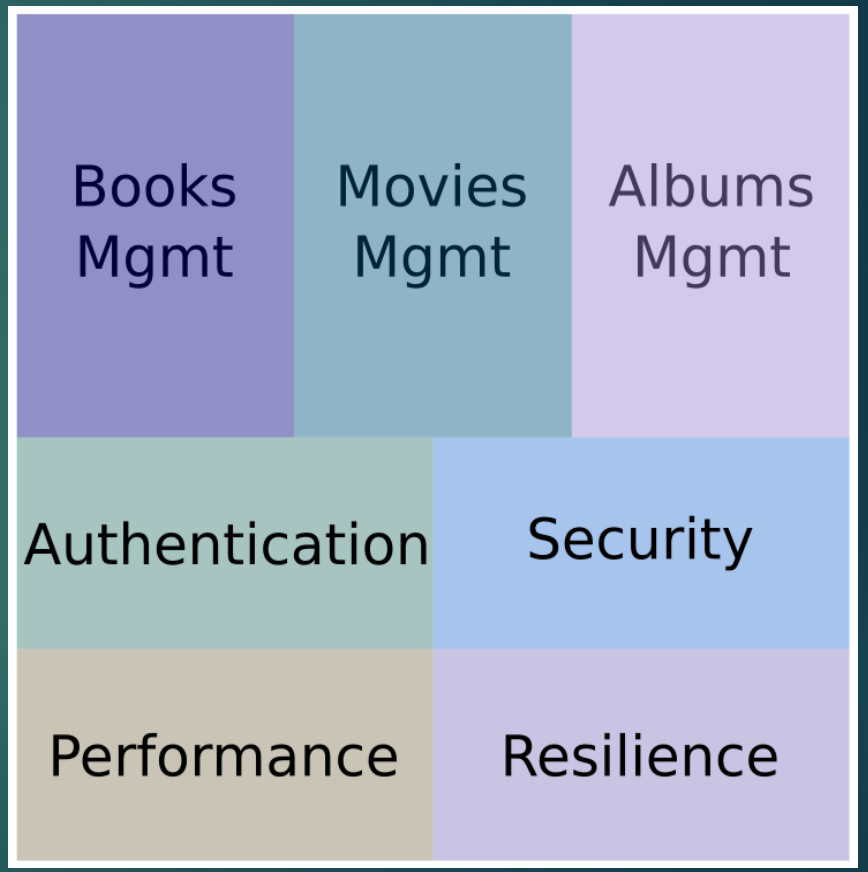


Depuis la formalisation du besoin, on aboutit après un travail d'analyse et de conception au développement d'un système qui implante un certain nombre d'interfaces

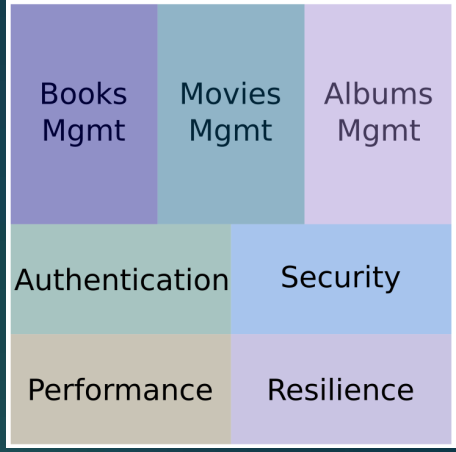
Quels sont les problèmes posés par cette solution ?

Application monolithique

- Applications dotées de nombreuses fonctionnalités, toutes regroupées en un seul exécutable logique
- Avec l'évolution des besoins et l'élargissement du périmètre fonctionnel, une application peut grossir et se complexifier au point où plus grand monde n'arrive à comprendre son architecture
- De fait, les capacités d'évolution d'une application monolithiques s'amenuisent avec temps
  - L'organisation, les usages et les compétences changent, les standards et les technologies évoluent
  - Le passage à l'échelle s'avère être de plus en plus difficile et sa **réactivité** se dégrade

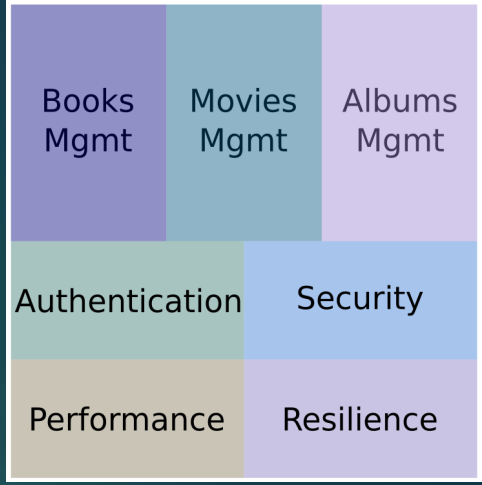


Architectures monolithiques Problèmes



- Les avancées technologiques ces dernières années et les demandes croissantes en fonctionnalités posent de grands défis dans la conception, le développement et l'opérationnalisation de systèmes informatiques.
  - Les architectures logicielles "classiques", monolithiques, ne répondent pas à ces défis et de nouvelles architectures plus flexibles, plus résilientes et plus faciles à maintenir sont aujourd'hui nécessaires....
- **Évolutivité**: Il peut être coûteux de faire évoluer l'ensemble de l'application alors que bien souvent une seule partie fait l'objet d'une forte demande.
  - L'idéal serait de pouvoir ne faire évoluer que les composants nécessaires, ce qui se traduirait par une utilisation plus efficace des ressources.
- **Goulets d'étranglement**: Toute modification, aussi minime soit-elle, nécessite le redéploiement de l'ensemble de l'application. Cela peut ralentir les processus de développement et de déploiement.
  - L'idéal serait de permettre un déploiement indépendant des différents services qui constituent l'application, ce qui faciliterait des mises à jour plus rapides et plus fréquentes.

Architectures monolithiques Problèmes



- **Gestion de la pile technologique**: Une application monolithique est généralement limitée à une seule pile technologique.
  - Il serait plus pratique de pouvoir développer et combiner des services sur la base de technologies, de cadres et de langages différents qui conviennent le mieux à leurs exigences spécifiques.
- **Gestion de la complexité**: Au fur et à mesure qu'une application monolithique se développe, sa base de code peut devenir lourde et complexe, ce qui la rend difficile à comprendre et à maintenir.
  - Le fait de décomposer l'application en éléments composables plus petits rendrait l'application plus facile à développer et à maintenir.
- **Fiabilité**: Dans une architecture monolithique, un bogue dans n'importe quelle partie de l'application peut potentiellement entraîner l'effondrement de l'ensemble du système.
  - Une meilleure isolation des défaillances n'entraînerait pas nécessairement l'arrêt de l'ensemble de l'application.

Architectures monolithiques Problèmes

- **Manque de flexibilité**: Les fournisseurs de solutions ont plus que jamais besoin de se reposer sur des pratiques **plus agiles**.
  - Différentes équipes pourraient travailler simultanément sur différents services, ce qui accélérerait le développement et l'innovation.
- **Dépendances aux fournisseurs**: Les architectures monolithiques peuvent conduire à un verrouillage du fournisseur ou de la plateforme.
  - Il faudrait pouvoir utiliser des services provenant de différents fournisseurs ou plateformes, ce qui réduirait le risque de verrouillage.
- **Gestion du travail**: La gestion d'une grande équipe sur une base de code monolithique unique peut s'avérer difficile.
  - L'idéal serait de se reposer sur une approcher qui permette à des équipes plus petites et plus ciblées de s'approprier des services spécifiques, ce qui améliorerait la coordination et l'efficacité.

Illustration des problèmes

- En 2013, le site web amazon.com a connu une panne importante qui a entraîné une perte d'environ 66 240 dollars par minute, soit un total de près de 2 millions de dollars sur la base de son chiffre d'affaires net de 2012. Cet incident met en évidence l'impact financier que les pannes de système peuvent avoir sur une entreprise, en particulier pour les sociétés opérant à l'échelle d'Amazon.
  - https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/?sh=7b9f109c495c

- En 2011, le site web de Walmart a dû faire face à la forte charge de trafic du Black Friday, ce qui a entraîné des ruptures de pages de paiement, des paniers d'achat vides et des erreurs de connexion. Cette situation a non seulement frustré les clients, mais a également entraîné des pertes de ventes et nui à la réputation de l'entreprise. En outre, les magasins physiques ont été le théâtre d'actes de violence, ce qui a encore terni l'événement.
  - https://techcrunch.com/2011/11/25/walmart-black-friday/?guccounter=1

Pourquoi la réactivité ?

- Les serveurs web sont configurés avec un pool de threads fixe
  - Dimensionné selon un usage attendu
  - Avec des ressources adéquates
- Si un serveur fait face à un pic d'utilisation non prévu, cela peut avoir des conséquences fâcheuses sur la qualité de service
  - Et sur la perception du client, avec potentiellement des conséquences fâcheuses
- Les plus grands ont subi de grandes pertes pour des problèmes de réactivité
  - https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/?sh=7b9f109c495c
  - https://techcrunch.com/2011/11/25/walmart-black-friday/?guccounter=1

Pourquoi la réactivité ?

- Les problèmes mentionnés dans la planche précédente mettent en exergue plusieurs propriétés que l'on serait en droit d'exiger pour une application
  - **Résilience**: capacité d'un système à rester réactif en toute situation.
  - **Élasticité**: capacité à rester réactif quelle que soit la charge.
  - **Scalabilité**: pour supporter l'élasticité, le système doit être en mesure d'adapter ses capacités en fonction de la charge.
- La réactivité se mesure notamment par la **latence**, à savoir le temps d'attente pour la satisfaction d'une requête
- Les piliers de la réactivité en Java sont:
  - Le traitement asynchrone des données
  - Les appels non bloquants
  - La **programmation fonctionnelle – Quels en sont les intérêts ?**

Réactivité: Par quels moyens ?

- Dans le cas où vous gérez l'infrastructure physique supportant l'exécution de vos applications
  - Il vaut faut procéder à un compromis entre scalabilité horizontale et scalabilité verticale
  - Les critères pour la recherche de compromis sont:
    - Les coûts, l'évolutivité, la topographie des nœuds d'exécution, la fiabilité ou la performance
- Dans le cas où vous déployez sur une infrastructure existante (IaaS), la plupart des fournisseurs ont des offres élastiques
  - Tenant compte de différentes exigences:
    - Puissance de calcul, latence, mémoire, stockage, etc.

Réactivité: Par quels moyens ?

- Au niveau logiciel, un moyen de contribuer à rendre le système réactif est de respecter le principe d'**isolation** entre composants fonctionnels
  - En effet, la dégradation de la réactivité d'une fonction, ou même son arrêt, ne doit pas impacter la réactivité des autres fonctions
- Le respect du principe d'isolation a un impact fort sur l'architecture du système
  - Il est en effet nécessaire d'assurer un **découplage fort** des composants du système pour y arriver
  - On parle alors de service, ou plus précisément de **micro-service**
- En plus du découplage fort, il est généralement recommandé de **dupliquer** les composants sur différents nœuds physiques pour des questions de résilience

https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling
http:/ /microservices. io/ patterns

Réactivité: Par quels moyens ?

- Dans le cas où vous gérez l'infrastructure physique supportant l'exécution de vos applications
  - Il vaut faut procéder à un **compromis entre scalabilité horizontale et scalabilité verticale**
  - Les critères pour la recherche de compromis sont: Les coûts, l'évolutivité, la topographie des nœuds d'exécution, la fiabilité ou la performance
- Dans le cas où vous déployez sur une infrastructure existante (IaaS), la plupart des fournisseurs ont des offres élastiques tenant compte de différentes exigences: Puissance de calcul, latence, mémoire, stockage, etc.

Réactivité: Par quels moyens ?

- Au niveau logiciel, un moyen de contribuer à rendre le système réactif est de respecter le principe d'**isolation** entre composants fonctionnels
  - En effet, la dégradation de la réactivité d'une fonction, ou même son arrêt, ne doit pas impacter la réactivité des autres fonctions

- Le respect du principe d'isolation a un impact fort sur l'architecture du système
  - Il est en effet nécessaire d'assurer un **découplage fort** des composants du système pour y arriver
  - On parle alors de service, ou plus précisément de **micro-service**

- En plus du découplage fort, il est recommandé de **dupliquer** les composants sur différents nœuds physiques

Scalabilité verticale / horizontale

- La mise à l'échelle verticale consiste à ajouter des ressources à un serveur ou à un système existant afin d'en augmenter la capacité. Il peut s'agir d'une mise à niveau de l'unité centrale, de la mémoire vive, du stockage ou d'autres composants d'une seule machine.
  - La mise à l'échelle verticale est généralement plus simple à mettre en œuvre car elle n'implique pas la complexité de la coordination entre plusieurs systèmes ou des changements dans l'architecture de l'application.

- La mise à l'échelle horizontale consiste à ajouter des machines ou des instances à un pool de ressources afin de répartir la charge et d'augmenter la capacité. Cette approche est privilégiée dans les systèmes distribués, tels que les architectures en microservices, où les charges de travail peuvent être réparties sur plusieurs serveurs.

Scalabilité verticale

- **Avantages** :
  - Simplicité : Plus facile à mettre en œuvre et à gérer puisqu'elle ne concerne qu'un seul système.
  - Impact immédiat : La mise à niveau du matériel permet d'améliorer rapidement les performances du système.
  - Compatibilité : Moins de risques de devoir modifier le code de l'application.
- **Inconvénients** :
  - Limites physiques : Les contraintes physiques et technologiques limitent les possibilités de mise à niveau d'un système unique.
  - Temps d'arrêt : La mise à niveau du matériel peut nécessiter des temps d'arrêt, ce qui affecte la disponibilité.
  - Coût : le matériel haut de gamme peut être coûteux et la rentabilité diminue à mesure que l'on s'approche des limites supérieures de la technologie disponible.

Scalabilité horizontale

- **Avantages** :
  - Évolutivité : Évolution pratiquement illimitée par l'ajout de machines en fonction des besoins.
  - Flexibilité : Possibilité d'augmenter la capacité pour répondre à la demande et de la réduire lorsque la demande diminue, ce qui permet d'optimiser l'utilisation des ressources et les coûts.
  - Tolérance aux pannes : Amélioration de la résilience et de la disponibilité puisque la défaillance d'un nœud n'entraîne pas l'effondrement de l'ensemble du système.
- **Inconvénients** :
  - Complexité : Plus complexe à mettre en œuvre et à gérer en raison de la nature distribuée de l'architecture.
  - Frais généraux : Nécessite des mécanismes pour l'équilibrage de la charge, la cohérence des données et la communication entre les services.
  - Coût initial : Peut impliquer des coûts d'installation initiaux plus élevés et une plus grande complexité opérationnelle.

Reactive Manifesto

Selon le "Reactive Manifesto" (https://www.reactivemanifesto.org/):

"Les systèmes modernes font face à des contraintes de plus en plus fortes. Pour y répondre, les systèmes doivent être plus robustes, plus résilients, plus flexibles et mieux placés pour répondre aux exigences modernes [...]

Nous voulons des systèmes **réactifs, résilients, élastiques et pilotés par les messages**. Nous les appelons systèmes réactifs.

Les systèmes construits en tant que systèmes réactifs sont plus flexibles, faiblement couplés et évolutifs. Ils sont donc plus faciles à développer et à modifier. Ils sont beaucoup plus tolérants à l'égard des défaillances et, lorsque celles-ci se produisent, ils y font face avec élégance plutôt qu'en catastrophe. Les systèmes réactifs offrent aux utilisateurs un retour d'information interactif efficace."

Reactive Manifesto

Le système doit être responsive :

Le système réagit en temps voulu, dans la mesure du possible. La réactivité est la pierre angulaire de la convivialité et de l'utilité, mais plus encore, la réactivité signifie que les problèmes peuvent être détectés rapidement et traités efficacement. Les systèmes réactifs s'attachent à fournir des temps de réponse rapides et cohérents, en établissant des limites supérieures fiables afin de fournir une qualité de service constante. Ce comportement cohérent simplifie à son tour le traitement des erreurs, renforce la confiance de l'utilisateur final et l'incite à interagir davantage.

Le système doit être résilient :

Le système reste réactif en cas de défaillance. Cela ne s'applique pas seulement aux systèmes hautement disponibles et critiques - tout système qui n'est pas résilient ne sera pas réactif après une panne. La résilience est obtenue par la réplication, le confinement, l'isolation et la délégation. Les défaillances sont contenues dans chaque composant, isolant les composants les uns des autres et garantissant ainsi que certaines parties du système peuvent tomber en panne et se rétablir sans compromettre le système dans son ensemble. La reprise de chaque composant est déléguée à un autre composant (externe) et la haute disponibilité est assurée par la réplication si nécessaire. Le client d'un composant n'a pas à gérer ses défaillances.

Reactive Manifesto

Le système doit être élastique :

Le système reste réactif sous une charge de travail variable. Les systèmes réactifs peuvent réagir aux changements du débit d'entrée en augmentant ou en diminuant les ressources allouées pour servir ces entrées. Cela implique des conceptions sans points de contention ni goulots d'étranglement centraux, ce qui permet de partager ou de répliquer les composants et de répartir les entrées entre eux. Les systèmes réactifs prennent en charge les algorithmes de mise à l'échelle prédictifs, ainsi que réactifs, en fournissant des mesures de performance pertinentes en temps réel. Ils assurent l'élasticité de manière rentable sur des plates-formes matérielles et logicielles de base.

Le système doit être "Message-driven" :

Les systèmes réactifs s'appuient sur le passage de messages asynchrones pour établir une frontière entre les composants qui garantit un couplage faible, l'isolation et la transparence de l'emplacement. Cette frontière fournit également les moyens de déléguer les défaillances sous forme de messages. L'utilisation du passage de messages explicite permet la gestion de la charge, l'élasticité et le contrôle du flux en façonnant et en surveillant les files d'attente de messages dans le système et en appliquant une pression de retour si nécessaire. L'utilisation de la messagerie transparente comme moyen de communication permet à la gestion des pannes de fonctionner avec les mêmes constructions et la même sémantique à travers un cluster ou au sein d'un seul hôte. La communication non bloquante permet aux destinataires de ne consommer des ressources que lorsqu'ils sont actifs, ce qui réduit la surcharge du système.

Micro-services: Présentation

- Les micro-services tentent d'apporter une réponse aux problèmes posés par les applications monolithiques par un style architectural visant:
  - Un couplage faible entre composants
  - Une meilleure cohérence dans l'hétérogénéité des composants
  - Une plus grande réactivité face aux différents événements pouvant survenir pendant tout le cycle de vie du logiciel (panne, pic d'utilisation, ...)

- Les micro-services permettent:
  - Une meilleure évolution dans le temps des applications
  - Une meilleure maintenabilité
  - Une hétérogénéité contrôlée
  - Un passage à l'échelle facilité
  - Une résilience accrue

Avantages des micro-services

- **Évolutivité** : Les micro-services peuvent être mis à l'échelle de manière indépendante pour répondre à la demande. La défaillance du site web de Walmart démontre la nécessité de systèmes capables de s'adapter dynamiquement aux pics de trafic, ce qui est un point fort des micro-services.

- **Résilience** : L'architecture en micro-services peut isoler les défaillances, empêchant ainsi qu'un seul point de défaillance n'entraîne l'arrêt de tout le système. La panne d'Amazon montre le coût élevé des temps d'arrêt, soulignant la valeur d'une architecture résiliente que les micro-services peuvent fournir.

Avantages des micro-services

- **Agilité** : Les micro-services permettent un déploiement plus rapide des mises à jour et des corrections. En réponse à des problèmes ou à des demandes changeantes, les entreprises peuvent mettre à jour ou faire évoluer des services individuels sans redéployer l'ensemble de l'application, ce qui permet d'éviter les pannes ou d'y remédier rapidement.

- **Évolutivité** : Les micro-services peuvent être mis à l'échelle de manière indépendante pour répondre à la demande. La défaillance du site web de Walmart démontre la nécessité de systèmes capables de s'adapter dynamiquement aux pics de trafic, ce qui est un point fort des micro-services.

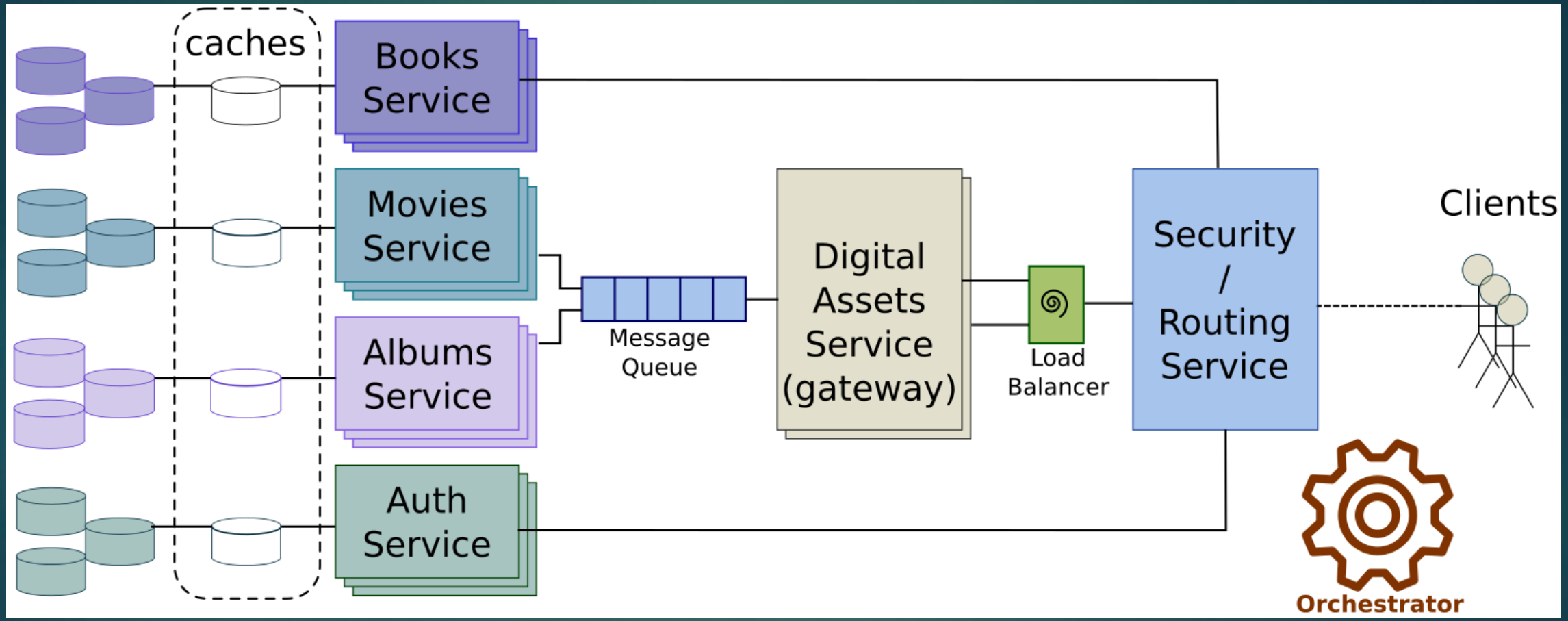
Avantages des micro-services

- **Diversification des technologies** : Les micro-services favorisent l'utilisation de différentes technologies entre les services, ce qui permet d'adopter la technologie la plus adaptée aux besoins de chaque service. Cela aurait pu contribuer à optimiser les performances et la fiabilité de fonctionnalités spécifiques, telles que le processus d'encaissement de Walmart.

- **Décentralisation** : En décentralisant le contrôle et les données, les micro-services réduisent le risque d'incohérences et de défaillances généralisées des données. Cette approche pourrait atténuer les problèmes tels que ceux rencontrés par Walmart, où des défaillances à l'échelle du système ont entraîné une mauvaise expérience client.

- **Expérience utilisateur** : En fin de compte, l'architecture en micro-services favorise une meilleure expérience client en garantissant que les services sont disponibles, réactifs et à jour. Les incidents d'Amazon et de Walmart illustrent l'impact direct des défaillances du système sur la satisfaction des clients et la réputation de l'entreprise.

Exemple d'architecture en micro-services



- Cet exemple est plus conforme à ce qu'on attend d'une application réactive - **Pourquoi ? Quels avantages ?**

Micro-services

- Les micro-services tentent d'apporter une réponse aux problèmes posés par les applications monolithiques par un style architectural visant:
  - Un couplage faible entre composants
  - Une meilleure cohérence dans l'hétérogénéité des composants
  - Une plus grande réactivité face aux différents événements pouvant survenir pendant tout le cycle de vie du logiciel
- Les micro-services permettent:
  - Une meilleure évolution dans le temps des application
  - Une meilleure maintenabilité
  - Une hétérogénéité contrôlée
  - Un passage à l'échelle facilité
  - Une résilience accrue
- **Questions:**
  - Dans quelle mesure cette approche peut impacter l'organisation du travail ?
  - Quelles sont les avantages et inconvénients de cette approche ? En particulier sur les tests ?

Passage aux micro-services

- **Développement**: Le développement d'un micro-service est fortement contraint par les performances; il nécessite un cadre de programmation particulier, permettant de créer des applications réactives, à faible empreinte mémoire. Exemple: Micronaut

- **Conteneurisation**: Une fois le micro-service implanté, il faut le conteneuriser de manière à pouvoir le déployer n'importe où, avec notamment **docker**

- **Communication**: Le manifeste Reactive stipule que les applications doivent être message-driven de manière à assurer un couplage plus lâche et une meilleure isolation. Exemple: Kafka

Passage aux micro-services

- **Orchestration**: Les micro-services nécessitent des outils permettant que l'application soit toujours dans un état acceptable; que les micro-services puissent communiquer de manière transparente, qu'ils soient relancés automatiquement en cas de panne ou qu'ils soient dupliqués en cas de pic de charge. Exemple: Kubernetes

- **Tests:** De fait, le test d'applications à base de micro-services nécessitent des frameworks particuliers permettant le test de chaque micro-service en mimant son environnement. Exemple: Testcontainers

- **Configuration**: La configuration d'applications à base de micro-services nécessitent une approche centralisée pour plus de cohérence. Exemple: Spring Cloud Config

Passage aux micro-services

- **Logging**: Comme pour la configuration, les applications à base de micro-services nécessitent également une approche centralisée permettant d'analyser la chaîne de traitements et remonter à la source du problème. Exemple: Zipkin ou Jaeger

- **Monitoring:** Les décisions de reconfiguration de l'application sont basées sur les observations faites sur l'ensemble des micro-services. Là encore, nous avons besoins de centralisée les traces d'exécution, les visualiser et éventuellement prédire de potentiels problèmes. Exemples: Prometheus / Grafana

Méthodologie des 12 facteurs (Heroku)

- Assurer le suivi de vos changement avec un système de contrôle de versions
- Gérer de manière explicite toutes les dépendances (maven, pip, …)
- Centraliser la configuration des services (Spring cloud, Ansible)
- Utiliser des interfaces standards afin de s'abstraire des services techniques (JPA pour les BDD)
- Bien séparer les étapes de construction de l'application (Compilation, test, déploiement, exécution)
- Les services doivent être stateless
- Les services doivent être complétement autonomes, et embarquer toutes leur dépendance (ex: Spring / Jetty)
- La concurrence doit être supportée au niveau des processus, les threads étant vus comme des détails d'implantation
- L'arrêt d'un service ne doit pas générer d'effet de bord
- Éviter les gaps entre environnements de développement et de production
- Gérer les logs de manière standardisée et centralisée
- Automatiser autant que possible les tâches d'administration par des scripts

Exemple de success story

- Spotify, le service de streaming musical, a migré d'une architecture monolithique vers une architecture en micro-services en 2013-2014. Avant cela, leur application backend était un gros bloc monolithique Java déployé sur une centaine de serveurs.

- Les problèmes rencontrés avec cette architecture monolithique étaient :
  - Des déploiements longs et risqués à cause de la taille du monolithe
  - Des temps de build très longs
  - Des problèmes de scalabilité, l'architecture n'était pas adaptée à la croissance de Spotify
  - Un couplage fort rendant les changements difficiles et lents

- Spotify a donc décidé de migrer vers une architecture en micro-services, en découpant progressivement son monolithe en petits services autonomes. Chaque service peut être développé, déployé et scalé indépendamment par une équipe dédiée.

Exemple de success story (Cont'd)

- Les bénéfices observés ont été :
  - Des déploiements beaucoup plus rapides et fréquents (passage de un par semaine à plusieurs par jour)
  - Une bien meilleure scalabilité / résilience
  - Des temps de build réduits permettant un développement plus rapide
  - Plus d'autonomie pour les équipes qui maîtrisent leur service de bout en bout
- Spotify utilise son propre outil de déploiement continu (Helios) pour gérer ses centaines de micro-services dans un cluster.
- Cet exemple montre bien les avantages que peut apporter une architecture en micro-services pour une application à grande échelle comme Spotify, même si la migration depuis un monolithe est un processus long et complexe qui doit être bien planifié.

Méthodologie des douze facteurs

- La **méthodologie des douze facteurs** est un ensemble de douze bonnes pratiques pour développer des applications destinées à fonctionner comme un service.
  - Elle a été élaborée à l'origine par Heroku pour les applications déployées en tant que services sur leur cloud en 2011.
  - Au fil du temps, elle s'est avérée suffisamment générique pour tout développement de logiciel en tant que service (SaaS).

Méthodologie des douze facteurs

- Les douzes bonnes pratiques sont les suivantes:
  - Assurer le suivi de vos changement avec un système de contrôle de versions
  - Gérer de manière explicite toutes les dépendances (maven, pip, …)
  - Centraliser la configuration des services (Spring cloud, Ansible)
  - Utiliser des interfaces standards afin de s'abstraire des services techniques (JPA pour les BDD)
  - Bien séparer les étapes de construction de l'application (Compilation, test, déploiement, exécution)
  - Les services doivent être stateless
  - Les services doivent être complétement autonomes, et embarquer toutes leur dépendance (ex: Spring / Jetty)
  - La concurrence doit être supportée au niveau des processus, les threads étant vus comme des détails d'implantation
  - L'arrêt d'un service ne doit pas générer d'effet de bord
  - Éviter les gaps entre environnements de développement et de production
  - Gérer les logs de manière standardisée et centralisée
  - Automatiser autant que possible les tâches d'administration par des scripts

Les bases

Principes de développement

- Le développement logiciel repose sur 3 volets:
  - Les données: types primitifs, structures
  - Les traitements: algorithmie
  - L'organisation: architecture, librairies, composants, déploiement

- Considérant ces 3 volets et les objectifs d'agilité / développement rapide, il est nécessaire d'adopter des techniques permettant à la fois une meilleure séparation des préoccupations et une meilleure capitalisation / réutilisation

- **Citer des exemples de telles techniques**

Principes de développement

- Depuis l'avènement du développement logiciel, beaucoup de techniques ont été proposées afin d'améliorer la **productivité** et la **qualité** des produits dans des contraintes de **coût** et **délai** de plus en plus fortes
- Parmi ces techniques, nous pouvons citer:
  - Des paradigmes:
    - Approche objet
    - Approche fonctionnelle
    - Programmation par aspect
  - Des règles de bonne pratique:
    - Patrons de conception
    - Inversion de contrôle
  - Des outils de configuration / test

IoC: Présentation

- L'IoC permet de décharger les développeurs de la gestion du cycle de vie des objets
  - Chargement, dépendances, suppression
- Les objets et leurs dépendances sont gérées par un conteneur
- Les objets peuvent être injectés au démarrage ou en cours d'exécution

IoC: Présentation

- Le conteneur est chargé et configuré au démarrage de l'application
  - Il se charge ensuite du cycle de vie de tous les objets constituants l'application, et de leurs dépendances
  - Il repose sur l'utilisation d'une factory chargée de la gestion des Beans, la "Bean Factory"
  - La Bean Factory est configurée avec le contexte applicatif "Application Context"
- Le conteneur utilise les annotations afin de gérer le cycle de vie des objets
  - Les annotations sont supportées de manière native par Java
  - Ils peuvent être utilisées à la compilation ou en cours d'exécution

IoC: Concepts fondamentaux

- Application Context
  - Élément central d'une application Spring
  - Implante le principe de IoC
  - Encapsule la Bean Factory
  - Fournit les métadonnées servant à la création de Beans
  - Une application peut avoir plus d'une instance de Application Context
- La Bean Factory assure la création de Beans
  - Comprenant les singletons
  - Assure l'ordre de création des Beans (important pour la gestion des dépendances)

IoC: Configuration

- La configuration du contexte d'une application Spring peut se faire de deux manières
  - Fichiers XML (plus vraiment utilisés)
  - Annotations

- L'utilisation d'annotations présentent plusieurs avantages
  - Un seul langage, le java
  - Vérification à la compilation
  - Meilleure intégration dans les IDE

- Spring a accès aux variables d'environnement du système

Annotations

- Le mécanisme d'annotation a été proposé pour la première fois par Java et repris depuis dans plusieurs langages (python, C#, go, …)
- Il permet de préciser la sémantique des différents éléments: attributs, opérations ou classes
- Les annotations peuvent être exploitées à la compilation ou à l'exécution
  - Compilation => Instructions destinées au compilateur
  - Exécution => Instructions destinées à l'interpréteur (framework)
- L'exploitation des annotations nécessitent généralement des capacités de réflexion et d'introspection
- Java fournit un certain nombre d'annotations pour gérer l'évolution des APIs ou documenter le code: @Override, @Deprecated, @SuppressWarnings
- Les frameworks basés sur Java, comme Spring, Micronaut ou Quarkus, ajoute leur propre jeu d'annotations

Annotations: Exemple

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
      String first();
      String last();
 }

@Author(first = "Oompah", last = "Loompah")
Book book = new Book();

Class<Book> clObj = Book.class
AnnotatedElement e = (AnnotatedElement) clObj;
if (e.isAnnotationPresent(Author.class)) {
  Annotation a = e.getAnnotation(Author.class);
  Author auth = (Author)a;
  System.out.println(auth.first());
}
```

- Alternative aux fichiers de configuration XML
- Par exemple, le paquetage javax.validation.constraints standardise un certain nombre de contraintes de validation des données
  - @NotNull, @Past, @Size

IoC: Configuration – Cont'd

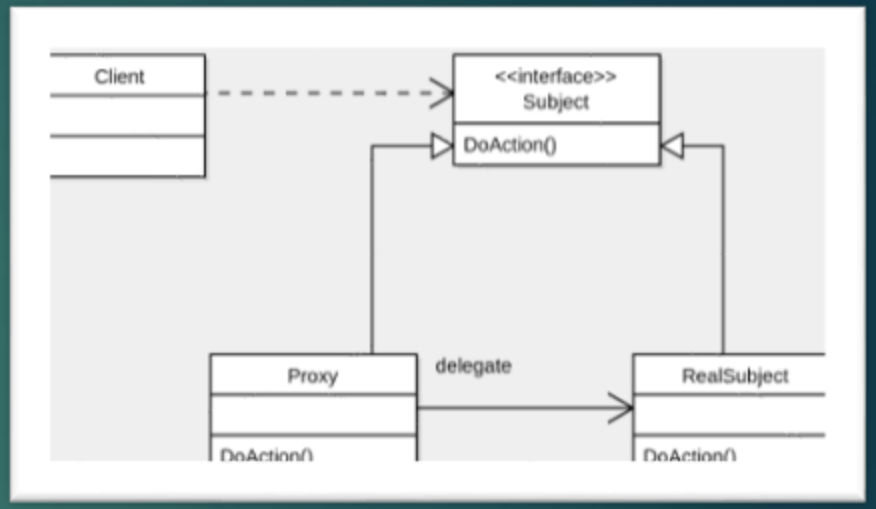
- D'autres variables, propres à l'application, peuvent être déclarées via des fichiers de propriétés
  - Les variables d'environnement ont la priorité sur les variables issues des fichiers de propriétés
- Il est possible de configurer différents profiles
  - Utilisation de l'annotation "Profile('name')" sur les Beans
  - Et le paramétrage de la JVM: -Dspring.profiles.active=name

IoC: Langage d'expression SPEL

- Pour des configurations plus complexes, Spring offre un langage d'expression
  - Apporte plus de contrôle et de flexibilité
  - Exemple: @Value("#{new Boolean(environment['spring.profiles.active'] != 'dev')}")
- Une expression SPEL débute par un #
  - Elle se base sur la syntaxe de Java et dispose de variables prédéfinies
- Documentation:
  - https://docs.spring.io/spring-framework/docs/3.0.x/reference/expressions.html

Beans: Principes de fonctionnement

- En Spring, tous les objets sont des proxies
  - Pour des raisons de performances (virtual proxies)
- Permet d'ajouter du comportement dynamiquement
  - Notamment par la Programmation par Aspect (AoP)
- Les méthodes privées et appels internes ne sont pas accessibles via les proxies
  - Source de bugs dans l'utilisation de Spring si on n'y prête pas attention



Chargement automatique des Beans

- La configuration du contexte applicatif permet, en partie, de configurer la Bean Factory
  - En fournissant notamment des méthodes permettant d'instancier des Beans
- La configuration des composants peut être automatisée par le scanning
  - Il faut alors autoriser la configuration du contexte applicatif à charger certains Beans automatiquement
  - La chargement automatique de Beans repose sur le scanning

Chargement automatique des Beans

- Le chargement automatique des composants passe par 2 annotations
  - Une annotation "@ComponentScan" avec en paramètres les packagages où chercher les composants à charger
  - Une annotation "@Component" sur les Beans à charger

- L'annotation "@Component" est spécialisée par d'autres annotations
  - L'annotation "@Service" sert à indiquer que le Bean à charger implante une logique métier
  - L'annotation "@Repository" sert à indiquer que le Bean à charger sert à gérer la persistance d'objets métiers

Chargement automatique des Beans

- Le contexte de l'application scanne les composants dans les paquetages passés et dans tous leurs sous-paquetages
  - La définition de ces composants est automatiquement chargée
  - L'injection de leurs dépendances est réalisée essentiellement sur la base de l'annotation @Autowired
  - L'injection de valeurs est réalisée sur la base de l'annotation @Value
  - Le scanning doit être explicitement appelé au démarrage de l'application
  - Dans le cas d'une application Spring Boot, le scanning est implicite
- Lorsqu'il existe plusieurs implémentations pour un même composant, il est possible d'en sélectionner une en particulier grâce à l'annotation @Qualifier

Cycle de vie des Beans

1. Instanciation de la Bean Factory
    1. Chargement de la définition des beans: Annotations ou XML, proxies uniquement
    2. Post-traitement de la définition des beans

2. Pour chaque Bean
    1. Instanciation: dans l'ordre pour respecter les dépendances
    2. Exécution des setters: injection des données et des dépendances
    3. Initialisation: Pré-initialisation avec @PostConstruct

3. La méthode annotée @PreDestroy est appelée juste avant que l'objet soit nettoyé

Portée des Beans

- Singleton: portée par défaut, une seule instance par contexte
- Prototype: chaque référence créé une nouvelle instance
  - Les objets sont nettoyés par le GC dès qu'ils ne sont plus utilisés
  - Intéressant pour les données "transient"
- Session: similaire au prototype
  - Utilisé dans un environnement web uniquement
  - Une instance par session utilisateur
- Request: similaire à la session
  - Une instance par requête
- Les Beans n'accèdent pas au contexte de l'application
  - Excepté les Beans "Context-Aware" - à utiliser avec précaution

TP: IoC

- Créer un projet java simple avec maven
- Créer un paquetage et ajouter deux classes comme indiqué dans les encarts
  - A compléter
- Créer une classe "Main" avec une méthode main comme indiqué
  - Exécuter la classe Main
- Passer par une interface et fournir plusieurs implantations

```
public class Hello {
  private String message;
  public String getMessage()
{
    return message;
  }
}
```

```
public class Display {
  private Hello hello;
  public void displayMessage() {
    System.out.println(hello.getMessage());
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Hello hello = new Hello("Hello World");
    Display display = new Display(hello);
    display.displayMessage();
  }
}
```

TP: IoC

- Il s'agit de mettre en œuvre le principe de l'IoC
- Il faut tout d'abord importer deux librairies Spring
  - Spring-core
  - Spring-context
- Ensuite, il s'agit de définir la configuration du contexte
  - Cela passe par la spécification d'une classe annotée @Configuration
  - La classe de configuration possède deux méthodes permettant d'instancier
    - Un bean Hello et un bean Display
    - Les deux méthodes sont annotées @Bean
- Il faut enfin modifier la classe Main
  - Pour instancier un objet ApplicationContext
  - Nous choisissons une implantation basée sur les annotations: AnnotationConfigApplicationContext
  - Nous pouvons alors demander le chargement du bean Display et appeler la méthode displayMessage

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
</dependency>
  <dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
</dependency>
```

TP: Configuration

- Créer dans le dossier "resources" un fichier hello.properties
  - Avec une entrée "app.greeting=Hello World"
- Ajouter à la configuration du contexte ce fichier comme source de propriétés
  - Par l'ajout de l'annotation @PropertySource("classpath:hello.properties")
  - Les propriétés définies dans ce fichier sont maintenant disponibles pour l'application
- Supprimer le constructeur de la classe Hello et injecter la valeur de la propriété app.greeting
  - Par l'ajout de l'annotation @Value("${app.greeting}") sur un attribut de la classe

TP: Configuration

- Dans la configuration du Main au niveau de l'outil, ajouter une variable d'environnement app.greeting avec la valeur "Hello SPRING"
  - Exécuter et observer le résultat

- Créer dans l'outil une deuxième configuration "dev"
  - Utiliser SPEL pour configurer le message à afficher selon les profils; tester
    - "Hello Dev" si profil dev, "Hello Prod" sinon
    - Paramétrage de la JVM: -Dspring.profiles.active=dev

TP: Configuration

- Nous allons maintenant configurer le chargement automatique des beans

- Il faut tout d'abord indiquer à la configuration de scanner les beans à charger automatiquement au démarrage
  - Par l'annotation @ComponentScan et en indiquant le paquetage racine à partir duquel rechercher les composants à charger automatiquement
  - Par l'annotation @Component sur les beans à charger automatiquement

- Supprimer le bean Hello de la configuration

TP: Configuration

- Marquer la classe Hello avec l'annotation @Component
- Modifier la classe Display en conséquence
  - Suppression du constructeur
  - Injection de la dépendance au bean Hello par l'utilisation de l'annotation @Autowired
  - Tester et observer
- Supprimer toutes les méthodes de la configuration et marquer la classe avec l'annotation @Component
  - Tester et observer

AOP: Présentation

- Les principes du paradigme objet supportent une meilleure capitalisation / réutilisation
  - Par l'encapsulation et l'emploi de différents patrons de conception
  - La POO améliore le développement logiciel au niveau architectural
  - Elle reste cependant limitée au niveau comportemental
- Le paradigme AOP permet de mieux séparer les préoccupations au niveau comportemental
  - La définition d'un objet résulte généralement de la composition de diverses préoccupations: métier, performance, sécurité, authentification, etc.
  - Le but de l'AOP est un découplage et une meilleure réutilisation de préoccupations transverses
- L'AOP se combine bien à l'IoC
  - Elle permet un tissage en cours d'exécution de différentes préoccupations
  - Le pattern Proxy utilisé par l'IoC se prête bien au tissage de comportement en cours d'exécution

AOP: Principes

- Les aspects représentent des blocs de code réutilisables pouvant être injecté à l'application en cours d'exécution
  - Évite la duplication de code et permet une meilleure réutilisation de bouts de code
  - Facilite la maintenance
  - Améliore la qualité, notamment en rendant le code plus lisible

- Exemples d'application:
  - Logging
  - Gestion des transactions
  - Mise en cache
  - Sécurité

AOP: Concepts

- **Aspect:** module définissant des greffons et leurs points d'activation
- **Greffon (Advice):** un programme activable à un certain point d'exécution du système, précisé par un point de jonction
- **Tissage (Weaving) :** insertion statique ou dynamique dans le système logiciel de l'appel aux greffons
- **Point de coupe (*Pointcut*) :** endroit du logiciel où est inséré un greffon par le tisseur d'aspect
- **Point de jonction (Join Point)** endroit spécifique dans le flot d'exécution du système, où il est valide d'insérer un greffon (avant, autour de, à la place ou après l'appel de la fonction)
- **Préoccupation transverse:** sous-programmes distinct couvrant un aspect de la programmation

AOP: AspectJ

- AspectJ est un langage supportant le paradigme AOP
- Il a été proposé et initialement développé par Xerox au début des années 2000
  - Sur le base du langage Java
  - Dans l'environnement Eclipse
- Il supporte le tissage statique et dynamique
- Il est bien intégré dans l'environnement SPRING
- Pour aller plus loin avec AspectJ
  - https://www.eclipse.org/aspectj/doc/released/progguide/index.html

TP: AOP

- Ajouter une dépendance à Log4j dans le fichier pom.xml et importer les librairies
- Configurer log4j pour afficher les logs sur la sortie standard (fichier log4j.properties)
- Modifier le code des classes Hello et Display afin de tracer les appels aux opérations
  - Respectivement getMessage() et displayMessage()
  - En instanciant un Logger au niveau de chaque classe
- Exécuter et observer le résultat

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-reload4j</artifactId>
</dependency>
```

```
log4j.rootLogger=INFO, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{dd-MM-yyyy HH:mm:ss} %-5p %c{1} - %m%n
```

TP: AOP

- Nous allons maintenant "aspectiser" les traces
- Il s'agit de créer un aspect qui va tisser le comportement du log aux fonctions
- Il faut pour ce faire ajouter une dépendance à AspectJ dans le fichier pom.xml
- Nous allons créer une annotation @Loggable que nous utiliserons pour injecter du comportement aux fonctions

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Loggable {}
```

TP: AOP

- Ensuite nous créons notre aspect qui définit
  - Un point de coupe
  - Des greffons: avant, pendant et après l'appel
- L'aspect est un simple bean chargé automatiquement et annoté @Aspect
- Le point de coupe est une méthode
  - Ne retournant rien
  - Annoté @Pointcut avec en paramètre la condition du tissage (annotation, appel, exception,…)

TP: AOP

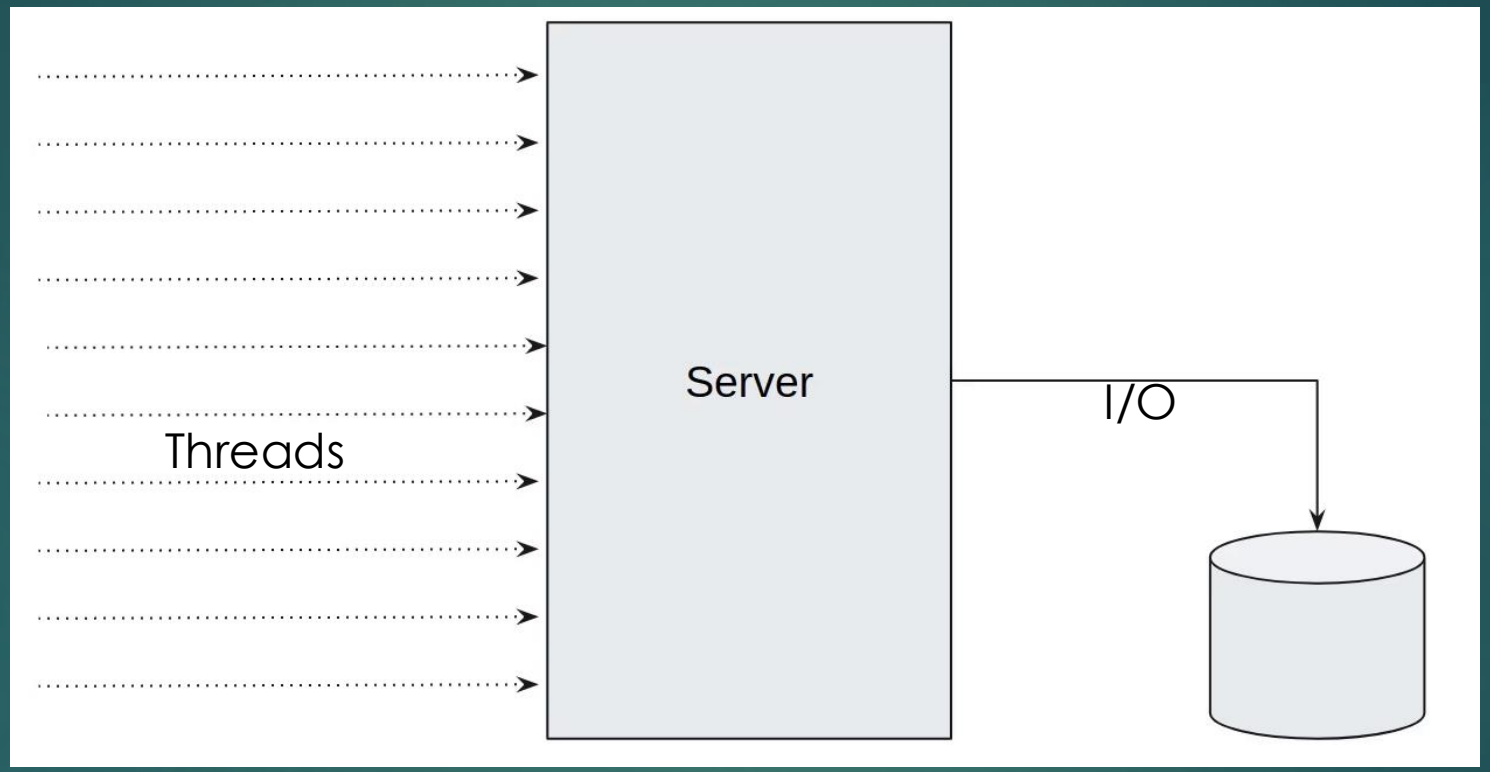
- Nous allons mettre en place 3 types de greffon:
  - Celui exécuté avant l'appel à l'opération identifié par l'annotation @Before
  - Celui exécuté pendant l'appel à l'opération identifié par l'annotation @Around
  - Celui exécuté après l'appel à l'opération identifié par l'annotation @AfterReturning
- Il est également nécessaire d'activer les aspects au niveau de la configuration
  - Par l'utilisation de l'annotation @EnableAspectJAutoProxy
- Tester et observer

TP: AOP

- Mettre en œuvre un autre aspect qui modifie la valeur de retour de la méthode getMessage de la classe Hello
- Implanter un aspect qui log le nombre d'appels à chaque méthode
- Implanter un aspect qui décryptent les chaînes de caractères avant d'appeler une méthode et qui cryptent les chaînes de caractères en sortie

Patterns

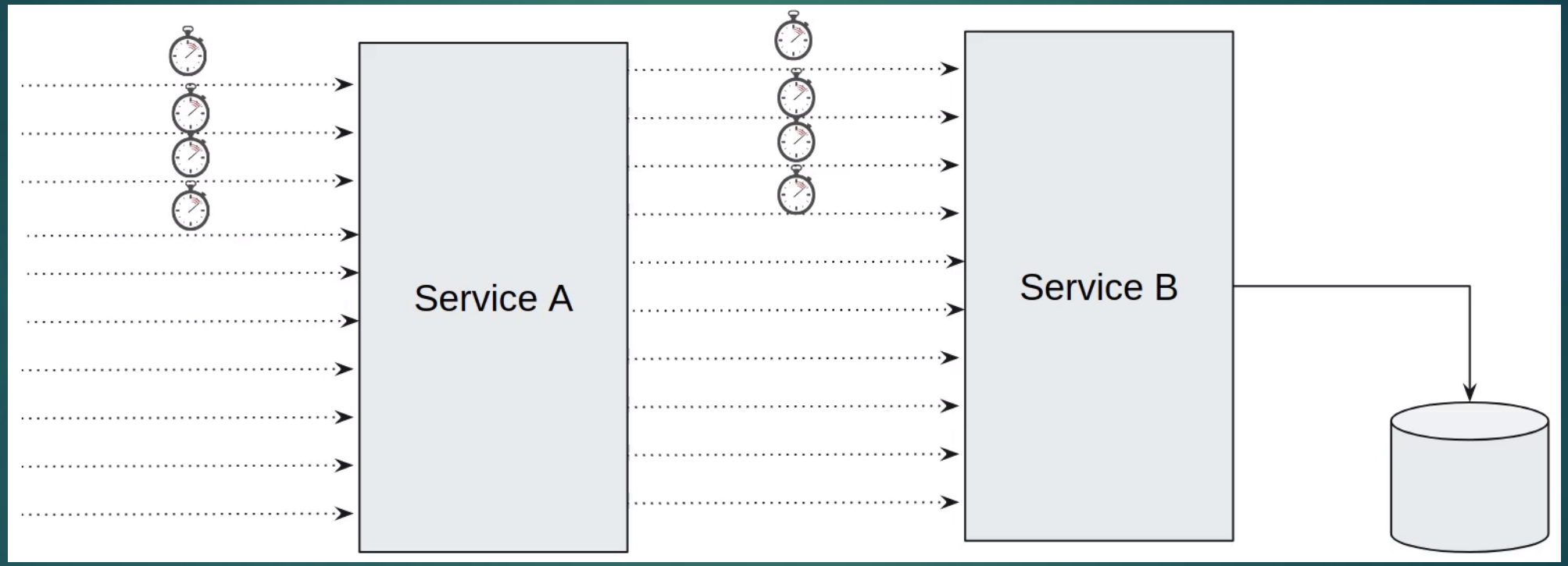
Fonctionnement classique d'un serveur



Une requête => Un thread

Quels sont les principaux problèmes de cette approche ?

Fonctionnement d'un serveur dans un contexte de micro-services

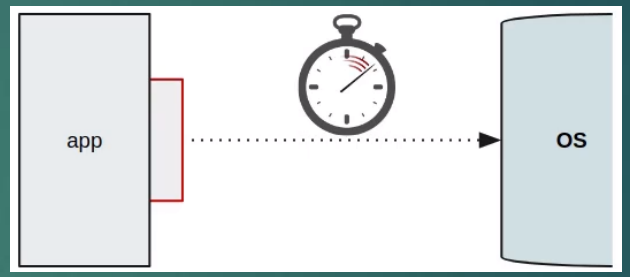
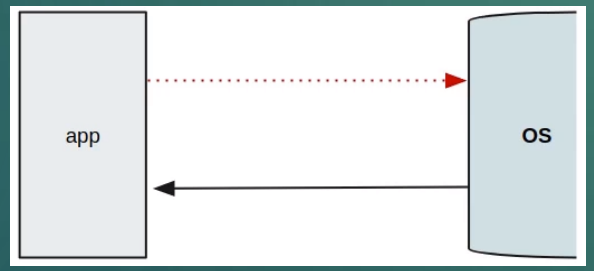
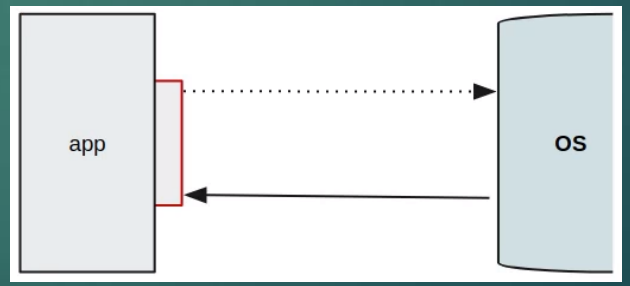


Approche consommatrice de ressources peu scalable

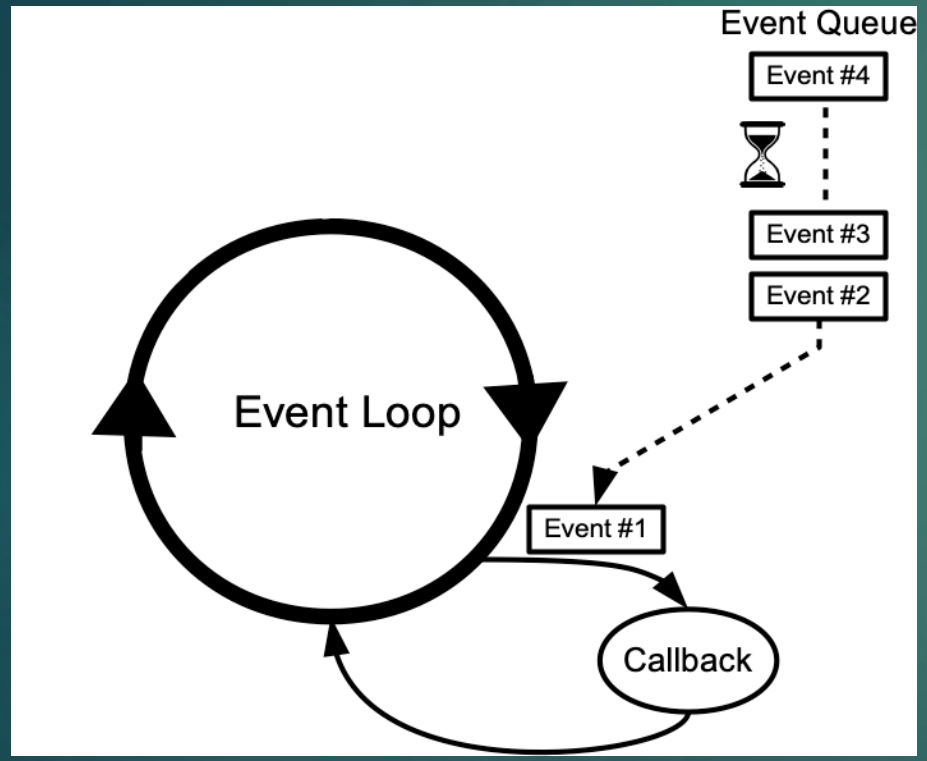
Quelles solutions mettre en œuvre pour traiter ce problèmes ?

Quels problèmes sont posés par cette approche ?

Patterns de communication

	Synchrone	Asynchrone
Bloquant		
Non Bloquant		

Event-Driven programming



- Boucle infinie qui scrute une queue d'événements et exécute des tâches associées (callback)
- **Quel est le problème de cette approche ?**

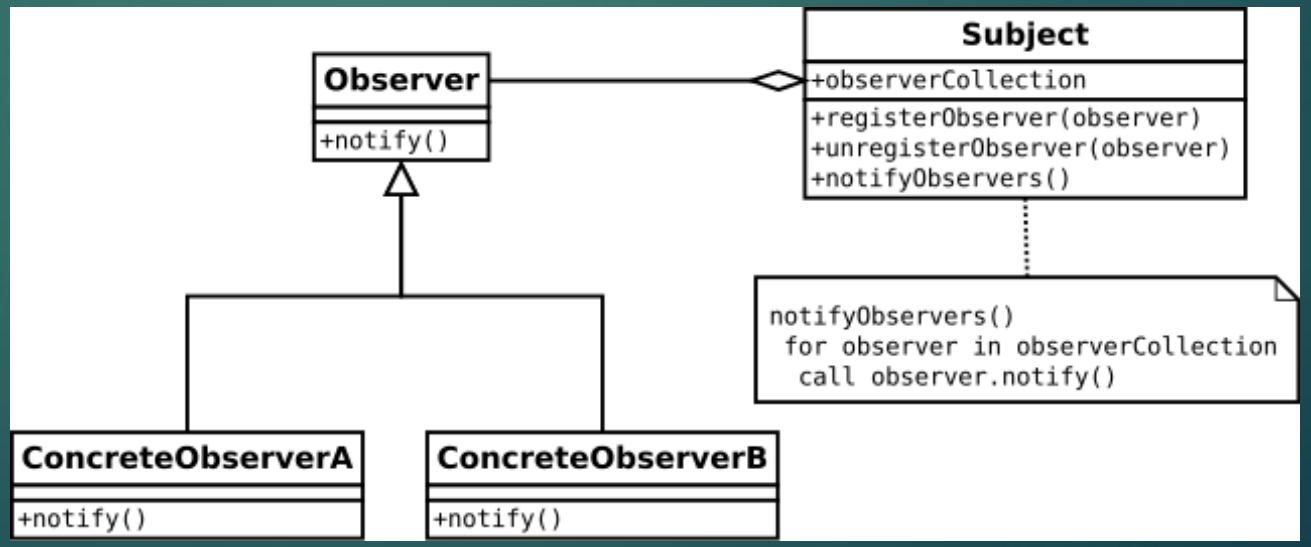
"Call-back Hell"

```
userRepository.getUser(username, (user) => {
    let userId = user.getUserId();
    orderService.getOrders(userId, (orders) => {
        paymentService.getStatus(orders, (results) => {
            ...
            ...
        });
    });
});
```

À partir du moment où les fonctions ont besoin de consommer des données produites par d'autres fonctions, il faut alors imbriquer les appels afin de gérer leur asynchronicité

Observer

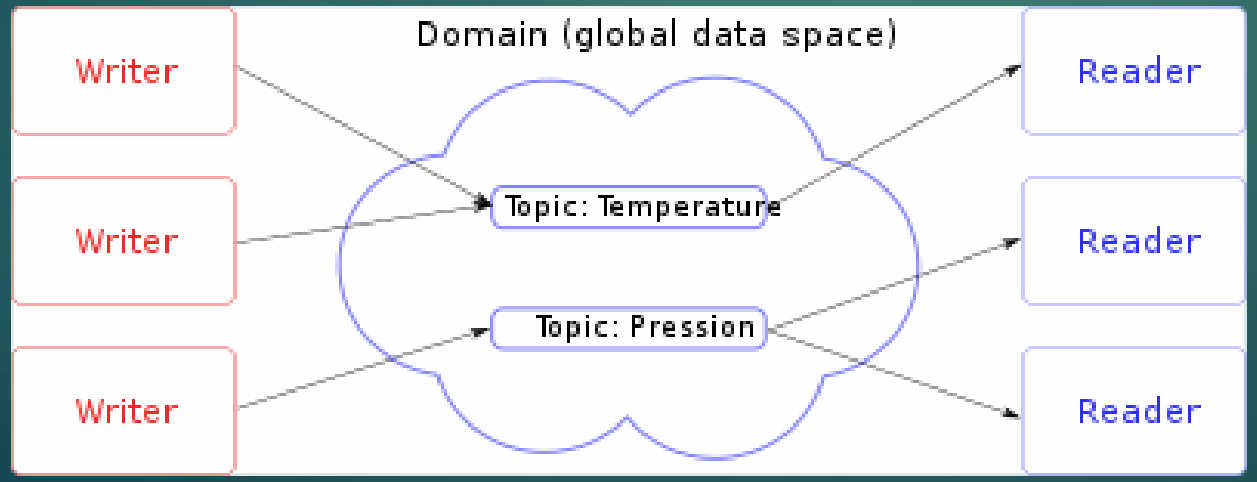
- Permet de limiter le couplage entre objets aux seuls phénomènes (événements) observables
- Un sujet observable notifie ses changements d'état à ses abonnés (observateurs)
  - Chaque abonné est responsable des conséquences des changements observés sur le sujet



Publish / Subscribe

Mécanisme de publication et d'abonnement dans lequel:

- Les diffuseurs (publishers) n'émettent pas des messages à des destinataires (subscribers) particuliers
- Les diffuseurs émettent des messages à des catégories particulières (topics) sans avoir à se soucier si ces derniers seront consommés par des destinataires quelconques
- À l'inverse, les destinataires s'abonnent à des catégories sans soucier de savoir si il y aura des producteurs, ni même qui produira des messages



Quelle différence avec le pattern Observer ?

Quelles technologies utilisent ce pattern ?

Quelles en sont les limites ?

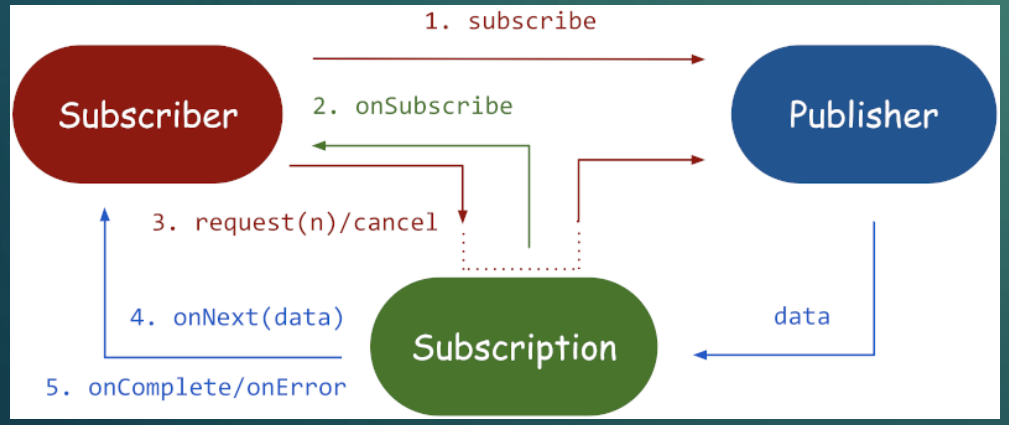
Limites du Publish / Subscribe

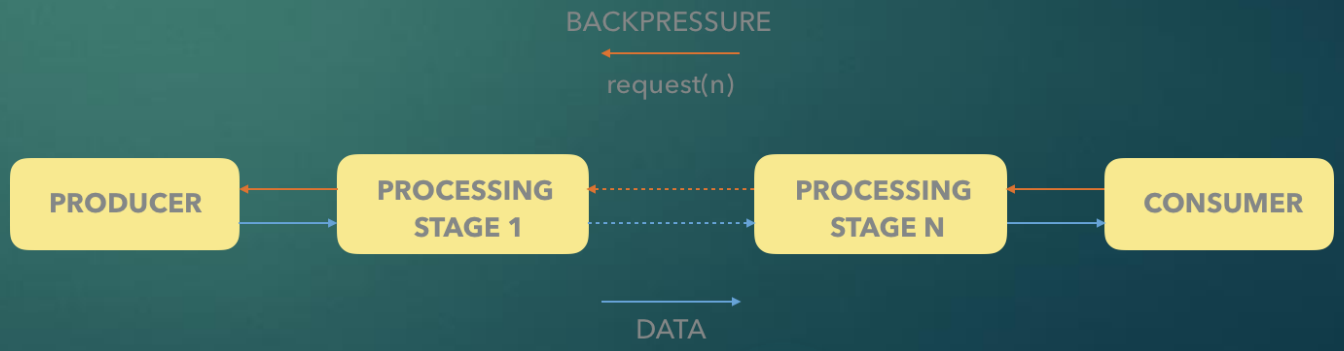
- Mélange des préoccupations
- Le publish / Subscribe est un choix d'implantation
  - Il ne devrait pas être gravé dans le marbre
- Ne passe pas à l'échelle
  - Production de données même si il n'y a pas de composants pour les consommer
  - Gâchis dans l'utilisation des ressources
- Mauvaise gestion des erreurs
- Mauvaise gestion des tests
  - Impossible de tester les composants sans avoir à charger tout le contexte

Réactivité et micro-services

Reactive Streams dans Java

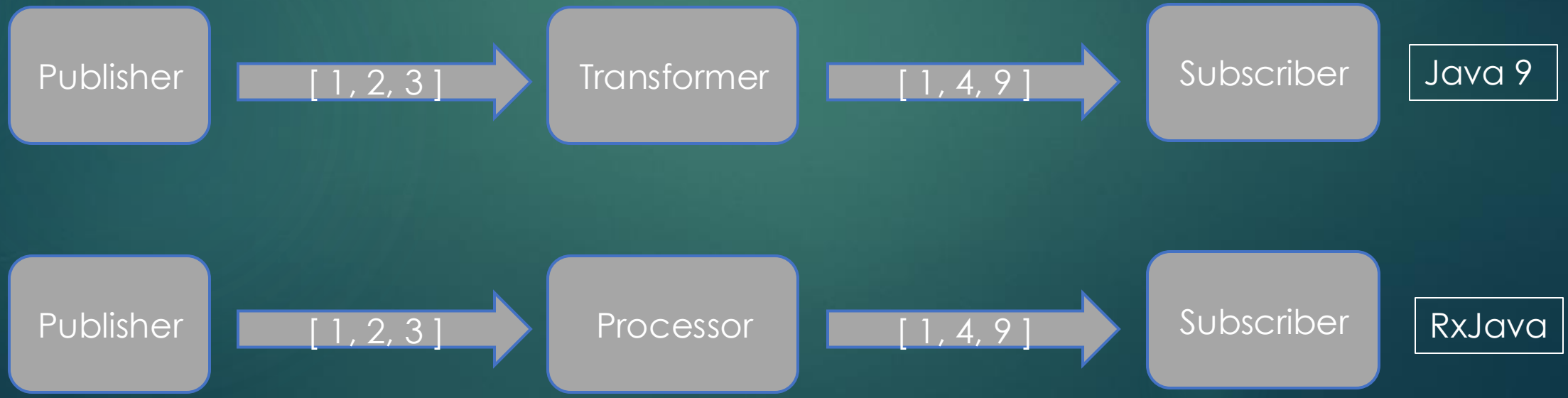
- Initiative visant à établir un standard pour le traitement de flux asynchrones non-bloquants
  - Introduit dans la version 9
- Interfaces définies dans le paquetage java.utils.concurrent.Flow
  - Permettent de créer des programmes asynchrones basés sur des séquences observables d'événements
  - Étendent le patron de conception "Observer" en ajoutant un itérateur
    - Évite de produire de la données s'il n'y a pas de consommateur





Exercice: Reactive Streams

- Implanter la chaîne représentée par ce schéma:
  - Un pubisher publie une série d'entiers
  - Un transformer (publisher / suscriber) met ces entiers au carré
  - Un suscriber stocke les résultats dans un tableau
- Compléter l'exercice Reactive Demo



Reactive Streams: Back-pressure

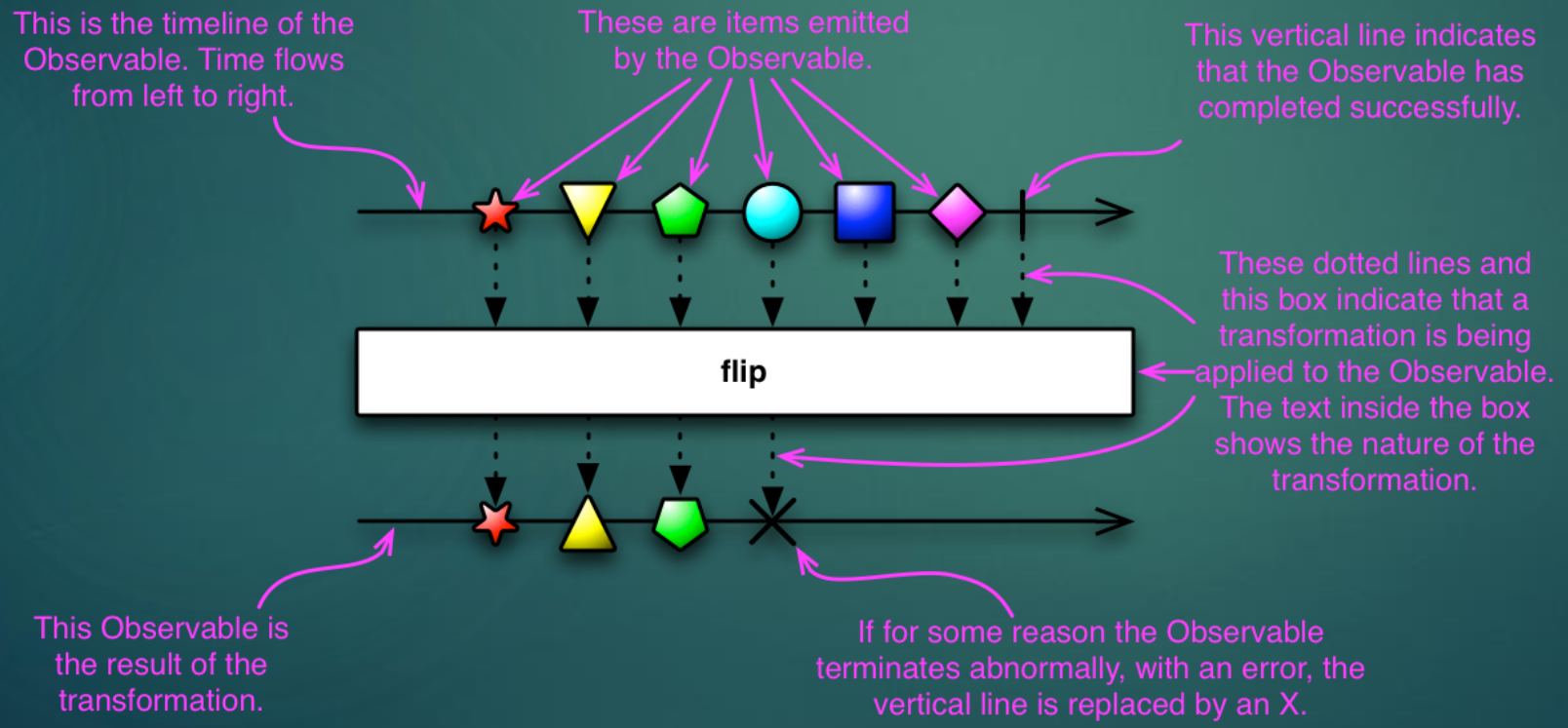
Selon le "Reactive Manifesto", le back-pressure est défini comme suit:

Lorsqu'un composant a du mal à suivre, le système dans son ensemble doit réagir de manière raisonnable. Il est inacceptable que le composant sous tension connaisse une défaillance catastrophique ou qu'il abandonne des messages de manière incontrôlée. Puisqu'il ne peut pas faire face et qu'il ne peut pas tomber en panne, il doit communiquer le fait qu'il est sous tension aux composants en amont et leur demander de réduire la charge. Cette contre-pression est un mécanisme de retour d'information important qui permet aux systèmes de répondre gracieusement à la charge plutôt que de s'effondrer sous celle-ci. La contre-pression peut remonter jusqu'à l'utilisateur, auquel cas la réactivité peut se dégrader, mais ce mécanisme garantit la résilience du système en cas de charge et fournit des informations qui peuvent permettre au système lui-même d'appliquer d'autres ressources pour aider à répartir la charge.

Avantages des Reactive Streams

Basé sur le pattern observer étendu avec des itérateurs

Un observateur s'abonne à un Observable. Ensuite, cet observateur réagit à tout élément ou séquence d'éléments émis par l'Observable. Ce modèle facilite les opérations simultanées car il n'est pas nécessaire de bloquer en attendant que l'Observable émette des objets, mais il crée une "sentinelle" sous la forme d'un observateur qui est prêt à réagir de manière appropriée au moment où l'Observable le fera.



L'avantage des observables est qu'ils peuvent être composés de manière à créer des chaînes de traitements asynchrones.

Cela apporte beaucoup de flexibilité et d'élégance à la programmation

Mix entre push et pull

Reactor

Présentation

- Reactor est né de la nécessité de réaliser des applications logiciels pouvant facilement passer à l'échelle et pouvant gérer de grandes quantités de données
  - Dans le cadre du projet Spring XD (https://docs.spring.io/spring-xd/docs/current-SNAPSHOT/reference/html/)
  - En optimisant l'utilisation des ressources de mémoire et de calcul
  - En optimisant les temps de réponse
  - En offrant un cadre cohérent aux développeurs
- Reactor a été conçu autour d'un pattern comportemental, le "**Reactor pattern**"
  - Événements asynchrones / traitements synchrones
  - https://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf

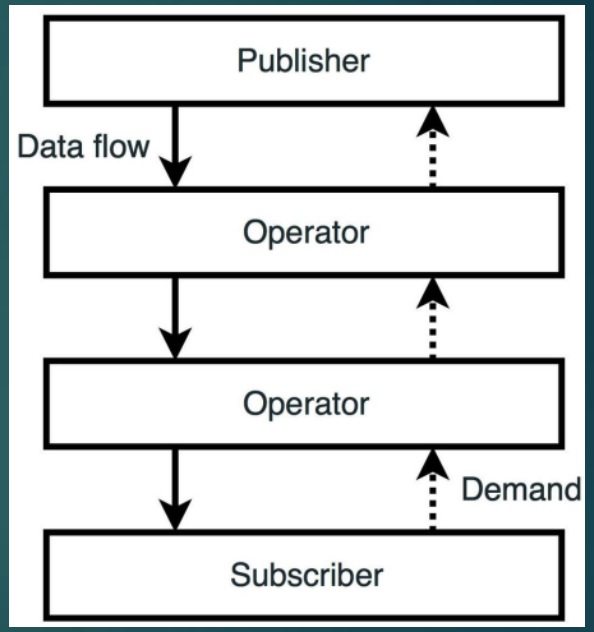
Objectifs

Une grande partie de la complexité de la création d'applications Big Data réelles est liée à l'intégration de nombreux systèmes disparates en une solution cohérente pour toute une série de cas d'utilisation. Les cas d'utilisation courants rencontrés lors de la création d'une solution big data complète sont les suivants

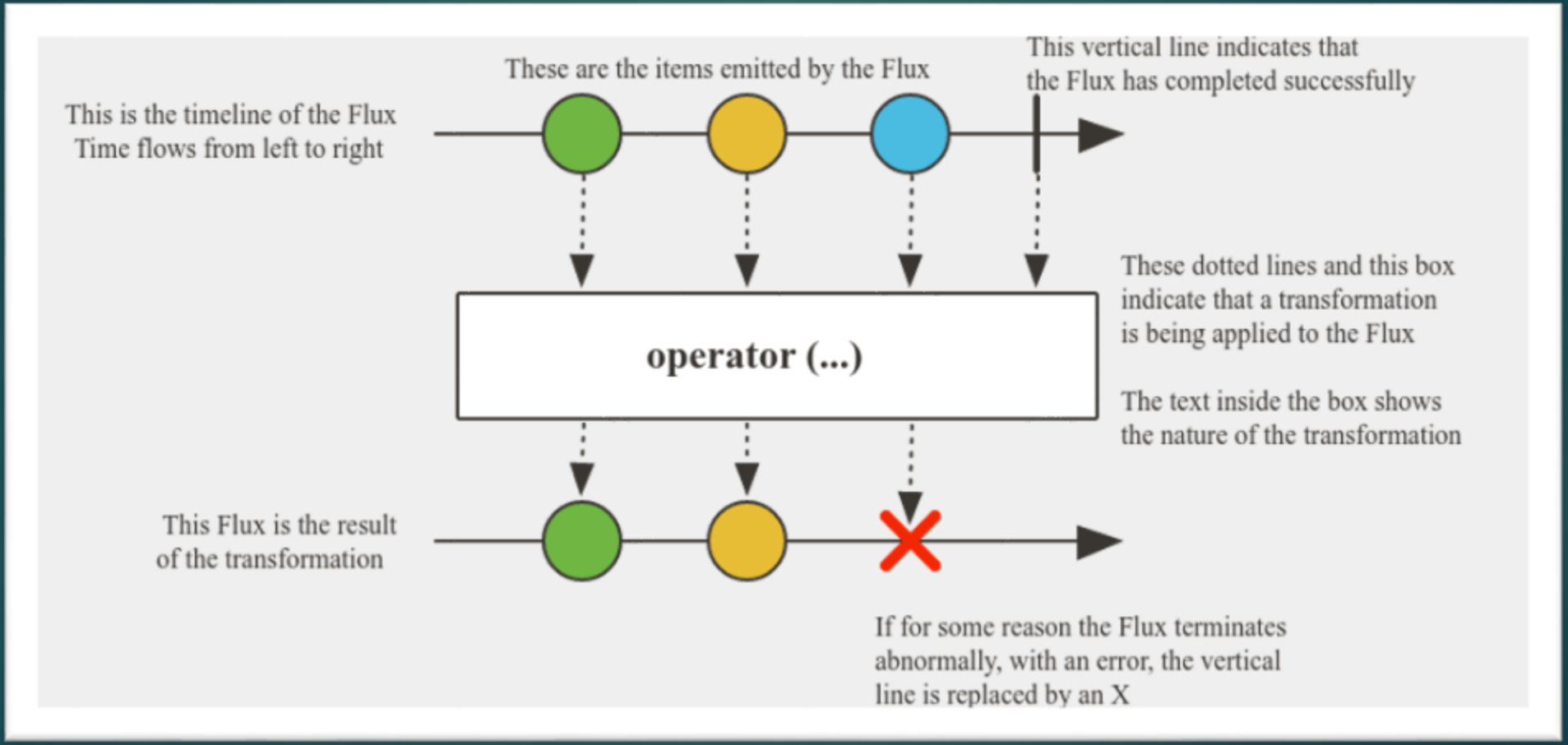
- **L'ingestion de données distribuées à haut débit** à partir d'une variété de sources d'entrée dans un store big data tel que HDFS ou Splunk.
- **Analyse en temps réel** au moment de l'ingestion, par exemple, collecte de métriques et comptage de valeurs.
- **Gestion du flux** via des batchs combinant des interactions avec des systèmes d'entreprise standard (par exemple, SGBDR) ainsi que des opérations Hadoop (par exemple, MapReduce, HDFS, Pig, Hive ou HBase).
- **Exportation de données à haut débit**, par exemple de HDFS vers un SGBDR ou une base de données NoSQL.
- **Composition** La création de valeur repose la composition de services, comprenant leur agrégation et les échanges de données basés sur un couplage faible.

Concepts

- Nous allons retrouver dans Reactor les même concepts que ceux proposés par RxJava: Publisher, Processor, Subscriber
- Avec un cadre de développement plus rigoureux
- Tout en supportant différents paradigmes d'exécution
  - Push → subscription.request(Long.MAX_VALUE)
  - Pull → subscription.request(1)
  - Push-pull → adaptation des rythmes de production / consommation
- Reactor fournit 2 types de publishers
  - **Flux** → 0..N éléments produits, tout comme un **Stream**
  - **Mono** → 0..1 éléments produits, tout comme un **Optional**
  - Avec des opérations de conversion possibles entre les deux



Concepts


These are the items emitted by the Flux
This vertical line indicates that
the Flux has completed successfully
This is the timeline of the Flux
Time flows from left to right
These dotted lines and this box
indicate that a transformation
is being applied to the Flux
operator (...)
The text inside the box shows
the nature of the transformation
This Flux is the result
of the transformation
If for some reason the Flux terminates
abnormally, with an error, the vertical
line is replaced by an X

Mono

- Création
  - Mono.just(element): à utiliser pour des données existantes
  - Mono.fromSupplier(lambda): à utiliser pour des données à calculer
  - Mono.fromRunnable(runnable): à utiliser pour des données asynchrones
  - Mono.fromCallable(callable): à utiliser pour des données à calculer
  - Mono.fromFuture(completableFuture): à utiliser pour des données asynchrones
  - Mono.empty()
  - Mono.error(err)
- Exercices :
  - Compléter la classe MonoCreate
  - Tester les différentes méthodes de création et observer les différences

Mono

- Exécuter le code de MonoSupplier
  - Qu'observez-vous ?
  - Comment y remédier ?

Mono

- Dans le code précédent, on peut observer que les 3 pipelines sont exécutées dans le thread Main
  - On observe en conséquence un comportement non bloquant
- Reactor permet d'exécuter les publishers de manière asynchrone selon différentes stratégies
  - Il faut dans ce cas veiller à bloquer le thread main jusqu'à ce que les threads créés se terminent
  - Il est possible également dans ce cas de bloquer le thread main jusqu'à la complétion de threads créés (Mono::block)
    - Cela créer implicitement un souscripteur qui se bloque en attente du résultat du publisher
    - À éviter autant que possible car cela va à l'encontre de la philosophie réactive

Flux

- Création
  - Flux.just(elements...)
  - Flux.from(publisher)
  - Flux.fromIterable(it)
  - Flux.fromStream(stream)
  - Flux.empty()
  - Flux.error(throwable)
- Génération
  - Flux.range(start, count)
  - Flux.interval(duration)

Exercice:
Tester les différentes méthodes

Flux - Debug

- Il est possible d'afficher les détails des événements dans la production / consommation de données
- Flux::log
  - Ex: Flux.range(1, 10).log().subscribe(System.out::println)

Flux - Subscription

- L'utilisation de la méthode **subscribe()** génére un subscriber avec une implémentation par défaut
- Il est possible de fournir une implémentation custom avec la méthode **subscribeWith()**
  - Il faut fournir une implémentation pour toutes les méthodes de l'interface Subscriber
  - On peut alors manipuler de manière explicite l'objet Subscription créé lors de la souscription

Flux / Mono conversion

- Il peut être utile dans certaines situations de conversion un Mono en Flux, et inversement
  - Flux.from(mono)
- La conversion Flux vers Mono pose la question de la cardinalité des éléments
  - 0..N vers 0..1
  - Plusieurs stratégies sont possibles alors:
    - On sélectionne un seul élément du Stream: next(), à combiner avec un filtre pour récupérer l'élément voulu
    - Générer un mono d'un élément représentant le tableau des éléments contenus dans le flux, possible que dans certaines situations
- Exercice: compléter la classe FluxToMono

Génération d'un flux (potentiellement) infini

- Jusqu'à présent, nous avons étudié des flux finis
- Reactor fournit la possibilité de créer des flux infinis que l'on peut arrêter selon certaines conditions, de différentes manières:
  - Flux::create(sink) - cf. FluxGenerate
  - Flux::generate(synchronousSink), stateless, une seule émission autorisée dans la définition, mais qui est exécutée à l'infini (loop implicite)
  - Flux::generate(stateSupplier, biFunction), idem mais statefull
  - Flux::push, pareil au generateur mais un seul thread par émission
- Exercice: compléter la classe FluxGenerate

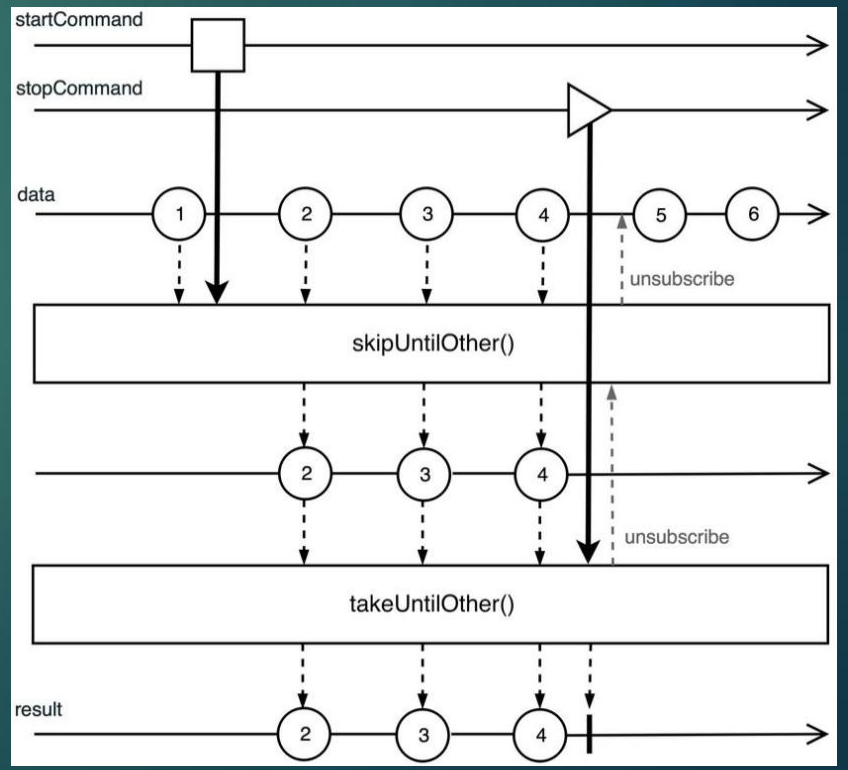
Hooks

- Reactor permet d'ajouter des hooks aux événements associés à la publication
- Ces méthodes sont généralement de la forme doOnXXXX
  - XXXX représentant l'événement sur lequel on désire ajouter un hook
- Exercice: Modifier la classe FluxGenerate afin de tester ces différents hooks

Opérateurs

- Avec Reactor, nous retrouvons à peu de choses près les mêmes **opérateurs** que RxJava
  - Map
  - Filtre (take, takeLast, takeUntil, …)
  - Collection (just, collect, collectSortedList,…)
  - Réduction (any, all, count, …)
  - Combinaison (concat, merge, zip, …)
  - Batch (buffer et variants)
  - Cf. https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

```
Flux.range(0,100)
        .flatMap(v -> Mono.fromCallable(() -> {
                Thread.sleep(500);
                return v;
        }))
        .skipUntilOther(Mono.delay(Duration.of(5, ChronoUnit.SECONDS)))
        .takeUntilOther(Mono.delay(Duration.of(10, ChronoUnit.SECONDS)))
        .subscribe(System.out::println);
```



Pour savoir quel opérateur choisir:
https://projectreactor.io/docs/core/release/reference/#which-operator

Opérateurs

- Take(n): retourne un flux contenant n éléments
  - Attention à l'utilisation avec un flux infini
- Map(function): applique une fonction de transformation
- Filter(predicate): filtre selon le prédicat passé en paramètre
- Handle(biConsumer): map + filter
- LimiteRate(num, pourcentage): permet de limiter la taille du pipeline entre le publisher et le subscriber
- DelayElements(duration): permet de produire avec une certaine latence

Opérateurs

- Timeout(duration, fallback): permet de limiter l'attente sur le publisher et de fournir un autre publisher le cas échéant
- DefaultIfEmpty: valeur à retourner si absence de donnée(s)
- SwitchIfEmpty: remplace le publisher si absence de donnée(s)
- Transform: permet de capitaliser / réutiliser des opérateurs complexes
- SwitchOnFirst: permet de remplacer un flux sur la base du premier élément transmit

Opérateurs

- FlatMap: permet de "mettre à plat" des publishers
  - Ex:

```
UserService.getUsers()
  .flatMap(user -> OrderService.getOrders(user.getUserId())) //Flux
  .subscribe(Util.subscriber());
```

- ConcatMap: Similaire à FlatMap à la différence que l'ordre dans la transformation est préservée
- Exercice: exécuter la classe FluxFlatMap et observer le résultat

Gestion des erreurs

- La réalisation de systèmes réactifs repose un paradigme de concurrence / communication qui favorise un couplage faible entre composants
  - Les communications reposent sur la notion de signal
  - Les erreurs sont des signaux particuliers qui sont bien gérés par ce paradigme
- De fait, et conformément à la logique métier, il est possible lorsque une erreur se produit de:
  - Retourner une valeur par défaut (onErrorReturn)
  - Arrêter le flux (onErrorComplete)
  - Changer le flux nominal (onErrorResume)
  - Dans les 2 cas précédents, la souscription est annulée quand une erreur se produit
    - On peut utiliser la méthode onErrorContinue pour éviter d'interrompre le flux
  - Retenter l'exécution de l'opération qui a échouée (retry, retryBackoff)
  - De déférer l'exécution à d'autres flux selon les conditions d'exécution (defer)
- Exercice: compléter la classe FluxErrors afin de tester les différentes stratégies

Gestion du temps

- Comme nous l'avons vu dans les slides précédents, il est possible de définir la durée de vie des événements dans le cache: buffer (duration)
  - Il est également possible d'utiliser la méthode window et ses variantes pour jouer sur la durée et la taille des fifos
- Il est possible de générer des événements à intervalles régulier avec la méthode delayElements(duration)
  - Et même calculer des intervalles entre deux événements avec la méthode elapsed()

```
Logger log = Loggers.getLogger("app-logger");
Flux.range(0,5)
        .delayElements(Duration.ofMillis(100))
        .elapsed()
        .subscribe(e -> log.info("Elapsed {} ms: {}", e.getT1(), e.getT2()));
Thread.sleep(1000);
```

Cold vs Hot publishers

- Jusqu'à présent, nous n'avons utilisé des publishers que de type "cold"
  - Les publishers ne sont pas instanciés tant qu'il n'y a pas de souscripteur
  - Une instance par souscripteur
- Les hot pubishers sont instanciés une seule fois et émettent pour tous les souscripteurs
  - VoD vs direct stream
- La méthode **share** permet de générer un hot publisher à partir d'un cold publisher
- La méthode **publish** permet de préciser davantage les propriétés du partage (ex: nombre de souscripteur min pour lancer l'émission)
  - La méthode **autoconnect** permet de lancer l'émission même en l'absence de souscripteurs
  - La méthode **cache** permet de mettre en cache les données émises pour un replay
- **Exercice:** Modifier la classe FluxDelay afin d'illustrer le concept de Hot Publisher

Combinaisons

- Reactor supporte différentes manières de combiner des publishers
- La méthode **startWith** ajoute au début d'un flux des éléments venant d'un autre flux / collection
- La méthode **concatWith** fait l'inverse, elle ajoute à la fin d'un flux des éléments venant d'un autre flux / collection
  - Cf. Les méthodes concat* pour aller plus loin
- La méthode **merge** permet de fusionner plusieurs publishers de manière non séquentielle, au contraire des méthodes précédentes
- La méthode **zip** permet de fusionner les données (une à une) provenant de différents publishers
- La méthode **combineLatest** permet de fusionner les dernières données créées par chaque publisher à combiner

Batching

- Reactor supporte différentes stratégies pour le traitement par lots
- La méthode **buffer**(n) collecte les données produites par lots de n éléments (liste)
  - La méthode buffer(duration) collecte les données sur une durée
  - La méthode bufferTimeout combine les deux (capacité et durée)
- La méthode **window** diffère de la méthode buffer dans la mesure où elle renvoie un flux pour chaque lot, au lieu d'une liste
- La méthode **groupBy** diffère de la méthode window dans la mesure où elle renvoie différentes flux créés selon certains critères

(Dé)Matérialisation

- Il peut être parfois utile de considérer un flux de données comme un flux de signaux, et inversement

```
Flux.range(1, 3)
        .doOnNext(e -> log.info("data : {}", e))
        .materialize()
        .doOnNext(e -> log.info("signal: {}", e))
        .dematerialize()
        .collectList()
        .subscribe(r-> log.info("result: {}", r));
```

Ordonnanceurs

- Comme pour RxJava, Reactor fournit un certain nombre d'ordonnanceurs utilisables à travers les opérateurs publishOn et subscribeOn
  - single() : optimisé pour les exécutions à faible latence
  - parallel(): optimisé pour les exécutions non-bloquantes rapides
  - elastic(): optimisé pour les exécutions bloquantes / non-bloquantes plus longues
  - boundedElastic(): équivalent à elastic() avec un nombre de tâches actives borné
  - immediate(): pour une exécution immédiate (pas d'ordonnancement)
  - fromExecutorService(): pour une exécution basée sur l'utilisation d'une instance de ExecutorService (API Java)

```
Flux.range(1,10)
        .map(i -> i + 1)
        .map(i -> i * 2)
        .map(i -> i + 1)
        .subscribe(System.out::println);
```

```
Flux.range(1,10)
        .parallel(2)
        .runOn(Schedulers.parallel())
        .map(i -> i + 1)
        .map(i -> i * 2)
        .map(i -> i + 1)
        .subscribe(System.out::println);
```

Manipulation des I/O

- Qu'il s'agisse d'accéder à des fichiers ou à des bases de données
  - À des ressources qu'il est nécessaire d'ouvrir ou de fermer
  - Reactor offre l'équivalent de la construction try-with-resources en Java
- Exercice: Modifier la classe Titanic, remplacer les streams par des Flux.

```
Flux<String> ioRequestResults = Flux.using(
        Connection::newConnection,
        connection -> Flux.fromIterable(connection.getData()),
        Connection::close
);
```

Gestion du back-pressure

Malgré le mécanisme de back-pressure permettant une meilleure résilience du système, il est possible de surcharger un consommateur

- Les traitements par batch peuvent améliorer la situation, mais pas toujours
- Il existe d'autres mécanismes permettant de gérer de manière plus fine le back-pressure
- Le fifotage dans une queue non-bornée des données envoyées (onBackPressureBuffer)
- La suppression des données les plus récentes ne pouvant être traitées (rythme de production > rythme de consommation → onBackPressureDrop)
- La suppression des données les plus âgées ne pouvant être traitées (rythme de production > rythme de consommation → onBackPressureLast)

```
Flux.range(1, 10000000)
        .subscribeOn(Schedulers.parallel())
        .onBackpressureLatest()
        .publishOn(Schedulers.single())
        .concatMap(Mono::just, 1)
        .subscribe(ts);
```

Transformations entre Streams

- Il est possible de définir avec Reactor des fonctions permettant de générer un flux à partir d'un autre
  - En spécifiant une fonction binaire
  - Deux types de transformation: statique ou dynamique

```
Function<Flux<Integer>, Flux<Object>> transfo = stream -> stream
        .index()
        .map(Tuple2::getT1);
Flux.range(100,5)
        .map(i -> i * 2)
        .transform(transfo)
        .subscribe(System.out::println);
```

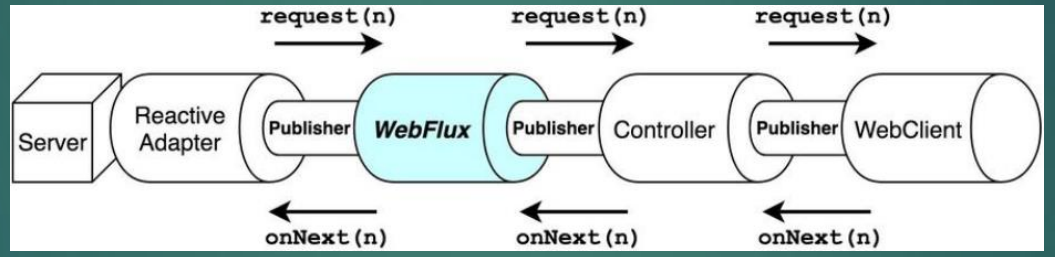
```
Random random = new Random();
Function<Flux<Integer>, Flux<Integer>> composition = stream -> {
        if (random.nextBoolean()) {
                return stream.doOnNext(e -> System.out.println(e * 2));
        } else {
                return stream.doOnNext(e -> System.out.println(e * 3));
        }
};
Flux<Integer> flux = Flux.range(100, 5).transform(composition);
flux.subscribe();
```

Tests

- Reactor fournit un objet permettant de tester des flux
  - Step Verifier
- StepVerifier.create(flux).expectNext(i).verifyComplete()
- StepVerifier.create(flux).expectNext(i).verifyError(err)

Intégration avec Spring Web

- Le module **WebFlux** repose sur la couche Reactive Stream Adapter
  - Il utilise Reactor 3
  - Il assure une meilleure intégration avec les serveurs web (Tomcat, Netty, …)
- WebFlux fournit en particulier la class WebClient qui permet des interactions non-bloquantes avec les clients web




```
@GetMapping
public Flux<String> getHello() {
    return Flux.fromArray(new String[] {"Hello", "World"});
}
```

```
WebClient client = WebClient.create("http://localhost:8080");
Flux<String> flux = client.get().uri("/hello").retrieve().bodyToFlux(String.class);
flux.subscribe(System.out::println);
```

Quarkus

Présentation

- Quarkus est un framework Java open source, créé par Red Hat, conçu spécifiquement pour les besoins des architectures cloud natives, des microservices et des environnements de conteneurs comme Kubernetes.

- La principale motivation de Quarkus est d'optimiser l'expérience Java en réduisant la consommation mémoire et les temps de démarrage, offres cruciales pour le cloud, le serverless et le déploiement à grande échelle.

Caractéristiques fonctionnelles

- Prise en charge de la programmation impérative et réactive : capacité à combiner des approches traditionnelles et modernes (event-driven).
- Compatible avec la JVM et la compilation native via GraalVM, permettant de générer des binaires ultra rapides et légers, adaptés au serverless et au FaaS.
- Mode “live reload” en développement : modification du code avec rechargement automatique de l’application, ce qui accélère l’itération.

Caractéristiques fonctionnelles

- Support des standards et bibliothèques Java : Eclipse MicroProfile, CDI, Hibernate ORM (JPA), RESTEasy (JAX-RS), Kafka, Camel, etc.
- Extensions Quarkus : ajout de fonctionnalités via des modules (REST, sécurité, persistance, etc.) et configuration centralisée.
- Sécurité, résilience, monitoring, configuration unifiée, gestion de l'observabilité grâce à l'intégration avec MicroProfile et autres outils du cloud.

Caractéristiques extra- fonctionnelles

- Démarrage ultra-rapide : jusqu'à 300 fois plus rapide qu'une stack Java classique.
- Empreinte mémoire réduite : parfois 10 fois moins de mémoire que des solutions traditionnelles (Spring Boot, par exemple).
- Optimisé pour le cloud, les conteneurs, le serverless et Kubernetes.
- Sécurité et gestion avancée des configurations et secrets, adaptée au cloud.

Ecosystème

- Large compatibilité avec les outils du cloud : Kubernetes, Docker, OpenShift.
- Intégration avec GraalVM pour la compilation native.
- Extensions nombreuses et variées pour s'adapter aux besoins (REST, GraphQL, Kafka, monitoring, etc.).
- Documentation officielle, guides pratiques, et communauté en croissance rapide.
- Support des standards comme MicroProfile (API Java pour la gestion des aspects transversaux des microservices).

Historique

- Quarkus est lancé par Red Hat en mars 2019, avec comme objectif d'optimiser Java pour le cloud native et les microservices.
- Les premières versions (1.x) introduisent la compilation native GraalVM, un démarrage rapide et de nombreuses extensions (RESTEasy, Hibernate ORM, CDI, etc.).
- Quarkus 2.x (2021-2023) apporte la programmation réactive, l'autoconfiguration Dev Services, plus de portabilité Kubernetes et le support LTS.

Historique

- Quarkus 3.x (2023-2025) représente une évolution majeure : adoption Java 17+ (Java 21 recommandé), Hibernate ORM 6, Virtual Threads, Dev UI et CLI, refonte des extensions, nouveau cycle LTS avec maintenance améliorée.

- Les versions majeures sortent tous les ans, chaque nouvelle version apporte innovations ou ruptures techniques, la communauté Quarkus est très active et l'écosystème (Kubernetes, MicroProfile, GraalVM…) s'élargit continuellement.

Quarkus CLI

- Quarkus propose un outil en ligne de commande officiel, appelé Quarkus CLI, qui facilite la gestion et le développement de projets

- Installation
  - Linux: *sdk install quarkus*
  - Windows: *choco install quarkus*
  - *MacOS: brew install quarkus*

- Alternative: https://code.quarkus.io/

Quarkus CLI

- **Création de projets** : Commande quarkus create pour générer un nouveau projet avec les extensions et configurations souhaitées.
- **Gestion des extensions** : Ajouter, retirer ou lister les extensions avec quarkus extension add, remove, list.
- **Développement en mode live reload** : Lancer l'application en mode développement avec quarkus dev, pour bénéficier du live coding.
- **Build et tests** : Compiler (quarkus build), tester (quarkus test) et packager un projet simplement.

Quarkus CLI

- **Contrôle des images de conteneur** : Générer des images Docker ou autres via les commandes CLI intégrées.
- **Déploiement** : Support du déploiement sur Kubernetes, OpenShift, Knative etc. directement via des commandes CLI.
- **Gestion des plugins/updates** : Mise à jour des extensions ou du projet, gestion des plugins CLI.

Graal VM

- JDK haute performance basé sur un compilateur juste à temps (JIT).
- Il permet de compiler les applications en binaires natifs performants.
- Il prend en charge la programmation polyglotte permettant l'utilisation de bibliothèques de plusieurs langages dans une seule application.

Graal VM

- Il fournit des outils avancés pour le débogage, la surveillance, le profilage et l'optimisation de la consommation des ressources entre les différents langages.

- GraalVM et Micronaut ont un haut niveau d'intégration et de compatibilité, ce qui facilite la création d'images natives d'applications Micronaut à l'aide de GraalVM.

Première application

- Installation de SDKMan
  - curl -s "https://get.sdkman.io" | bash
  - sdk update
- Installation de Quarkus
  - sdk install quarkus
- Installation de Graal (pour créer des images natives)
  - sdk install java 23.1.8.r21-mandrel #Recommandé pour quarkus
  - native-image --version #composant pour générer des images natives
- Création de l'application
  - quarkus create app com.akfc.training.hello --extensions=resteasy

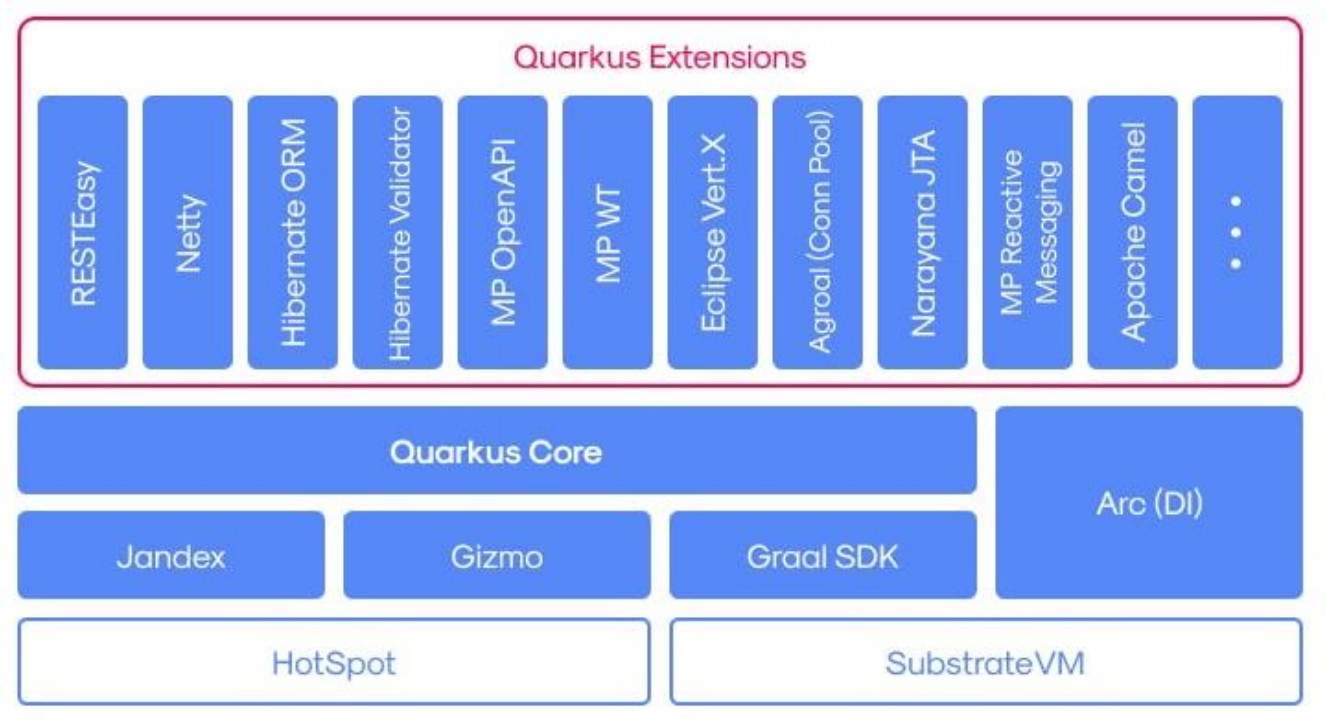
Exercice 1 : Première application

```java
@Path("/hello")
public class HelloCtrl {

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

```java
@QuarkusTest
public class HelloCtrlTest {

    @Test
    public void testHelloEndpoint() {
        given()
            .when().get("/hello")
            .then()
                .statusCode(200)
                .body(is("Hello Quarkus!"));
    }
}
```

Test
$ ./gradlew :users:test
Exécution
$ ./gradlew :users:quarkusDev
Génération d'une image native
$ ./gradlew :users:buildNative
Génération d'une image docker déployable
$ ./gradlew :users:build -Dquarkus.container-image.build=true

Architecture



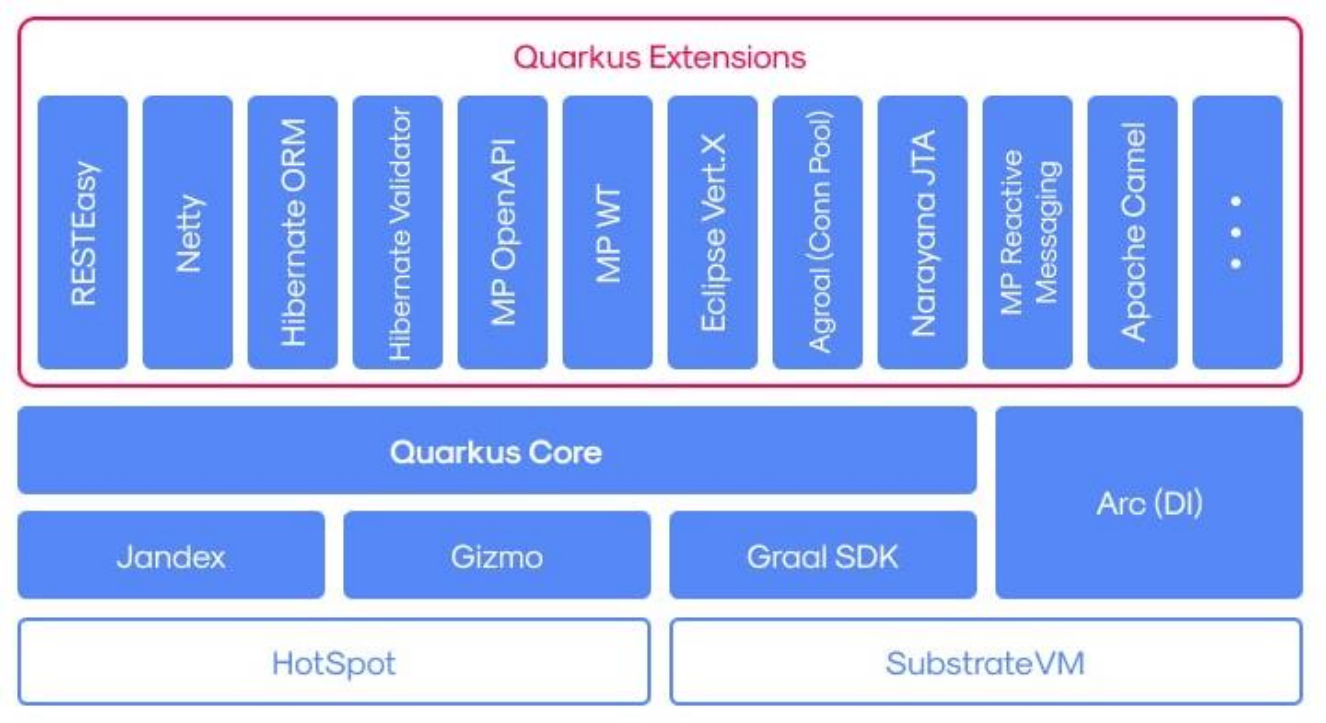
Runtimes JVM ou natif

- **HotSpot** : exécution classique sur la JVM standard.
- **SubstrateVM** : exécution via GraalVM en natif, pour un démarrage ultra-rapide et une faible consommation mémoire.

Composants bas niveau

- **Jandex** : indexe les méta-données Java pour accélérer la découverte des annotations et l'injection de dépendances.
- **Gizmo** : génère du bytecode à la compilation pour réduire l'usage de la réflexion (reflection).
- **Graal SDK** : supporte la compilation d'applications en images natives via GraalVM ou Mandrel.
- **Arc (DI)** : implémentation légère et performante de CDI pour l'injection de dépendances.

Architecture



Quarkus Core

- Cœur du framework, il assure la logique principale, la gestion des cycles de vie, la configuration centralisée, la compilation à la volée, l'injection de dépendances, etc.
- Il orchestre la collaboration entre les extensions et les composants techniques sous-jacents.

Quarkus Extensions

- Les extensions permettent d'intégrer facilement des technologies et frameworks (RESTEasy pour les APIs REST, Netty pour le serveur HTTP, Hibernate ORM pour la persistance, OpenAPI, Vert.X, Camel, etc.).
- Elles offrent une abstraction pour interfacer l'application Quarkus avec un large écosystème, tout en profitant des optimisations Quarkus (statique, buildtime, etc.).

Configuration

- Une application Quarkus bénéficie des mêmes mécanismes de configuration que Spring-Boot
- application.properties pour définir un ensemble de clé / valeur; éventuellement préfixé
- L'annotation @ConfigProperty pour injecter une valeur définie dans le fichier application.properties
  - @ConfigProperty("my.app.prop1")
  - @ConfigProperty(name = "my.app.prop1", defaultValue = "Salut")
- Environnements:
  - `%dev.greeting.message=Hello`
  - ./gradlew -Dquarkus.profile=prod :users:quarkusDev
- **Exercice 2** : Injecter une valeur message qui diffère en fonction de l'environnement

Configuration

- Pour des structures de configuration complexes (regroupement de propriétés, objets imbriqués), il est recommandé de définir une interface annotée
- **Exercice 3**: Injecter le message et le nom à travers une structure de données

```
@ConfigMapping(prefix = "greeting")
public interface HelloConfig {
    String message();
    String name();
}
```

```
@Inject
private HelloConfig config;
```

Inversion de Contrôle

- Contrairement à Spring qui utilise la réflexion pour gérer l'injection de dépendances, Quarkus résout les dépendances à la compilation

- Afin d'améliorer les performances, Quarkus a fait le choix de:
  - N'utiliser la réflexion qu'en dernier ressort
  - Éviter les proxies
  - Réduire l'empreinte mémoire
  - Améliorer les temps de chargement

Beans

- Dans Quarkus, c'est le container CDI (nommé ArC) qui gère la découverte, l'instanciation et le cycle de vie des beans : il n'expose pas d'API directe équivalente à la BeanFactory du monde Spring.

- Pour déclarer un bean dans Quarkus, il suffit d'annoter une classe avec un scope (@ApplicationScoped, @RequestScoped, @Singleton…), et le container CDI gère tout de manière transparente.

- On ne manipule pas, ni n'instancie explicitement, de bean factory ou de contexte d'application. L'injection, la sélection de beans (avec @Inject, @Qualifier…), ou l'injection dynamique (@Inject Instance<T>) sont assurées par le CDI au runtime.

Beans et cycle de vie

- Création: automatique par le container CDI lorsqu'une injection est demandée ou en fonction du scope.
- @PostConstruct : méthode appelée juste après l'instanciation, pour l'initialisation.
- Utilisation/Injection : le bean est utilisé en fonction de son scope (partagé, temporaire, etc.).
- @PreDestroy : méthode appelée avant destruction pour du "cleanup" (shutdown du pool par exemple).
- Destruction: selon le scope (fin de requête, arrêt de l'application, etc.).

Beans et scope

- **@ApplicationScoped**
  - Un seul et unique bean créé et partagé tout au long de la vie de l'application.
  - Instancié "lazily" (à la première invocation) grâce à un proxy.
  - Détruit uniquement à l'arrêt de l'application.
  - Idéal pour les services sans état lié à l'utilisateur ou à une requête.

- **@Singleton**
  - Uniquement une instance, créée "eagerly" (dès le démarrage) sans proxy.
  - Utilisé pour les composants de très bas niveau qui doivent exister immédiatement.
  - Performances légèrement meilleures, mais à utiliser si tu veux explicitement éviter le proxy.

Beans et scope

- **@RequestScoped**
  - Une nouvelle instance pour chaque requête HTTP.
  - Instanciée et détruite à chaque cycle de requête.
  - Permet de stocker un état temporaire lié à une requête/utilisateur.
- **@Dependent**
  - Aucune instance partagée : une nouvelle instance à chaque injection, liée à l'injecteur.
  - Cycle de vie "dépendant" du bean consommateur.
  - Utilisé pour les beans légers et sans état.
- **Autres scopes personnalisés**
  - Extensions Quarkus ou CDI peuvent fournir d'autres scopes comme @TransactionScoped pour la gestion de transaction.

Beans: Cycle de vie

- Dans le cas où un bean possède plusieurs constructeur, il est possible d'indiquer lequel utiliser
  - Par l'annotation @Inject

- Un bean est considéré immutable si toutes ses propriétés sont marquées "final"
  - Les beans immuables sont thread-safe : plusieurs threads peuvent les partager sans synchronisation.
  - En mode @ApplicationScoped, ils sont instanciés une seule fois et réutilisés partout, ce qui évite les problèmes de concurrence.
  - C'est cohérent avec la philosophie "build-time initialization" de Quarkus : ce qui est figé à la création réduit les coûts d'exécution.

Beans: Sélection

- Dans le cas où plusieurs implémentations d'une même interface sont fournies par différents beans, différents mécanismes sont offerts par Quarkus pour sélectionner l'une ou l'autre implémentation
- Le qualifier – Utilisation d'annotations customs
- Le nom (@Named) – Alternative légère au qualifier
- Par défaut (@DefaultBean)
- Par priorité (@Alternative et @Priority)
- Par profil (@ifBuildProfile)
- Sélection dynamique (@Any)

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD})
public @interface Dev {}
```

```
@ApplicationScoped
@Dev
public class DevService implements HelloService {...}
```

```
@Inject
@Dev
private HelloService hello;
```

Introspection des beans

- L'introspection dans Quarkus repose sur une approche statique et anticipée (build-time), radicalement différente de la réflexion dynamique classique utilisée par de nombreux frameworks, pour rester compatible avec GraalVM (et donc avec les binaires natifs).
- **Jandex** parcourt les classes du projet et construit un index des informations de type, d'annotations, et de signatures.
  - Cet index est enregistré dans META-INF/jandex.idx et lu par Quarkus et ses extensions au démarrage.
  - Il permet au framework de "voir" les annotations (ex. @Inject, @Entity, @Path, etc.) sans utiliser la réflexion.
- **Gizmo** est utilisé pour générer du code au build-time — un peu comme si Quarkus écrivait, compilait et intégrait automatiquement certaines classes.
  - Il remplace les opérations de réflexion en générant des classes spécialisées capables d'invoquer directement les méthodes ou d'accéder aux champs.
  - Exemple : Quand RESTEasy ou Jackson doivent sérialiser un objet, Quarkus peut, via Gizmo, produire une classe de sérialisation spécifique au lieu de recourir à java.lang.reflect.

Introspection des beans

- Le système de build Quarkus repose sur un cycle d'augmentation en plusieurs étapes, chaque extension apportant ses propres Build Steps :
  - Les Build Steps utilisent les Build Items (objets de configuration) pour indiquer quelles classes doivent être introspectées ou rendues accessibles.

- Les extensions Quarkus (RESTEasy, Panache, Hibernate, etc.) tirent parti de cette architecture :
  - Elles lisent l'index Jandex pour détecter les entités, endpoints ou services CDI.
  - Elles génèrent au besoin le bytecode additionnel requis par Gizmo.
  - Elles exportent ensuite les métadonnées nécessaires vers le runtime (notamment via Arc, le conteneur CDI de Quarkus).

Introspection des beans

Avantage	Explication
Performance	Pas de scan de classes ni d'opérations réflexives au démarrage.
Compatibilité native	Respecte les contraintes de GraalVM (monde fermé).
Faible empreinte mémoire	Les classes d'analyse (ex. processeurs d'annotations) ne sont pas embarquées dans le runtime.
Sécurité renforcée	Moins de manipulation dynamique limite l'exposition des APIs internes.

AOP

- Micronaut fait un usage intensif de la programmation par aspect que nous avons pu avoir au travers de nombreux exemples
  - Pour la validation, à travers l'API javax.validation
  - Pour la programmation asynchrone (@EventListener)
  - Pour la mise en cache (@Cacheable)
  - Pour l'ordonnancement de tâches (@Scheduled)
- Il permet de définir ses propres aspects
  - Par une interface simplifiée
  - 2 types d'advice: Around et Introduction

AOP: Advices

- **Around**
  - Décorateur
  - Équivalent au Around que nous avons déjà vu avec AspectJ
- **Introduction**
  - Permet d'injecter une implémentation aux méthodes d'interface
- Dans les deux cas, il faut:
  - Déclarer une annotation
  - Spécifier un intercepteur

```
@Singleton
public class HelloBean {

    @Loggable
    public String sayHello(@NotBlank String message) {
        return message;
    }
}
```

```
@Around
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Loggable {}
```

```
@Singleton
@InterceptorBean(Loggable.class)
public class LoggableInterceptor implements MethodInterceptor<Object, Object> {

    private static final Logger LOGGER = LoggerFactory.getLogger(LoggableInterceptor.class);

    @Override
    public Object intercept(MethodInvocationContext<Object, Object> context) {
        Object p = context.getParameterValueMap().values().stream().findFirst();
        LOGGER.info(context.getMethodName() + " called with param " + p);
        return context.proceed();
    }
}
```

AOP: Quelques advices utiles

- @Cacheable
  - Indique une méthode "cacheable" dans un cache spécifié au niveau de l'application
- @Retryable
  - Dans le cas d'applications réparties, il peut être utile de tenter de réexecuter des requêtes qui ont échoué
- @Scheduled
  - Permet d'exécuter une tâche de manière périodique
  - Rythme ou délai
  - Cron: @Scheduled(cron = "0 15 10 ? * MON")
- Quelques annotations relatives à l'injection de Streams
  - Exple: @ExecuteOn

```
micronaut:
    caches:
        my-cache:
            maximum-size: 20
```

```
micronaut:
    executors:
        scheduled:
            type: scheduled
            core-pool-size: 30
```

```
@Scheduled(fixedRate = "5m")
void everyFiveMinutes() {
    System.out.println("Executing everyFiveMinutes()");
}
```

```
@Retryable(attempts = "5", delay = "2s")
public Book findBook(String title) {}
```

Gestion des événements

- Micronaut fournit une API permettant une communication par événements: ApplicationEventPublisher
  - Les événements sont synchrones par défaut → Expliciter les mécanismes de synchronisation le cas échéant

```
public class SampleEvent {

    private String message = "Hello World";

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

```
@Singleton
public class HelloEmitter {

    @Inject
    ApplicationEventPublisher<SampleEvent> publisher;

    public void publishEvent() {
        publisher.publishEvent(new SampleEvent("message "));
    }
}
```

```
@Singleton
public class HelloReceiver implements ApplicationEventListener<SampleEvent> {

    @Override
    public void onApplicationEvent(SampleEvent event) {
        System.out.println("Receiving sample event " + event.getMessage());
    }
}
```

```
@Singleton
public class HelloReceiver {

    @EventListener
    @Async
    public void onSampleEvent(SampleEvent event) {
        System.out.println("Receiving sample event " + event.getMessage());
        counter ++;
    }
}
```

Gestion des événements

Micronaut définit un certain nombre d'événements auxquels il est possible de s'abonner

- En particulier sur les événements en lien avec le cycle de vie des beans
  - Il faut passer pour cela par une factory (@Factory)
  - Et utiliser des hooks particuliers:
    - Interface BeanCreatedEventListener<MyFactory>
    - Interface BeanInitializedEventListener<MyFactory>

Gestion des erreurs

- Les erreurs peuvent être gérées au niveau local
  - Au niveau du contrôleur
  - Par l'annotation @Error
- Elles peuvent être gérées au niveau global
  - Par la même annotation
  - En positionnant le paramètre global à true

```
@Controller
public class ErrorHandler {

    @Error(global = true, status = HttpStatus.NOT_FOUND)
    public HttpResponse<JsonError> notFound(HttpRequest request) {
        JsonError err = new JsonError("Object Not Found")
                .link(Link.SELF,  Link.of(request.getUri()));
        return HttpResponse.<JsonError>notFound().body(err);
    }
}
```

```
@Get("/error")
@Produces(MediaType.TEXT_PLAIN)
public HttpResponse<String> error() {
    throw new HttpStatusException(HttpStatus.NOT_FOUND, "Resource not found");
}

@Error(status = HttpStatus.NOT_FOUND)
public HttpResponse<JsonError> notFound(HttpRequest request) {
    JsonError err = new JsonError("Object Not Found").link(Link.SELF, Link.of(request.getUri()));
    return HttpResponse.<JsonError>notFound().body(err);
}
```

Configuration du serveur

- Micronaut étant basé sur Spring Boot, on retrouve à peu de chose près les mêmes variables de configuration
  - Centralisées dans le fichier application.yml
  - Simplifié et hautement customisable
- Pour ce qui est du routage, la syntaxe des annotations est assez proche de ce que l'on peut retrouver sur Spring Boot
  - Mais annotations spécifiques à Micronaut tout de même
  - Exple: @Controller("/movies") @Get("/{id}")
  - Il est possible de versionner les API avec l'annotation @Version(string)

Template	Description
/books/{id}	Simple match
/books/{id:2}	A variable of two characters max
/books{/id}	An optional URI variable
/book{/id:[a-zA-Z]+}	An optional URI variable with regex
/books{?max,offset}	Optional query parameters
/books{/path:.*}{.ext}	Regex path match with extension

Annotation	Description
@Body	Binds from the body of the request
@CookieValue	Binds a parameter from a cookie
@Header	Binds a parameter from an HTTP header
@QueryValue	Binds from a request query parameter
@Part	Binds from a part of a multipart request
@RequestAttribute	Binds from an attribute of the request. Attributes are typically created in filters
@PathVariable	Binds from the path of the request
@RequestBean	Binds any Bindable value to single Bean object

JPA

- Micronaut utilise les mêmes drivers JDBC que n'importe quelle autre framework pour la connection aux bases de données
- Il implante sa propre couche JPA, conformément aux API standards

```
<dependency>
<groupId>io.micronaut.sql</groupId>
<artifactId>micronaut-hibernate-jpa</artifactId>
<scope>compile</scope>
</dependency>
```

Beans: Validation

- Micronaut fournit sa propre implémentation de l'API javax.validation
  - Importé lors de la création du projet
- Il est également possible de définir ses propres contraintes
  - En définissant des nouvelles annotations
  - En spécifiant une factory pour instancier des validateurs
- https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraints/package-summary

Programmation réactive

- Micronaut s'intègre bien avec RxJava et Reactor
  - Par une encapsulation des deux implémentations
  - Pas besoin de passer par des adaptateurs
  - Il suffit juste de déclarer les dépendances adéquates

```
<dependency>
    <groupId>io.micronaut.reactor</groupId>
    <artifactId>micronaut-reactor</artifactId>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>io.micronaut.reactor</groupId>
    <artifactId>micronaut-reactor-http-client</artifactId>
    <scope>compile</scope>
</dependency>
```

R2DBC

- Reactive Relational Database Connectivity
- Apporte une couche de programmation réactive pour l'accès aux bases de données
- Approche bien plus scalable et cloud-ready

```
ConnectionFactory connectionFactory = ConnectionFactories
        .get("r2dbc:h2:mem:///testdb");
Mono.from(connectionFactory.create())
            .flatMapMany(connection -> connection
            .createStatement("SELECT firstname FROM PERSON WHERE age > $1")
            .bind("$1", 42)
            .execute())
        .flatMap(result -> result
            .map((row, rowMetadata) -> row.get("firstname", String.class)))
        .doOnNext(System.out::println)
        .subscribe();
```

R2DBC

- L'utilisation de R2DBC nécessite l'utilisation de connecteurs dédiés
- Les librairies R2DBC fournissent des interfaces facilitant le requêtage des bases de données en mode réactif

```
<dependency>
<groupId>io.micronaut.data</groupId>
<artifactId>micronaut-data-r2dbc</artifactId>
<version>3.3.0</version>
</dependency>
```

```
<dependency>
<groupId>io.r2dbc</groupId>
<artifactId>r2dbc-postgresql</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
```

```
r2dbc:
    datasources:
        default:
            url: r2dbc:postgresql://localhost:6543/multimedia
            username: ${JDBC_USER:cenotelie}
            password: ${JDBC_PASSWORD:cenotelie}
```

Tests

- Micronaut supporte plusieurs frameworks de tests
  - Spock
  - JUnit 5
  - Kotest 5
  - Rest Assured

Tests

- Micronaut repose sur JUnit5 pour les tests
- Il fournit en plus quelques classes utiles
  - Un client Http (Comportement injecté par annotation)
  - Un serveur embarqué

```
@MicronautTest
public class HelloControllerTest {
    @Inject
    EmbeddedServer server;
    @Inject @Client("/")
    HttpClient client;
    ...
}
```

Tests avec Spock

- Ajouter la dépendance suivante:

```
<dependency>
<groupId>io.micronaut.test</groupId>
<artifactId>micronaut-test-spock</artifactId>
<scope>test</scope>
</dependency>
```

- Exemple:

```
package io.micronaut.test.spock;

public interface MathService {

   Integer compute(Integer num);

}
```

```
package io.micronaut.test.spock
import jakarta.inject.Singleton

@Singleton
class MathServiceImpl implements MathService
{
   @Override
   Integer compute(Integer num)
   { return num * 4 // should never be called }
}
```

```
@MicronautTest
class MathServiceSpec extends Specification
{
   @Inject
   MathService mathService

   @Unroll
   void "should compute #num times 4"()
   {
      when: def result = mathService.compute(num)
      then: result == expected
      where: num | expected
                 2 | 8
                 3 | 12
   }
}
```

Aspects Traverses

Micro-services: considérations

- L'avantage des micro-services, comparés aux applications monolithiques, est indéniable
- Cependant, l'intégration des micro-services peuvent poser des problèmes à cause des comportements émergents
  - Même avec l'adoption d'interfaces standards (ex: HTTP + REST)
- Sur un plan théorique, il est quasiment impossible de démontrer la correction fonctionnelle d'une application
  - Ni même ses performances
- Le moyen le plus efficace dont nous disposons pour tenter de maîtriser les comportements reste les tests et le monitoring
  - Tests unitaires
  - Tests d'intégration
  - Tests E2E
  - Tests de performance

  Suppose de disposer d'une chaîne d'intégration / déploiement continue
- De ce fait, il est impératif que le déploiement d'une application basée sur une architecture en micro-services dispose d'un système de logs et d'exploitation des logs (monitoring) efficace

Micro-services: considérations

- Une architecture à base de micro-services a un cycle de vie différent celui d'une application monolithique
- Les composants fonctionnent selon le principe de séparation des préoccupations
  - Ils n'ont la responsabilité que d'un sous-ensemble de préoccupations
    - Sécurité
    - Authentification
    - Stockage
    - Messages
    - …
  - Ils sont indépendants et doivent pouvoir être remplacés à chaud pour assurer une continuité de service
  - Leur exécution doit être orchestré par un composant dédié

Micro-services: cycle de vie

- L'intégration et le déploiement continus d'une application à base de micro-services est de fait différent de celui d'une application monolithique
- Que ce soit pour l'intégration ou le déploiement
  - Ils peuvent être exécutés à chaud et par parties
  - Beaucoup des composants de l'application sont des COTS
- Questions:
  - Quelle implication pour le versionnement de l'application ?
  - De quelle(s) fonctionnalité(s) avons-nous besoin pour gérer une composition à chaud ?

Service de découverte

- Avec les problématiques de déploiement à chaud mentionnés précédemment, il est nécessaire de disposer d'un service de découverte de services
- Il existe dans la littérature, plusieurs composants offrant cette fonctionnalité, classés en deux catégories
  - Les services de nommage statique (DNS)
  - Les services de nommage dynamique (registre de services), pouvant également être classés en deux sous-catégories:
    - La responsabilité de la découverte est confiée au service (ex: Eureka)
    - La responsabilité de la découverte est confiée au serveur (ex: Kubernetes)
- Concernant l'enregistrement d'un service, il existe là encore deux approches
  - Le service est responsable de son enregistrement (self-registration)
  - L'enregistrement est assurée par un tiers qui scrute le réseau afin de mettre à jour l'ensemble des services disponibles
  - **Question:** avantages et inconvénients de chaque approche ?

Liveness

- Dans une architecture à base de micro-services à chaud, il est nécessaire de disposer de métriques afin de déterminer si un service est toujours vivant. Il existe pour ce faire différentes stratégies
  - Au niveau service:
    - **Timeout**: Si un service ne reçoit pas une réponse à sa requête au bout d'une certaine durée, à déterminer selon les application, alors le service cible est considéré comme mort
    - **Retry**: Ne pas recevoir une réponse au bout d'une certaine ne signifie pas toujours que le service cible est mort, le message peut avoir été perdu. On peut dans ce cas essayer un certain nombre de fois, à déterminer, avant de considérer le service cible comme mort
    - Handshake: Le service source s'assure que le service cible est vivant avant de lui envoyer quoi que ce soit
  - Au niveau application
    - **Bulkhead**: Il faut prêter attention au fait que la perte d'un service n'entraîne pas la perte de l'application par un effet domino par une meilleure isolation des préoccupations
    - **Circuit breaker**: Consiste à couper des connections entre services pour éviter une propagation d'erreur
- **Questions:**
  - Quels patterns mettriez-vous en place pour assurer une grande disponibilité de service ?
  - Quelles règles de bonnes pratiques mettriez-vous en place ?

Micro-services et Patterns

- Selon la nature de votre application et le domaine ciblé, différents patterns sont applicables
  - Comportement: synchrone, asynchrone
  - Communication: Message Queue, Base de donnée partagée, volume partagé
  - Sécurité: Secured Message, Secured Infrastructure, Firewall
- Coordination: Orchestration, Chorégraphie
  - Orchestration: une intelligence centrale joue le rôle de chef d'orchestre
  - Chorégraphie: l'intelligence est distribuée et c'est un objectif partagé qui guide les comportements
  - **Question:** Quelles sont les avantages et inconvénients des deux approches ?

Logging et Monitoring

- Dans une application à base de micro-services, chaque service produit des logs
- Il est nécessaire, afin d'avoir une vue d'ensemble, de pouvoir agréger tous les logs à des fins de visualisation / analyse en temps réel
- Il existe des outils permettant de centraliser les logs sur la base de requêtes
  - Elastic Search / Logstash
  - Prometheus
- À combiner avec d'autres outils pour visualiser ces logs
  - Le plus souvent sous la forme de séries temporelles
  - Kibana (avec ES)
  - Grafana (avec Prometheus ou ES)
- Dans l'analyse de problèmes pouvant arriver à l'exécution, l'utilisation seule des logs peut limiter nos capacités d'analyse pour tenter de comprendre la chaîne d'événements ayant aboutie à une erreur (probablement due à des comportements émergents)
  - Il peut être intéressant dans ce cas d'utiliser des outils de traces distribuées
  - Ex: Zipkin ou Jaeger
- Il peut être intéressant de combiner ces outils de monitoring à des outils de ML afin d'anticiper les problèmes (maintenance préventive) ou d'automatiser sa résolution (ne serait-ce que par sa planification avec des outils dédiés comme PagerDuty)

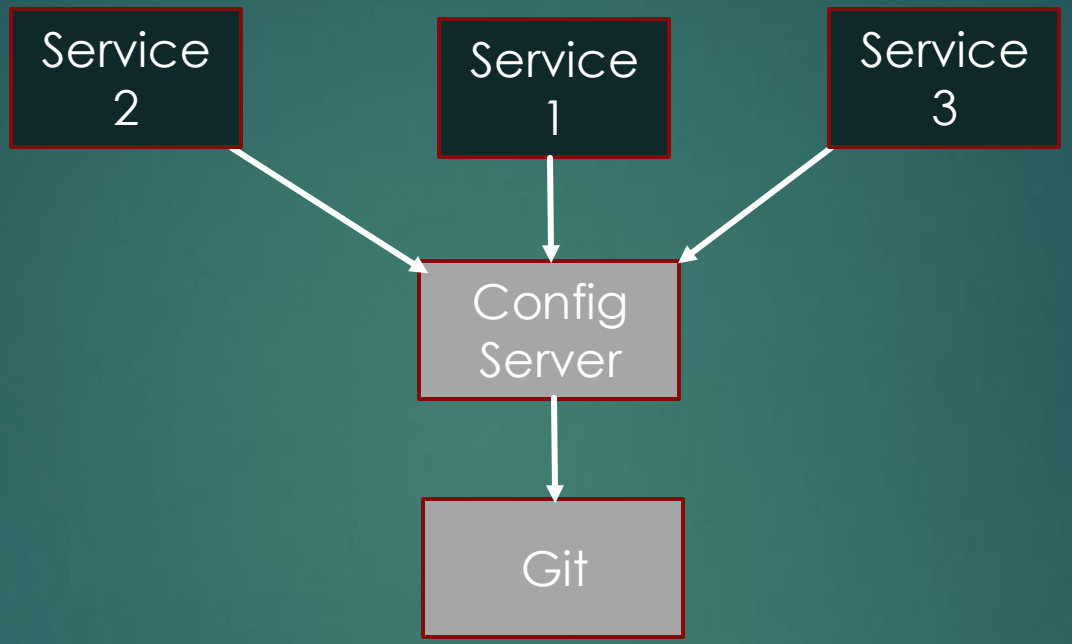
Micro-services: Challenges

- Lorsque plusieurs micro-services doivent être déployés, la configuration est embarquée dans le binaire, rendant difficile son changement à chaud
  - La modification requiert alors un redéploiement
  - => Spring Cloud Config Server
- Passage à l'échelle
  - Scale up/down du nombre d'instances en fonction de la charge
  - Avec un load-balancing dynamique
  - => Eureka (naming), SCB(load balancing), Feign (Easier Rest Clients), SCG
- Résilience
  - Gestion efficace des pannes
  - Basé sur un monitoring en temps réel
  - => Zipkin (Tracing distribué), Resilience4j(Fault Tolerance)

Spring Cloud

- Spring Cloud fournit des outils aux développeurs pour construire rapidement certains des modèles communs dans les systèmes distribués
  - Gestion de la configuration
  - Découverte de services
  - Circuit Breaker
  - Routage intelligent
  - Micro-proxy,
  - Bus de contrôle
  - …
- Grâce à Spring Cloud, les développeurs peuvent rapidement mettre en place des services et des applications qui mettent en œuvre ces schémas.
  - Ils fonctionneront bien dans n'importe quel environnement distribué, y compris l'ordinateur portable du développeur, les centres de données " bare metal " et les plates-formes gérées telles que Cloud Foundry.
- Spring Cloud représente un ensemble de projets visant à fournir un support pour chacun de ces aspects (https://spring.io/projects/spring-cloud)

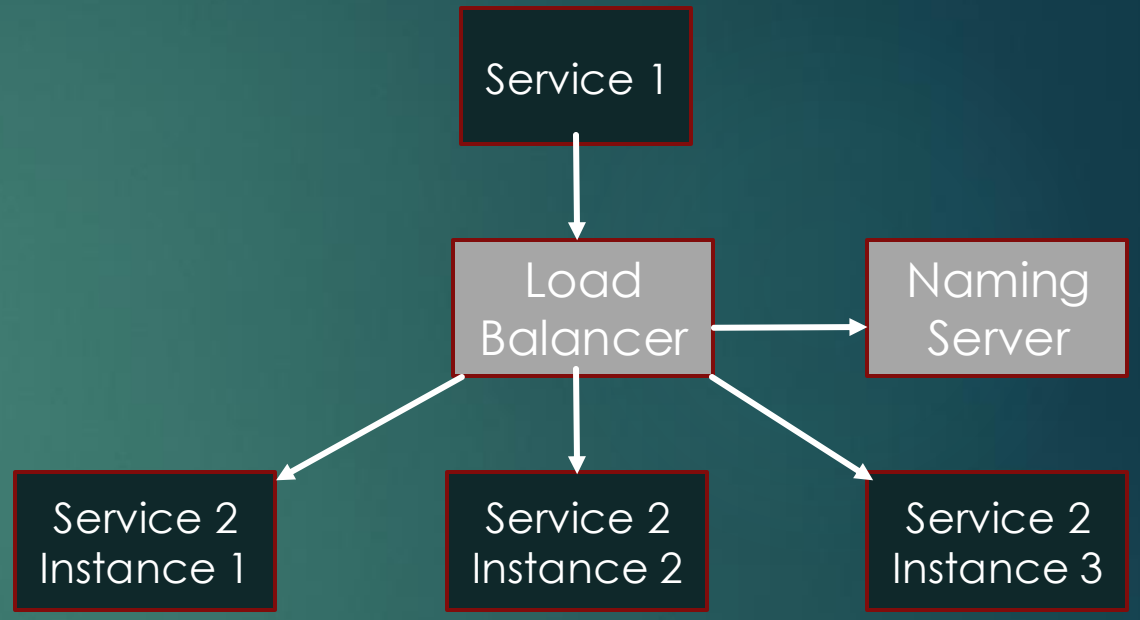
Configuration Centralisée



- La configuration de chaque service peut être centralisée
  - On utilise pour cela un repo Git qui va stocker les propriétés à centraliser
  - On instancie un serveur cloud config qui va pointer le repo git
  - On paramètre chaque service pour pointer le serveur de configuration

Load Balancing

- Question: quelles conditions préalables sonts nécessaires pour le load balancing ?



Circuit breaker

- Dans le cas où un service tombe ou n'atteint plus les performances attendues, c'est toute la chaîne qui est impactée
  - Quand un service est appelé, il faut que ce dernier puisse notifier les services appelants qu'il a bien reçu la demande.
  - Après un certain délai sans réponse, l'appelant peut renvoyer sa demande ou alors trouver une alternative

  - C'est précisément la fonctionnalité qu'apporte la brique Resilience4J
  - Resilience4j est une bibliothèque légère de tolérance aux fautes conçue pour la programmation fonctionnelle.
    - Elle fournit des fonctions d'ordre supérieur (décorateurs) pour améliorer n'importe quelle interface fonctionnelle, expression lambda ou référence de méthode avec un coupe-circuit, un limiteur de taux, une répétition ou une cloison.
    - https://resilience4j.readme.io/

Circuit Breaker

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
<groupId>io.github.resilience4j</groupId>
<artifactId>resilience4j-spring-boot3</artifactId>
<version>2.0.2</version>
</dependency>
```

- Ajouter les dépendances suivantes au service 1
- Ajouter une opération au niveau du contrôleur
  - Faire en sorte que le contrôleur appelle une opération inexistante sur le service 2
  - Annoter cette opération avec l'annotation @Retry(name = "mycall")
  - Imprimer un message sur les logs à chaque appel
  - Exécuter et appeler cette opération. Qu'observez-vous ?
- Basé sur le nom de l'opération, il y a différentes manières de configurer le circuit breaker
  - Nombre d'essais
  - Temps d'attente entre essais
  - Fallback
  - ...

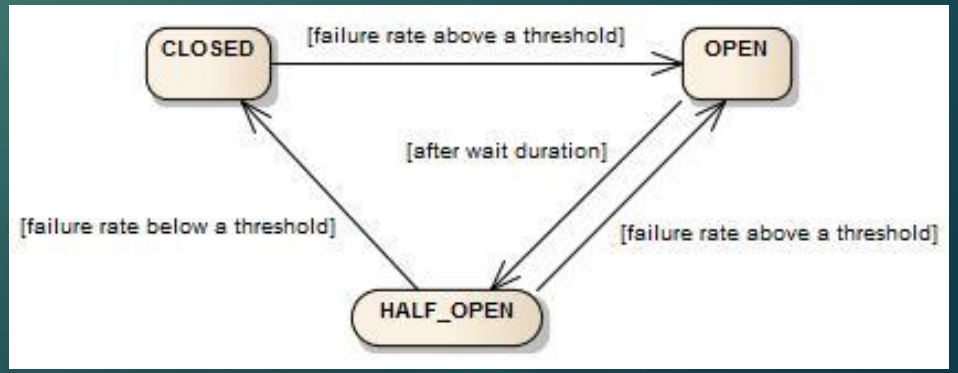
```
@GetMapping("/v2")
@Retry(name = "mycall", fallbackMethod = "getMessageV2Fallback")
public Message getMessageV2() {...}
```

```
resilience4j.retry.instances.mycall.maxAttempts=5
resilience4j.retry.instances.mycall.waitDuration=1s
```

```
public Message getMessageV2Fallback(Exception e) {
  String title = config.getDefaultTitle();
  String content = config.getDefaultContent();
  return new Message(title, content);
}
```

Circuit Breaker

- Au bout d'un certain et d'un certain nombre d'essai, si le service appelé ne répond pas, on peut conclure qu'il est tombé
  - Dans ce cas, il est inutile d'insister et il est inutile de continuer de l'appeler
  - L'annotation @CircuitBreaker apporte cette intelligence et permet d'éviter de perdre du temps en coupant le circuit et en renvoyant une réponse par défaut
    - Il permet également de renvoyer une réponse par défaut en cas d'un grand nombre de requêtes
  - La question qui peut se poser est la suivante:
    - Que se passerait-il si le service qui était défaillant est réparé ?
- Le CircuitBreaker possède différents états
  - Par défaut, il est fermé
  - Au dessus d'un certain nombre d'échecs, il est ouvert
  - Après un certain temps, il réessaye et passe:
    - Dans l'état fermé si le service répond de nouveau
    - Dans l'état ouvert si il est toujours défaillant
  - Le seuil et la durée sont à configurer



Circuit Breaker

- Rate Limiter
  - Il est possible de limiter le nombre d'appels sur une opération
  - Il faut pour cela annoter l'opération avec l'annotation @RateLimiter(name="mycall")
  - Ensuite, il faut ajouter les propriétés suivantes à l'application

```
resilience4j.ratelimiter.instances.mycall.limitForPeriod=2
resilience4j.ratelimiter.instances.mycall.limitRefreshPeriod=10s
```

- Bulkhead
  - Il est possible de limiter le nombre d'appels concurrents
  - Il faut pour cela annoter l'opération avec l'annotation @Bulkhead(name="mycall")
  - Ensuite, il faut ajouter les propriétés suivantes à l'application

```
resilience4j.bulkhead.instances.mycall.maxConcurrentCalls=10
```

Cloud Function

- L'architecture serverless est un modèle dans lequel le fournisseur de services cloud (AWS, Azure ou Google Cloud) est responsable de l'exécution d'un morceau de code en allouant de manière dynamique les ressources.
- Il ne facture que la quantité de ressources utilisées pour exécuter le code. Le code est généralement exécuté dans des conteneurs sans état pouvant être déclenchés par divers événements,
- Le code envoyé au fournisseur de cloud pour l'exécution est généralement sous la forme d'une fonction. Par conséquent, serverless est parfois appelé "*Functions as a Service*" ou "*FaaS*". Voici les offres FaaS des principaux fournisseurs de cloud :
  - AWS: AWS Lambda
  - Microsoft Azure: Azure Functions
  - Google Cloud: Cloud Functions
- Spring Cloud Function est un projet dont les objectifs de haut niveau sont les suivants :
  - Promouvoir l'implémentation de la logique métier via des fonctions.
  - Découpler le cycle de développement de la logique métier de toute cible d'exécution spécifique afin que le même code puisse s'exécuter en tant que point de terminaison Web, processeur de flux ou tâche.
  - Prendre en charge un modèle de programmation uniforme pour tous les fournisseurs de services sans serveur, ainsi que la possibilité de fonctionner de manière autonome (localement ou dans un PaaS).
  - Activer les fonctionnalités de Spring Boot (auto-configuration, injection de dépendances, métriques) sur les fournisseurs sans serveur.
  - Faire abstraction de tous les détails du transport et de l'infrastructure, ce qui permet au développeur de conserver tous les outils et processus familiers, et de se concentrer fermement sur la logique métier.

Cloud Function

- Ajouter la dépendance suivante au projet

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-function-web</artifactId>
</dependency>
```

- Implanter les fonctions à déployer

```
@Configuration
public class AppConfig {

  @Autowired
  private MessageService messageService;

  @Bean
  public Supplier<List<Message>> allMessages() {
    return () -> messageService.getMessages();
  }

  @Bean
  public Consumer<Message> createMessage() {
    return (message) -> messageService.addMessage(message);
  }
}
```

Cloud Function

- Pour un déploiement chez un fournisseur particulier, il est nécessaire d'ajouter les dépendances appropriées
  - Et configurer la chaîne maven / gradle en conséquence

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-function-adapter-aws</artifactId>
</dependency>
```

- Exercice: Implanter une fonction de transformation et la tester

Cloud Batch

- Spring Batch est un framework batch léger et complet conçu pour permettre le développement d'applications batch robustes, vitales pour les opérations quotidiennes des systèmes d'entreprise
- Spring Batch fournit des fonctions réutilisables qui sont essentielles dans le traitement de grands volumes d'enregistrements, tels que :
  - La journalisation/traçage,
  - La gestion des transactions,
  - Les statistiques de traitement,
  - Le redémarrage des tâches,
  - La gestion des ressources.
- Spring Batch offre également une interface d'administration basée sur le web (Spring Cloud Data Flow), des techniques d'optimisation des performances et une intégration avec d'autres projets Spring.

Cloud Batch - Objectifs

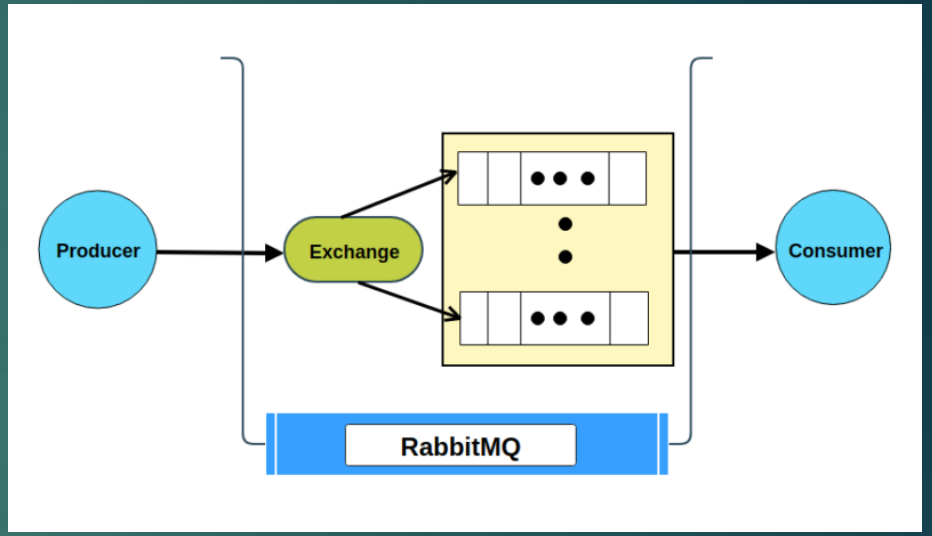
- Laissez les développeurs de lots utiliser le modèle de programmation Spring afin de le permettre de se concentrer sur la logique métier et ne pas avoir à se préoccuper de l'infrastructure.
- Prévoir une séparation claire des préoccupations entre l'infrastructure, l'environnement d'exécution des lots et l'application des lots.
- Fournir des services d'exécution communs et essentiels sous forme d'interfaces que tous les projets peuvent mettre en œuvre.
- Fournir des implémentations simples et par défaut des interfaces d'exécution de base qui peuvent être utilisées "out of the box".
- Faciliter la configuration, la personnalisation et l'extension des services en utilisant le cadre Spring dans toutes les couches.
- Tous les services de base existants doivent pouvoir être facilement remplacés ou étendus, sans aucun impact sur la couche d'infrastructure.
- Fournir un modèle de déploiement simple, avec les JAR d'architecture complètement séparés de l'application, construits à l'aide de Maven.

Cloud Batch - Scénarios

- Exécuter périodiquement le processus de traitement par lots.
- Traitement simultané par lots : traitement parallèle d'un travail.
- Traitement par étapes, axé sur les messages d'entreprise.
- Traitement par lots massivement parallèle.
- Redémarrage manuel ou programmé en cas d'échec.
- Traitement séquentiel des étapes dépendantes (avec extension aux lots pilotés par le flux de travail).
- Transaction par lot entier, pour les cas où la taille du lot est faible ou pour les procédures stockées ou les scripts existants.

Cloud Stream

- Spring Cloud Stream est un framework permettant de créer des applications de microservices basées sur les messages.
- Spring Cloud Stream s'appuie sur Spring Boot pour créer des applications Spring autonomes de niveau production et utilise Spring Integration pour assurer la connectivité avec les courtiers de messages.
- Il fournit un middleware supportant plusieurs fournisseurs, introduisant les concepts de sémantique persistante de publication et d'abonnement, de groupes de consommateurs et de partitions.



Cloud Stream

- Integration avec RabbitMQ
  - Ajouter la dépendance suivante
- Les dernières versions de Spring Cloud Stream s'appuient sur la programmation fonctionnelle pour définir des canaux de communications entre services

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

Java Functional Interface	Application Type
Function<T,R>	Processor Hence , **it will create both input and output channel.**
Consumer<T>	Consumer. Hence, **it will create only input channel.**
Supplier<T>	Publisher/Producer Hence, **it will create only output channel**.

Cloud Stream : Producteur

```
@Slf4j
@Service
public class MessagesWriter {

  @Bean
  public Supplier<Message> produceMessages() {
    log.info("Producing message");
    return () -> new Message("my title", "my content");
  }
}
```

```
spring.cloud.function.definition=produceMessages
```

Cloud Stream: Consommateur

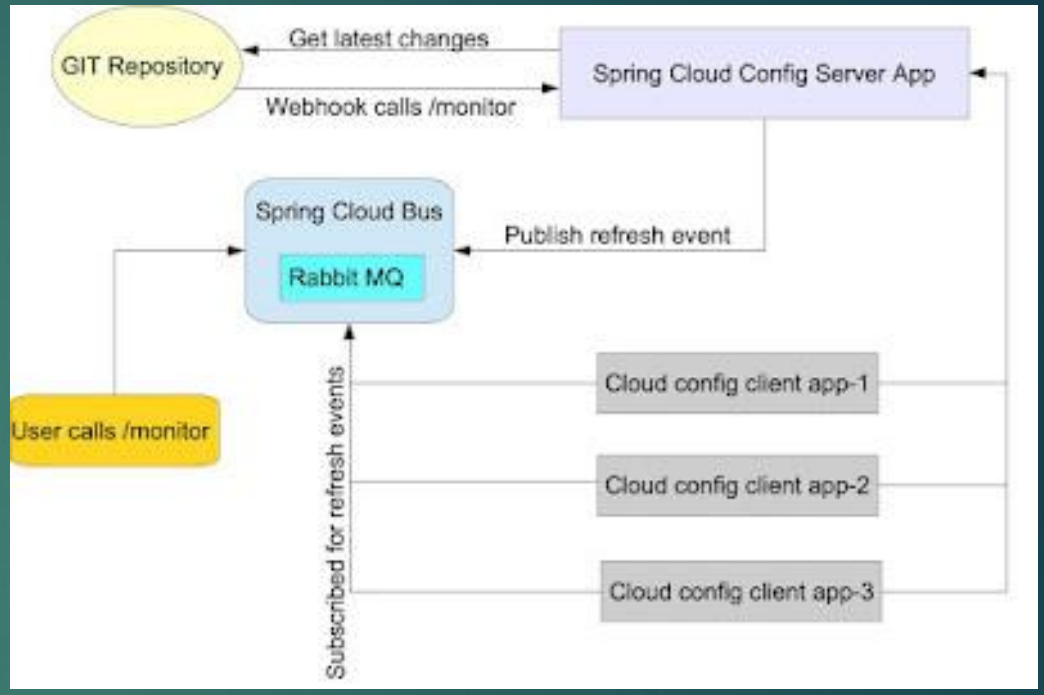
```java
@Slf4j
@Service
public class MessagesListener {

  @Bean
  public Consumer<Message> consumeMessages() {
    return m -> log.info("Received message: {}", m);
  }
}
```

```
spring.cloud.function.definition=consumeMessages
spring.cloud.stream.bindings.consumeMessages-in-0.destination=produceMessages-out-0
spring.cloud.stream.bindings.consumeMessages-in-0.group=messages-consumers
spring.cloud.stream.bindings.consumeMessages-in-0.consumer.concurrency= 10
spring.cloud.stream.bindings.consumeMessages-in-0.consumer.max-attempts= 3
```

Question: quel est l'intérêt de cette approche ?
Exercice: Implanter une transformation entre les 2 canaux

Cloud Bus

- Message Broker permettant le broadcast d'événement basé sur le pattern publish / subscribe
- Il s'appuie sur des technologies de Message Queueing existants (RabbitMQ, Kafka, etc.)
- Il permet notamment de gérer les changements



Cloud Bus: Changements de config

- Config Server

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-config-monitor</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.password=training
spring.rabbitmq.username=training
```

- Service

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

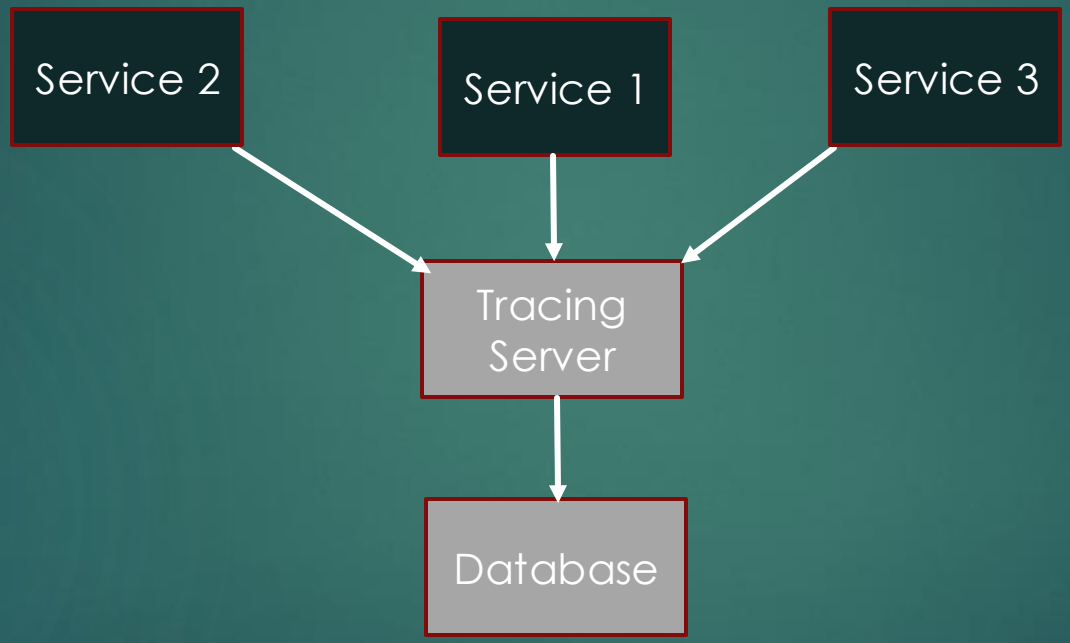
```
spring.cloud.bus.enabled=true
spring.cloud.bus.refresh.enabled=true
```

  - Ajouter l'annotation @Refreshscope à l'application
  - Ajouter un hook sur github

Observation

- Contrairement au monitoring, l'observation est pro-active
- Elle nécessite de mettre en place un certain nombre d'actions
  - Collecte des données: métriques, logs, traces
  - Mettre en œuvre une intelligence afin de détecter des anomalies
- OpenTelemetry représente un ensemble d'outils, standards, SDKs afin de
  - Générer, collecter et exporter les données de télémétrie

Traces distribuées



Zipkin: Mise en place

- Lancer une instance avec docker-compose
- Ajouter les dépendances suivantes au service
- Configurer le service afin d'envoyer les données de télémétrie
- Activer l'actuator sur chaque service
- Relancer le service
  - Faire une requête
  - Observer les changements sur Zipkin
  - Faites une requête sur l'Api Gateway
    - Que remarquez-vous ?

```
zipkin:
  image: openzipkin/zipkin
  container_name: zipkin
  restart: unless-stopped
  ports:
    "9411:9411"
```

```
<dependency>
<groupId>io.micrometer</groupId>
<artifactId>micrometer-observation</artifactId>
</dependency>
<dependency>
<groupId>io.micrometer</groupId>
<artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>
<dependency>
<groupId>io.opentelemetry</groupId>
<artifactId>opentelemetry-exporter-zipkin</artifactId>
</dependency>
```

```
management.tracing.sampling.probability=1.0
logging.pattern.level=%5p [${spring.application.name},%X{traceId:-},%X{spanId:-}]
```

Zipkin: Configuration d'OpenFeign

- Ajouter la dépendance suivante pour les services utilisant OpenFeign

```
<dependency>
<groupId>io.github.openfeign</groupId>
<artifactId>feign-micrometer</artifactId>
</dependency>
```

- Simplement relancer le service après avoir ajouté la dépendance

Génération d'une image docker

- Lorsque l'application a été testée, validée, il faut la déployer
- Micronaut fournit les outils permettant de générer une image docker ayant une empreinte mémoire minimale
  - Mvn mn:build-docker

Pour aller plus loin

Micronaut fournit un ensemble d'extensions, supportées par des annotations, qui facilitent l'intégration de différentes technologies

- Langages: Java, Kotlin et Groovy
- Programmation réactive: Reactor, RxJava 2/3
- Bases de données:
  - SQL et NoSQL
  - Migrations: Liquibase et Flyway
- Messaging: JMS, Kafka, MQTT, RabbitMQ, Pulsar
- Cache: Redis, Hazelcast
- Autres: Emails, Chatbots, Discovery,…
- Cf. https://docs.micronaut.io/index.html