

# Micro-Services Réactifs avec Quarkus

ALI KOUDRI – ALI.KOUDRI@GMAIL.COM

# Agenda

- ▶ Introduction: Contexte et motivations des microservices
- ▶ Principes
- ▶ Bases
  - ▶ Annotations
  - ▶ IoC et AOP
  - ▶ Programmation fonctionnelle et réactive
- ▶ Quarkus
  - ▶ Introduction
  - ▶ Persistence
  - ▶ Cache
  - ▶ Service
  - ▶ Presentation
  - ▶ Documentation
  - ▶ Tests
  - ▶ Réactivité
  - ▶ Aspects transverses

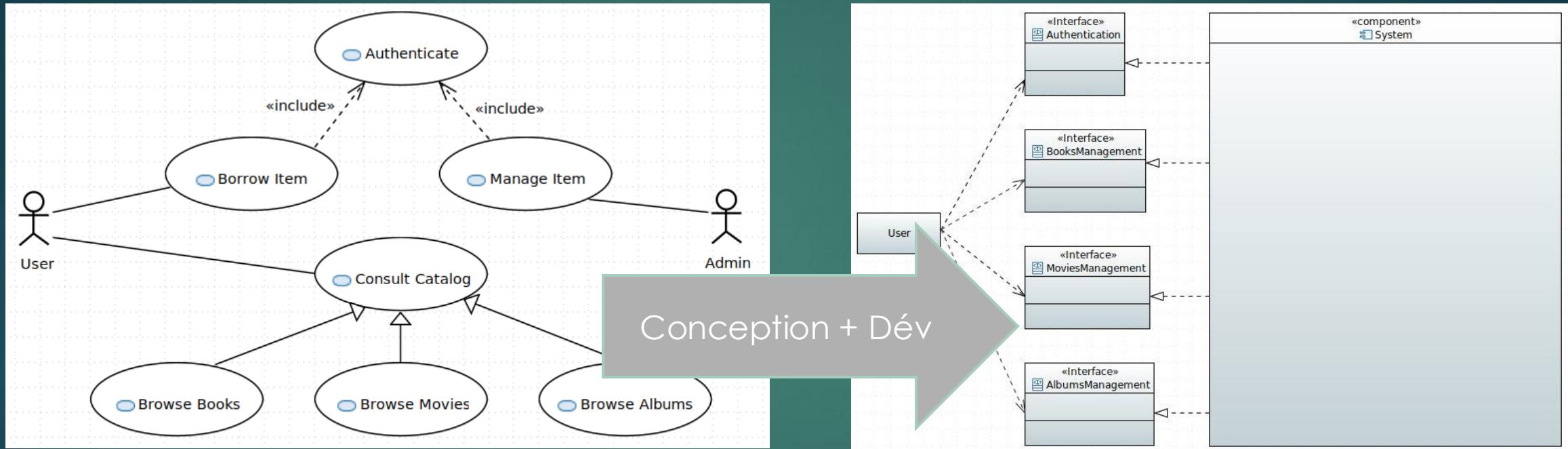
# Prérequis

- ▶ Java 17
- ▶ Docker
- ▶ Cloner les dépôts suivants:
  - <https://github.com/akoudri/spring-core-training.git>
  - <https://github.com/akoudri/multimedia-quarkus.git>

# Introduction

# Exemple d'application

- ▶ Application de gestion de contenus multimédia

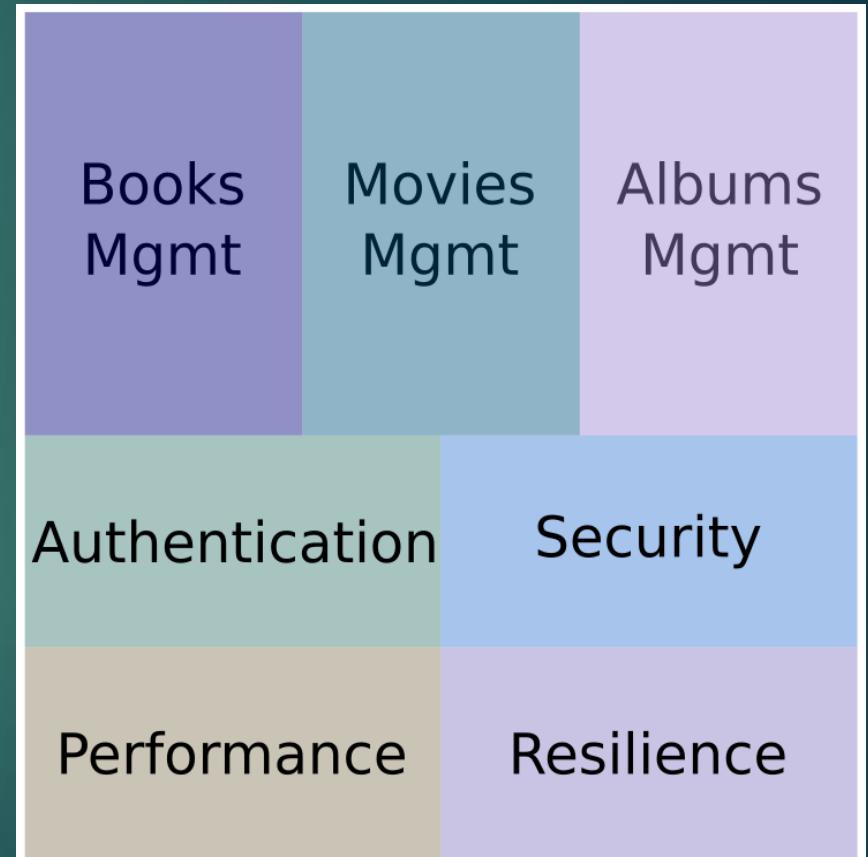


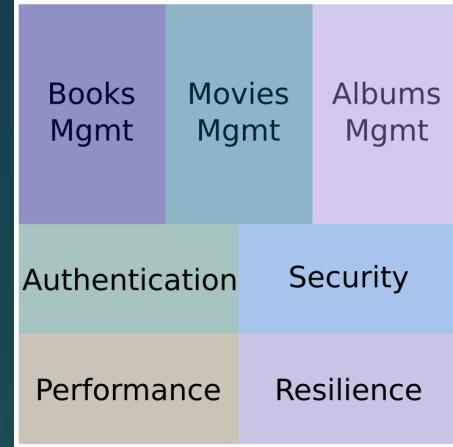
Depuis la formalisation du besoin, on aboutit après un travail d'analyse et de conception au développement d'un système qui implante un certain nombre d'interfaces

**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

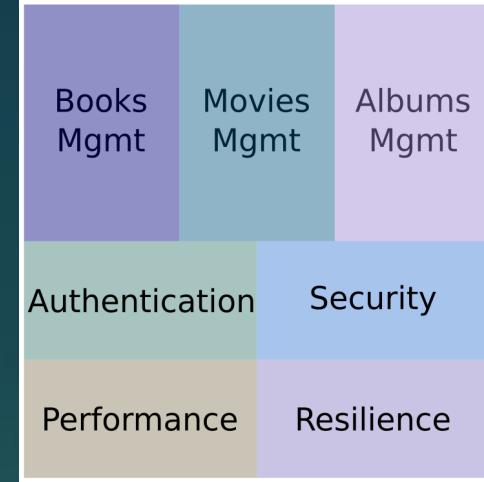




# Architectures monolithiques

## 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.



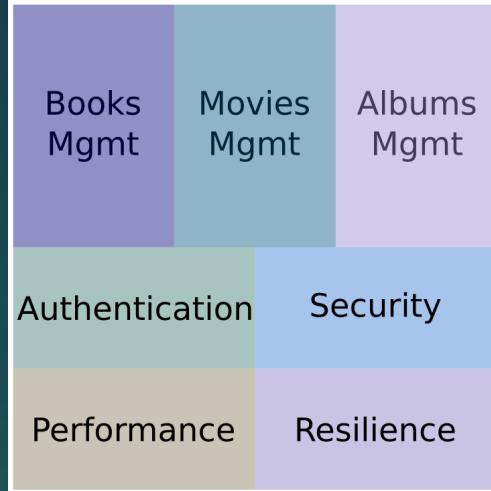
# Architectures monolithiques

## 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 approche 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 noeuds 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 **duplicer** 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.

# 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 goulets 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.

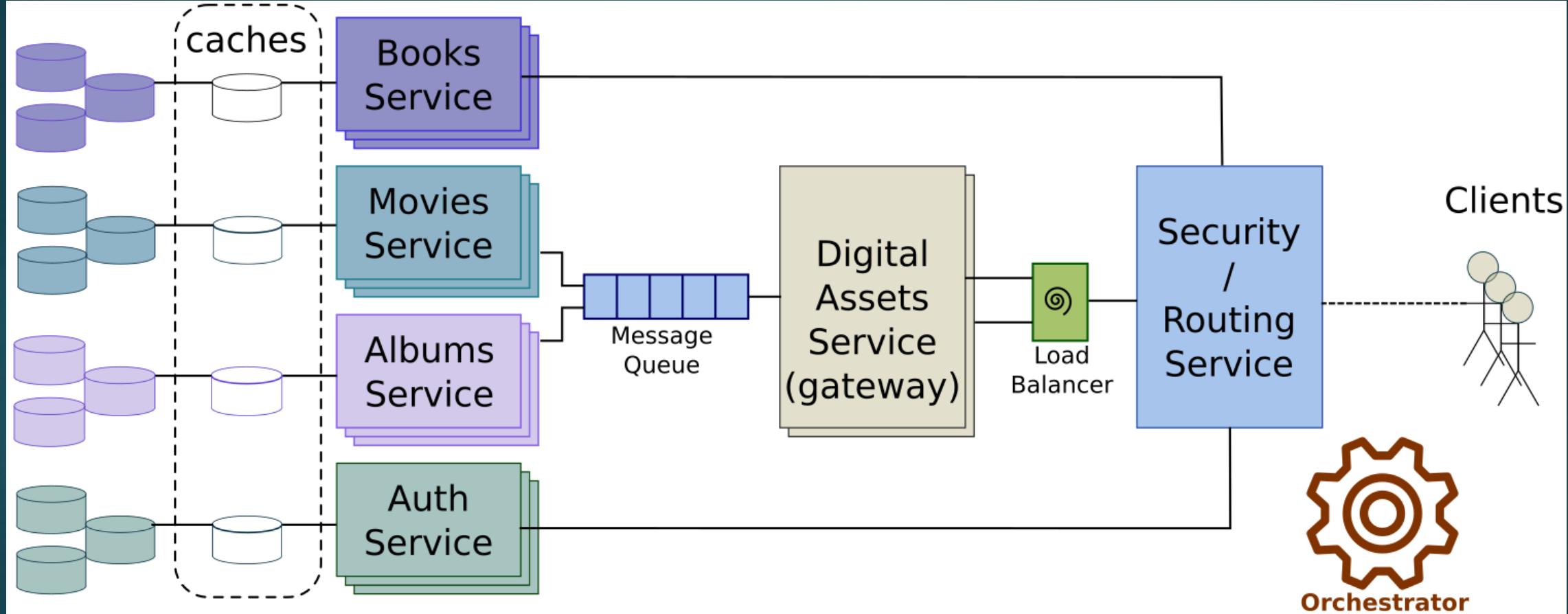
# Avantages des micro-services

- ▶ **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 ?**

# 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 besoin de centraliser les traces d'exécution, les visualiser et éventuellement prédire de potentiels problèmes. Exemples: Prometheus / Grafana

# 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 découplant 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 complètement 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 ou réactive
    - 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 constitutants 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 "ApplicationContext"
- ▶ 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, ajoutent 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

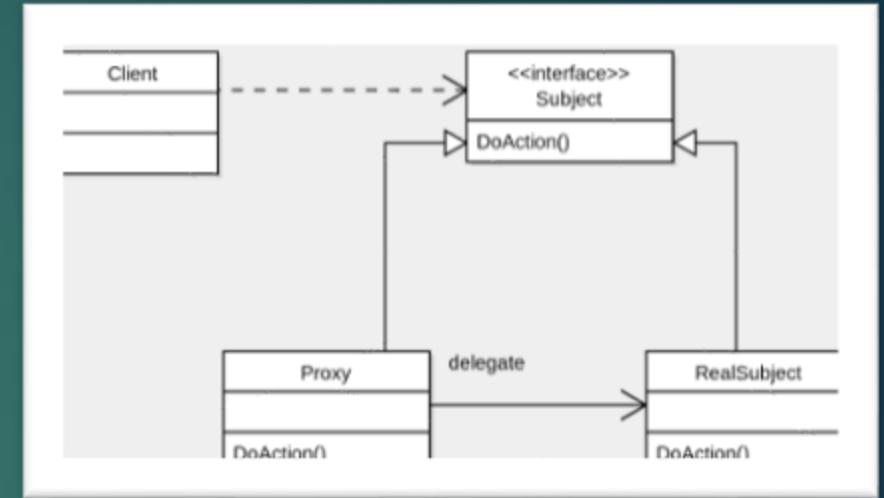
- ▶ D'autres variables, propres à l'application, peuvent être déclarées via des fichiers de propriétés
  - ▶ Les variables d'environnement ont la priorité sur les variables issues des fichiers de propriétés
- ▶ Il est possible de configurer différents profiles
  - ▶ Utilisation de l'annotation "Profile("name")" 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
  - ▶ Le 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 packages 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 crée 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

# 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 la 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

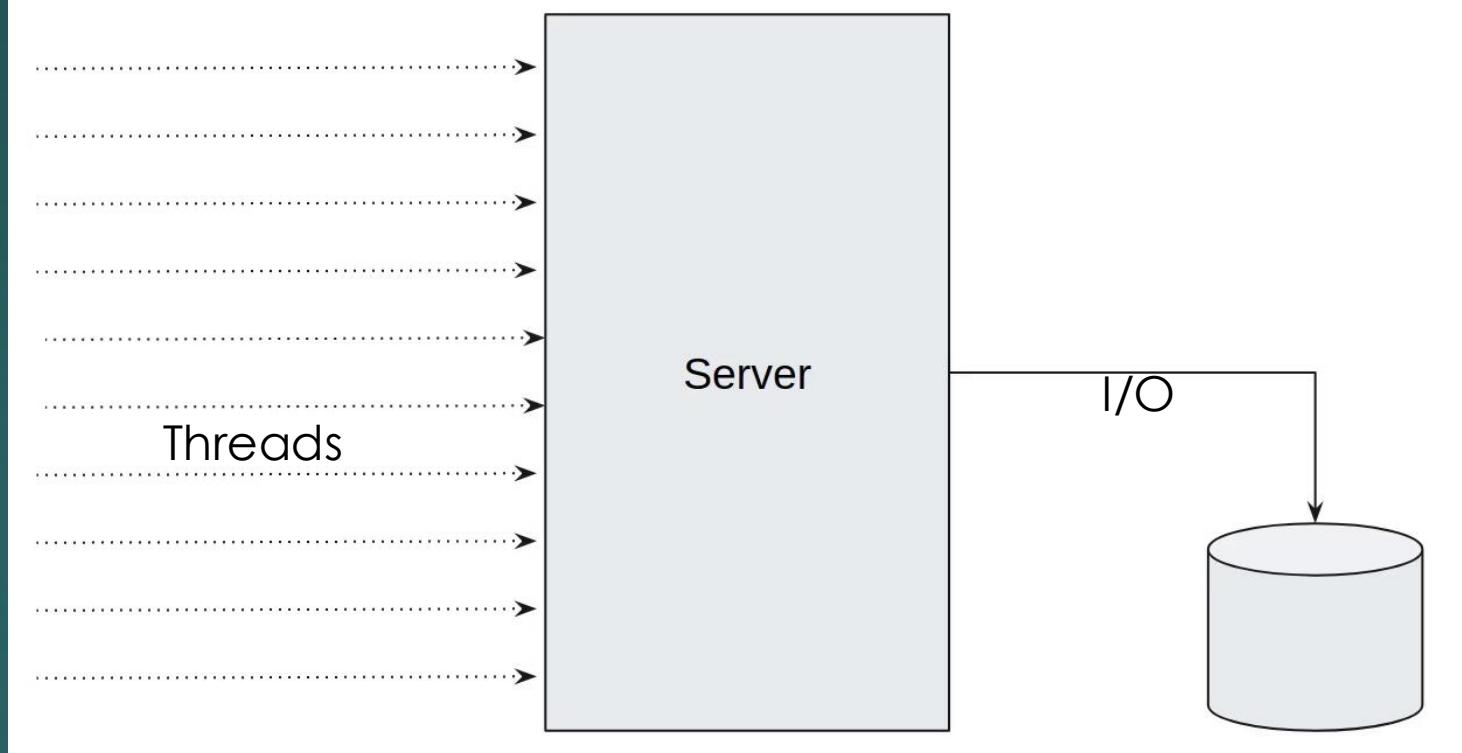
# TP: AOP

- ▶ 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

# Programmation Réactive



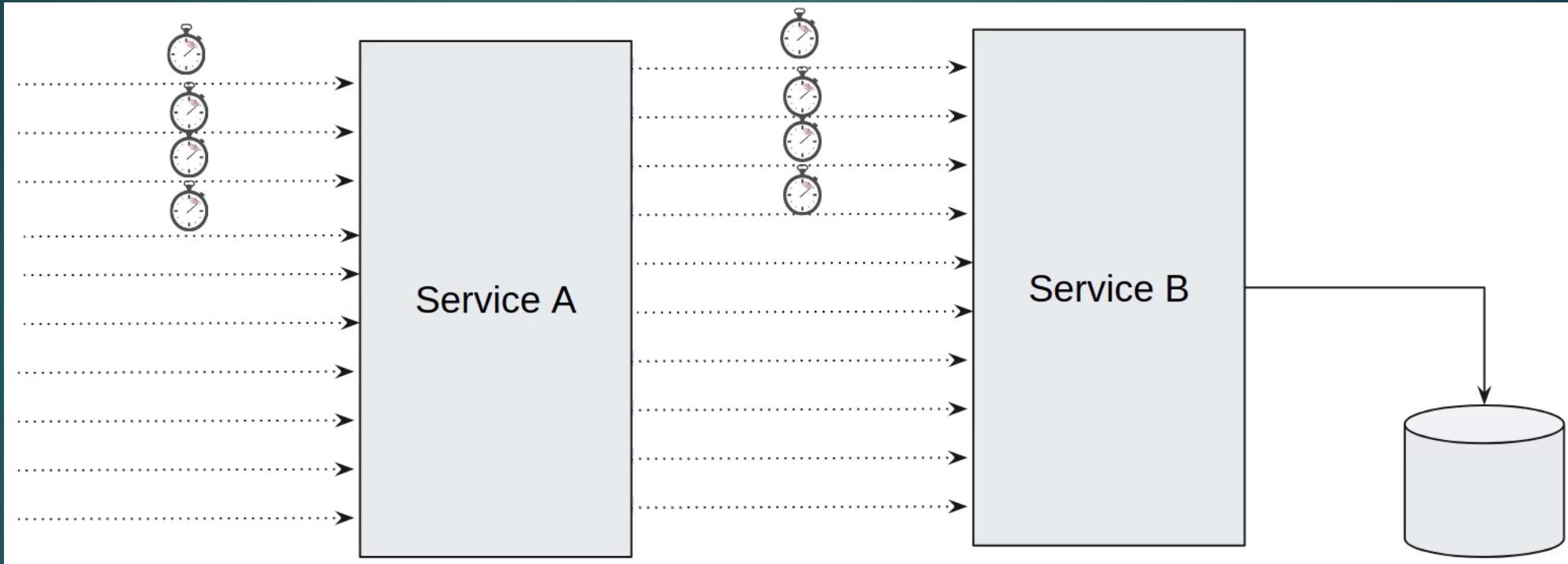
# Fonctionnement classique d'un serveur



Une requête => Un thread

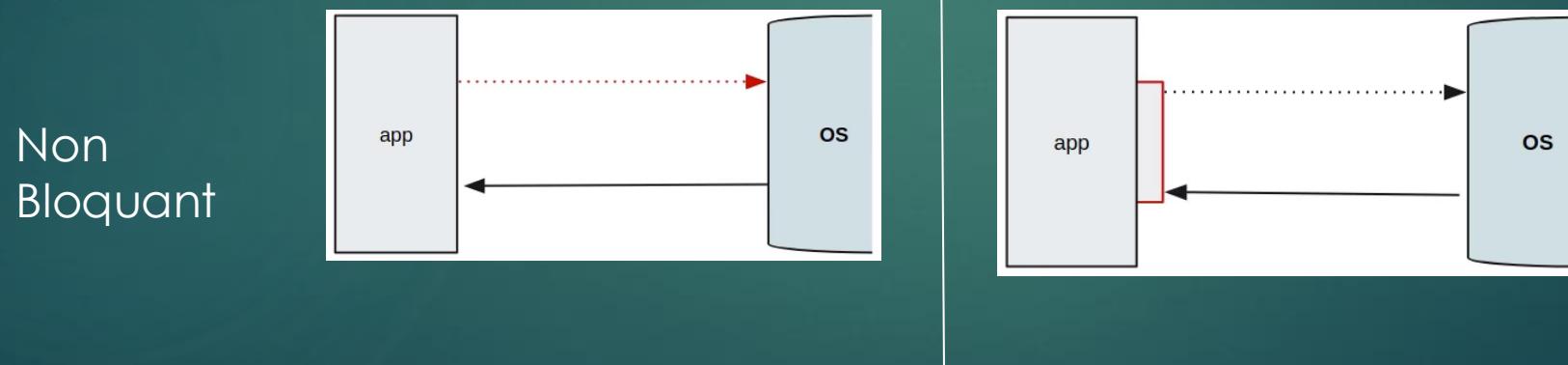
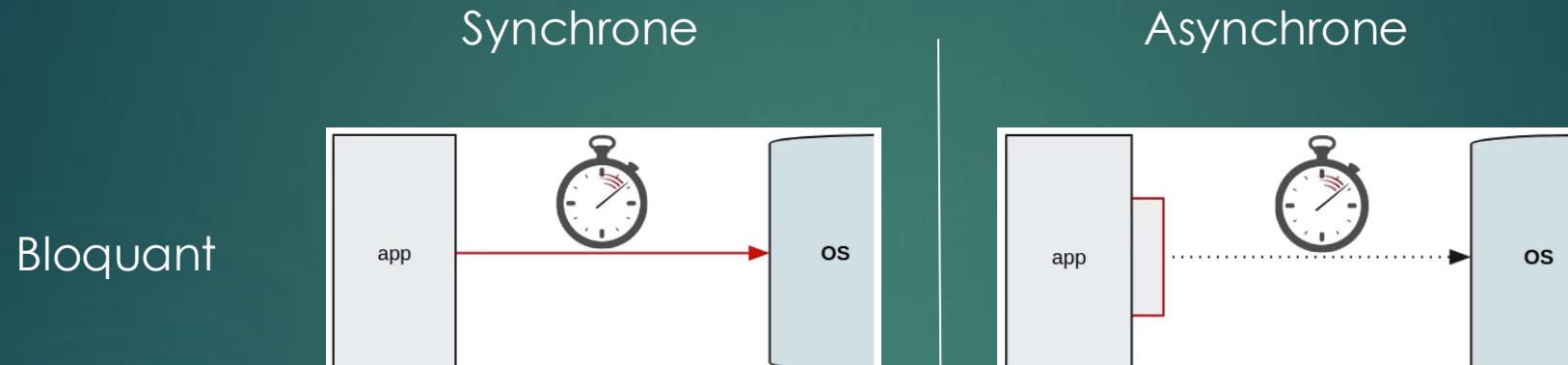
**Quels sont les principaux problèmes de cette approche ?**

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

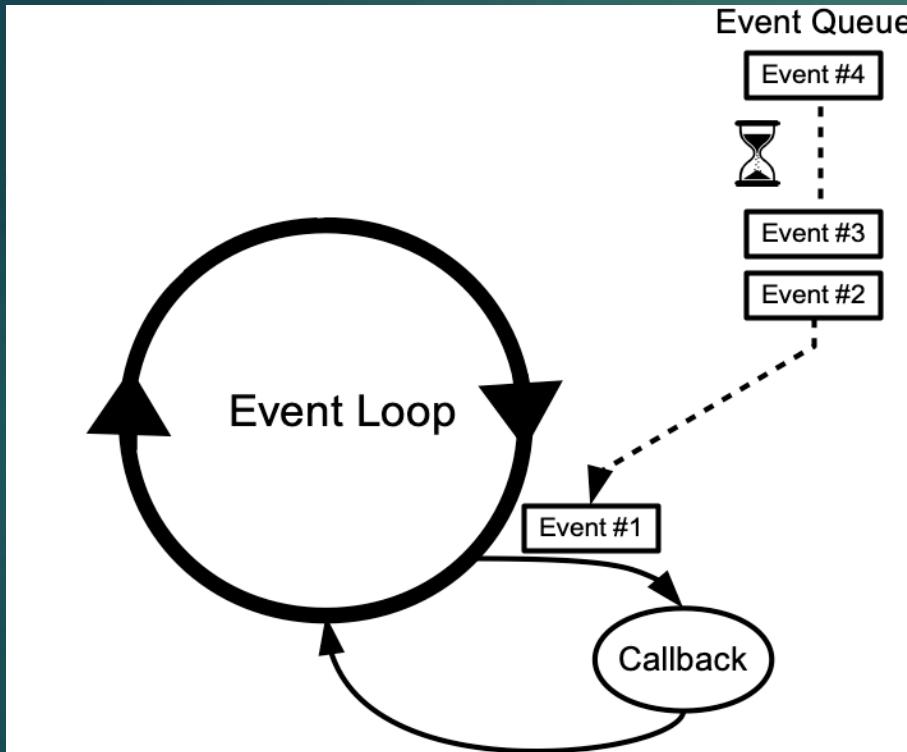


Approche consommatrice de ressources peu scalable  
**Quelles solutions mettre en œuvre pour traiter ce problèmes ?**  
**Quels problèmes sont posés par cette approche ?**

# Patterns de communication



# Event-Driven programming



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

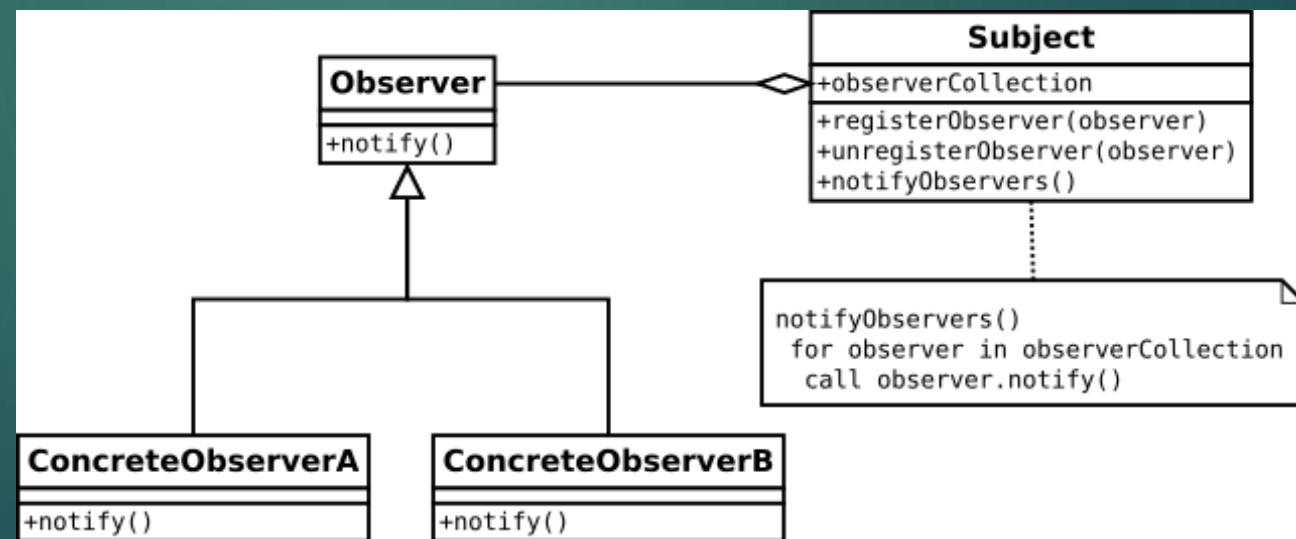
# "Call-back Hell"

```
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

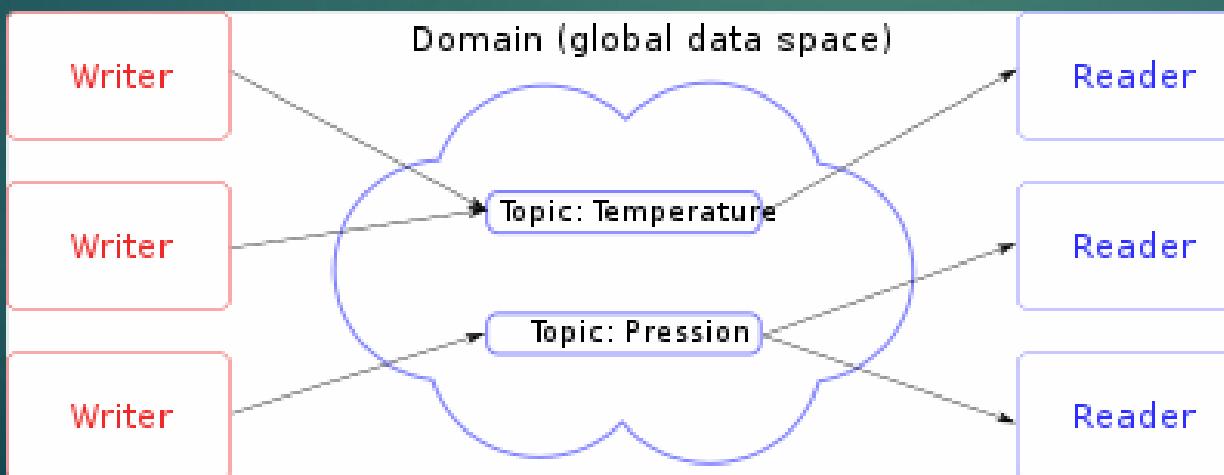
- ▶ Permet de limiter le couplage entre objets aux seuls phénomènes (événements) observables
- ▶ Un sujet observable notifie ses changements d'état à ses abonnés (observateurs)
  - ▶ Chaque abonné est responsable des conséquences des changements observés sur le sujet



# 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 ?**

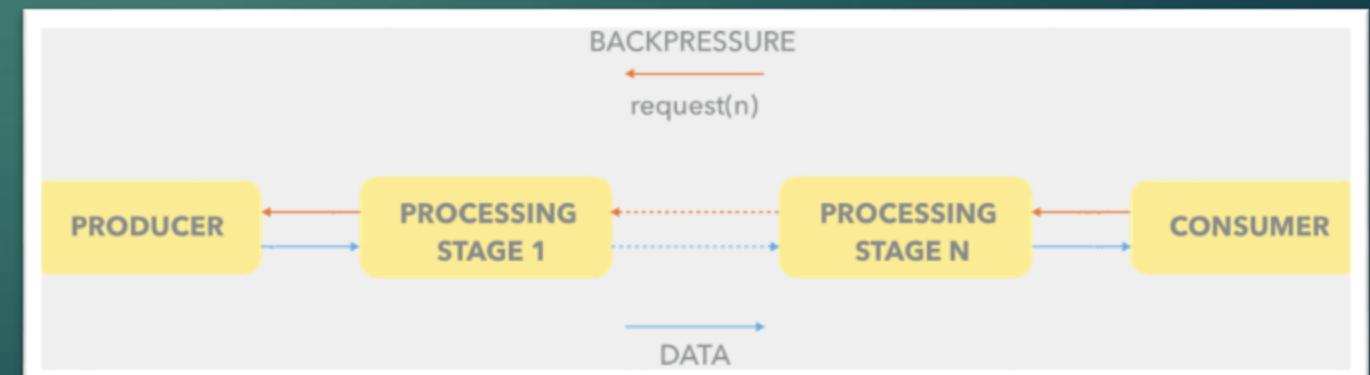
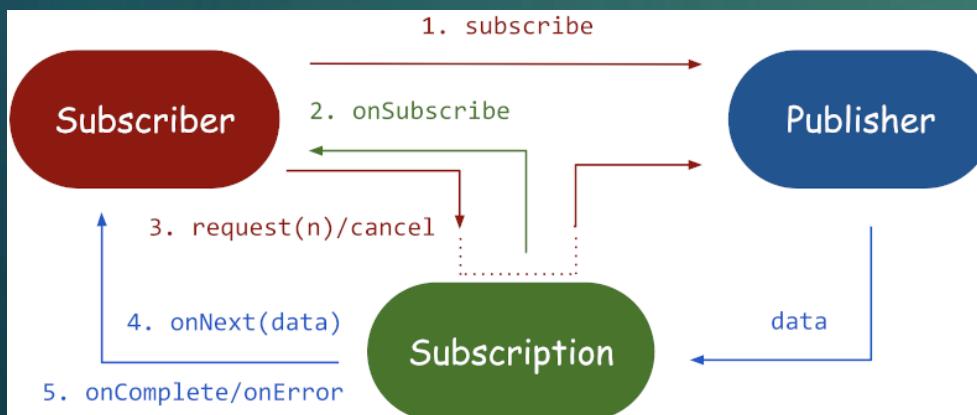
**Quelles en sont les limites ?**

# 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

# 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.util.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 publisher 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

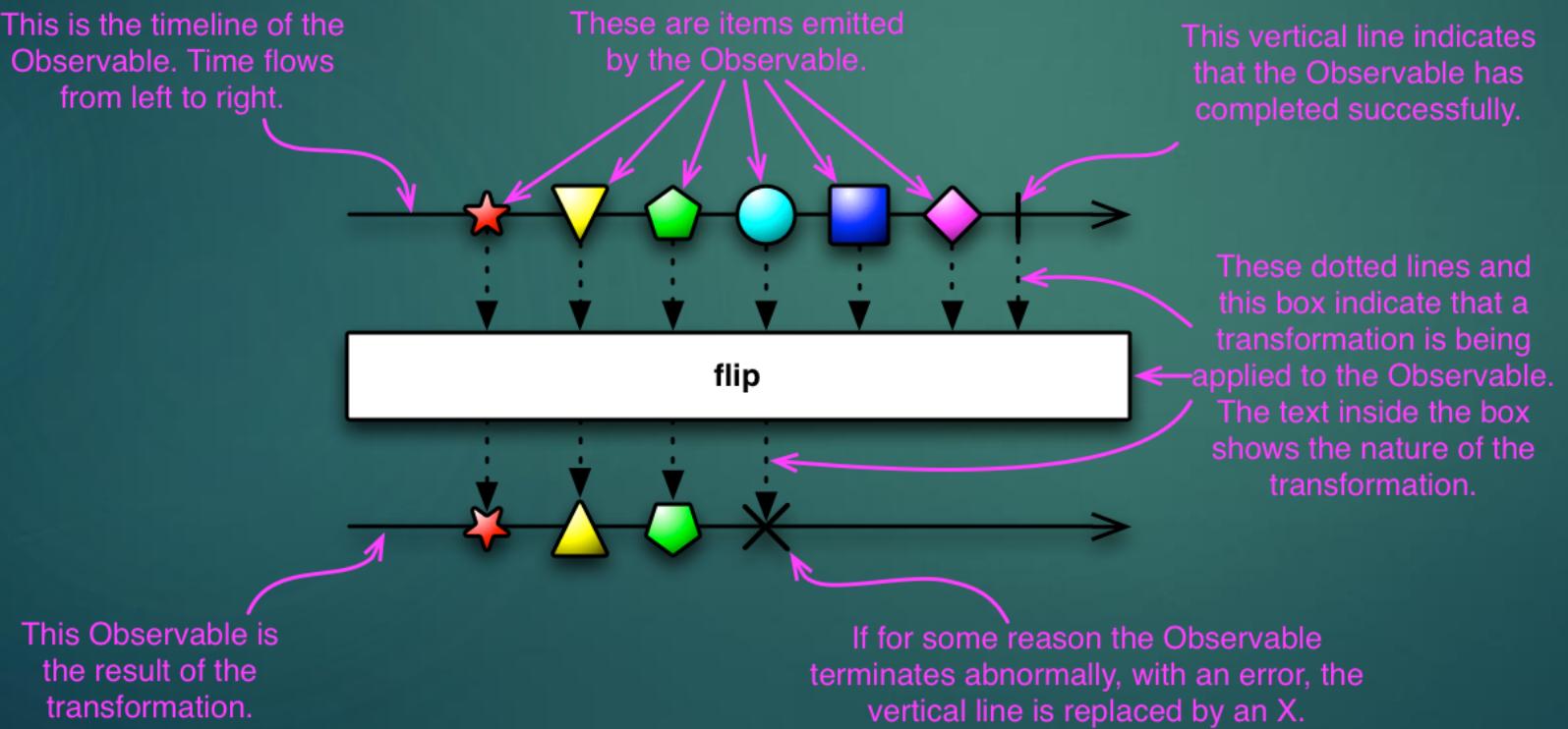
Selon le "Reactive Manifesto", le back-pressure est défini comme suit:

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>

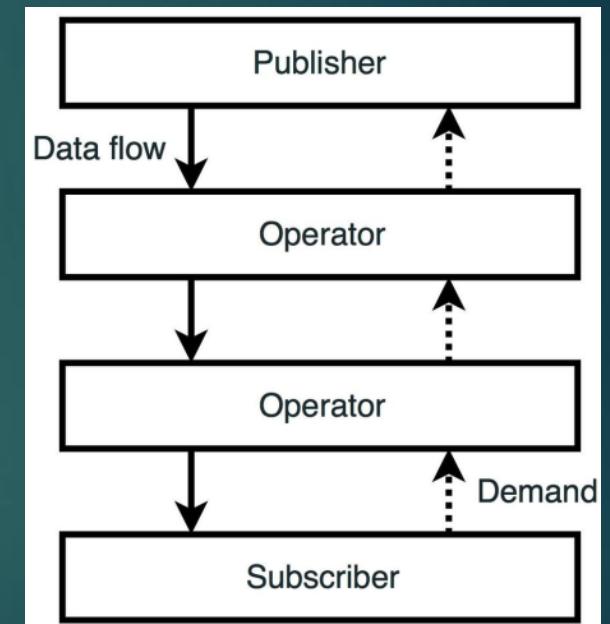
# Objectifs

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

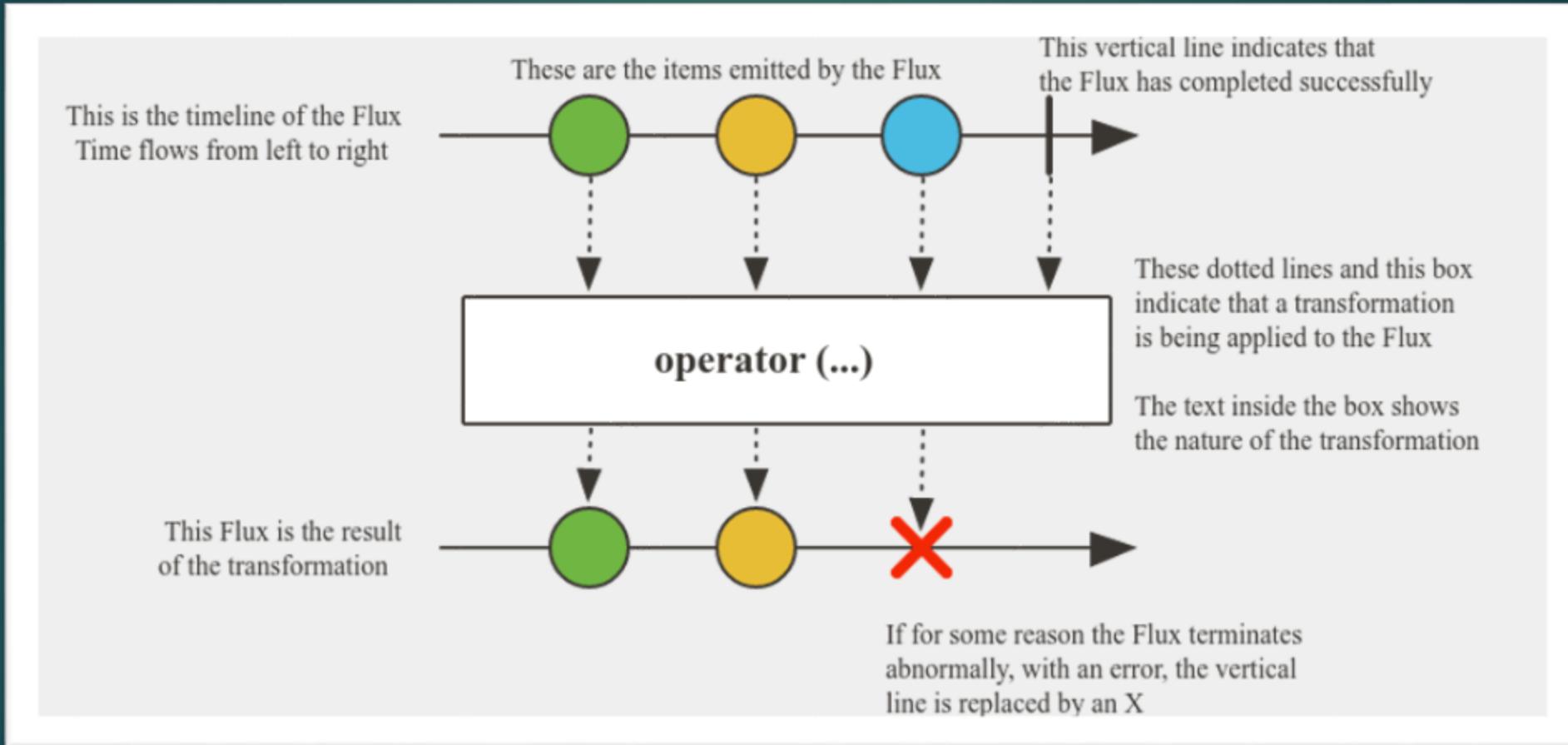
- ▶ **L'ingestion de données distribuées à haut débit** à partir d'une variété de sources d'entrée dans un store big data tel que HDFS ou Splunk.
- ▶ **Analyse en temps réel** au moment de l'ingestion, par exemple, collecte de métriques et comptage de valeurs.
- ▶ **Gestion du flux** via des batchs combinant 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



# 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()** génère 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

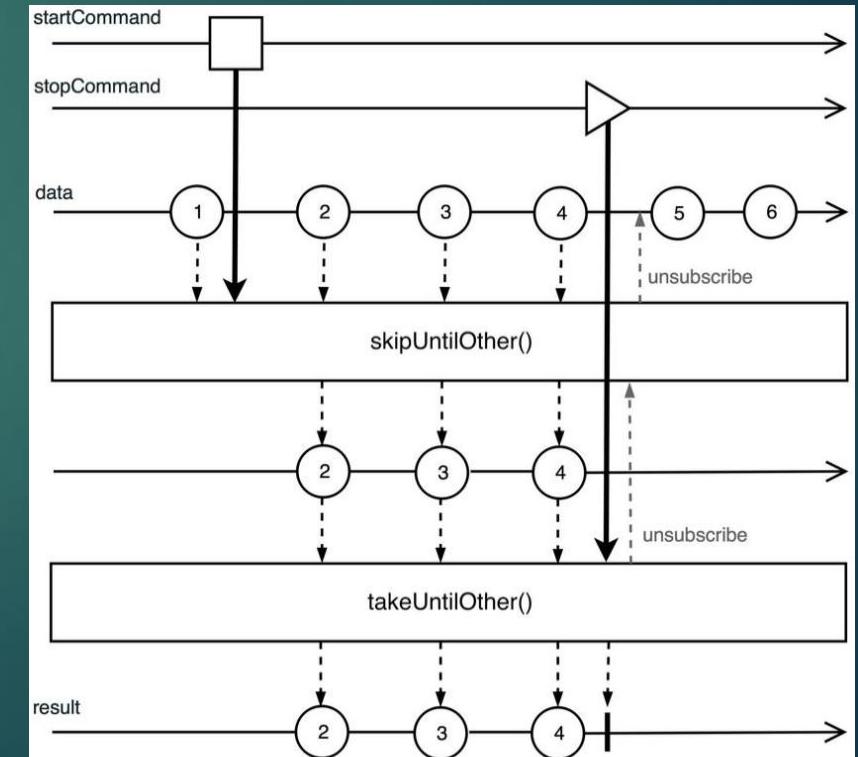
- ▶ Reactor permet d'ajouter des hooks aux événements associés à la publication
- ▶ Ces méthodes sont généralement de la forme doOnXXXX
  - ▶ XXXX représentant l'événement sur lequel on désire ajouter un hook
- ▶ Exercice: Modifier la classe FluxGenerate afin de tester ces différents hooks

# 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>

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

```
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);
```



# 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 prédictat 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 déferer 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 publishers 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)`

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

# Graal VM - Principes

- ▶ **Analyse statique au moment de la compilation** : GraalVM examine tout le code accessible de l'application et de ses dépendances sous une hypothèse de monde fermé (closed-world). Seules les classes, méthodes et champs qui seront effectivement utilisés sont inclus dans l'exécutable. Tout code inutile est éliminé (tree shaking).
- ▶ **Compilation Ahead-Of-Time** : Toute la compilation du code Java vers le code machine natif est réalisée à la construction, contrairement à la JVM classique avec compilation Just-In-Time (JIT) à l'exécution. L'exécutable résultant démarre ainsi très rapidement et bénéficie d'une exécution performante immédiatement.
- ▶ **SubstrateVM** : Une couche runtime légère incluse dans l'exécutable fournit les fonctionnalités basiques de la JVM (gestion mémoire, threads, GC simplifié) mais en natif, sans JVM installée.

# Graal VM - Principes

- ▶ **Initialisation à la compilation** : Certaines classes peuvent être initialisées à la compilation (init at build time) plutôt qu'au démarrage, réduisant les temps de démarrage et les vérifications dynamiques.
- ▶ **Heap snapshotting** : À la compilation, l'état initial de la mémoire (objets statiques, etc.) est capturé dans l'image native, ce qui accélère encore le lancement.
- ▶ **Résultat** : Un binaire léger, autonome (sans JVM requise), qui démarre en millisecondes, consomme moins de mémoire, avec une surface d'attaque réduite et une meilleure intégration en containers.

# 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

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

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

```
@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
```

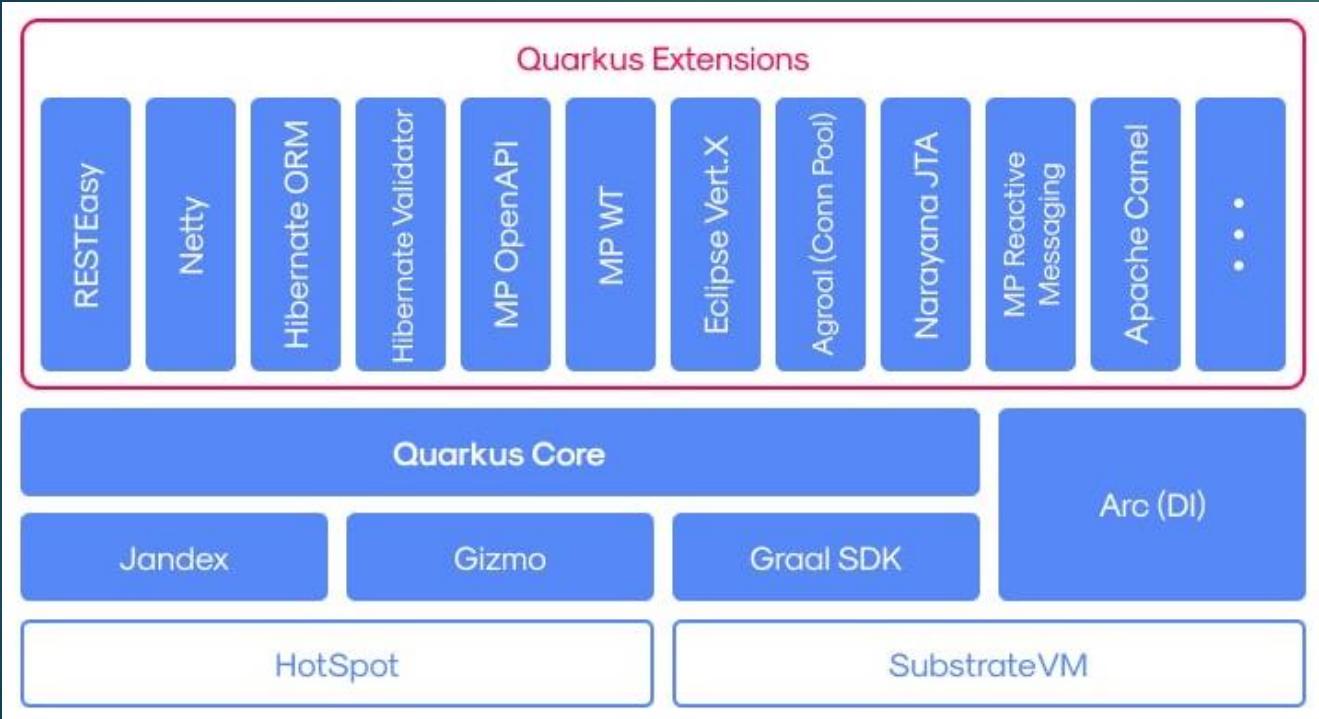
Génération d'une image native

```
$ ./gradlew :users:buildNative
```

Génération d'une image docker déployable

```
$ ./gradlew :users:build -Dquarkus.container-image.build=true
```

# Architecture



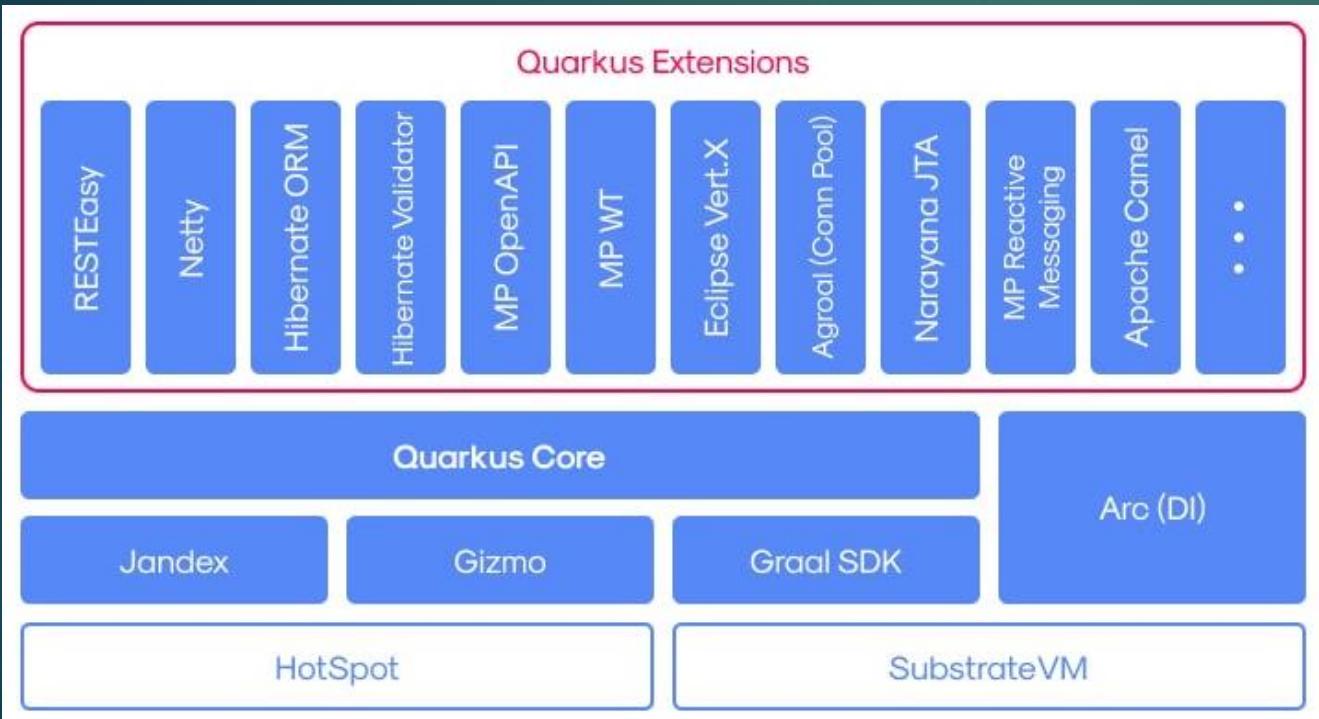
## Runtimes JVM ou natif

- **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 métadonnées 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: le fichier **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

- ▶ Le framework Quarkus permet de faire de l'AOP, mais différemment des frameworks comme Spring.
- ▶ Quarkus utilise principalement les interceptors CDI (Contexts and Dependency Injection), qui permettent d'obtenir des fonctionnalités AOP (avant/après exécution, autour, interception sur annotation personnalisée, etc.) sur les beans CDI.
- ▶ Il n'implémente pas nativement AspectJ, mais offre des possibilités similaires à travers les interceptors, avec une approche orientée compilation (build-time) pour maximiser la performance, en accord avec sa philosophie cloud-native.

# AOP: Interceptors

- ▶ Dans Quarkus, la mise en place d'un interceptor CDI consiste à :
  - ▶ Créer une annotation de liaison (interceptor binding),
  - ▶ Coder l'interceptor avec @AroundInvoke,
  - ▶ Appliquer à une classe ou une méthode métier.
- ▶ C'est l'approche native pour réaliser de l'AOP avec Quarkus sans surcouche.

```
@Logged  
@Interceptor  
@Priority(Interceptor.Priority.APPLICATION)  
public class LoggingInterceptor {  
    @AroundInvoke  
    public Object log(InvocationContext ctx) throws Exception {  
        System.out.println(">> Appel de : " + ctx.getMethod().getName());  
        return ctx.proceed(); // Exécute la méthode interceptée  
    }  
}
```

```
@InterceptorBinding  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE, ElementType.METHOD})  
public @interface Logged {}
```

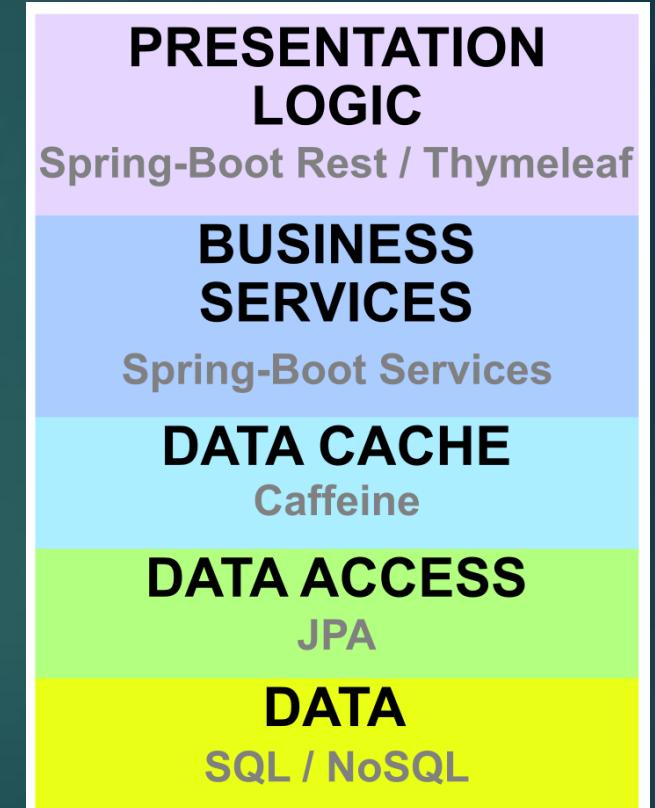
```
@ApplicationScoped  
@Logged  
public class DevService implements HelloService {...}
```

# AOP: Quelques advices utiles

- ▶ **@CacheResult** et **@CacheInvalidate** : Pour le caching transparent des méthodes ou des résultats (extension quarkus-cache).
- ▶ **@Retry** : Pour la gestion automatique des tentatives de réexécution en cas d'échec, souvent utilisée avec quarkus-smallrye-fault-tolerance.
- ▶ **@CircuitBreaker** : Pour activer le pattern circuit breaker et protéger une méthode/service des défaillances en cascade (quarkus-smallrye-fault-tolerance).
- ▶ **@Timeout** : Pour définir un délai maximal d'exécution autorisé, passée ce délai l'appel échoue et peut être traité de façon alternative.
- ▶ **@Bulkhead** : Pour limiter le nombre d'exécutions concurrentes d'une méthode donnée.
- ▶ **@Fallback** : Pour spécifier une alternative ("plan B") à appeler lors d'une erreur ou d'un timeout.
- ▶ **@Scheduled** : Pour planifier l'exécution périodique d'une méthode (quarkus-scheduler).

# Anatomie d'une application moderne

- ▶ Cette architecture suit un modèle commun aux applications Spring Boot, séparant les préoccupations en couches distinctes:
  - **Logique de présentation** : La couche supérieure utilise Spring Boot REST pour traiter les requêtes HTTP et Thymeleaf comme moteur de templating pour le rendu des vues côté serveur.
  - **Business Services** : Cette couche est mise en œuvre à l'aide des services Spring Boot, qui contiennent généralement la logique métier de base de l'application.
  - **Cache de données** : Caffeine est utilisé comme solution de mise en cache, ce qui permet d'améliorer les performances en stockant les données fréquemment consultées en mémoire.
  - **Accès aux données** : Java Persistence API (JPA) est utilisé pour l'accès aux données. JPA est une spécification pour l'ORM (Object-Relational Mapping) dans les applications Java.
  - **Données** : L'application peut fonctionner avec différents types de bases de données.
- ▶ Elle permet d'améliorer l'organisation, la maintenabilité et l'évolutivité de l'application.
- ▶ L'utilisation de Spring Boot dans l'ensemble de la pile garantit la cohérence et tire parti de ses fonctions d'auto-configuration et de gestion des dépendances.

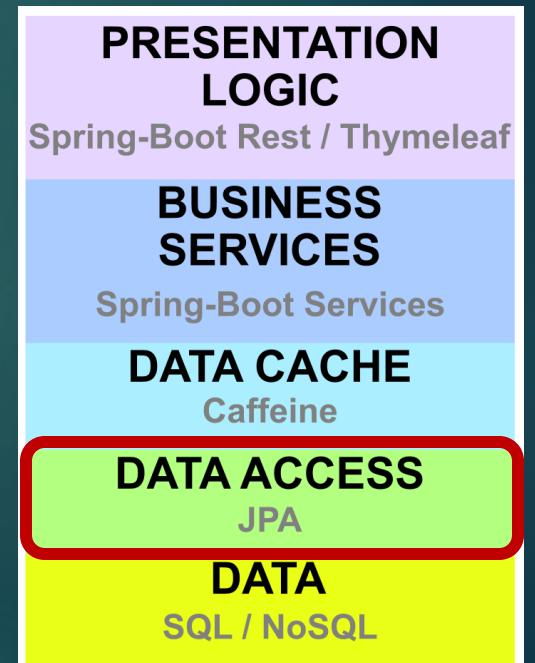
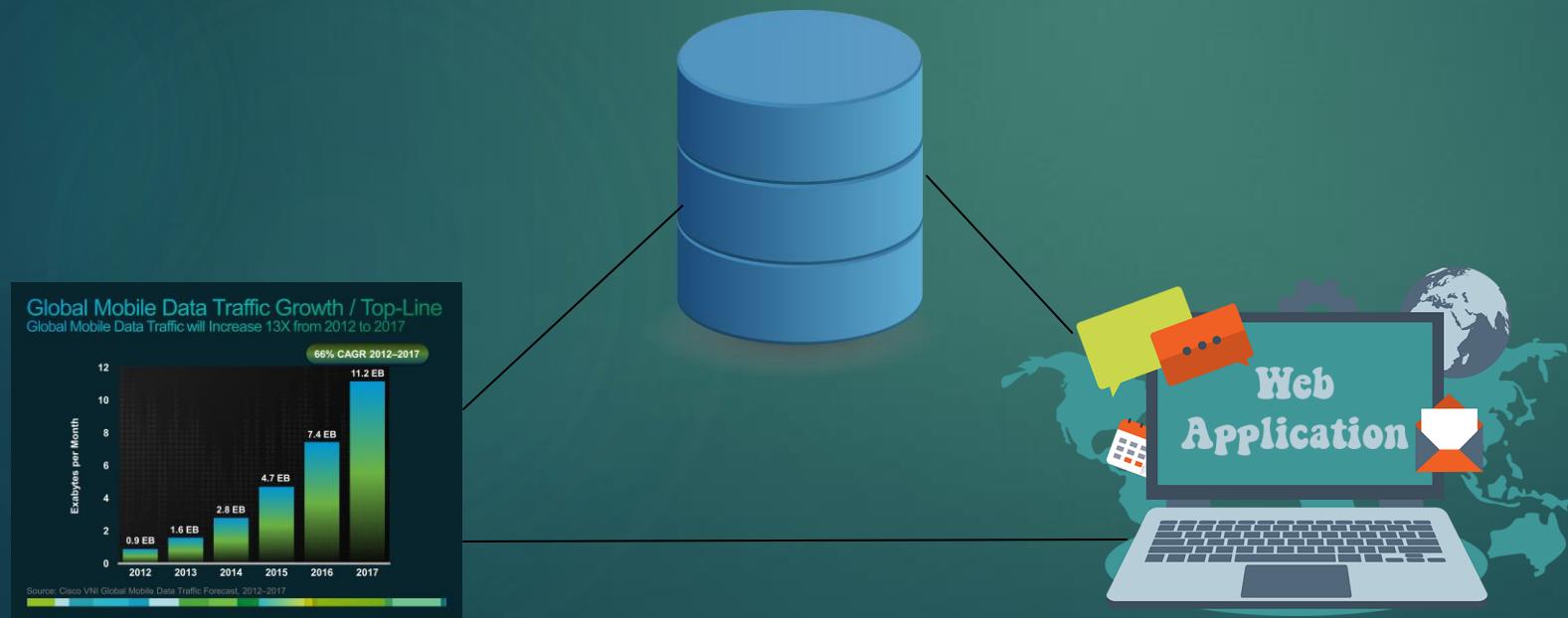




Persistence

# Persistiance des données

- ▶ Les applications manipulent des données
  - ▶ Ces données peuvent provenir de différentes sources: fichiers ou base de données
  - ▶ Ces données peuvent être stockées pour pouvoir être réutilisées ou partagées
  - ▶ La persistiance permet aux données d'exister même quand les applications qui les manipulent ne s'exécutent plus



# JPA: Présentation

- ▶ JPA permet de cacher au développeur les détails techniques de la persistance
  - ▶ Qui est du ressort de la base de données
  - ▶ Cela permet de se focaliser sur les problèmes métiers
- ▶ Les API de JPA sont définies dans les paquetages jakarta.persistence.\*
  - ▶ Cf. <https://jakarta.ee/specifications/persistence/3.2/jakarta-persistence-spec-3.2>
- ▶ Les API de JPA ne définissent que des interfaces
  - ▶ Qui sont implantées par différents outils:
    - ▶ Eclipse Link, Hibernate, OpenJPA ou Data Nucleus
  - ▶ Par exemple, SPRING utilise Hibernate



# JPA – Java Persistence API

- ▶ Java Persistence API (JPA) est une spécification qui définit comment gérer et accéder aux données relationnelles dans les applications Java.
- ▶ JPA se compose de trois éléments principaux : les entités, les gestionnaires d'entités et les unités de persistance.
  - ▶ Les **entités** sont des classes Java qui correspondent à des tables et des colonnes de base de données. Elles sont annotées avec @Entity et d'autres annotations pour spécifier leurs propriétés et leurs relations.
  - ▶ Les **gestionnaires d'entités** sont des objets qui fournissent des méthodes de création, de mise à jour, de suppression et d'interrogation des entités. Ils sont obtenus à partir d'une fabrique de gestionnaires d'entités, qui est configurée avec une unité de persistance.
  - ▶ Les **unités de persistance** sont des groupes logiques d'entités qui partagent la même connexion à la base de données et la même configuration. Elles sont définies dans un fichier persistence.xml qui spécifie les classes d'entités, le pilote de base de données, l'URL de connexion et d'autres propriétés.

# JPA – Java Persistence API

- ▶ JPA prend en charge différents types de mapping entre les entités et les tables de la base de données, tels que one-to-one, one-to-many, many-to-one et many-to-many. Il prend également en charge l'héritage, le polymorphisme et les objets intégrés.
- ▶ JPA fournit un langage de requête appelé **JPQL** (Java Persistence Query Language) qui permet d'écrire des requêtes dans une syntaxe de type SQL, mais en utilisant les noms et les attributs des entités au lieu des noms et des colonnes des tables. Il prend également en charge les requêtes par critères, qui sont des requêtes programmées à l'aide d'une API fluide.
- ▶ JPA n'est pas une implémentation spécifique mais une norme qui peut être mise en œuvre par différents fournisseurs. Parmi les fournisseurs JPA les plus populaires, on peut citer Hibernate, EclipseLink et OpenJPA.

# ORM – Object Relational Mapping

- ▶ La persistance d'objets dans des bases de données relationnelles passe par une traduction entre deux paradigmes
  - ▶ Le paradigme objet d'une part, basé sur des objets et des pointeurs
  - ▶ Le paradigme entité/relation d'autre part, basé sur des colonnes et des clefs
  - ▶ **Question:** Quels sont les différences fondamentales entre les deux paradigmes ?
- ▶ Lorsque l'on veut persister des objets java dans des bases relationnelles, il est nécessaire de définir un mapping entre les deux mondes
  - ▶ C'est précisément l'objectif d'un ORM
  - ▶ Et c'est ce que JPA permet de spécifier
- ▶ Les avantages de l'utilisation d'un ORM sont nombreux:
  - ▶ Gain de productivité
  - ▶ Meilleure séparation des préoccupations
  - ▶ Meilleure maintenabilité

# JPA: Configuration

Propriété	Description	Exemple
quarkus.datasource.db-kind	Type de SGBD : postgresql, mysql, mariadb, h2, etc.	quarkus.datasource.db-kind=postgresql
quarkus.datasource.jdbc.url	Chaine de connexion JDBC	quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/ma_db
quarkus.datasource.username	Nom d'utilisateur SQL	quarkus.datasource.username=myuser
quarkus.datasource.password	Mot de passe SQL	quarkus.datasource.password=secret
quarkus.datasource.jdbc.max-size	Taille max du pool de connexions JDBC	quarkus.datasource.jdbc.max-size=8
quarkus.datasource.jdbc.min-size	Taille min du pool	quarkus.datasource.jdbc.min-size=2
quarkus.hibernate-orm.database.generation	Stratégie de gestion du schéma (update, drop-and-create, none)	quarkus.hibernate-orm.database.generation=update
quarkus.hibernate-orm.log.sql	Affichage des requêtes SQL dans les logs	quarkus.hibernate-orm.log.sql=true

# JPA: Configuration des beans

- ▶ Comme pour les beans, il y a deux manières de configurer JPA
  - ▶ Par les annotations ou par les fichiers XML
  - ▶ Dans le cas où les deux sont présents, les fichiers XML ont la priorité
- ▶ La définition des mappings est basée sur l'utilisation de métadonnées sous forme d'annotations
- ▶ Les objets devant être persistés dans des bases de données sont marqués par l'annotation @Entity
- ▶ Une entité est un composant scannable
- ▶ Elle définit le mapping entre:
  - ▶ Objet et table
  - ▶ Propriétés et colonnes
  - ▶ Contraintes d'intégrités
- ▶ Elle est manipulable comme un objet
  - ▶ On peut lui ajouter des méthodes par exemple

```
@Entity
@Table(
    name = "persons",
    uniqueConstraints = {
        @UniqueConstraint(columnNames = {"firstName", "lastName"})
    }
)
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", nullable = false)
    private Long id;
    private String firstName;
    @Column(length = 50)
    private String lastName;
    private int age;
}
```

# JPA: Règles de mapping

- ▶ L'annotation **@Entity** sert à indiquer que les instances d'une classe seront persistées dans une table
- ▶ Les champs de la classe doivent être publiques et non statiques ni finaux
- ▶ Sans aucune autre information, JPA applique des règles de mapping par défaut, à savoir:
  - ▶ Entité ⇒ Table de même nom
  - ▶ Attribut ⇒ Colonne de même nom
- ▶ Sur les types également, JPA applique des règles de mapping par défaut, selon les SGBD
  - ▶ String ⇒ varchar(255)
  - ▶ Long ⇒ BIGINT
  - ▶ LocalDate, LocalDateTime => DATE, DATETIME
  - ▶ ...
- ▶ Ces règles sont cependant modifiables par des annotations
  - ▶ Nom des tables et des colonnes
  - ▶ Champs requis
  - ▶ Taille des champs
  - ▶ ...

# JPA: Annotations pour les entités

- ▶ **@Table:** Précise le nom de la table concernée par le mapping
- ▶ **@Column:** Associe un champ de la table à la propriété
- ▶ **@Id:** Associe un champ de la table à la propriété en tant que clé primaire
- ▶ **@GeneratedValue:** Demande la génération automatique de la clé primaire
- ▶ **@Basic:** représente la forme de mapping la plus simple; utilisée par défaut
- ▶ **@Transient:** demande de ne pas tenir compte du champ lors du mapping
- ▶ Chacune de ces annotations possède un certain nombre d'attributs
  - ▶ Cf. <https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html>
- ▶ Il est également possible d'utiliser des objets dont les propriétés seront agrégées aux objets qui les référencent
  - ▶ Par l'annotation **@Embeddable**
  - ▶ Intéressant quand plusieurs objets partagent des propriétés communes
  - ▶ Ex: Personne → Adresse

# JPA: Relations entre entités

- ▶ Les objets en Java sont reliés par des références. Dans les bases de données, les tables sont reliées par des jointures, basées sur des clefs
- ▶ Dans le cas de références à des tableaux d'objets ou des références croisées, JPA offre également un certain nombre d'annotations pour préciser le mapping
- ▶ Il y a différents types de relations entre entités et JPA permet d'annoter les références afin de spécifier les arités des relations et les cycles de vie associés
  - ▶ 1..1 : **@OneToOne** ⇒ Stratégie de mapping: JoinColumn
  - ▶ 1..\* : **@OneToMany** ⇒ Stratégie de mapping: JoinTable
  - ▶ \*..1 : **@ManyToOne** ⇒ Stratégie de mapping: JoinColumn
  - ▶ \*..\* : **@ManyToMany** ⇒ Stratégie de mapping: JoinTable
- ▶ Pour chaque type de relation, il est possible d'en préciser les propriétés
  - ▶ **Optional**: La relation n'est pas obligatoire
  - ▶ **MappedBy**: nom de la référence inverse
  - ▶ **Cascade**: Action à réaliser en cas de modification (CRUD)

# JPA: Relations entre entités

- ▶ Pour chaque classe référencée, un identifiant est généré
  - ▶ Il est possible de changer cet identifiant par l'annotation **@JoinColumn**
- ▶ Pour les arités n-aires, il est possible de générer une table d'association par l'annotation **@JoinTable**
  - ▶ Le choix d'utiliser l'annotation **@JoinTable** plutôt que **@JoinColumn** est une question de compromis entre performance et empreinte mémoire
- ▶ Chargement des entités référencées
  - ▶ **Eager**: chargée systématiquement
  - ▶ **Lazy**: chargée à la demande
  - ▶ Le choix de l'une ou l'autre stratégie est une question de compromis entre performance et empreinte mémoire

```
@Basic(fetch = FetchType.LAZY)  
private Appointments[] appts;
```

```
@ManyToOne  
 @JoinTable(name = "rdv_patient",  
 joinColumns = @JoinColumn(name = "patient_id"),  
 inverseJoinColumns = @JoinColumn(name = "rdv_id"))  
private Appointment appt;
```

# JPA: Cascading

- ▶ ALL: Regroupe toutes les stratégies (y compris celles spécifiques à Hibernate)
- ▶ PERSIST: Propage la persistance aux enfants
- ▶ MERGE: Copie les propriétés d'un parent aux enfants
- ▶ REMOVE: La suppression du parent entraîne la suppression des enfants
- ▶ REFRESH: La mise à jour du parent entraîne la mise à jour des enfants
- ▶ DETACH: Les enfants sont détachés lorsque leurs parents sont détachés
- ▶ REPLICATE (Hibernate): Synchronise les données dans le cas de sources de données multiples
- ▶ SAVE\_UPDATE (Hibernate): Propage les opérations de sauvegarde des parents aux enfants
- ▶ LOCK (Hibernate): Les enfants sont réattachés lorsque leurs parents sont réattachés

# JPA: Héritage entre entités

- ▶ Peu de SGBD supporte l'héritage entre tables
- ▶ Pour autant, l'héritage est une caractéristique importante du paradigme Objet
- ▶ JPA supporte l'héritage entre entités avec différentes stratégies de mapping possibles
  - ▶ MAPPED\_SUPER\_CLASS: La classe parent n'est pas persistée
  - ▶ SINGLE\_TABLE: Une seule table utilisée pour une hiérarchie de classes
  - ▶ JOINED: L'héritage est représenté par une jointure entre table parente et tables filles
  - ▶ TABLE\_PER\_CLASS: l'héritage se manifeste par une table par entité

```
@MappedSuperclass  
public abstract class Person {...}
```

```
@Entity  
@DiscriminatorValue("Patient")  
public class Patient extends Person {..}
```

```
@Entity  
@Inheritance(strategy = InheritanceType.JOINED)  
@DiscriminatorColumn(name = "person_type")  
public class Person {...}
```

```
@Entity  
@DiscriminatorValue("Patient")  
public class Patient extends Person {..}
```

```
@Entity  
@Inheritance(strategy =  
InheritanceType.SINGLE_TABLE)  
@DiscriminatorColumn(name = "person_type")  
public class Person {...}
```

```
@Entity  
@DiscriminatorValue("Patient")  
public class Patient extends Person {..}
```

```
@Entity  
@Inheritance(strategy =  
InheritanceType.TABLE_PER_CLASS)  
public class Person {...}
```

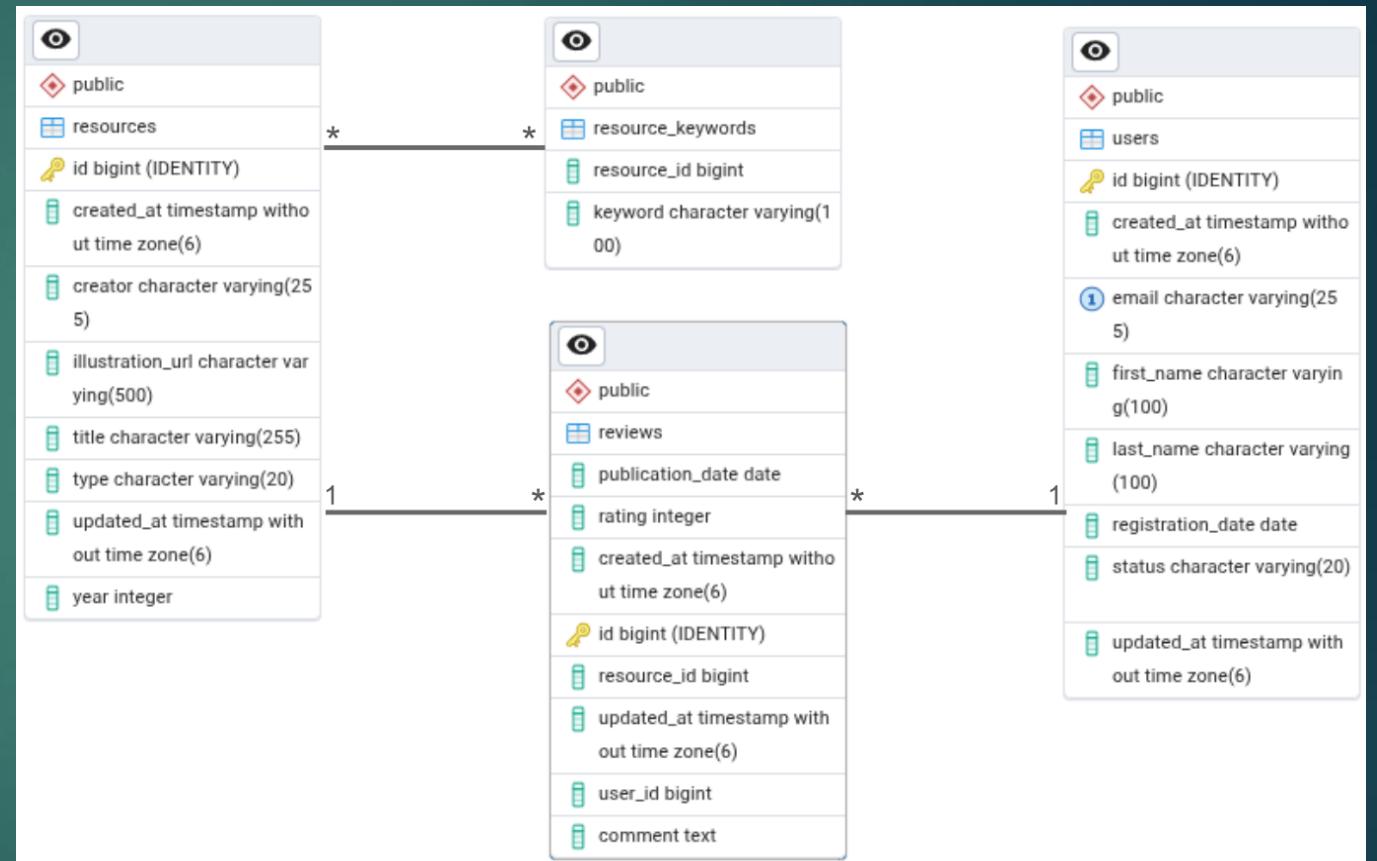
```
@Entity  
public class Patient extends Person {..}
```

# Panache

- ▶ Panache est une surcouche ORM proposée par Quarkus pour simplifier la définition, la manipulation et l'accès aux entités de données, en rendant la persistence "fun et triviale" pour les développeurs Java.
- ▶ Principes et objectifs
  - Réduction du code boilerplate : Panache cache la complexité d'Hibernate ORM et permet d'exprimer rapidement des opérations fréquentes, comme les recherches, créations et suppressions d'entités, avec une syntaxe minimale.
  - Active Record ou Repository : Les entités peuvent hériter de PanacheEntity (modèle Active Record : les méthodes CRUD et query sont sur l'entité elle-même) ou utiliser le pattern PanacheRepository (pour un découpage Clean Architecture).
  - Champs publics : Panache pousse à rendre les propriétés publiques pour alléger la syntaxe (pas de getter/setter de base).
  - Requêtes fluides : Les méthodes utilitaires comme find, list, persist sont directement accessibles sur l'entité ou le repository, acceptant de la syntaxe HQL ou JPQL simplifiée.

# Exercice 4: Modèle de données

- ▶ Nous allons implanter une solution de gestion de bibliothèque multimédia
- ▶ Spécifier les entités pour implanter ce modèle de données
- ▶ Tester différentes stratégies et observer l'effet sur la définition des schémas (avec PgAdmin4)
- ▶ **PS:** vous n'avez que le modèle de revues à implémenter, sur la base des exemples fournis



# Annotations de validation Jakarta

- ▶ **@NotNull** : la valeur ne doit pas être nulle.
- ▶ **@NotBlank** : le champ (String) doit être non nul et non vide (et comporter au moins un caractère non-espace).
- ▶ **@NotEmpty** : le champ (String, Collection, Map) doit contenir au moins un élément.
- ▶ **@Size(min, max)** : constraint la taille, longueur ou nombre d'éléments (String, Collection, tableau).
- ▶ **@Min / @Max** : bornes minimales et maximales sur une valeur numérique.
- ▶ **@Positive / @Negative** : doit être strictement positif ou négatif.

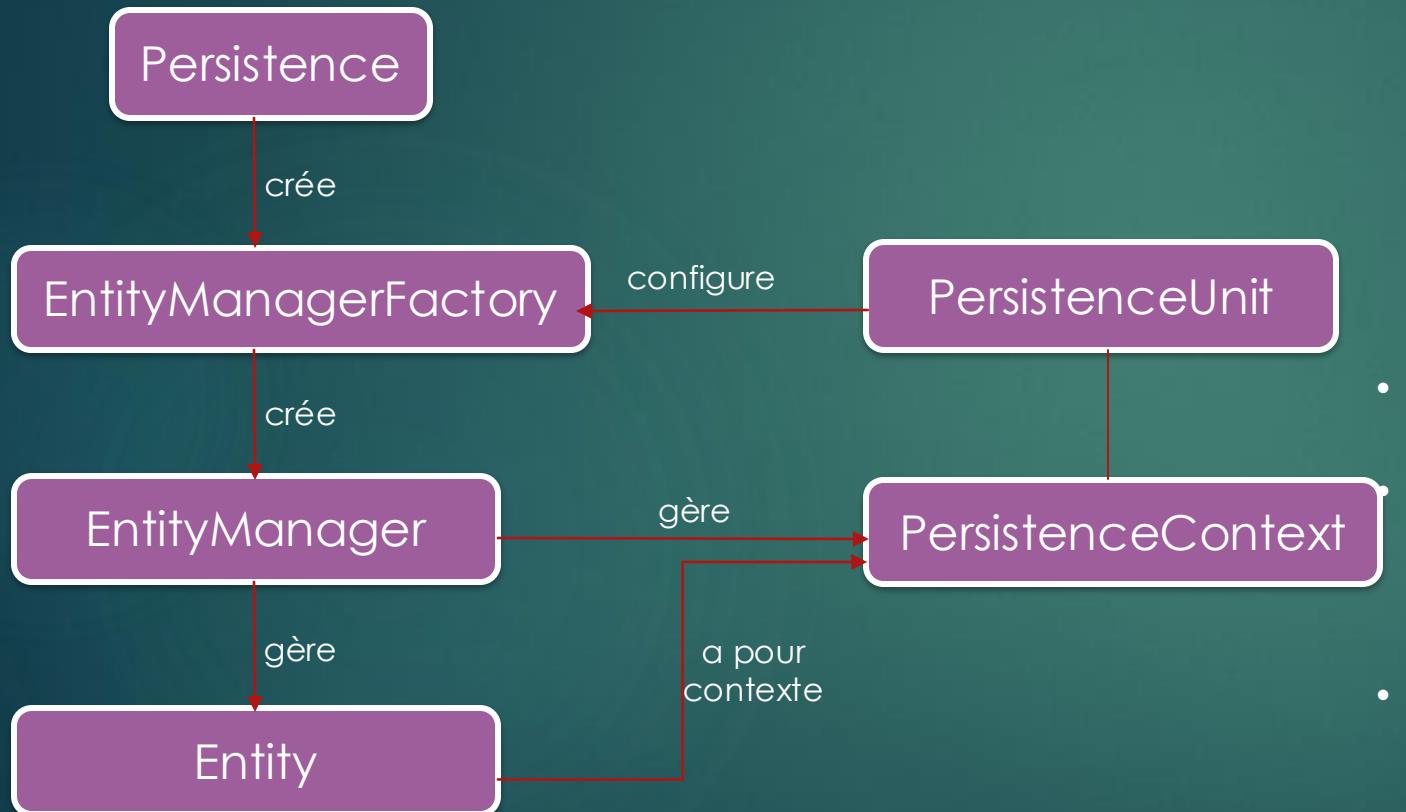
# Annotations de validation Jakarta

- ▶ **@Email** : la chaîne doit représenter une adresse email valide.
- ▶ **@Pattern(regexp="...")** : la valeur doit matcher une expression régulière donnée.
- ▶ **@Past / @PastOrPresent** : la date doit être dans le passé (ou présent).
- ▶ **@Future / @FutureOrPresent** : la date doit être dans le futur (ou présent).
- ▶ **@Digits(integer, fraction)** : limite le nombre de chiffres avant/après la décimale.
- ▶ **@AssertTrue / @AssertFalse** : la valeur booléenne doit être vraie/faux.
- ▶ **Exercice 4 (Cont'd)**: Préciser les contraintes de validation sur le modèle de données

# JPA: Manipulation des données

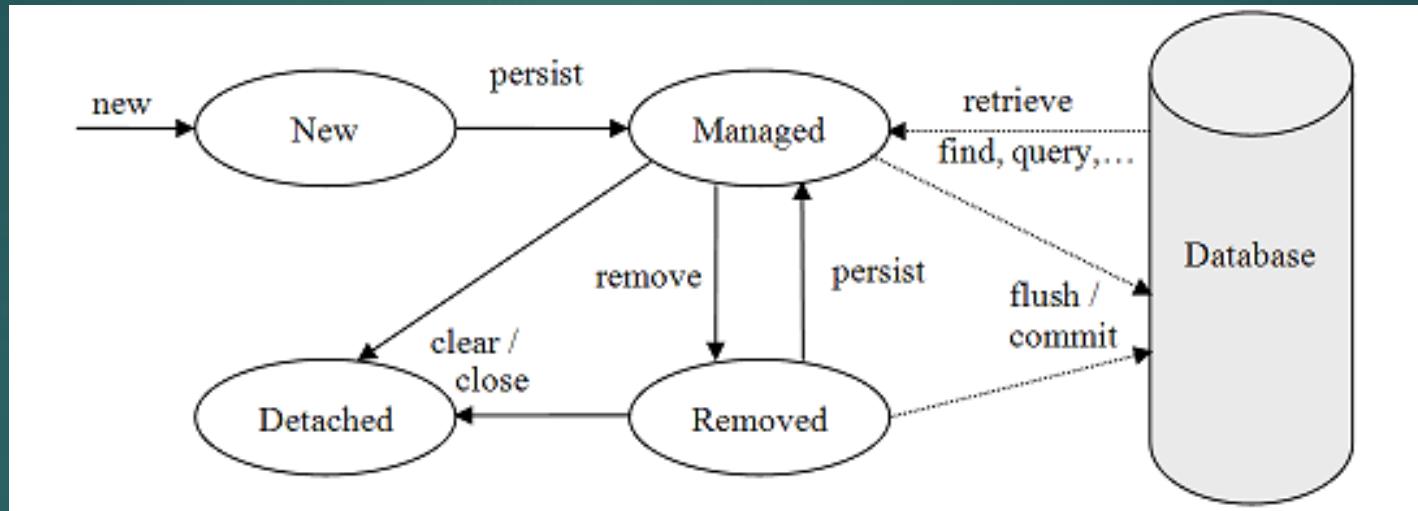
- ▶ Lorsque les entités ont été définies, il faut pouvoir les manipuler
  - ▶ À travers les opérations du CRUD
- ▶ Pour ce faire, JPA met à notre disposition un objet dédié: **Entity Manager**
  - ▶ Supportant toutes les opérations du CRUD et de modifier la base de données de manière programmatique et plus concise
  - ▶ Cf. <https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html>
- ▶ JPA fournit le langage JPQL permettant de faire des requêtes SQL
  - ▶ Quelle que soit la technologie choisie
  - ▶ Avec une expressivité supérieure à SQL
  - ▶ Qui renvoie des objets

# JPA: Manipulation des données



- L'EntityManagerFactory est créé à partir des propriétés d'accès à la base de données
  - URL, Driver, login et mot de passe
  - Définis dans le fichier application.properties dans le cas de SPRING
  - Ou à partir d'un fichier de configuration XML dans le cas où il y aurait plusieurs unités de persistance
- L'unité de persistance les propriétés de la source de données  
La factory a pour charge de gérer le cycle de vie des EntityManager
  - Notamment pour gérer les transactions (Utilisation de l'annotation @Transactional)
  - Ces objets sont gérés dans un contexte de persistance thread-safe

# Entité: Cycle de vie



- Une entité gérée est sous la responsabilité de son manager
  - Elle est synchronisée avec la base de données
- Lorsqu'elle est détachée, elle devient désynchronisée et peut être nettoyée par le GC
  - Si elle n'est plus référencée par aucun objet.

# Panache Entity

- ▶ La classe PanacheEntity dans Quarkus joue le rôle d'une entité métier (donnée persistée) dotée de nombreuses méthodes utilitaires facilitant l'accès aux données, sans pour autant être un équivalent direct avec un EntityManager.
- ▶ Rôles:
  - **Entité métier** : Une classe étendant PanacheEntity (ou PanacheEntityBase) devient une entité JPA classique, comportant ses propriétés persistées.
  - **Méthodes utilitaires** : Panache enrichit l'entité de nombreuses méthodes statiques (comme find(), persist(), delete(), list(), etc.), permettant d'exécuter les opérations de persistence et de recherche directement depuis la classe, sans déclarer d'EntityManager ni de repository particulier.
  - **Active Record** : Ce modèle s'inspire du pattern Active Record : l'entité manipule ses propres données et peut se charger elle-même de sa persistance, ce qui évite la verbosité de JPA traditionnel.

# Panache Entity vs Entity Manager

- ▶ Pas d'EntityManager direct : Panache encapsule la plupart des fonctionnalités classiques d'EntityManager (persist, merge, remove, find, etc.) en les proposant sous forme de méthodes de classe ou d'instance.
- ▶ Utilisation simplifiée : Pas besoin d'injecter ni de manipuler un EntityManager manuellement - la gestion du contexte, des transactions et du cycle de vie est entièrement prise en charge par Quarkus/Hibernate.
- ▶ Repository possible : Si toutefois vous préferez séparer la logique métier et l'accès aux données, il est possible d'utiliser (à la place ou en complément) PanacheRepository ou PanacheRepositoryBase.

# Panache Entity ou Panache Entity Base ?

- ▶ Étendre PanacheEntity :
  - Si vous voulez un identifiant primaire auto-généré (id de type Long).
  - Pour la majorité des cas CRUD standards, le mode "Active Record".
  
- ▶ Étendre PanacheEntityBase :
  - Si vous devez personnaliser l'identifiant (autre type, autre nom, composite key).
  - Pour une intégration avec une base existante ou une structure qui n'utilise pas de clé primaire simple Long.

# Active Record ou Repository ?

- ▶ Utilisez le modèle Active Record (PanacheEntity ou PanacheEntityBase) pour des applications simples où la logique métier et l'accès aux données peuvent cohabiter dans la même classe (exemples : microservices CRUD, démos, APIs simples).
- ▶ Privilégiez le modèle Repository (PanacheRepository<T>) pour des projets plus complexes, des règles métier avancées, ou si vous voulez séparer la persistance des entités.

# JPQL: Java Persistence Query Language

- ▶ Langage de requête proche du SQL avec une plus grande expressivité
  - ▶ Ex: `SELECT p FROM Person p WHERE p.age > 20`
- ▶ Les résultats des requêtes sont retournés sous forme de collections d'entités
- ▶ Syntaxe générale:
  - ▶ `SELECT <clause> FROM <clause> WHERE <clause> ORDER BY <clause> GROUP BY <clause> HAVING <clause>`
  - ▶ `UPDATE <clause> SET <clause> WHERE <clause>`
  - ▶ `DELETE <clause> WHERE <clause>`
- ▶ Cf. [https://en.wikibooks.org/wiki/Java\\_Persistence/JPQL](https://en.wikibooks.org/wiki/Java_Persistence/JPQL)

# JPQL: Requêtes

- ▶ 2 Types de requêtes:
  - ▶ Celles renvoyant une liste d'objet → TypedQuery
  - ▶ Celles ne renvoyant rien → Query
- ▶ Les requêtes sont paramétrables
  - ▶ Elles peuvent contenir des paramètres désignés par une syntaxe dédiée
  - ▶ Les paramètres sont mappées par des valeurs à travers la méthode setParameter
- ▶ Les requêtes renvoient une liste d'objets (getResultSet) ou des streams (getResultStream)
  - ▶ Pour tirer avantage de la programmation fonctionnelle

# JPQL: Fonctions

- ▶ Des fonctions peuvent servir de clause dans les SELECT
- ▶ JPQL définit un certain nombre de fonctions utilisables dans les SELECT
  - ▶ Statistiques: AVG, COUNT, MAX, MIN, SUM
  - ▶ Comparaisons: EQUAL, LESS THAN, LESS THAN OR EQUAL, BETWEEN, NOT BETWEEN
  - ▶ Mathématiques: SQRT
  - ▶ Chaînes de caractères: LENGTH, CONCAT, DATES

# PanacheQL

- ▶ PanacheQL est la syntaxe de requête simplifiée proposée par Quarkus Panache pour faciliter l'interrogation des entités ORM.
- ▶ Sa syntaxe permet d'écrire des fragments de requêtes très proches de JPQL/HQL, mais pensés pour privilégier la concision et la lisibilité dans les opérations courantes.
- ▶ Principes:
  - Fragment de clause WHERE/ORDER BY : La plupart du temps, on écrit simplement la condition de filtrage (WHERE) et/ou de tri (ORDER BY), en utilisant le nom des propriétés de l'entité et la syntaxe des opérateurs classiques.
  - Abstraction de la query complète : Panache contextualise automatiquement la requête pour l'entité concernée, donc plus besoin d'écrire SELECT ou FROM.
  - Interopérabilité JPQL/HQL : PanacheQL est un sous-ensemble du JPQL standard, ce qui garantit que toutes les propriétés, associations, et opérateurs sont compatibles.

# PanacheQL

- ▶ Pour des requêtes plus complexes, vous pouvez toujours basculer sur le JPQL/HQL complet en précisant la clause SELECT ou FROM dans la méthode (utile pour des projections, des DTO, ou de l'agrégat).
- ▶ PanacheQL offre de la pagination, du tri et prend en charge les projections via .project() pour adapter le résultat à des DTOs.

```
@PrePersist  
protected void onCreate() {  
    createdAt = LocalDateTime.now();  
    updatedAt = LocalDateTime.now();  
    if (publicationDate == null) {  
        publicationDate = LocalDate.now();  
    }  
}  
  
@PreUpdate  
protected void onUpdate() {  
    updatedAt = LocalDateTime.now();  
}  
  
/**  
 * Find all reviews for a specific resource  
 */  
public static List<Review> findByresourceId(Long rld) {  
    return list("resourceId ORDER BY publicationDate DESC", rld);  
}
```

<https://quarkus.io/guides/hibernate-orm-panache>

# Panache Query

- ▶ La classe PanacheQuery<T> est l'interface essentielle de Quarkus Panache pour manipuler des résultats de requêtes de façon fluide, **typée** et **paginée**.
- ▶ Rôle de PanacheQuery
  - Encapsulation d'une requête : Un objet PanacheQuery est retourné par les méthodes find() ou findAll() sur une entité Panache ou un repository. Il représente une requête potentiellement paginée, triée, ou filtrée.
  - API typée et riche : Permet de manipuler les résultats, la pagination, la navigation dans les pages, l'accès direct aux éléments ou aux streams.

```
public List<Resource> findAllPaginated(int pageIndex, int pageSize) {  
    return findAll()  
        .page(io.quarkus.panache.common.Page.of(pageIndex, pageSize))  
        .list();  
}
```

# API Criteria

- ▶ Les requêtes JPQL présentent quelques limitations
  - ▶ Ce sont des chaînes de caractères, statiques
  - ▶ Elles ne peuvent être vérifiées à la compilation
  - ▶ Si le modèle de données change, il est difficile d'évaluer l'impact sur la spécification des requêtes
- ▶ Pour pallier à ce problème, JPA fournit l'API Criteria
  - ▶ Permet de spécifier des requêtes entièrement en Java
  - ▶ Type-safe, en ligne avec le modèle de données
  - ▶ Plus pratique pour écrire des requêtes dynamiques (expressivité de Java)
  - ▶ Aussi performant que JPQL
- ▶ L'API Criteria fournit des équivalents à toutes les constructions JPQL
  - ▶ SELECT → select()
  - ▶ FROM → from()
  - ▶ ...

```
public List<Patient> getPatient(String name) {  
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();  
    CriteriaQuery<Patient> query = builder.createQuery(Patient.class);  
    Root<Patient> root = query.from(Patient.class);  
    query.select(root);  
    query.where(builder.equal(root.get("name").as(String.class), name));  
    return entityManager.createQuery(query).getResultList();  
}
```

# Requêtes natives

- ▶ Utilisation du SQL propriétaire
- ▶ Attention à la syntaxe
  - ▶ Les SGBD ne sont pas tous complètement conformes au SQL

```
public List<Patient> getPatients(String name) {  
    return entityManager  
        .createNativeQuery("SELECT * FROM Patient WHERE name = :name")  
        .setParameter("name", name)  
        .getResultList();  
}
```

# Panaches - Règles de bonnes pratiques

- ▶ Champs publics : Les champs sont souvent publics pour alléger la syntaxe, sauf si des invariants ou de la logique métier s'imposent sur les accès.
- ▶ Utilisation des méthodes utilitaires :
  - ▶ Préferez les méthodes fournies par Panache (find, list, persist, ...) pour plus de clarté et de concision.
  - ▶ Pour les requêtes complexes, vous pouvez toujours utiliser les méthodes JPQL/HQL standard ou injecter l'EntityManager si besoin.

# Panaches - Règles de bonnes pratiques

- ▶ Gestion des erreurs et transactions :
  - ▶ Utilisez l'annotation @Transactional sur les méthodes (service ou resource) qui modifient la base.
  - ▶ Vérifiez les accès concurrents et la gestion du rollback (surtout en production ou sur des microservices distribués).
- ▶ Tests et Indexation : Vos entités doivent être bien indexées, particulièrement en multi-module ou si tu utilises des librairies externes qui embarquent des entités Panache.
- ▶ Projection, DTO : Pour les lectures en masse ou les vues non-entity, utilisez des DTO ou la projection (pas nécessairement des entités Panache).
- ▶ **Exercice 4 (Cont'd)**: Complétez le support du cycle de vie des entités; vous choisirez pour cela l'approche qui convient – Pensez également à supporter la pagination.

# Quarkus Repository

- ▶ Quarkus propose bien une couche équivalente à CrudRepository ou JpaRepository de Spring à travers le pattern repository de Panache
  - ▶ Vous pouvez utiliser les interfaces PanacheRepository<T> ou PanacheRepositoryBase<T, ID> qui apportent toutes les méthodes CRUD de base pour tes entités.
- ▶ Fonctionnement
  - ▶ PanacheRepository<T> : À étendre dans une classe de repository annotée @ApplicationScoped pour chaque entité métier. Cette interface apporte toutes les opérations usuelles : find, persist, delete, list, count, etc..
  - ▶ PanacheRepositoryBase<T, ID> : À utiliser si la clé primaire n'est pas de type Long ou si vous voulez personnaliser ce type.

# Quarkus Repository – Bonnes pratiques

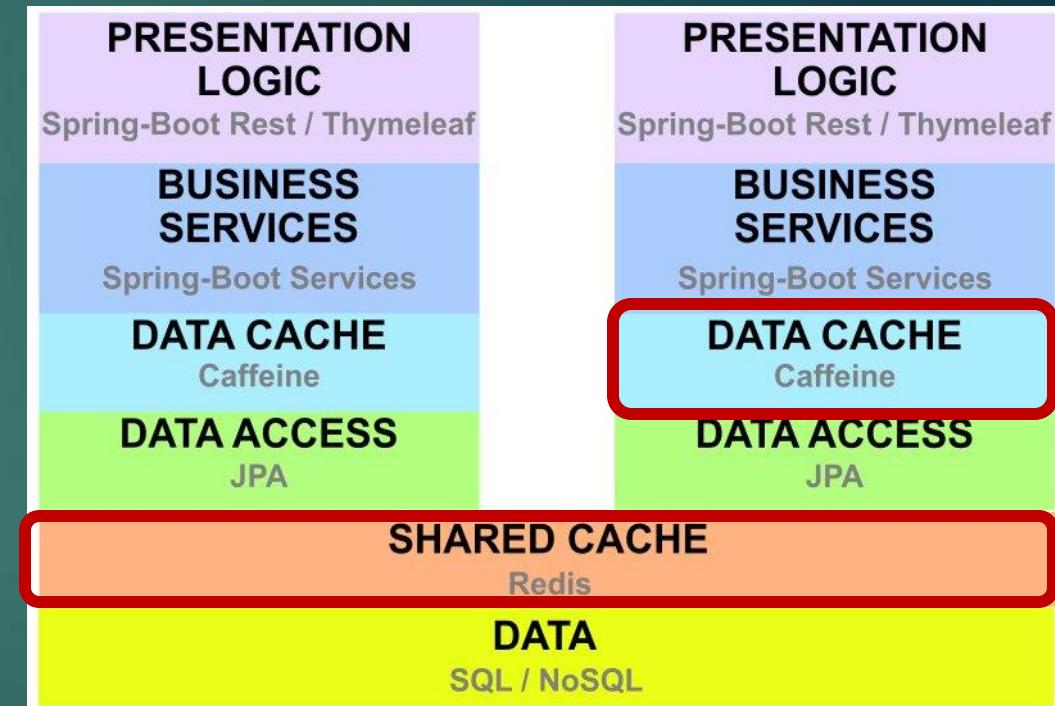
- ▶ **Pas de génération automatique** : L'analyse des noms de méthodes comme chez Spring n'est pas supportée, vous devez donc définir la requête ou la logique dans le corps de la méthode.
- ▶ Préférez des méthodes courtes, claires et typées selon le domaine métier.
- ▶ Les conventions sont guidées par la simplicité et la lecture du code : findByX pour une recherche basique, des méthodes métier plus explicites pour la complexité.

# Cache



# Cache

- ▶ Quarkus propose plusieurs options pour la mise en cache selon les besoins : **cache applicatif** par annotations, **cache second niveau** Hibernate pour les entités, et intégrations avec des **caches distribués** comme Redis ou Infinispan.
- ▶ Les caches fonctionnent sur les méthodes publiques/package-private, jamais privées.
- ▶ Attention aux problématiques liées à la **cohérence des données** : les caches doivent être invalidés lors des changements (mise à jour, suppression) pour éviter de servir des données obsolètes.
- ▶ En microservices, le choix entre cache local et cache distribué dépend du besoin de cohérence, de volume, et de tolérance à la latence.



# Cache applicatif

- ▶ Principale extension : quarkus-cache, basée sur Caffeine (local, in-memory).
- ▶ Annotations principales :
  - ▶ **@CacheResult** : met en cache le résultat d'une méthode (souvent utilisée sur des services ou endpoints REST). Le cache est indexé sur les paramètres ; configurable avec cacheName et d'autres options (timeout, clé personnalisée...).
  - ▶ **@Cache Invalidate** : invalide l'entrée de cache associée à un ou plusieurs paramètres, souvent utilisée après modification ou suppression de données.
  - ▶ **@Cache Invalidate All** : vide complètement tout le cache sélectionné.
  - ▶ **@CacheKey** : précise quels paramètres sont utilisés comme clé pour le cache, en cas de méthode multi-arguments.
- ▶ Configuration fine via application.properties, dont nom du cache, taille, stratégie....

# Cache de second niveau

- ▶ Annotation : @Cacheable sur les entités.
- ▶ Les valeurs des champs (sauf collections et relations, qui nécessitent l'annotation @org.hibernate.annotations.Cache + CacheConcurrencyStrategy).
- ▶ Les queries JPQL peuvent être rendues cacheables avec le hint Hibernate org.hibernate.cacheable.
- ▶ Permet d'éviter la répétition des accès DB pour les entités fréquemment lues.

# Cache distribué

- ▶ Redis : extension Quarkus Redis Cache, configuration avancée et expiration paramétrable, adapté cloud/scalabilité.
- ▶ Infinispan : permet le partage de cache entre plusieurs instances, configuré soit embarqué, soit en mode client distant.
- ▶ Configuration : backend de cache paramétrable (`quarkus.cache.type`), timeouts, réduction du cold-start.

# Cache - Risques

- ▶ Cohérence des données : le principal défi est d'assurer que tous les niveaux de cache s'invalident (ou se mettent à jour) correctement lors des modifications : une donnée modifiée ne doit pas rester obsolète dans un cache local, distribué ou ORM.
- ▶ Complexité : plus il y a de couches de cache, plus il faut ajouter de la logique de gestion : invalidation croisée, gestion des timeouts/TTL, restauration en cas de défaillance, synchronisation sur écriture.
- ▶ Stratégie d'invalidation : il faut prévoir des stratégies claires (événements, timeouts, write-through/back, notification/polling) pour synchroniser tous les caches lorsqu'une donnée est changée.
- ▶ Surveillance et analyse : surveiller régulièrement le taux de hit/miss, la latence, et détecter rapidement les problèmes de désynchronisation.

# Cache – Bonnes pratiques

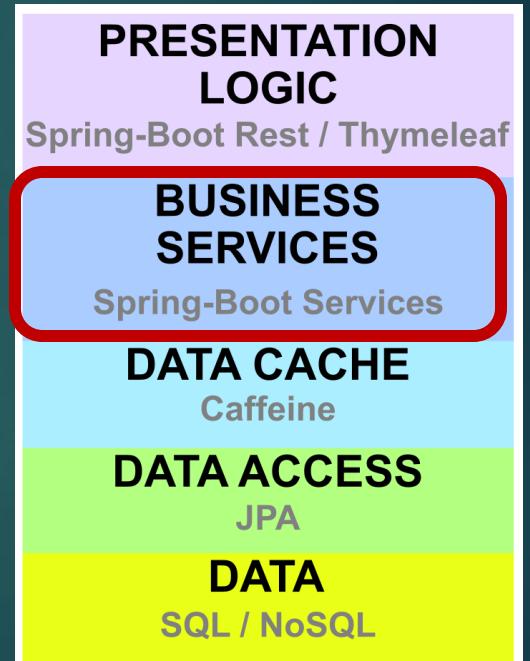
- ▶ Cache local : in-process pour ultra-rapidité (Caffeine), utile pour calculs/filtres très fréquents.
- ▶ Cache distribué : toujours utiliser pour les données partagées/fortement concurrentes, ou pour supporter l'auto-scaling (Redis/Infinispan).
- ▶ Cache ORM : pour réduire l'accès en lecture à la base, mais toujours en coordonnant invalidation avec le cache applicatif.
- ▶ Éviter la redondance sur le même axe métier : utilité de chaque niveau doit être motivée (ex : cache REST pour les réponses, cache ORM pour la persistence, distribué pour le cluster).
- ▶ Mettre en œuvre des tests et des outils de monitoring pour valider la bonne synchronisation des données.

# Services



# Couche services - Principes

- ▶ Isolation du métier :
  - ▶ La couche service encapsule toutes les règles métier, les validations complexes, et les orchestrations métier.
  - ▶ Elle ne doit pas dépendre des détails techniques de la persistence, de l'API ou des frameworks sous-jacents.
- ▶ Réutilisabilité et testabilité :
  - ▶ Les services sont conçus pour être réutilisés par plusieurs endpoints, controllers ou autres services.
  - ▶ Ils doivent être facilement testables indépendamment de la couche d'accès aux données (repository) ou de la couche de présentation.



# Couche services - Principes

- ▶ Injection de dépendances (DI) :
  - ▶ Utilise la CDI (@Inject) pour injecter repository, services externes, et outils techniques nécessaires au métier.
  - ▶ Les dépendances doivent être minimisées, de préférence injectées via le constructeur pour la facilité de test.
- ▶ Atomicité et gestion transactionnelle :
  - ▶ Toutes les opérations qui modifient l'état métier (création, suppression, update) sont annotées avec @Transactional pour garantir la cohérence et le rollback en cas d'exception.

# Couche services - Principes

- ▶ Règles métier centralisées :
  - ▶ Toutes les validations complexes (multichamps, workflows, triggers spécifiques) sont traitées dans le service, ni dans le contrôleur ou les entités.
  - ▶ Les erreurs métier sont systématiquement remontées sous forme d'exceptions ou de résultats typés.
- ▶ Découplage API/DTO :
  - ▶ La couche service dialogue avec ses clients via des DTOs ou des objets métier purs : elle ne doit pas exposer les entités ORM, ni dépendre du format des endpoints REST ou GraphQL.

# Couche services - Principes

- ▶ Organisation par domaine
  - ▶ Un service correspond à un domaine fonctionnel précis ("UserService", "ReviewService", "CatalogService"), pour limiter la taille et maintenir la cohérence du métier.
  - ▶ Les utilitaires transverses (conversion, mapping, vérification) sont isolés dans des services techniques dédiés si nécessaire.
- ▶ Gestion des erreurs métier
  - ▶ Les services remontent des exceptions ou codes d'erreur explicitement typés, facilitant la gestion des cas d'erreur côté API.

# Couche services – Bonnes pratiques

- ▶ La bonne pratique pour déclarer un service métier dans Quarkus est d'utiliser **@ApplicationScoped** provenant de CDI/Jakarta ; cette annotation garantit que la classe est traitée comme un bean singleton, injectable partout dans l'application.
  - **@ApplicationScoped** : à placer sur toutes tes classes métier (service, business logic) pour garantir que leur instance est unique et gérée par le conteneur CDI.
  - **@Transactional** : ajouter sur les méthodes ou la classe si le service réalise des opérations d'écriture/modification sur la base de données afin d'assurer la cohérence transactionnelle.

# Couche services - Exemple

```
@ApplicationScoped
public class ReviewService {

    @Inject
    ReviewRepository reviewRepository;

    @Transactional
    public ReviewResponse createReview(@Valid CreateReviewRequest request, String createdBy) {
        validateCreateRequest(request);
        ...
        return ReviewResponse.from(review);
    }
}
```

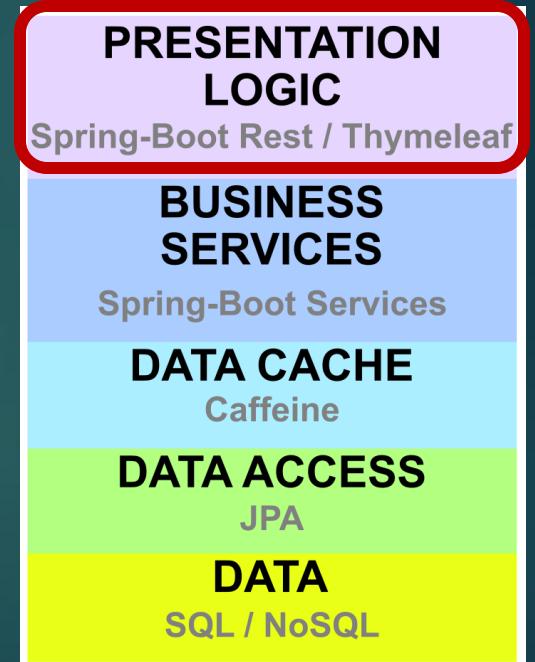
**Exercice 5 :** Implémenter les couches services de chaque micro-services  
Vous devrez y intégrer toute la logique métier, la couche DTO et sa validation  
Pensez à bien définir ce qui est de l'ordre du transactionnel

**Exercice 6 :** Veillez à implémenter les différents niveaux de cache pour garantir une bon niveau de performance

# Présentation

# Couche présentation / REST

- ▶ La couche "Contrôleurs REST" dans Quarkus repose sur l'API JAX-RS (Jakarta REST), enrichie par le framework pour offrir une gestion moderne, performante et flexible des endpoints HTTP.
- ▶ Quarkus offre un certain nombre d'avantages:
  - Structure RESTful intuitive, support complet pour les standards HTTP et conventions REST.
  - Injection CDI, validation automatique.
  - Gestion fine des erreurs métier et techniques.
  - Documentation et testabilité via OpenAPI.
  - Support réactif pour haute performance et IO intensive.



# Couche présentation / REST

## Bases

- ▶ Classe Java annotée avec @Path pour définir l'URI d'accès (ex : /users).
- ▶ Chaque méthode exposée comme endpoint utilise un annotateur HTTP : @GET, @POST, @PUT, @DELETE, @PATCH, etc.
- ▶ Les paramètres de requête, d'URL et de header sont injectés avec : @PathParam, @QueryParam, @HeaderParam.
- ▶ Négociation de contenu via :
  - ▶ @Produces (type MIME retourné, ex: application/json)
  - ▶ @Consumes (type MIME accepté, ex: application/json)

# Couche présentation / REST

## Aspects transverses

- ▶ Gestion des entrées/sorties
  - Les payloads sont automatiquement convertis entre JSON et objets Java grâce à Jakarta JSON-B ou Jackson selon configuration.
  - Les validations Bean/Jakarta sont exécutées automatiquement sur les DTO injectés.
- ▶ Injection et découplage: La resource peut injecter services métiers, repositories ou clients externes via CDI (@Inject dans la resource REST).
- ▶ Support de la gestion d'erreur: Utilisation des ExceptionMappers pour personnaliser les réponses d'erreur, globales ou spécifiques.

# Couche présentation / REST

## Aspects transverses

- ▶ Asynchrone et réactif
  - Support natif des méthodes Uni<T> (Mutiny) ou CompletionStage<T> pour la gestion d'API non bloquante.
  - Les endpoints REST réactifs permettent d'améliorer la scalabilité et la gestion de IO.
- ▶ Documentation OpenAPI
  - Les endpoints sont automatiquement documentés et exposés via une interface Swagger/OpenAPI, grâce à Quarkus MicroProfile OpenAPI (/q/openapi, /q/swagger-ui en dev).

# Couche présentation / REST

## Passage de paramètres

Type de paramètre	Annotation Quarkus	Exemple dans la méthode
Chemin (Path)	@PathParam	getUser(@PathParam("id") Long id)
Requête (Query)	@QueryParam	search(@QueryParam("kw") String kw)
Corps (Body)	Aucun, type objet Java	addUser(User user)
En-tête HTTP	@HeaderParam	process(@HeaderParam("Auth") String a)
Cookie	@CookieParam	stats(@CookieParam("level") String lvl)
Formulaire	@FormParam	submit(@FormParam("email") String email)
Matriciel (Matrix)	@MatrixParam	getCheese(@MatrixParam("type") String t)

# Couche présentation / REST

## Exemple

```
@Path("/resources")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class ResourceResource {

    @Inject
    ResourceService resourceService;

    @GET
    @Path("/{id}")
    public ResourceResponse getResource(
        @Parameter(description = "Resource ID", required = true, example = "1")
        @PathParam("id") Long id) {
        return resourceService.getResourceById(id);
    }

    @DELETE
    @Path("/{id}")
    public Response deleteResource(
        @Parameter(description = "Resource ID", required = true)
        @PathParam("id") Long id) {
        resourceService.deleteResource(id);
        return Response.noContent().build();
    }
}
```

# Gestion des exceptions

- ▶ La gestion des exceptions dans Quarkus repose sur des principes de robustesse, de clarté des réponses API, et d'extensibilité, quel que soit le type d'application (REST, service ou batch).
- ▶ Elle associe des conventions Java, du Jakarta REST, et quelques bonnes pratiques spécifiques au framework. La gestion des erreurs dans Quarkus repose sur des principes de robustesse, de clarté des réponses API, et d'extensibilité, quel que soit le type d'application (REST, service ou batch).
- ▶ Quarkus propose de multiples options pour la gestion d'exception, allant de la propagation naturelle à la personnalisation fine des réponses via ExceptionMapper, jusqu'à l'adoption de standards modernes (RFC 7807/9457) et une gestion fine de la validation automatique, tout en assurant la sécurité et la lisibilité des réponses APIs.

# Gestion des exceptions

- ▶ Exceptions standards et propagation
  - Les erreurs inattendues (NullPointerException, SQLException...) ou métier (ex: ResourceNotFoundException) sont propagées naturellement.
  - Quarkus gère automatiquement la conversion de certaines exceptions communes (404, 400, etc.) en réponses HTTP associées lorsqu'on expose une API REST.
- ▶ **ExceptionMapper : personnalisation centralisée**
  - Mise en place possible de un ou plusieurs ExceptionMappers (@Provider + implements ExceptionMapper<T>) pour attraper les exceptions métier ou techniques et retourner des réponses structurées, typées ou localisées.
  - Cela permet de transformer toutes les exceptions d'un type donné en réponse HTTP homogène,

# Gestion des exceptions

- ▶ Spécifications modernes : RFC 7807 & RFC 9457
  - Utilisation possible de l'extension **Quarkus Problem** pour générer des réponses d'erreur JSON standard "Problem Details" (RFC 7807 ou RFC 9457) : cela donne des erreurs interopérables, standardisées, faciles à traiter côté client.
- ▶ Gestion validation Bean/Jakarta
  - Lors des erreurs de validation (annotations jakarta.validation), Quarkus retourne automatiquement un code 400 et un JSON listant les violations.
  - Il est possible personnaliser l'aspect et le format des erreurs de validation via un ExceptionMapper sur ConstraintViolationException.

# Gestion des exceptions

- ▶ Personnalisation d'erreur globale
  - ▶ Pour définir une page d'erreur ou un format de réponse personnalisé pour tout le projet :
    - Implémenter un ExceptionMapper global
    - Ou utiliser les outils de Quarkus pour page d'erreur HTML/JSON adaptée selon l'Accept header du client.
- ▶ Erreurs réactives & Mutiny
  - ▶ Avec l'utilisation de Mutiny, les exceptions sont gérées dans les pipes avec .onFailure().recoverWithItem() ou .onFailure().invoke(...)
  - ▶ Il est possible de mapper les exceptions à des codes précis REST.

# Exceptions: Bonnes pratiques

- ▶ Ne jamais exposer de stacktrace, ni de détails d'implémentation, dans les erreurs retournées en production.
- ▶ Journaliser les erreurs serveur (500) avec le plus de contexte possible, en ignorant celles prévues de type 4xx.
- ▶ Structurer toujours les erreurs projetées (JSON, RFC 7807/9457) pour que les clients puissent les traiter automatiquement.
- ▶ Centraliser le mapping des exceptions pour garantir cohérence, sécurité, audit et internationalisation.
- ▶ Penser à la gestion spécialement pour le mode cluster ou microservices : préférence à des codes et formats interopérables pour la communication service-to-service.

# Gestion des exceptions

## Exemple

```
@Provider
public class ResourceNotFoundExceptionMapper implements ExceptionMapper<ResourceNotFoundException> {

    @Override
    public Response toResponse(ResourceNotFoundException exception) {
        Long resourceId = exception.getResourceId();

        ErrorResponse error = ErrorResponse.of(
            exception.getMessage(),
            "RESOURCE_NOT_FOUND",
            Map.of("resourceId", resourceId)
        );

        return Response.status(Response.Status.NOT_FOUND) // HTTP 404
            .entity(error)
            .build();
    }
}
```

# Documentation

- ▶ Quarkus s'appuie sur les annotations de MicroProfile OpenAPI pour enrichir la documentation générée de tes APIs REST ; elles permettent de décrire des informations globales sur l'API comme sur chaque endpoint ou schéma.
- ▶ Principales annotations:
  - **@OpenAPIDefinition**  
Pour spécifier les informations générales de l'API (titre, version, contact, tags, description globale, licence...).
  - **@Tag** : Pour regrouper et décrire les endpoints par fonctionnalités (ex : "user", "admin").
  - **@Operation** : Pour documenter/préciser un endpoint : son nom d'opération, une description, un résumé, les codes de retours attendus, et les paramètres
- ▶ Pour tester: [http://{{base\\_url}}:{{port}}/q/swagger-ui](http://{{base_url}}:{{port}}/q/swagger-ui)

# Documentation Exemple

```
@GET  
@Operation(  
    summary = "List all resources",  
    description = "Retrieves a paginated list of all multimedia resources in the catalog"  
)  
@APIResponses({  
    @APIResponse(  
        responseCode = "200",  
        description = "List of resources (may be empty)",  
        content = @Content(  
            mediaType = MediaType.APPLICATION_JSON,  
            schema = @Schema(implementation = ResourceResponse.class),  
            examples = @ExampleObject(  
                name = "List of resources",  
                value = """""  
                [  
                    {  
                        "id": 1,  
                        "title": "1984",  
                        "type": "BOOK",  
                        "status": "AVAILABLE"  
                    },  
                    ...  
                ]  
            )  
        )  
    })  
})  
public List<ResourceResponse> getAllResources(...)
```

# Exercice 7: API REST+ Documentation

- ▶ Sur la base des services, il vous est demandé d'implanter les contrôleurs REST de chaque micro-service
- ▶ Vous devrez prendre en considération les éléments suivants:
  - Gestion fine et centralisée des erreurs
  - Documentation
- ▶ Rendez-vous ensuite sur la documentation afin de tester vos API

# Présentation : Qute

- ▶ Qute est le moteur de template natif de Quarkus, spécialement conçu pour produire du HTML dynamique, des e-mails, ou tout contenu textuel côté serveur dans des applications Java modernes.
- ▶ Points forts
  - Parfaitement adapté à la création de vues HTML server-side modernes.
  - Typé, rapide, évolutif et compatible cloud natif.
  - Permet la génération d'e-mail, la production de rapports, et même d'autres types de texte.
- ▶ Qute est donc à la fois moderne et performant, et constitue le choix naturel pour le templating serveur dans Quarkus, avec une expérience de développement très fluide et sûre.
- ▶ Alternatives: Freemarker, Thymeleaf, JSF

# Qute: Fonctionnalités

- ▶ Qute permet de générer dynamiquement des pages HTML côté serveur à partir de templates, de partials, de snippets et de model data.
- ▶ Il est rapide, type-safe, orienté microservices/cloud, fonctionne en mode natif (GraalVM) et s'intègre parfaitement à la stack Quarkus.
- ▶ Les templates .html sont stockés dans src/main/resources/templates et consommés facilement via des contrôleurs annotés @CheckedTemplate ou par injection de TemplateInstance.
- ▶ Syntaxe proche de Thymeleaf, Handlebars ou Mustache, avec validation à la compilation et auto-complétion (plugins IDE).

# Qute : Principales constructions

- ▶ Les principales constructions du langage Qute sont basées sur une syntaxe Java-like orientée templates et conçue pour faciliter l'édition côté serveur, en rendant vos fichiers réactifs et très proches du modèle de programmation Java.
- ▶ Les plus utilisées incluent l'inclusion de variables, les sections conditionnelles, les boucles et les sections de bloc avec gestion fine des données.
- ▶ Qute offre des blocs de contrôle puissants (`{#if}`, `{#for}`, `{#include}`, sections nommées), l'interpolation Java-like, ainsi que la validation très stricte typée à la compilation, ce qui le distingue nettement d'autres moteurs de template classiques.
- ▶ <https://quarkus.io/guides/qute>

# Qute : Exemple

```
@GET  
@Path("/{id}")  
public Object view(  
    @PathParam("id") Long id,  
    @QueryParam("message") String message,  
    @QueryParam("error") String error  
) {  
    try {  
        ResourceResponse resource = resourceService.getResourceById(id);  
        return Templates.view(resource, message, error);  
    } catch (ResourceNotFoundException e) {  
        return Response.seeOther(  
            URI  
                .create("/web/catalog?error=" + encodeMessage("Resource not found: " + id)))  
                .build();  
    }  
}
```

# Qute

```
<span class="badge fs-6">
  {#switch resource.status}
    {#case 'AVAILABLE'}bg-success{/case}
    {#case 'RESERVED'}bg-warning text-dark{/case}
    {#case 'UNAVAILABLE'}bg-secondary{/case}
 {/switch}>
  <i class="fas fa-circle">{resource.status}</i>
</span>

{#if resource.archived}
<span class="badge bg-danger fs-6">
  <i class="fas fa-archive"></i> ARCHIVED
</span>
{/if}
```

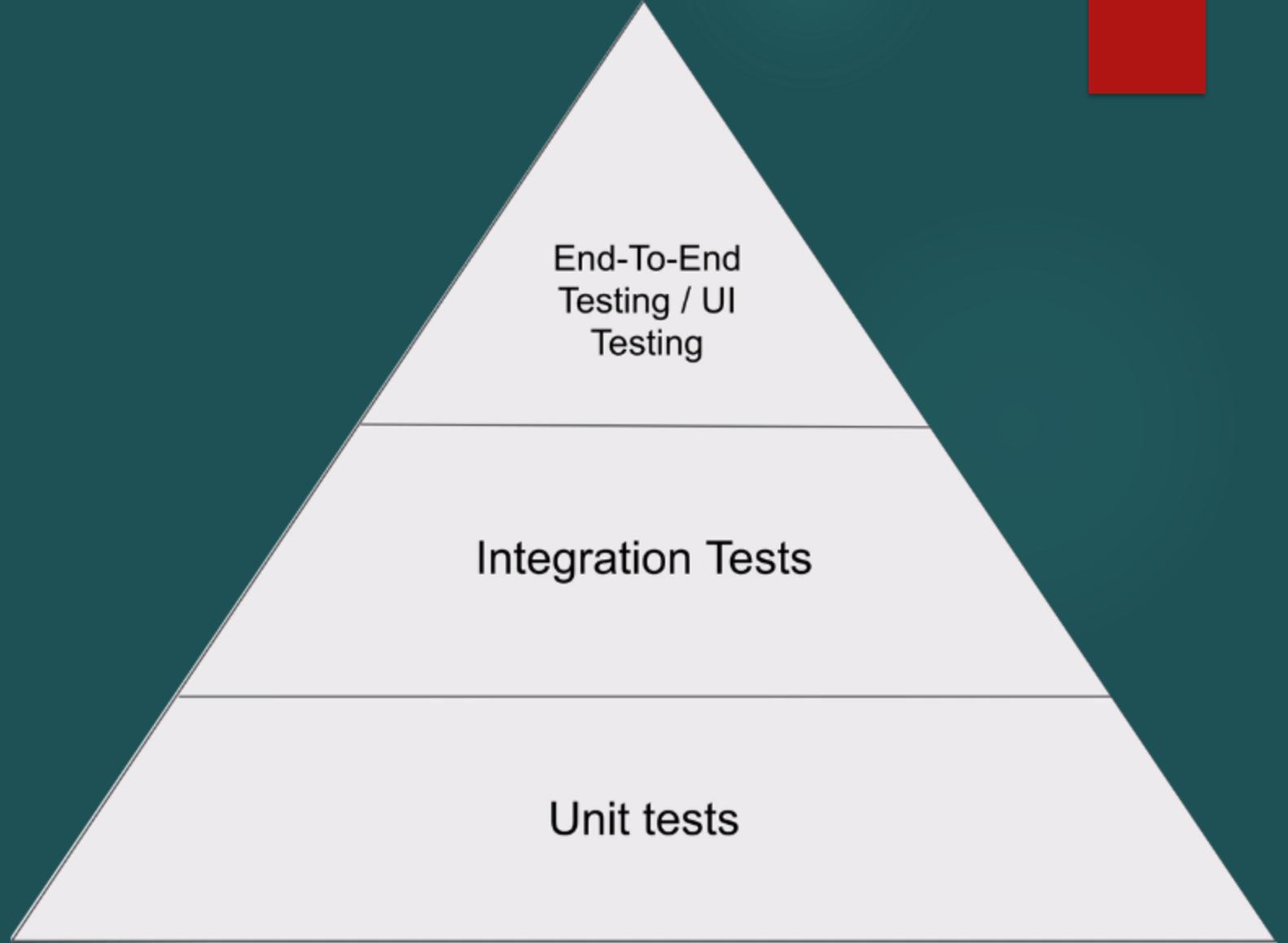
# Exercice 8 : Qute

- ▶ Sur la base de l'exemple fourni, vous devrez implémenter les différentes vues pour le micro-services reviews
- ▶ Questions:
  - Quels problèmes posent cette approche ?
  - Quelle solution mettre en œuvre pour les résoudre ?

# Tests



# Quoi tester ?



# Test unitaire - Définition

- ▶ Les tests unitaires sont une méthode de test de logiciels dans laquelle des unités ou des composants individuels d'un logiciel sont testés isolément du reste du code afin de déterminer s'ils fonctionnent conformément à la conception.
- ▶ Une unité est la plus petite partie testable d'un logiciel, et elle a généralement une ou quelques entrées et une seule sortie.
- ▶ Dans la programmation orientée objet, la plus petite unité est une méthode, qui peut appartenir à une classe de base/supérieure, une classe abstraite ou une classe dérivée/enfant. L'objectif des tests unitaires est de valider que chaque unité du logiciel fonctionne comme prévu.

# Test unitaire - Caractéristiques

- ▶ **Isolation** : Les unités sont testées indépendamment les unes des autres afin d'identifier les problèmes. Les mocks et les stubs sont souvent utilisés pour isoler l'unité testée de ses dépendances.
- ▶ **Faible portée** : Se concentre sur la vérification du comportement des unités individuelles, et non sur l'interaction entre elles.
- ▶ **Exécution rapide** : Les tests unitaires doivent être rapides à exécuter, ce qui facilite leur exécution fréquente au cours du développement.
- ▶ **Automatisés** : Les tests unitaires sont généralement automatisés et intégrés dans un processus de construction.

# Test unitaire – Principes FIRST

- ▶ **Fast** : Les tests doivent être exécutés rapidement. Les tests lents découragent une exécution fréquente.
- ▶ **Independent** : Les tests ne doivent pas dépendre les uns des autres. Le résultat d'un test ne doit pas influencer celui d'un autre. Cela permet d'éviter les échecs en cascade et facilite l'identification des problèmes.
- ▶ **Repeatable** : Les tests doivent produire le même résultat à chaque fois qu'ils sont exécutés, quel que soit l'environnement. Les dépendances externes ou les données aléatoires peuvent conduire à des résultats incohérents.
- ▶ **Self-Validated** : Les tests doivent automatiquement indiquer s'ils ont réussi ou échoué sans inspection manuelle. Cela permet une intégration facile dans les processus de construction automatisés.
- ▶ **Thorough & Timely** : Les tests doivent couvrir une partie significative de la base de code et être écrits au bon moment - idéalement avant ou en même temps que le code de production (TDD). Attendre plus tard rend plus difficile l'écriture de tests efficaces et augmente la probabilité d'apparition de bogues.

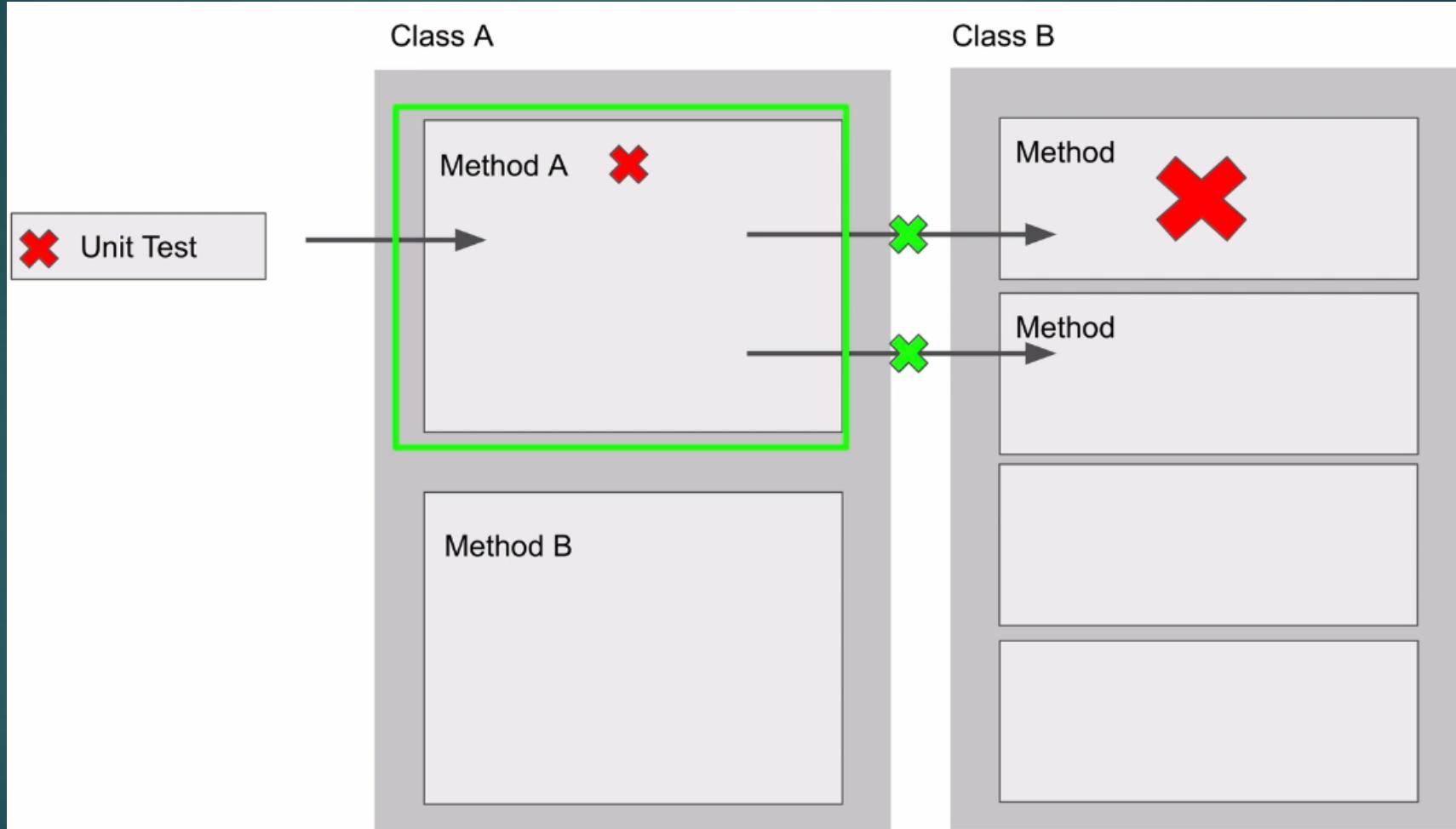
# Test unitaire – Défis liés aux dépendances

- ▶ Lorsqu'une classe (ClasseA) dépend d'une autre classe (ClasseB), plusieurs problèmes se posent lors des tests unitaires :
  - **Échecs en cascade** : Si un bogue existe dans ClassB, les tests de ClassA qui en dépendent peuvent également échouer, même si la logique de ClassA est correcte. Il est alors plus difficile d'identifier la source du problème. Cela peut conduire à une longue chaîne de débogage retracant l'échec à travers plusieurs classes.
  - **Lenteur de l'exécution des tests** : Si ClassB a une initialisation complexe ou interagit avec des ressources externes (bases de données, réseau, etc.), les tests impliquant ClassA seront lents en raison de cette dépendance.
  - **Tests fragiles** : Les changements dans l'implémentation de ClassB peuvent casser les tests de ClassA, même si le comportement de ClassA reste inchangé. Cela constitue une violation du principe d'indépendance des tests.

# Test unitaire – Défis liés aux dépendances

- ▶ Lorsqu'une classe (ClasseA) dépend d'une autre classe (ClasseB), plusieurs problèmes se posent lors des tests unitaires :
  - **Difficulté de mise en place/nettoyage:** La mise en place des conditions préalables nécessaires pour tester ClassA peut impliquer une initialisation complexe de ClassB et de ses dépendances, ce qui conduit à un code de test fragile et difficile à maintenir. De même, les opérations de nettoyage peuvent s'avérer complexes.
  - **Testabilité réduite :** Certaines dépendances, comme les connexions aux bases de données ou les services de réseau, peuvent être indisponibles ou peu pratiques à utiliser dans un environnement de test, ce qui rend difficile le test complet de ClassA.

# Test unitaire – Défis liés aux dépendances



# Test unitaire – Gestion des dépendances

- ▶ **Mocking** : Les mocks sont utilisés pour vérifier les *interactions* entre les objets. Ils permettent de vérifier si une méthode sur une dépendance a été appelée avec les arguments attendus, le bon nombre de fois et dans le bon ordre.
- ▶ **Stubbing** : Les stubs sont utilisés pour fournir des réponses automatiques aux appels de méthodes sur les dépendances. Ils se concentrent sur le contrôle du *comportement* de la dépendance, et non sur la vérification des interactions.
- ▶ **Autres approches :**
  - **Objets factices (Dummy objects)** : Ils sont passés en paramètre mais sont utilisés pour satisfaire les signatures de méthodes.
  - **Faux objets (Fake objects)**: Implémentations fonctionnelles mais inadaptées à la production (par exemple, une base de données en mémoire pour les tests).
  - **Espions (Spies)**: Combinaison de mock et de stub. Ils peuvent être utilisés pour vérifier les interactions (comme les mocks) et fournir des réponses stubbées (comme les stubs).

# Test unitaire - Structure

- ▶ Le modèle « Arrange, Act, Assert » (AAA) est une structure largement adoptée pour écrire des tests unitaires clairs et concis. Il sépare le test en trois parties distinctes :
  - **Organiser** : Mettre en place les conditions préalables nécessaires au test. Cela inclut la création d'objets, l'initialisation des données et la mise en place de mocks ou de stubs pour les dépendances. Essentiellement, vous préparez la scène pour l'action que vous voulez tester.
  - **Agir** : Exécuter la méthode ou le morceau de code testé. Il s'agit généralement d'une seule ligne de code qui invoque la méthode que vous testez. C'est l'action que vous évaluez.
  - **Assert** : Vérifier le résultat de l'action par rapport au comportement attendu. Cela implique l'utilisation d'assertions pour vérifier l'état des objets, les valeurs de retour ou tout effet secondaire. Cela permet de confirmer que l'action a produit le résultat souhaité.

# Test unitaire - Structure

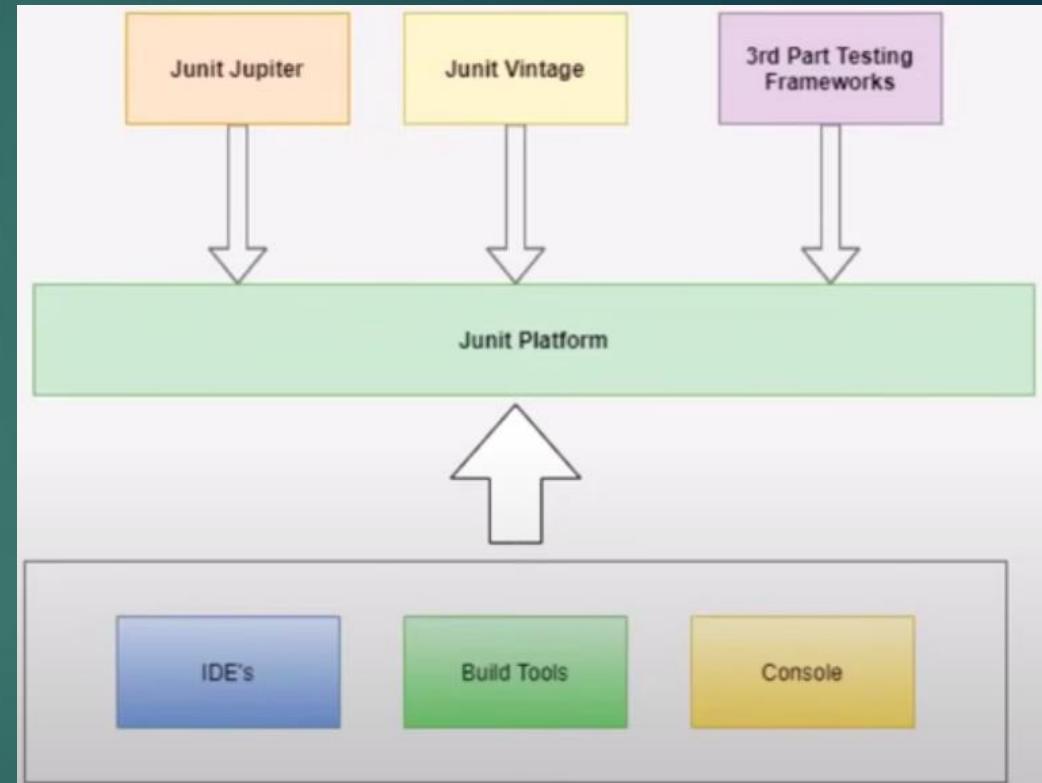
```
@Test void testCalculateArea() {  
    // Arrange  
    Circle circle = new Circle(5);  
    // Act  
    double area = circle.calculateArea();  
    // Assert  
    assertEquals(78.54, area, 0.01);  
}
```

# Test unitaire – Outils

- ▶ **JUnit** : Le cadre de test le plus utilisé en Java. Il fournit des annotations pour l'écriture des tests, des assertions pour la vérification des résultats attendus et des exécutants de tests pour l'exécution des tests. JUnit 5 est la version majeure actuelle.
- ▶ **Mockito** : Un framework de mocking qui simplifie la création de doubles de tests (mocks, stubs, spies, etc.). Il permet d'isoler l'unité testée en contrôlant le comportement de ses dépendances. Il est très flexible et s'intègre bien avec JUnit.
- ▶ **AssertJ** : Une bibliothèque d'assertions fluide qui fournit des assertions plus lisibles et plus expressives que les assertions intégrées de JUnit. Elle améliore la clarté du code de test et facilite le diagnostic des échecs.
- ▶ **TestNG** : Un autre cadre de test populaire qui offre des fonctionnalités avancées que l'on ne trouve pas dans JUnit, comme les tests paramétrés, les tests de dépendance et l'exécution parallèle des tests.
- ▶ **Hamcrest** : Une bibliothèque de correspondance qui fournit un ensemble riche de correspondance pour écrire des assertions plus flexibles et expressives. Souvent utilisée en conjonction avec JUnit ou TestNG. Cependant, les assertions fluentes d'AssertJ rendent souvent Hamcrest moins nécessaire.
- ▶ **PowerMock** : Un cadre pour simuler les méthodes statiques, les constructeurs, les classes finales et les méthodes privées. À utiliser avec précaution, car la simulation de ces éléments peut révéler des problèmes de conception. Une utilisation excessive peut conduire à des tests fragiles étroitement liés aux détails de l'implémentation.

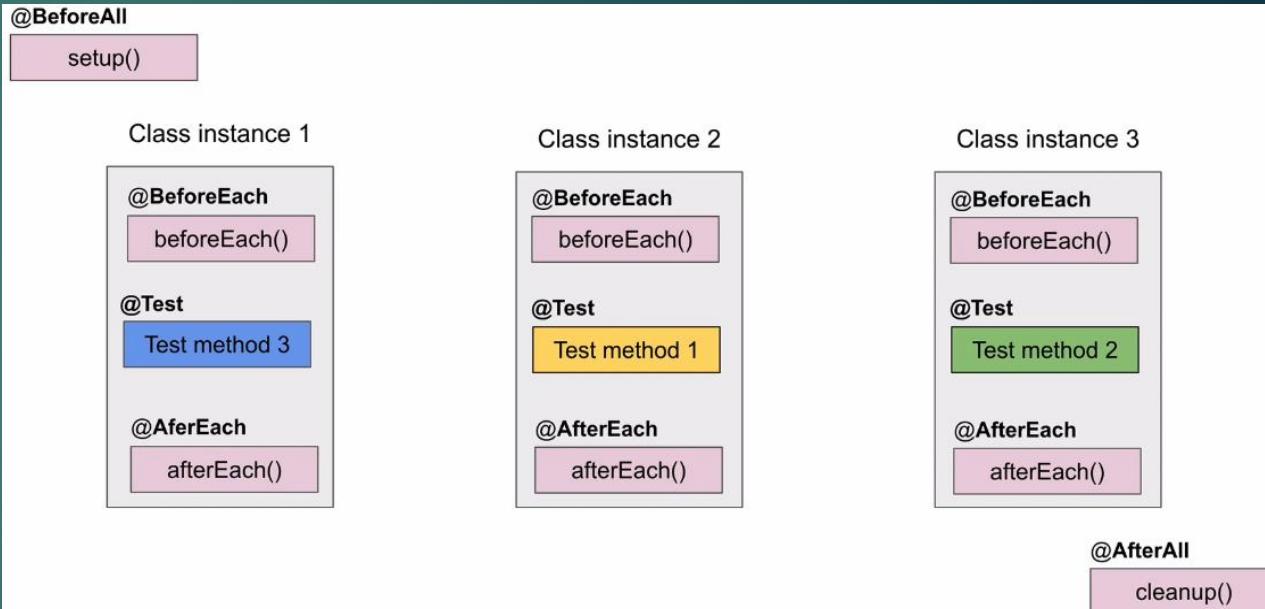
# JUnit

- ▶ Cadre pour spécifier et exécuter des tests reproductibles
- ▶ Permet de tester des classes ou des méthodes (SUT: Subject Under Test)
- ▶ Constitué de 4 composants:
  - ▶ Platform: Interface pour lancer des tests depuis un client
  - ▶ Jupiter: Fournit les constructions pour spécifier des tests et des extensions
  - ▶ Vintage: Moteur de test supportant une compatibilité ascendante
  - ▶ 3rd Party: Permet de créer son propre cadre de tests en réutilisant les briques de JUnit



# JUnit

- ▶ Nouveauté de la version 5:
  - ▶ Les tests imbriqués
  - ▶ Les tests dynamiques
  - ▶ Les tests paramétrés qui offrent différentes sources de données
  - ▶ Un nouveau modèle d'extension
  - ▶ L'injection d'instances en paramètres des méthodes de tests
- ▶ Cycle de vie
  - ▶ **BeforeAll**: appelée à l'instanciation de chaque classe de test
  - ▶ **BeforeEach**: appelée avant l'exécution de chaque test
  - ▶ **Test**: exécution du test
  - ▶ **AfterEach**: appelée après l'exécution de chaque test
  - ▶ **AfterAll**: appelée quand la classe est déchargée



# JUnit - Annotations

Annotation	Rôle
@Test	La méthode annotée est un cas de test. Cette annotation ne possède aucun attribut
@ParameterizedTest	La méthode annotée est un cas de test paramétré
@RepeatedTest	La méthode annotée est un cas de test répété
@TestFactory	La méthode annotée est une fabrique pour des tests dynamiques
@TestInstance	Configure le cycle de vie des instances de tests
@TestTemplate	La méthode est un modèle pour des cas de tests à exécution multiple
@DisplayName	Définit un libellé pour la classe ou la méthode de test annotée

[Source: <https://www.jmdoudoux.fr/java/dej/chap-junit5.htm>]

# JUnit- Annotations

Annotation	Rôle
@BeforeEach	La méthode annotée sera invoquée avant l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @Before de JUnit 4
@AfterEach	La méthode annotée sera invoquée après l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @After de JUnit 4
@BeforeAll	La méthode annotée sera invoquée avant l'exécution de la première méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @BeforeClass de JUnit 4. La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class
@AfterAll	La méthode annotée sera invoquée après l'exécution de toutes les méthodes de la classe annotées avec @Test, @RepeatedTest, @ParameterizedTest et @Testfactory. Cette annotation est équivalente à @AfterClass de JUnit 4. La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class.
@Nested	Indique que la classe annotée correspond à un test imbriqué
@Tag	Définit une balise sur une classe ou une méthode qui permettra de filtrer les tests exécutés. Cette annotation est équivalente aux Categories de JUnit 4 ou aux groups de TestNG
@Disabled	Désactive les tests de la classe ou la méthode annotée. Cette annotation est similaire à @Ignore de JUnit 4
@ExtendWith	Enregistre une extension

# JUnit - Assertions

Egalité	Nullité	Exceptions
<code>assertEquals()</code>	<code>assertNull()</code>	<code>assertThrows()</code>
<code>assertNotEquals()</code>	<code>assertNotNull()</code>	
<code>assertTrue()</code>		
<code>assertFalse()</code>		
<code>assertSame()</code>		
<code>assertNotSame()</code>		

<https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html>

# Test unitaire - Exemples

```
interface Shape { double getArea(); }
class Circle implements Shape
{
    // ... implementation ...
}
class Square implements Shape {
    // ... implementation ...
}
```

```
public class ShapeTest {
    @Test
    public void testAreaCalculation() {
        Shape circle = new Circle(5);
        assertEquals(78.54, circle.getArea(), 0.01);
        Shape square = new Square(5);
        assertEquals(25, square.getArea(), 0.01);
    }
}
```

# Test unitaire - Exemples

```
@ParameterizedTest
@MethodSource("pgcdTestData")
@DisplayName("Test PGCD")
void testPGCD(int a, int b, int expected) {
    int pgcd = calculator.pgcd(a, b);
    assertEquals(expected, pgcd, () -> "PGCD de " + a + " et " + b + " doit être " + expected);
}

private static Stream<Arguments> pgcdTestData() {
    return Stream.of(
        Arguments.of(12, 18, 6),
        Arguments.of(25, 15, 5),
        Arguments.of(42, 36, 6)
    );
}
```

# JUnit - Test de performance

```
@Test
void checkTimeout() {
    Assertions.assertTimeout(
        Duration.ofMillis(200), () -> { return ""; }
    );
}
```

# Mockito

- ▶ Mockito est un cadre de simulation qui vous permet de créer et de configurer des objets simulés.
  - Il est particulièrement utile lorsque vous devez tester un composant qui interagit avec d'autres composants complexes.
- ▶ En utilisant des objets fictifs, vous pouvez simuler le comportement de ces dépendances sans avoir à les instancier.
  - Cela est particulièrement utile dans les tests unitaires, lorsque vous souhaitez isoler le composant testé.
- ▶ JUnit et Mockito fonctionnent bien ensemble, ce qui vous permet d'écrire des tests complets qui couvrent à la fois la logique des composants individuels et leurs interactions avec les dépendances.
  - Mockito s'intègre parfaitement à JUnit et fournit des annotations telles que @Mock pour créer des objets fictifs et @InjectMocks pour les injecter automatiquement dans la classe testée.

# Mockito - Exemple

```
public class ServiceTest {  
  
    @Test  
    public void testServiceMethod() {  
        // Create a mock object of the repository  
        Repository mockRepository = mock(Repository.class);  
  
        // Configure the mock to return a specific value when a method is called  
        when(mockRepository.getData()).thenReturn("Expected Result");  
  
        // Create an instance of the service, injecting the mock repository  
        Service service = new Service(mockRepository);  
  
        // Call the method under test  
        String result = service.performAction();  
  
        // Verify the result  
        assertEquals("Expected Result", result);  
  
        // Verify interactions with the mock  
        verify(mockRepository).getData();  
    }  
}
```

# Test d'intégration - Définition

- ▶ Le test d'intégration est la phase du test de logiciels au cours de laquelle des modules logiciels individuels sont combinés et testés en tant que groupe.
- ▶ Il suit le test d'unité et précède le test du logiciel.
- ▶ L'objectif des tests d'intégration est de vérifier les exigences fonctionnelles, de performance et de fiabilité imposées aux principaux éléments de conception.
- ▶ Il met en évidence les défauts dans l'interaction entre les unités intégrées.

# Test d'intégration - Caractéristiques

- ▶ **Objectif** : vise à identifier les problèmes liés à la communication des données, à la compatibilité des interfaces et à la fonctionnalité combinée des unités intégrées.
- ▶ **Portée** : Se concentre sur l'interaction entre les unités, contrairement aux tests unitaires qui testent les unités de manière isolée. Il peut s'agir de tester les interactions au sein d'une même application ou entre plusieurs applications.
- ▶ **Complexité** : Plus complexe que les tests unitaires, car il s'agit de composants multiples et de leurs interactions.
- ▶ **Techniques** : Il existe différentes approches de test d'intégration, telles que le Big Bang, le Top-Down, le Bottom-Up et l'intégration Sandwich.
- ▶ **Dépendances** : Traite des dépendances réelles (bases de données, services externes, etc.) plutôt que des mocks ou des stubs utilisés dans les tests unitaires. Cela rend les tests d'intégration plus lents mais fournit des scénarios de test plus réalistes.

# Test d'intégration - Méthodologies

## ► Intégration du Big Bang :

- **Approche** : Tous les modules sont intégrés et testés en tant que système complet en une seule fois.
- **Avantages** : Simple à mettre en œuvre, en particulier pour les petits systèmes.
- **Inconvénients** : Il est difficile d'isoler les défauts. Si un problème survient, il est difficile d'identifier le module qui en est la cause. Risque élevé de découvrir de nombreux défauts à un stade avancé du processus de test. Ne convient pas aux systèmes complexes et de grande taille.

## ► Intégration descendante :

- **Approche** : Les tests commencent par le module de niveau supérieur et intègrent progressivement les modules de niveau inférieur. Les stubs sont utilisés pour simuler le comportement des modules de niveau inférieur qui n'ont pas encore été intégrés.
- **Avantages** : Il est plus facile de localiser les défauts au fur et à mesure de l'intégration. Test précoce des fonctionnalités de base.
- **Inconvénients** : Il peut être difficile de créer des stubs réalistes pour des modules complexes de niveau inférieur. Les modules de niveau inférieur sont testés de manière moins approfondie.

# Test d'intégration - Méthodologies

## ► Intégration ascendante :

- **Approche** : Les tests commencent par les modules de niveau inférieur et intègrent progressivement les modules de niveau supérieur. Les pilotes sont utilisés pour simuler le comportement des modules de niveau supérieur qui n'ont pas encore été intégrés.
- **Avantages** : Tests approfondis des modules de niveau inférieur. Il est plus facile de créer des pilotes que des stubs.
- **Inconvénients** : Des défauts de conception majeurs au niveau supérieur peuvent être détectés tardivement. Il est difficile d'observer le comportement du système tant que la plupart des modules ne sont pas intégrés.

## ► Intégration sandwich/mixte :

- **Approche** : Combine l'intégration descendante et ascendante. Le système est divisé en couches, et l'intégration descendante est réalisée sur les couches supérieures tandis que l'intégration ascendante est réalisée sur les couches inférieures. Elles se rejoignent au milieu.
- **Avantages** : Avantages de l'intégration descendante et de l'intégration ascendante. Peut être efficace pour les grands systèmes.
- **Inconvénients** : Nécessite une planification et une coordination minutieuses. Plus complexe à mettre en œuvre qu'une approche purement descendante ou ascendante.

# Test d'intégration – Outils

- ▶ **Spring Test** : Excellent pour les tests d'intégration des applications Spring. Il permet de charger les contextes Spring, d'injecter des dépendances et de tester divers composants Spring (contrôleurs, services, référentiels, etc.). Il prend en charge une grande partie du travail de base lié à la mise en place et à la suppression de l'environnement Spring pour les tests.
- ▶ **Arquillian** : Un cadre puissant spécialement conçu pour l'intégration et les tests fonctionnels des applications Java EE et Jakarta EE. Il gère le déploiement de l'application dans un conteneur (par exemple WildFly, Payara, Tomcat) et fournit un environnement de test qui interagit avec l'application déployée.
- ▶ **JUnit** : Bien qu'il s'agisse principalement d'un cadre de test unitaire, JUnit peut également être utilisé pour les tests d'intégration, en particulier lorsqu'il est associé à Spring Test ou à d'autres outils qui gèrent les dépendances et la configuration de l'application.
- ▶ **TestNG** : Comme JUnit, TestNG peut être utilisé pour les tests d'intégration. Ses fonctionnalités telles que l'injection de dépendances et les fournisseurs de données peuvent être utiles dans ce contexte.

# Test d'intégration – Outils

- ▶ **RestAssured** : Simplifie les tests des services web RESTful. Il fournit une API fluide pour effectuer des requêtes HTTP et vérifier les réponses, ce qui facilite le test des interactions avec des API externes ou des contrôleurs REST au sein de votre application.
- ▶ **WireMock** : Une bibliothèque flexible pour simuler des services HTTP. Utile dans les tests d'intégration lorsque vous devez simuler le comportement d'API ou de services externes dont votre application dépend. Offre plus de contrôle que la simple reproduction des réponses. Permet de simuler les conditions du réseau.
- ▶ **Testcontainers** : Une bibliothèque qui vous permet d'exécuter des conteneurs Docker dans vos tests. Ceci est incroyablement utile pour les tests d'intégration avec les bases de données, les files d'attente de messages, les serveurs web et d'autres composants d'infrastructure. Elle fournit un environnement propre et cohérent pour vos tests, éliminant le besoin d'une installation et d'une configuration manuelles des dépendances externes.

# Test d'intégration – Instanciation

- ▶ L'annotation `@TestInstance` dans JUnit Jupiter (JUnit 5) contrôle le cycle de vie des instances de test. Elle détermine combien d'instances de la classe de test sont créées pendant l'exécution du test. Elle propose deux modes :
  - ▶ **`@TestInstance(Lifecycle.PER_CLASS)`** : Une seule instance de la classe de test est créée et réutilisée pour toutes les méthodes de test. Ceci est différent du comportement par défaut (`Lifecycle.PER_METHOD`), où une nouvelle instance est créée pour chaque méthode de test.
  - ▶ **`@TestInstance(Lifecycle.PER_METHOD)`** : Il s'agit du comportement par défaut. Une nouvelle instance de la classe de test est créée pour chaque méthode de test. Cela garantit une isolation complète entre les tests, mais peut s'avérer moins efficace si la configuration est coûteuse.

# Test d'intégration – Instanciation

- ▶ **Avantages de @TestInstance(Lifecycle.PER\_CLASS) :**
  - ▶ **Partage de l'état entre les méthodes de test :** Permet de partager l'état (par exemple, objets, données) entre les méthodes @BeforeEach (avant chaque test), @AfterEach (après chaque test), @BeforeAll (avant tous les tests) et @AfterAll (après tous les tests) et les méthodes de test elles-mêmes. Sans PER\_CLASS, les méthodes @BeforeAll et @AfterAll doivent être statiques.
  - ▶ **Plus efficace pour une mise en place coûteuse :** Si la mise en place de l'environnement de test prend du temps (par exemple, démarrage d'une base de données, initialisation d'un objet volumineux), PER\_CLASS évite de répéter cette mise en place pour chaque méthode de test.
  - ▶ **Amélioration de l'organisation des tests :** Peut contribuer à améliorer l'organisation des tests en permettant à @BeforeAll et @AfterAll d'être non statiques, ce qui les rend plus intuitivement placés dans la structure de la classe de test.

# Test d'intégration – Instanciation

```
@TestInstance(Lifecycle.PER_CLASS)
class MyTests {
    private ExpensiveObject expensiveObject;

    @BeforeAll
    void setUp() {
        expensiveObject = new ExpensiveObject();
    }
}
```

# Test d'intégration – Ordre

- ▶ L'ordre d'exécution des tests peut être une considération importante, en particulier lorsqu'il s'agit de tests qui ont des dépendances ou des effets secondaires.
- ▶ JUnit 5 a introduit l'annotation **@TestMethodOrder**, qui permet de spécifier l'ordre d'exécution. Les options courantes sont les suivantes :
  - ▶ **@TestMethodOrder(MethodOrderer.OrderAnnotation.class)** : Utilise l'annotation **@Order** sur les méthodes de test pour définir l'ordre. Les valeurs les plus faibles sont exécutées en premier.
  - ▶ **@TestMethodOrder(MethodOrderer.DisplayName.class)** : Utilise les noms d'affichage des méthodes de test (ordre alphabétique).
  - ▶ **@TestMethodOrder(MethodOrderer.Random.class)** : Exécute les tests dans un ordre aléatoire. (Peut être utile pour découvrir des dépendances cachées entre les tests).
  - ▶ **@TestMethodOrder(MethodOrderer.Alphanumeric.class)** : Exécute les tests dans un ordre alphanumérique basé sur le nom de la méthode. C'est la valeur par défaut dans JUnit 5 si aucun **@TestMethodOrder** n'est spécifié.

# Test d'intégration – Ordre

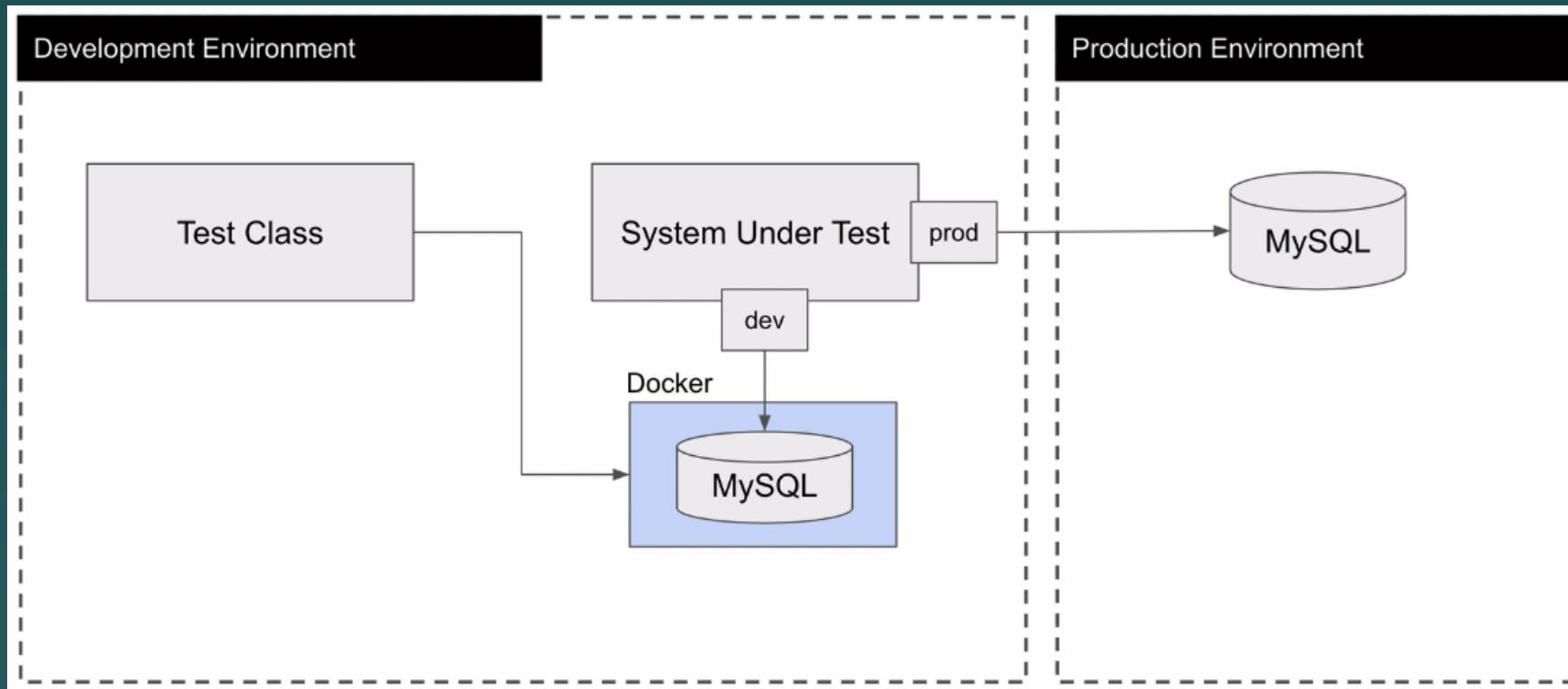
```
@TestInstance(Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class MyTests {
    @Test @Order(1)
    void testA() { /* ... */ }

    @Test @Order(2)
    void testB() { /* ... */ }
}
```

# TestContainers

- ▶ **Tests d'intégration** : Testcontainers est particulièrement utile pour les tests d'intégration lorsque les tests doivent interagir avec des systèmes externes tels que des bases de données, des courtiers en messages ou des services web. Il permet de tester des systèmes réels sans la complexité de la gestion de ces systèmes pendant les tests.
- ▶ **Tests d'interface utilisateur** : Grâce à la prise en charge de Selenium, Testcontainers peut être utilisé pour effectuer des tests d'interface utilisateur dans de vrais navigateurs fonctionnant dans des conteneurs Docker, garantissant ainsi que vos applications web fonctionnent correctement sur différents navigateurs et versions.
- ▶ **Tests de microservices** : Pour les architectures microservices, Testcontainers peut simuler l'ensemble de l'écosystème de services et de dépendances, ce qui permet de tester en profondeur les interactions entre les services.
- ▶ **Environnements d'intégration continue (CI)** : Comme Testcontainers utilise Docker, il s'intègre bien dans les pipelines d'intégration continue, permettant aux tests qui dépendent de services externes de s'exécuter de manière cohérente dans n'importe quel environnement supportant Docker.

# TestContainers



[https://testcontainers.com/modules/?language=j  
ava](https://testcontainers.com/modules/?language=java)

# Test E2E - Définition

- ▶ Le test de bout en bout (E2E) est une méthodologie de test de logiciels qui teste l'ensemble du produit logiciel du début à la fin, en simulant des scénarios d'utilisation réels.
- ▶ Il vise à valider le flux complet du système et à garantir que tous les composants intégrés fonctionnent ensemble comme prévu.

# Test E2E - Caractéristiques

- ▶ **Objectif** : Valider le système dans son ensemble, en s'assurant que tous les composants fonctionnent correctement et répondent aux exigences des utilisateurs.
- ▶ **Champ d'application** : Couvre l'ensemble de l'application, y compris toutes ses dépendances (bases de données, services externes, interface utilisateur, etc.)
- ▶ **Réalisme** : Simule des scénarios d'utilisation réels, offrant un degré élevé de confiance dans le comportement du système en production.
- ▶ **Complexité** : C'est la forme de test la plus complexe en raison de sa portée étendue.
- ▶ **Lenteur d'exécution** : Les tests E2E sont généralement lents à exécuter par rapport aux tests unitaires ou d'intégration.
- ▶ **Maintenance** : La maintenance peut s'avérer difficile en raison de la complexité et des dépendances impliquées. Les modifications apportées à une partie du système peuvent avoir une incidence sur de nombreux tests E2E.

# Test E2E – Outils

- ▶ **Selenium** : Un cadre largement utilisé pour automatiser les navigateurs web. Il permet d'écrire des tests qui interagissent avec des éléments web, naviguent sur des pages, remplissent des formulaires, cliquent sur des boutons et vérifient le comportement de l'application. Il prend en charge de nombreux navigateurs (Chrome, Firefox, Edge, Safari) et langages de programmation (Java, Python, C#, etc.).
- ▶ **Cypress** : Un cadre de test E2E moderne basé sur JavaScript, spécialement conçu pour les applications web. Il s'exécute directement dans le navigateur, ce qui permet de réaliser des tests rapides et fiables. Il met l'accent sur l'expérience du développeur et la facilité d'utilisation.
- ▶ **Playwright** : Un framework basé sur Node.js développé par Microsoft pour les tests de bout en bout inter-navigateurs des applications web modernes. Prend en charge Chromium, Firefox et WebKit. Il offre des fonctionnalités telles que l'attente automatique, l'interception du réseau et le traçage.

# Test E2E – Outils

- ▶ **Puppeteer** : Une bibliothèque Node fournissant une API de haut niveau pour contrôler Chrome ou Chromium via le protocole DevTools. Bien qu'elle soit souvent utilisée pour des tâches d'automatisation du navigateur, elle peut également être utilisée pour les tests E2E.
- ▶ **WebdriverIO** : Un cadre d'automatisation des tests basé sur JavaScript construit sur le protocole WebDriver. Il vous permet d'écrire des tests pour les applications web et mobiles dans divers cadres tels que Mocha, Jasmine et Cucumber.
- ▶ **RestAssured** : Bien que principalement utilisé pour les tests d'API, RestAssured peut également faire partie des tests E2E, en particulier lorsqu'il s'agit de tester la fonctionnalité du backend déclenchée par les interactions du frontend.
- ▶ **Testcontainers** : Ils peuvent être précieux dans les tests E2E en fournissant un environnement cohérent avec toutes les dépendances nécessaires (bases de données, files d'attente de messages, etc.) fonctionnant dans des conteneurs Docker.

# Micro-services & Tests

- ▶ 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

# Exercice 9 - Tests

- ▶ Sur la base des exemples mis à disposition, il vous est demandé de compléter les tests pour le micro-service reviews.
- ▶ Implémentez des tests unitaires pertinents
  - Sans Mock
  - Avec Mock (avec Mockito)
- ▶ Implémentez des tests d'intégration
  - Avec QuarkusTest
- ▶ Implémentez des tests E2E
  - Avec RestAssured et TestContainers

# Cycle de vie

- ▶ Quarkus offre des hooks natifs pour réagir aux événements du cycle de vie de l'application; Ces hooks permettent par exemple d'exécuter du code au démarrage, à l'arrêt, ou lors d'autres événements clés du runtime.
- ▶ Hooks de cycle de vie dans Quarkus
  - Démarrage de l'application : il suffit d'observer l'événement **StartupEvent** avec une méthode annotée **@Observes**. Le code placé ici s'exécutera quand l'application démarre.
  - Arrêt de l'application : même principe : il suffit d'observer **ShutdownEvent** pour exécuter un nettoyage ou des actions de fin.
  - Lifecycle des Beans CDI : il est également possible d'utiliser **@PostConstruct** pour initialiser un bean juste après son instantiation, ou **@PreDestroy** avant sa destruction.

# Exercice 10 : Cycle de vie

- ▶ Sur la base de l'exemple fourni, dans le micro-service catalog, il vous est demandé d'aller nourrir la base de données avec:
  - Des films - <https://developer.themoviedb.org/docs/getting-started>
- ▶ Vous nourrirez également la base de données des utilisateurs avec des utilisateurs fictifs

# Résilience

- ▶ 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



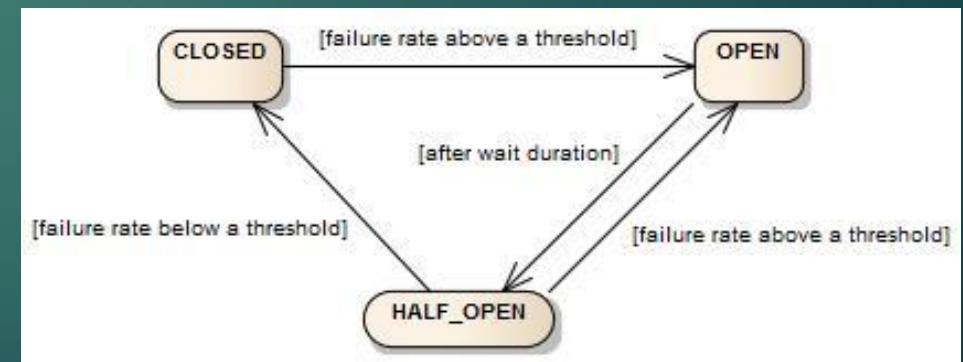
# Resilience4J

- ▶ **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/>
- ▶ **Question:** Comment les services communiquent-ils ?



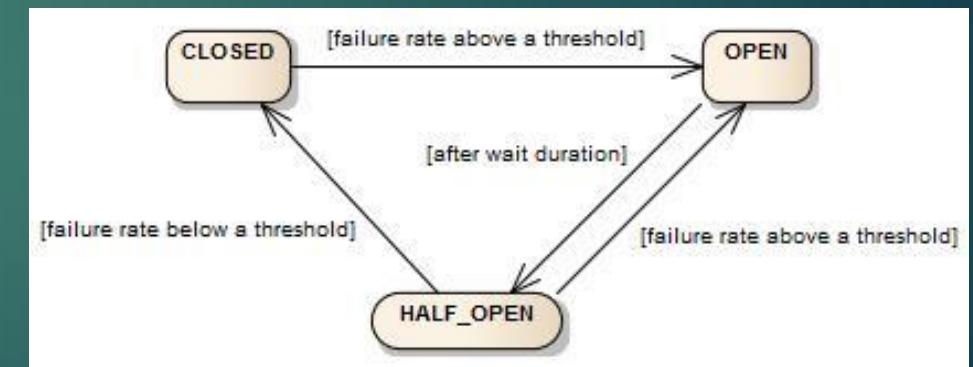
# Circuit Breaker

- ▶ 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 ou 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



# Circuit Breaker

- ▶ 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



# Rate Limiter et Bulkhead

## ► 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
```

# Resilience4J - Annotations

<b>@Timeout</b>	Limite la durée maximale d'exécution d'une méthode	Lève une exception si le délai est dépassé
<b>@Retry</b>	Réessaie automatiquement l'exécution en cas d'erreur	Configurable : nombre de tentatives, délai
<b>@CircuitBreaker</b>	Coupe l'exécution après un certain nombre d'échecs	Reprise après un délai configurable
<b>@Fallback</b>	Définit une méthode de secours (fallback) en cas d'échec	Appelée si les autres patterns échouent
<b>@Bulkhead</b>	Limite le nombre d'exécutions concurrentes	Empêche une surcharge de ressources
<b>@RateLimit</b>	Limite le nombre d'exécutions sur une période de temps	Utile pour les APIs externes/facturées

# Resilience4J - Exemples

```
@Timeout(2000)  
//Méthode interrompue après 2s  
public String appelExterne() { ... }
```

```
@Retry(maxRetries = 3, delay = 500, delayUnit = ChronoUnit.MILLIS)  
//3 tentatives espacées de 500 ms  
public String getData() { ... }
```

```
@CircuitBreaker(  
    requestVolumeThreshold = 4,  
    failureRatio = 0.5,  
    delay = 5000,  
    delayUnit = ChronoUnit.MILLISECONDS  
)  
//Le circuit s'ouvre après 50% d'échecs sur les 4 dernières requêtes  
//Il reste ouvert 5s avant de réessayer  
public String operationResiliente() { ... }
```

```
@Bulkhead(value = 5, waitingTaskQueue = 10)  
//5 appels simultanés autorisés  
public String serviceConcurrent() { ... }
```

```
@Fallback(fallbackMethod = "fallbackMethode")  
public String appelResilient() { ... }  
  
public String fallbackMethode() { return "Risque toléré, retour safe"; }
```

```
@RateLimit(value = 10, period = 1, periodUnit = ChronoUnit.MINUTES)  
//Pas plus de 10 appels par min  
public String callExternalApi() { ... }
```

# Exercice 11 : Résilience

- ▶ Les services sont enregistrés dans Consul pour la découverte dynamique. Nous voulons implémenter un scénario où le service catalog enrichit ses données en appelant les services reviews et users, avec une gestion complète de la résilience via SmallRye Fault Tolerance.
- ▶ Implémenter un endpoint /api/catalog/{id}/enriched qui :
  - ▶ Récupère une ressource du catalogue
  - ▶ Appelle le service reviews pour obtenir les avis associés
  - ▶ Pour chaque avis, appelle le service users pour obtenir les informations de l'auteur
  - ▶ Agrège toutes ces données dans une réponse enrichie
  - ▶ Gère la résilience avec : Retry, Circuit Breaker, Timeout, Fallback et Bulkhead
- ▶ Tester avec la commande "curl http://localhost:8081/resources/1/enriched | jq"

# Programmation réactive

## Mutiny

- ▶ Mutiny est une librairie Java moderne de programmation réactive, conçue pour rendre l'asynchrone simple, lisible et robuste, notamment dans l'univers Quarkus
- ▶ **Mutiny s'appuie sur Vert.x**
- ▶ Philosophie et objectifs
  - ▶ Orientée événement
  - ▶ API fluide et lisible
  - ▶ Lisibilité et robustesse

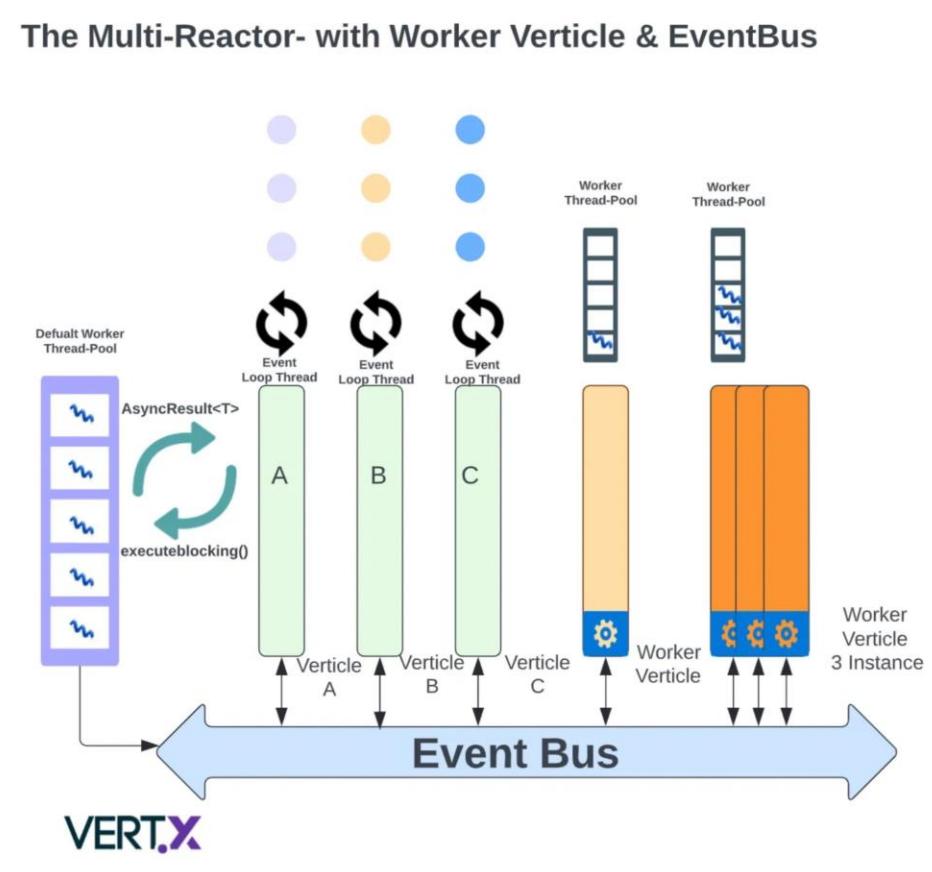
# Vert.x

- ▶ Vert.x est une boîte à outils et un framework événementiel incontournable pour développer des applications réactives, asynchrones et hautement performantes sur la JVM.
- ▶ Principes et architecture
  - Événementiel et asynchrone : Basé sur le modèle de programmation événementielle, Vert.x utilise un event loop (boucle d'événements) pour traiter des milliers de requêtes avec peu de threads, ce qui garantit une très forte scalabilité et des réponses rapides.
  - Polyglotte : Supporte plusieurs langages : Java, JavaScript, Kotlin, Scala, Groovy, Ceylon, etc. L'API est adaptée à chaque langage pour respecter ses conventions propres.
  - Basé sur Netty : S'appuie sur la bibliothèque Netty pour gérer le réseau en mode non bloquant, ce qui favorise sa performance et sa faible latence.

# Vert.x - Concepts

- ▶ **Verticle** : Unité de déploiement de code Vert.x. Chaque Verticle gère ses propres événements et peut être déployé, arrêté ou mis à l'échelle indépendamment.
- ▶ **Event Bus** : Bus de communication distribué, permettant d'envoyer des messages entre Verticles localement ou dans un cluster, et même vers le navigateur (JavaScript client). Idéal pour le découplage des microservices et des traitements.
- ▶ **Core & Extensions** : La partie « core » fournit des API réseau basiques (TCP, HTTP/WebSocket, datagramme, DNS, timers, stockage de données partagé/caché, cluster, haute disponibilité). Les extensions apportent du haut niveau : accès BDD, REST, monitoring, sécurité, etc.
- ▶ **Scalabilité** : Grâce à l'approche non bloquante et au partitionnement du travail par Verticles, Vert.x permet de gérer un énorme nombre de connexions concurrentes avec quelques threads seulement.

# Vert.x - Vue d'ensemble



- **Event Loop Threads (A, B, C)** : Ces threads assurent le traitement non-bloquant des événements. Dès qu'un événement arrive (ex : requête HTTP, message), il est traité rapidement, puis le thread passe immédiatement à l'événement suivant, évitant tout blocage.
- **Worker Verticles et Worker Thread Pools** : Ces verticles permettent d'exécuter des tâches potentiellement bloquantes hors des event loop threads, afin de ne jamais perturber la réactivité globale du système.
- **Event Bus** : L'Event Bus joue un rôle central dans Vert.x. Il permet l'échange de messages entre les différents verticles (event-loop ou worker), qu'ils soient dans le même processus ou distribués, réalisant un vrai découplage asynchrone et facilitant la scalabilité et la gestion d'instances multiples.

# Mutiny – Types fondamentaux

<b>Uni&lt;T&gt;</b>	Emet un seul événement : 1 résultat ou 1 erreur	Lecture non bloquante depuis une base ou HTTP
<b>Multi&lt;T&gt;</b>	Emet plusieurs événements sur la durée	Flux de messages d'un broker (Kafka, RabbitMQ...)

- Opérateurs typiques
  - `onItem().transform()` : transformer la donnée reçue
  - `onFailure().retry()` : appliquer une logique de résilience en cas d'erreur
  - `memoize()`, `merge()`, `delay()`, `filter()` : tous les classiques des API réactives modernes sont disponibles.
- <https://smallrye.io/smallrye-mutiny/3.0.0/>

# Principaux opérateurs

Objectif de l'Opérateur	Opérateur Reactor	Opérateur Mutiny
<b>Transformation (Synchrone)</b>	.map(T -> R)	.onItem().transform(T -> R)
<b>Transformation (Asynchrone 1 -&gt; 1)</b>	.flatMap(T -> Mono<R>)	.onItem().transformToUni(T -> Uni<R>)
<b>Transformation (Asynchrone 1 -&gt; N)</b>	.flatMap(T -> Flux<R>)	.onItem().transformToMulti(T -> Multi<R>)
<b>Filtrage</b>	.filter(T -> boolean)	.select().where(T -> boolean)
<b>Prendre N éléments</b>	.take(long n)	.select().first(long n)
<b>Erreur : Valeur par défaut</b>	.onErrorReturn(T)	.onFailure().recoverWithItem(T)
<b>Erreur : Flux alternatif</b>	.onErrorResume(err -> Mono<T>)	.onFailure().recoverWithUni(err -> Uni<T>)
<b>Erreur : Réessayer</b>	.retry(long n)	.onFailure().retry().atMost(long n)
<b>Combiner (Parallèle / "Zip")</b>	Mono.zip(...) / Flux.zip(...)	Uni.combine().all().unis(...)
<b>Combiner (Séquentiel)</b>	Flux.concat(...)	Multi.createBy().concatenating(...)
<b>Combiner (Mélangé)</b>	Flux.merge(...)	Multi.createBy().merging(...)
<b>Collecter en Liste</b>	.collectList()	.collect().asList()
<b>Contre-pressure (Jeter)</b>	.onBackpressureDrop()	.onOverflow().drop()
<b>Contre-pressure (Buffer)</b>	.onBackpressureBuffer()	.onOverflow().buffer(int size)

# Mutiny - Tests

- ▶ Vous créez un "abonné de test" (AssertSubscriber).
- ▶ Vous abonnez cet espion à votre Uni ou Multi.
- ▶ L'espion va enregistrer tous les événements qu'il reçoit (item, échec, complétion).
- ▶ À la fin, vous posez des questions (assertions) à l'espion :
  - "As-tu reçu un item ?"
  - "Est-ce que l'item était bien celui-ci ?"
  - "As-tu reçu une erreur ?"
  - "Le flux s'est-il bien terminé ?"

# Mutiny – Exemple de test

```
▶ @Test
  @DisplayName("Pattern 1: Test Uni success with UniAssertSubscriber")
void test1() {
    // Create a Uni that emits a value
    Uni<String> uni = Uni.createFrom().item("Hello Mutiny!");

    // Subscribe and get the result
    String result = uni
        .subscribe()
        .withSubscriber(UniAssertSubscriber.create())
        .awaitItem()
        .getItem();

    assertThat(result).isEqualTo("Hello Mutiny!");
}
```

# R2DBC

- ▶ R2DBC (Reactive Relational Database Connectivity) est une spécification et un ensemble de drivers qui permettent d'accéder aux bases de données relationnelles SQL de manière entièrement réactive et non bloquante.
- ▶ C'est à la fois une réponse à la limitation historique de JDBC — qui repose sur un modèle bloquant — et une fondation pour bâtir des applications Java modernes capables de scaler efficacement.

# R2DBC

- ▶ **Non-bloquant et réactif** : R2DBC s'appuie sur le standard Reactive Streams et permet d'exécuter des requêtes SQL en mode asynchrone, sans utiliser un thread par connexion comme avec JDBC. Les opérations retournent des flux (Publisher), manipulables avec des libs comme Mutiny, RxJava ou Reactor.
- ▶ **Compatible SQL** : R2DBC n'est pas une solution NoSQL : il fonctionne avec les bases classiques (PostgreSQL, MySQL/MariaDB, SQL Server, H2, Oracle, etc.) tant qu'un driver compatible est fourni.
- ▶ **API ouverte** : C'est une spécification ouverte destinée aux éditeurs de drivers, proche dans son concept de JDBC, mais orientée vers la réactivité et l'extensibilité.

# R2DBC

- ▶ **Connexion réactive** : les opérations de création de connexion, de requête et de récupération de résultats sont toutes asynchrones.
- ▶ **Support des transactions, isolation, batch, blobs/clobs** : R2DBC supporte les fonctionnalités de base attendues d'un pilote SQL moderne.
- ▶ **Utilisation de l'API** : Les retours des requêtes sont manipulés via des Publishers (typiquement un Publisher<Row>), facilement convertibles en modèles réactifs type Mutiny ou Project Reactor.

# Exercice 12

- ▶ Implémenter une version réactive du service "reviews", nommée "reactive reviews"
- ▶ Utiliser Mutiny afin d'implémenter des chaînes de traitement complètement réactives
- ▶ Utiliser un connecteur R2DBC pour se connecter de manière asynchrone à la base de données
- ▶ Tester les différents services avec le framework de test de mutiny

# Aspects Traverses



# 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 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 ?

# Services 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: Consul)
    - ▶ La responsabilité de la découverte est confiée au serveur (ex: Kubernetes)
- ▶ **Question:** avantages et inconvénients de chaque approche ?

# Message Queueing

- ▶ Les solutions de message queueing (RabbitMQ, Kafka, etc.) sont devenues absolument centrales dans les architectures modernes (microservices, cloud-native, event-driven) pour plusieurs raisons structurantes, présentées dans les diapos suivantes.
- ▶ L'adoption d'un message broker dans une application microservices permet de répondre à tous les impératifs modernes : disponibilité, découplage, robustesse, montée en charge, mais aussi simplicité de maintenance et d'évolutivité.

# Message Queueing - Avantages

## ► Découplage & Asynchronisme

- Découplage fort : Les producteurs (services qui envoient les messages) n'ont aucune dépendance directe sur les consommateurs (services qui les reçoivent), ce qui rend possible l'évolution, le déploiement, et le scaling indépendants de chaque composant.
- Communication asynchrone : Un service ne bloque jamais, il publie un message et délègue le traitement, ce qui fluidifie tous les processus et améliore les temps de réponse utilisateur même pour des tâches longues (PDF, email, facturation...).

## ► Architectures robustes & scalables

- Bufferisation & régulation : Le message broker absorbe les pics de charge ou les ralentissements d'un service, empêche la saturation du backend ou du réseau, rend le système plus tolérant aux montées en charge/brèves saturations.
- Scalabilité simple : On scale uniquement les workers qui consomment les queues les plus sollicités au lieu de surdimensionner toute l'appli ; typique dans la file d'attente d'emails, de génération de rapports ou de traitement vidéo.

# Message Queueing - Avantages

## ► Résilience & Fiabilité

- Garantie de livraison : Un message n'est jamais perdu (at-least-once delivery), même si une partie du système tombe temporairement. La persistance des messages évite la perte d'informations.
- Reprise automatique & tolérance aux pannes : Si un consommateur est indisponible, il reprend la main dès sa reconnexion sans rien perdre des messages publiés pendant la panne.

## ► Extension & évolutivité

- Ajout de fonctionnalités sans regression : On peut brancher à tout moment un ou plusieurs nouveaux consommateurs (notifications, BI, IA...) sans modifier le code legacy ni redéployer les producteurs.
- Multiplicité des usages : Pub/Sub, Work queues, broadcast, event sourcing — tous ces patterns sont rendus possibles et faciles à mettre en œuvre avec une queue de messages.

# RabbitMQ

- ▶ L'intégration de RabbitMQ dans Quarkus est réalisée de manière moderne, non-bloquante et très flexible grâce à l'extension officielle quarkus-messaging-rabbitmq, qui s'appuie sur SmallRye Reactive Messaging.
- ▶ Fonctionnement général
  - Producers et consumers : Ils publient ou consomment des messages RabbitMQ sans gérer de threads ou de connexions bas niveau.
  - Modèle réactif : Utilisation naturelle des types Mutiny (Uni, Multi), évitant tout blocage, compatible avec une architecture reactive/microservices.
  - Configuration déclarative : La configuration des échanges, queues, host, port... se fait essentiellement dans application.properties, ce qui rend le déploiement et l'évolution très simples.

# RabbitMQ - Exemples

```
mp.messaging.outgoing.user-events.connector=smallrye-rabbitmq  
mp.messaging.outgoing.user-events.exchange.name=library.users  
mp.messaging.outgoing.user-events.exchange.type=topic  
mp.messaging.outgoing.user-events.exchange.durable=true  
mp.messaging.outgoing.user-events.routing-keys=user.created,user.updated,user.suspended,user.deleted
```

```
@Inject  
@Channel("resource-events")  
Emitter<ResourceCreatedEvent> resourceEventsEmitter;  
  
resourceEventsEmitter.send(message);
```

```
@Incoming("resource-events")  
public void onResourceCreated(ResourceCreatedEvent event) {  
    LOG.infof("Received resource created event: %s", event);  
}
```

# Exercice 13 - RabbitMQ

- ▶ Mettre en place un échange RabbitMQ entre le service catalog (qui publie l'événement de création de ressource) et le service users (qui consomme les notifications pour informer les utilisateurs intéressés).
  - Ajouter une table de notifications afin d'informer les utilisateurs abonnés à un type de ressource de la création d'une resource
  - Définir dans un format partagé afin que les deux services parlent le même format.
- ▶ Tester et valider

# Approche Serverless

- ▶ 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](#)

# Approche Serverless - Quarkus

- ▶ **Démarrage ultra-rapide** : Quarkus a été conçu pour des temps de "cold start" très courts (surtout compilé en natif avec GraalVM), ce qui est indispensable dans un contexte serverless où chaque exécution est éphémère.
- ▶ **Consommation mémoire minimale** : L'empreinte mémoire de Quarkus, surtout en mode natif, permet de scaler horizontalement et d'éviter les coûts liés à la surconsommation de ressources dans le cloud.
- ▶ **Packaging flexible** : Tu peux générer des artefacts JAR traditionnels (pour Java 21), des images natives (GraalVM) ou des artefacts spécifiques (zip, fat jar, container) directement compatibles avec les principaux fournisseurs serverless.
- ▶ **Extensions dédiées** : Quarkus propose des extensions et guides pour générer des projets plug-and-play pour AWS Lambda, Azure Functions, Google Cloud Functions, ou Knative.
- ▶ **Quarkus Funqy** : Cette extension permet d'écrire des fonctions Java "agnostiques" du cloud, et de les déployer indifféremment sur Lambda, Azure Functions, Google Cloud, Knative, etc. Un même code peut être packagé pour plusieurs providers. <https://quarkus.io/guides/funqy>

# Batching

- ▶ Quarkus supporte le standard Java Batch Processing JSR-352 via l'extension officielle **quarkus-jberet**
  - ▶ <https://quarkus.io/extensions/io.quarkiverse.jberet/quarkus-jberet/>
- ▶ Cette extension permet de définir des jobs batch composés de steps (étapes) avec trois blocs principaux :
  - ▶ **reader** (lecture des données à traiter, fichier, DB, API...)
  - ▶ **processor** (traitement métier, transformation...)
  - ▶ **writer** (persistance ou production de résultats)
- ▶ Les jobs sont configurés via des artefacts XML ou annotations, et exécutés depuis l'application ou à la demande.
- ▶ Cette approche est idéale pour le traitement de gros volumes, les imports/exports, facturation, et calculs périodiques.

# Batching - 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.
- ▶ **Exercice:** Il vous est demandé d'analyser le micro-service **notifications**, basé sur Quarkus JBeret, afin de nettoyer périodiquement les notifications utilisateurs qui ont été marquées "lue".

# Observabilité

- ▶ 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

# Observabilité - Quarkus

- ▶ il est facile et natif d'intégrer la collecte et l'export de métriques avec OpenTelemetry dans Quarkus : l'écosystème Quarkus propose une extension dédiée (`quarkus-opentelemetry`) qui prend en charge la majeure partie de la configuration et du wiring automatiquement.
- ▶ Par défaut, les métriques sont exposées au format OpenTelemetry OTLP. Il est alors possible d'utiliser OpenTelemetry Collector pour router les métriques vers Prometheus, Jaeger, Grafana, ou d'autres backends d'observabilité.
- ▶ Il suffit juste d'activer l'extension et une ou deux propriétés pour bénéficier d'un monitoring complet.

# Observabilité - Quarkus

- ▶ Zero config par défaut : instrumente nativement toutes les requêtes HTTP, base, messaging, etc.
- ▶ Activation des trois “signals” observability (traces, métriques, logs) dès la version Quarkus 3.16+ avec la configuration simple.
- ▶ Compatible natif GraalVM : fonctionne aussi en images natives, ce qui n'est pas le cas de l'agent Java standard.
- ▶ Possibilité d'exporter les métriques Micrometer via l'exporteur OpenTelemetry (quarkus-micrometer-opentelemetry).
- ▶ L'annotation `@WithSpan` ou les APIs OpenTelemetry permettent d'ajouter des traces/métriques custom si souhaité.

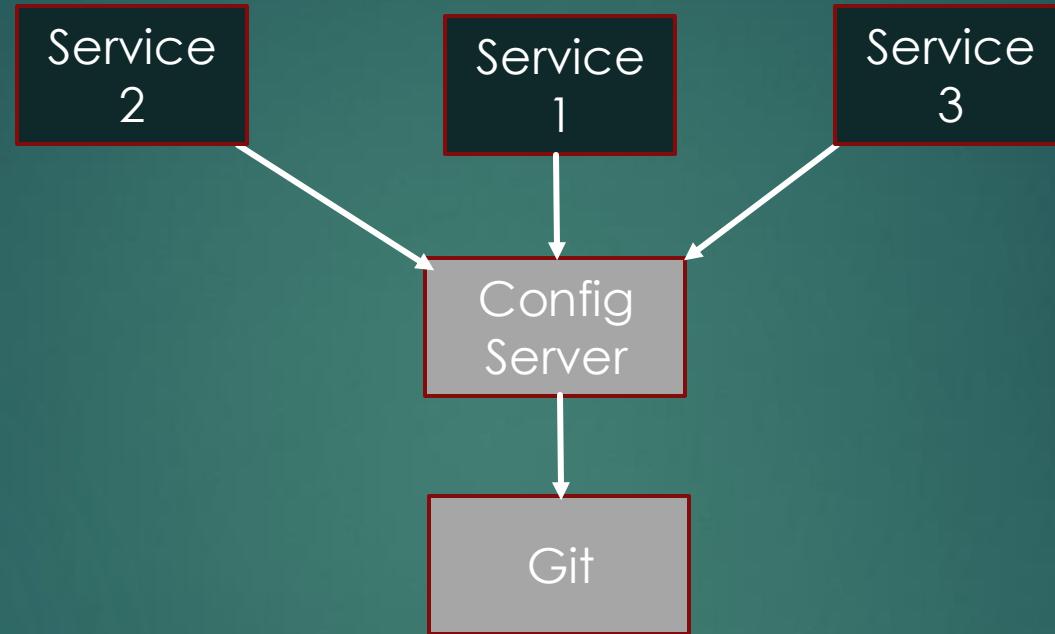
# Logging et Monitoring

- ▶ 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)
- ▶ **Exercice** : Sur la base de l'exemple fourni dans le micro-service catalog, implémenter les mécanismes d'observabilité pour le service reviews.

# 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 ?

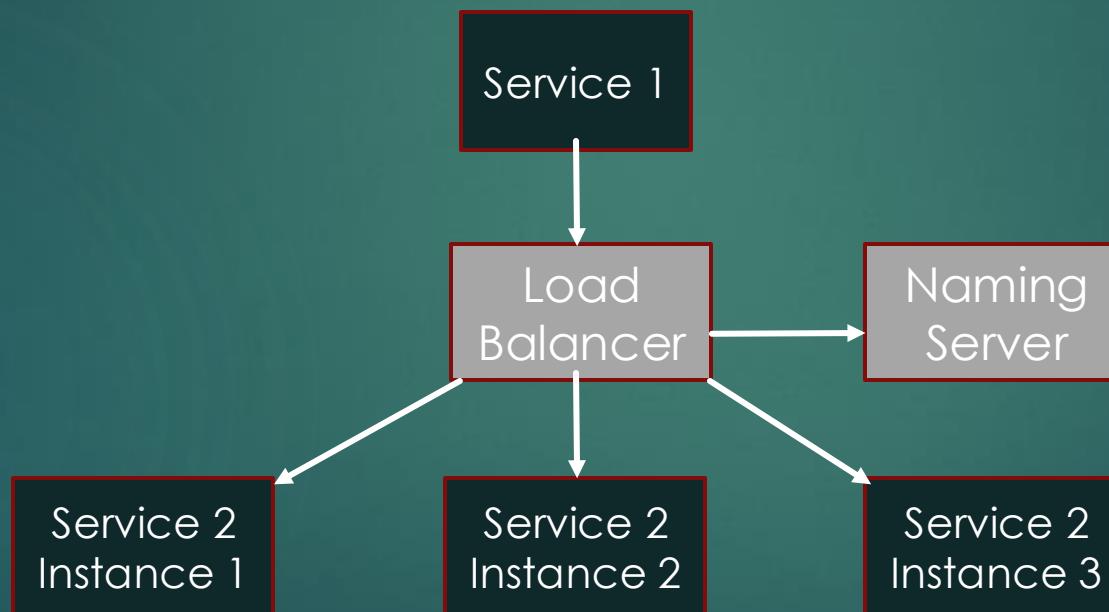
# Configuration Centralisée



- ▶ 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
- ▶ <https://github.com/wansors/lightweight-config-server>

# Load Balancing

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



# Pour aller plus loin

Quarkus 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 3
- ▶ Bases de données:
  - ▶ SQL et NoSQL
  - ▶ Migrations: Liquibase et Flyway
- ▶ Messaging: JMS, Kafka, MQTT, RabbitMQ, Pulsar
- ▶ Cache: Redis, Hazelcast
- ▶ Autres: Emails, Chatbots, Discovery,...