



Microservices avec Quarkus

David THIBAU – 2022

david.thibau@gmail.com



Agenda

- **Introduction**

- Architectures micro services
- Infrastructure de déploiement
- Quarkus
- Quarkus vs SpringBoot

- **Développer avec Quarkus**

- IDEs et outils
- Les extensions Quarkus
- CDI
- Configuration applicative, profils
- Trace, Debug, Test
- Développer des applications natives

- **API Restful avec Quarkus**

- Extensions, JAX-RS Front-end, Open API
- Sérialisations Jackson ou JSON-B
- Filtres http et intercepteurs, exemple CORS
- Rest Client, JWT Authentification
- Modèle réactif, resilience

- **Quarkus et la persistance**

- Configuration de sources de données
- JPA et Hibernate, Validation du modèle
- intégration aux moteurs de recherche via Hibernate Search
- Migration de schéma avec Liquibase, Flyway
- Support NoSQL et services cloud

- **Messaging**

- Support pour Kafka et ActiveMQ
- Messagerie réactive
- Intégration Kafka Streams

- **Sécurité**

- Architecture de la sécurité
- User realms
- OpenID
- Vault

- **Déploiement**

- Construction d'image
- Déploiement vers Kubernetes
- Déploiement vers les acteurs du cloud
- Observabilité des applications : healthcheck, distributed tracing, centralisation des traces



Introduction

Architectures micro services

Infrastructures de déploiement

Quarkus

Quarkus vs SpringBoot



Architecture

Une architecture micro-services implique la décomposition des applications en très petits services

- faiblement couplés
- ayant une seule responsabilité
- développés par des équipes full-stack indépendantes.



Bénéfices attendus

Scaling indépendant : Seuls les services les plus sollicités sont scalés
=> Économie de ressources

Mise à jour indépendantes : Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes
=> Agilité de déploiement

Maintenance facilitée : Les services sont plus petits
=> Corrections, évolutions plus rapide

Hétérogénéité des langages : Utilisation des langages les plus appropriés pour une fonctionnalité donnée

Isolation des fautes : Un dysfonctionnement peut être plus facilement localiser et isoler.

Equipe DevOps autonome : Full-stack team, Intra-Communication renforcée

=> Favorise le partage et les montées en compétences



Caractéristiques

Design piloté par le métier : La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)

Principe de la responsabilité unique : Chaque service est responsable d'une seule fonctionnalité et la fait bien !

Une interface explicitement publiée : Un producteur de service publie une interface qui peut être consommée

DURS (Deploy, Update, Replace, Scale) indépendants : Chaque service métier peut être indépendamment déployé, mis à jour, remplacé, scalé

Communication légère : REST sur HTTP, STOMP sur WebSocket, Server-Sent Events,



Contraintes

Réplication : Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)

Découverte : La scalabilité (automatisé selon certains métriques) nécessite que la localisation des services soit dynamique => Service de discovery

Monitoring : Les services sont surveillés en permanence. Des traces sont générées puis agrégées

Résilience : Les services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

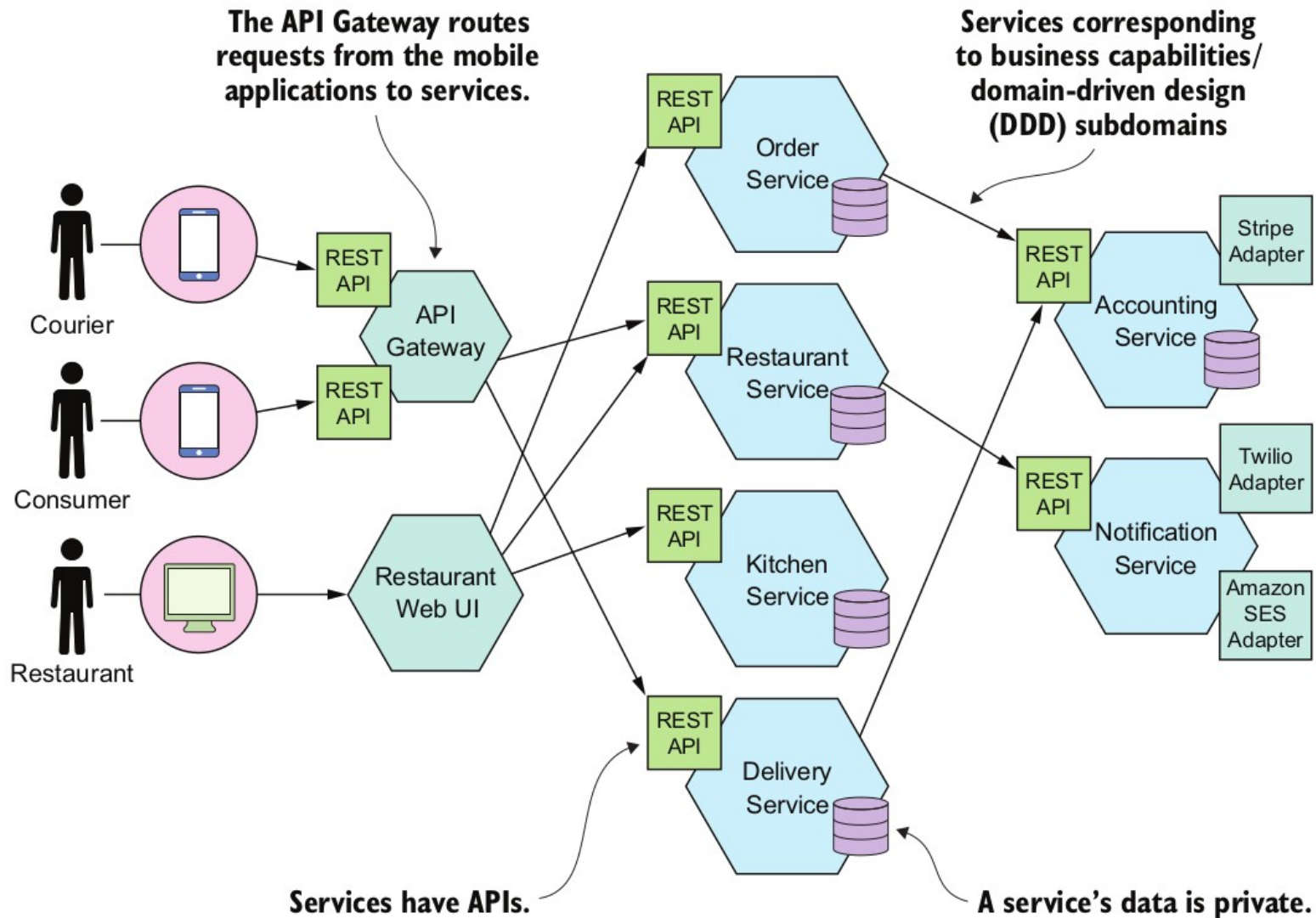
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.



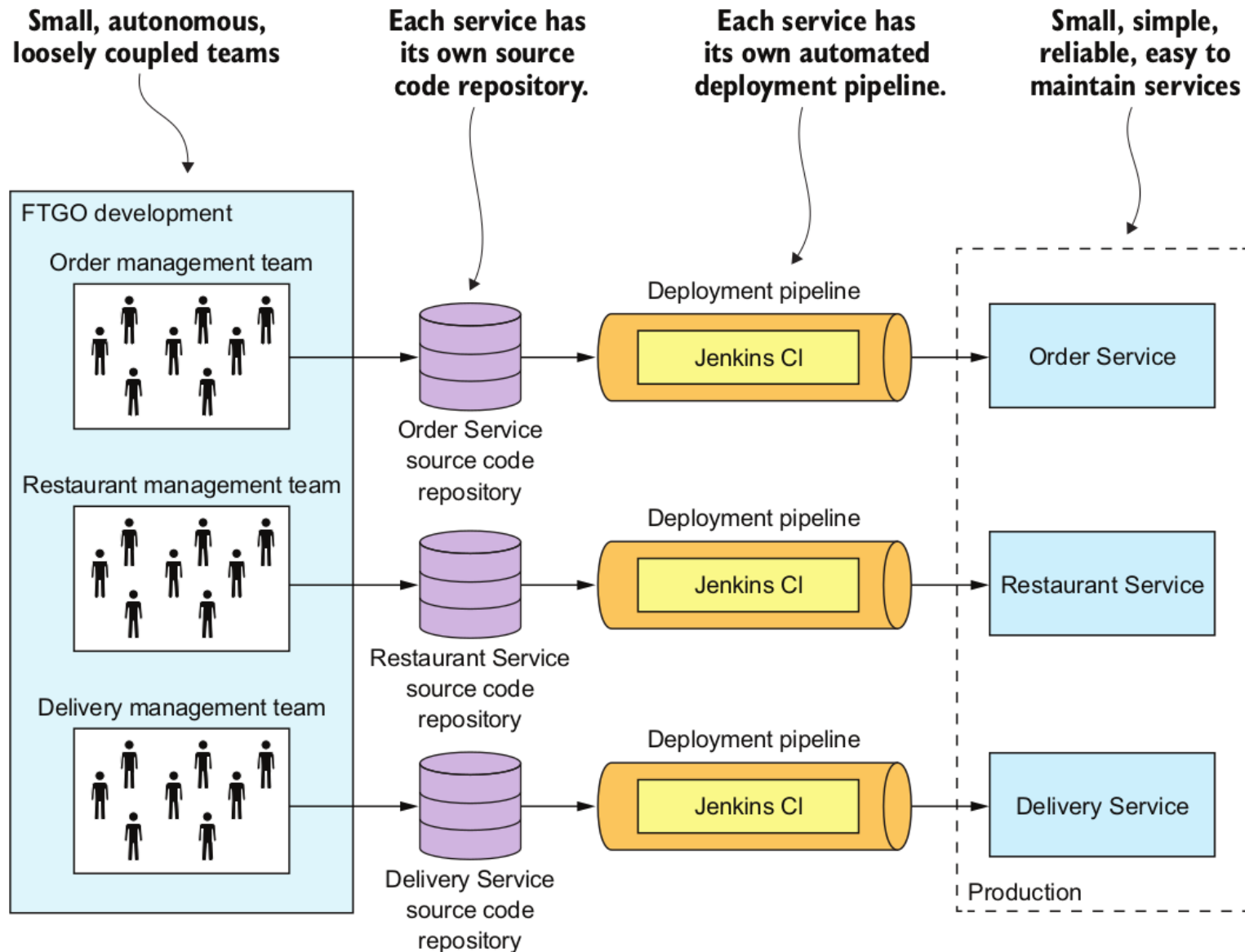
Inconvénients et difficultés

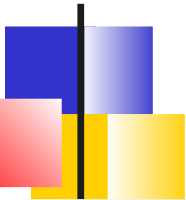
- Trouver la bonne décomposition est difficile.
Une mauvaise décomposition peut entraîner des couplages entre les micro-services
- Le côté distribué fait que le système complet est plus difficile à tester, déployer
- Le déploiement de fonctionnalités qui touche plusieurs services est plus délicat
- La migration d'une application monolithique existante vers les micro-services n'est pas simple

Une architecture micro-service



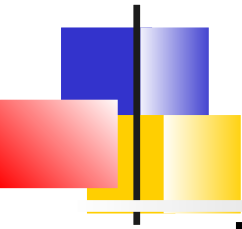
Organisation DevOps





Les 12 facteurs de réussite

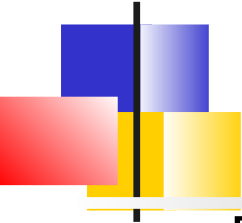
- I. Outil de scm** : Unique source de vérité
- II. Dépendances** : Déclarer explicitement et isoler les dépendances du code source
- III. Configuration** : Configuration externalisée (séparée) du code
- IV. Services** d'appui (backend) : Les services d'appui sont des ressources attachées, possibilité de switcher sans modification de code
- V. Build, release, run** : Distinguer clairement les phases de build, release et deploy. Permet la coexistence de différentes releases en production
- VI. Processus** : Exécute l'application comme un ou plusieurs processus stateless.
- VII. Port binding** : Application est autonome (pas de déploiement sur un serveur). Elle expose juste un port TCP
- VIII. Concurrence** : Montée en charge grâce au modèle de processus
- IX. Disposability** : Renforce la robustesse avec des démarrages et arrêts rapides
- X. Dev/prod parity** : Garder les environnements de développement, de pré-production et de production aussi similaires que possible
- XI. Logs** : Traiter les traces comme un flux d'événements
- XII. Processus d'Admin** : Considérer les tâches d'administration comme un processus parmi d'autres



Patterns micro-service

Les patterns concernant les architectures micro-services peuvent être découpés en 3 domaines :

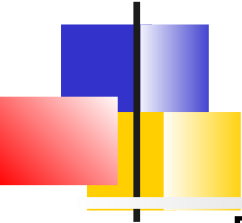
- ***Pattern d'infrastructure*** : Problématique en dehors du développement concernant l'infrastructure d'exécution des systèmes distribués
- ***Pattern applicatif d'infrastructure*** : Problématique d'infrastructure qui impacte le développement.
(Par exemple, quel mode de communication offre un message broker)
- ***Pattern applicatif*** : Problématique purement de développement.



Problèmes à résoudre et design patterns

Patterns applicatifs

- Quelle Décomposition pour mes services ?
Patterns : DDD/sous-domaines, Business Capability,
Comment définir mon API
- Comment maintenir la cohérence de mes données distribuées ?
Saga Pattern
- Comment requêter sur des données distribuées ?
CQRS Pattern
- Comment tester mes micro-services en isolation ?
Design By Contract
- Comment architecturer ma/mes bases de données ?



Problèmes à résoudre et design patterns

Patterns infrastructure applicative

- Quelle communication entre services ?
RPC, Asynchrone, Reactive, Messagerie transactionnelle
- Comment apporter de la résilience ?
Circuit-breaker pattern, Sondes,
- Quels sont les moyens de l'observabilité ?
- Service de discovery, infrastructure ou application ?

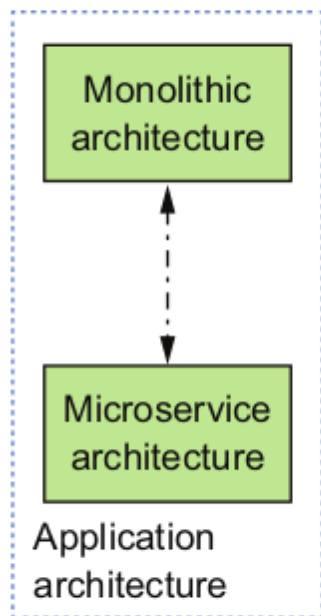
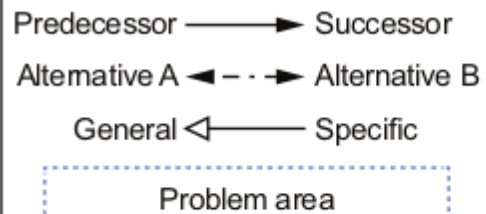


Patterns et problèmes à résoudre

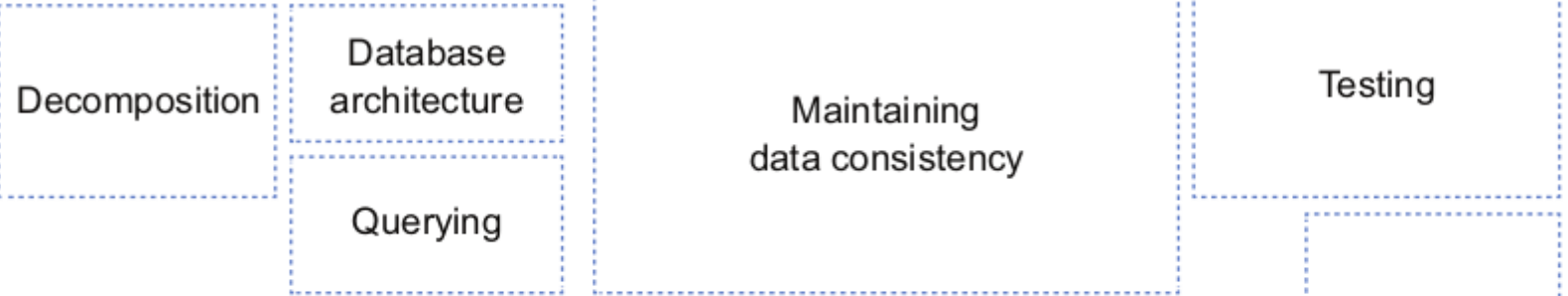
Pattern d'infrastructure

- Quelle infrastructure de déploiement est la plus adaptée ?
Hôtes uniques avec différents processus, Orchestration de Containers, Serverless
- Quels moyens pour exposer les services ?
Ips publics, Ingress Gateway

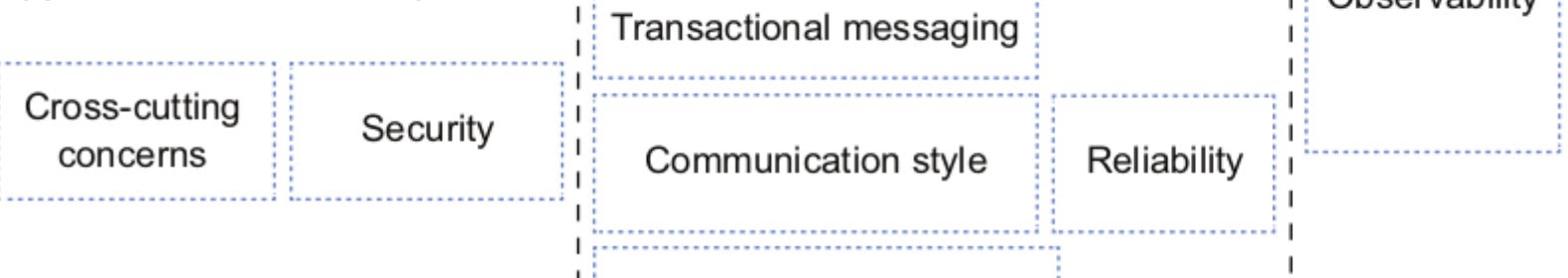
Key



Application patterns



Application infrastructure patterns



Infrastructure patterns



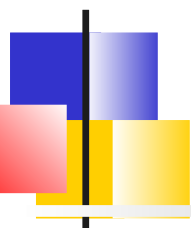
Microservice patterns

Communication patterns



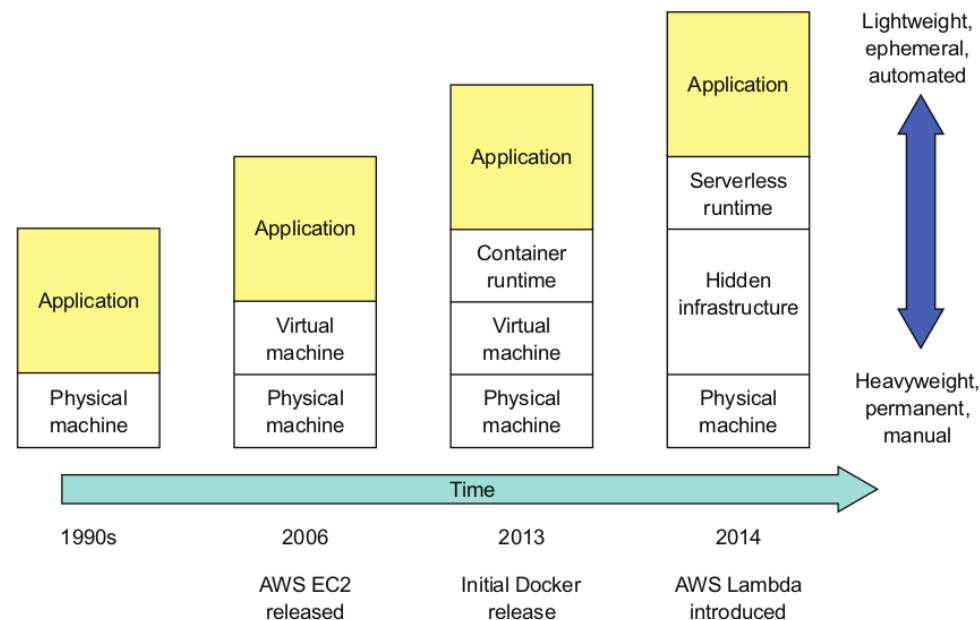
Introduction

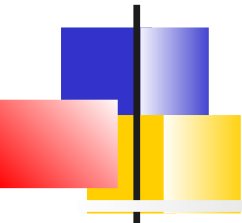
Architectures micro services
Infrastructures de déploiement
Quarkus
Quarkus vs SpringBoot



Infrastructure de déploiement

Même si plusieurs alternatives peuvent être envisagées, l'utilisation des technologies de container et d'orchestrateur de container sont à privilégier.



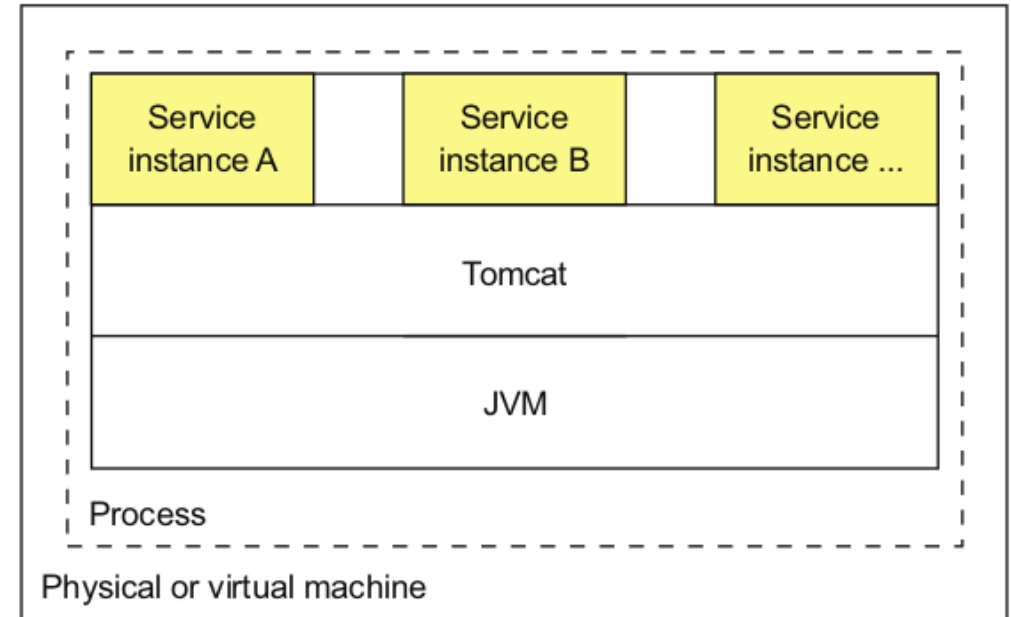
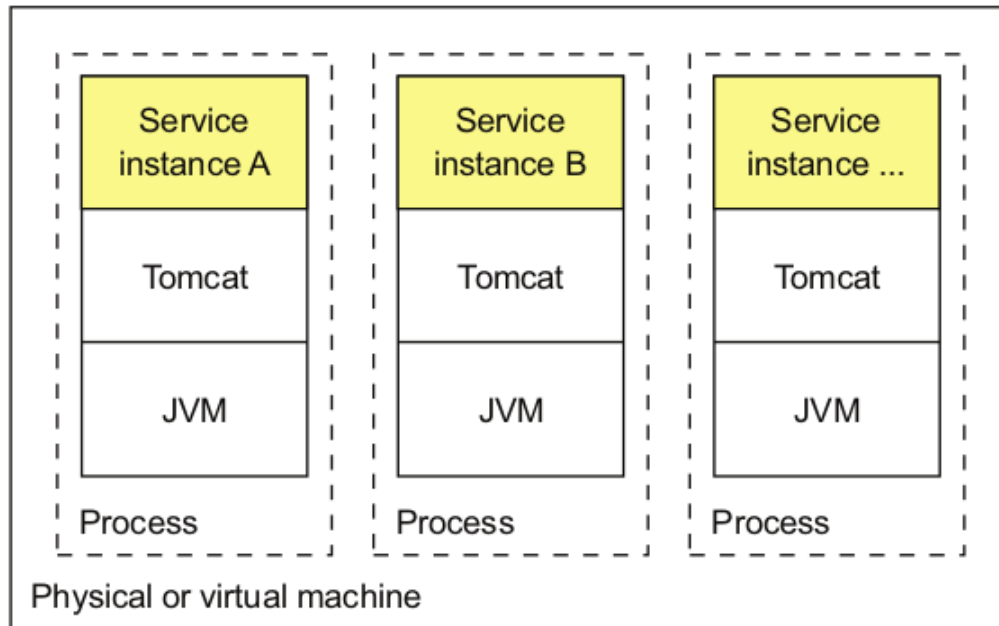


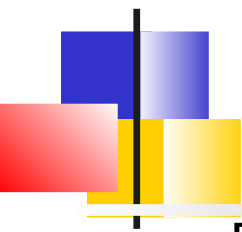
Format de packaging spécifique au langage

Une alternative est de packager les services dans un format spécifique au langage (ex : .war) et de le déployer sur une machine provisionnée (JDK + Tomcat)

- Soit chaque service est dans un processus distinct (a son propre Tomcat)
- Soit plusieurs services sont dans le même processus (plusieurs services déployés sur le même Tomcat)

Examples





Bénéfices / Inconvénients

Bénéfices

Déploiement rapide

Utilisation efficace des ressources surtout lorsque l'on exécute plusieurs instances sur la même machine ou le même processus

Inconvénients

Pas d'encapsulation de la pile technologique. Déploiements mutables.

Pas de possibilité pour limiter les ressources consommées par une instance de service.

Manque d'isolement lors de l'exécution de plusieurs instances de service sur la même machine.

Il est difficile de déterminer automatiquement où placer les instances de service.



Images VM

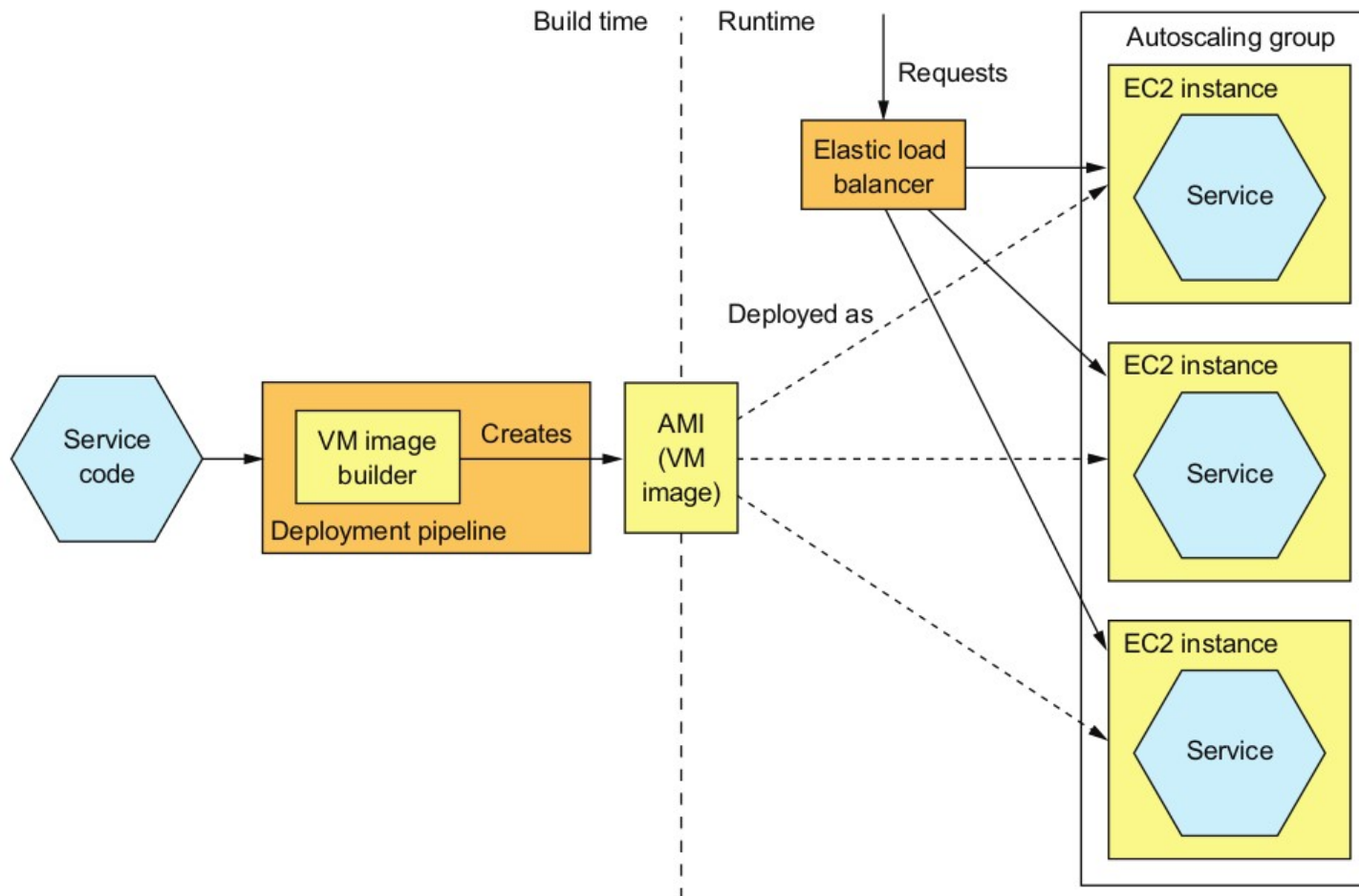
Deploy a service as a VM Pattern¹ :

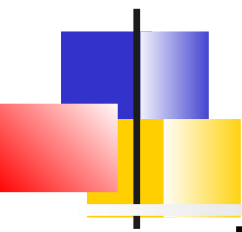
Déployer les services packagés comme des images VM. Chaque service est une VM

Le packaging peut s'automatiser dans les pipelines de build.

1. <http://microservices.io/patterns/deployment/service-per-vm.html>

Example





Bénéfices / Inconvénients

Bénéfices :

L'image VM encapsule la pile technologique =>
Déploiement immuable

Instances de service isolées.

Utilise une infrastructure cloud mature.

Inconvénients :

Utilisation peu efficace des ressources

Déploiements relativement lents

Surcharge d'administration du système



Service comme Container

Deploy a service as a container

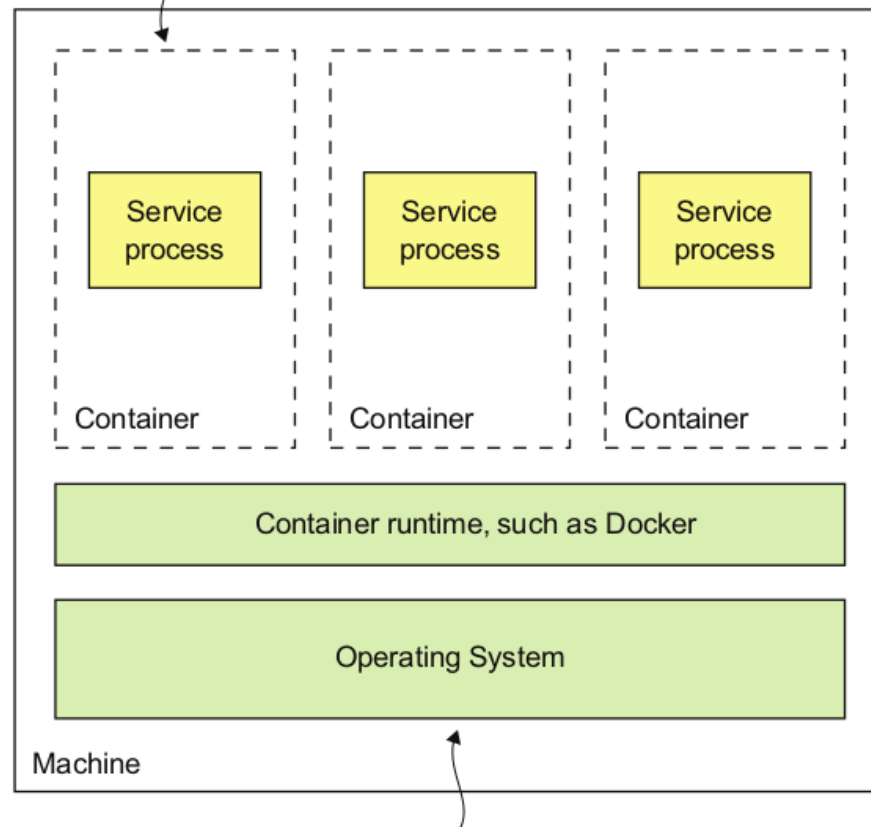
Pattern¹ : Déployé les services packagés comme des images de conteneur. Chaque service est un conteneur

Le packaging en image fait partie de la pipeline de déploiement

1. <http://microservices.io/patterns/deployment/service-per-container.html>

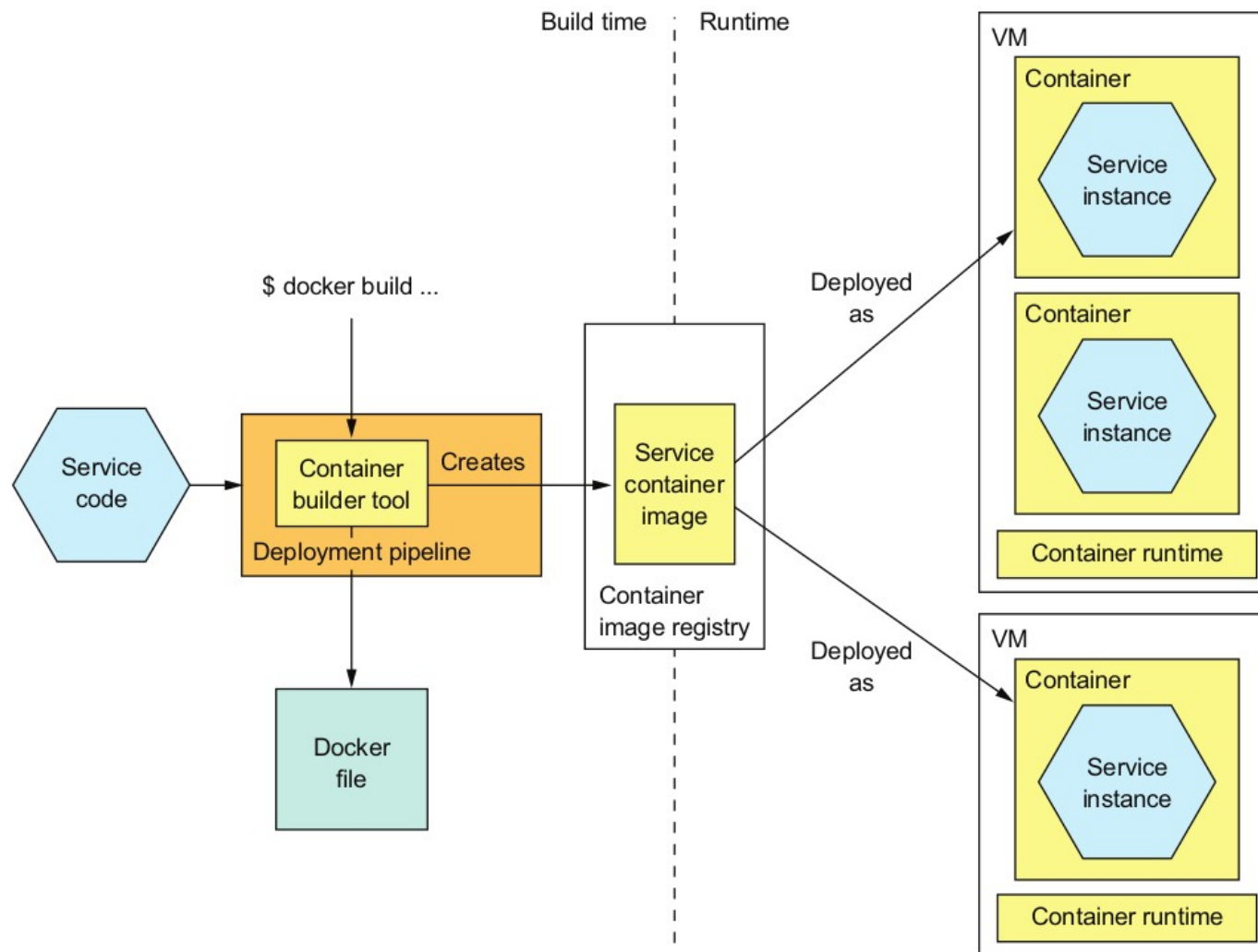
Exécution

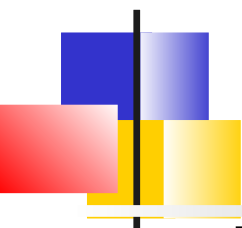
Each container is a sandbox that isolates the processes.



Shared by all of the containers

Déploiement





Bénéfices / Inconvénients

Bénéfices

Encapsulation de la pile technologique. Déploiements immuables

Les instances de service sont isolées.

Les ressources des instances de service sont limitées.

Inconvénients

Équipe DevOps responsable de l'administration des images du conteneur. (Patches de l'OS par exemple)

Administrer l'infrastructure du conteneur-runtime et éventuellement des VMs associés



Introduction

Architectures micro services
Infrastructures de déploiement

Quarkus

Quarkus vs SpringBoot



Le constat Quarkus

Les stacks Java traditionnelles ont été conçues pour les applications monolithiques avec de longs temps de démarrage et de grandes exigences de mémoire dans un monde

A l'époque le cloud, les conteneurs et Kubernetes n'existaient pas !



La réponse de Quarkus

Quarkus veut fournir un framework Java *Cloud native* pour GraalVM et HotSpot.

L'objectif est de faire de Java la plate-forme leader dans les environnements Kubernetes et serverless en offrant un framework capable de traiter le plus large éventail d'architectures d'applications distribuées.

Bien sûr, Quarkus est full OpenSource (Apache License 2.0)



Caractéristiques de Quarkus

- Offre une expérience de développement fluide grâce à une combinaison d'outils, de bibliothèques, d'extensions, etc.
- Déploiement vers Kubernetes facilité, (pas nécessaire de manuellement mettre au point les ressources Kubernetes)
- Apporte toutes les meilleurs librairies de Java
- Programmation réactive ou impérative



GraalVM

GraalVM ajoute un compilateur just-in-time optimisé, écrit en Java, à la machine virtuelle Java HotSpot.

En plus d'exécuter des langages basés sur Java et JVM, GraalVM (Truffle) permet d'exécuter JavaScript, Ruby, Python et un certain nombre d'autres langages populaires.

Avec GraalVM Truffle, Java et d'autres langages pris en charge peuvent interagir directement les uns avec les autres et transmettre des données dans les deux sens dans le même espace mémoire.



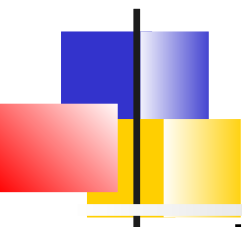
Exécutatble GraalVM Natif

Quarkus a un support pour créer des exécutables GraalVM Native qui démarrent beaucoup plus rapidement

Le compilateur natif utilise des techniques d'élimination de code mort afin d'embarquer les classes de la JVM absolument nécessaires

L'exécutable natif résultant a déjà exécuté la plupart du code de démarrage et sérialisé le résultat dans l'exécutable

Ces techniques sont valables aussi bien pour Kubernetes que pour des environnements bare-metal



Container first

L'idée centrale de Quarkus est de faire au build-time ce que les autres frameworks font à l'exécution time

- Analyse de la configuration, Scan classpath, Chargement dynamique et Instanciation, etc.

A la compilation,

- Quarkus prépare et initialise tous les composants utilisés par votre application.
=> Utilisation optimisée de la mémoire et temps de démarrage super-sonique
- Quarkus réduit l'utilisation de la réflexion, Il remplace les appels réfléchis par des invocations classiques
- En tant que framework IoC, le graphe d'injection est construit à la compilation



Kubernetes natif

Extension Kubernetes permettant d'automatiser un déploiement en une seule étape à l'aide de Jib, Docker et Source-to-Image (S2i)

Extensions serverless permettant de déployer vers des fournisseurs de cloud, notamment AWS Lambda, Azure Functions et Google Cloud Functions ainsi que Knative

Tracing et debugging avec OpenTracing

Micrometer pour les probes applicatives et les métriques

ConfigMaps et Secret pour la configuration applicative

Live coding, le code est directement visible sur le cluster Kubernetes de développement



Modèles de programmation

Quarkus fournit du support pour les modèles de programmation modernes

- **API RestFul** avec JAX-RS, JPA et MicroProfile Rest Client
- **Modèle réactif** avec Vert.x
- **Architecture Event-driven** pour Kafka ou AMQP
- **FaaS (Functions As A Service)** avec Funqy permettant d'être indépendant du fournisseur d'infrastructure



Expérience développeur

- ❖ Codage en direct où les modifications de code sont automatiquement reflétées dans votre application en cours d'exécution.
- ❖ A **unified format** for configuring all application layers
- ❖ Adoption of good programming practices in particular **interfaces**.
- ❖ **Dependency** on as few APIs as possible.
- ❖ **Testability**: Spring makes isolation of tested code easy.
- ❖ Declarative **technical services** (Transaction management, security for example).
- ❖ **Evolvutivity** : Spring's abstraction layers make it easy to switch from one technology to another (JDBC to JPA example)
- ❖ Use of **POJOs (Plain Old Java Objects)**.



Expérience développeur

Codage live : les modifications de code sont automatiquement reflétées dans votre application en cours d'exécution.

Configuration unifiée : Un unique fichier de configuration

Dogme : Quarkus se concentre sur la manière la plus simple et la plus utile d'utiliser une fonctionnalité donnée

Dev UI : Visualiser et configurer les extensions, les beans, accéder aux traces et suite de tests

Dev Services : Intégration automatique avec les services de support tels que les bases de données, les serveurs d'identité, etc.

Test en continu : A chaque changement de code les tests sont exécutés

CLI : Gestion des extensions et commandes de build

Développement Remote : Changement sur des fichiers locaux immédiatement disponibles dans un environnement conteneurisé.



Introduction

Architectures micro services
Infrastructures de déploiement
Quarkus

Quarkus vs SpringBoot



Standards

A la différence de Spring, Quarkus favorise les standards

- Standard Eclipse MicroProfile
- Contexts & Dependency Injection (CDI),
- Jakarta RESTful Web Services (JAX-RS)
- Java Persistence API (JPA)
- Java Transaction API (JTA)

Il s'appuie sur certains frameworks Eclipse Vert.x, Apache Camel et Hibernate



Eclipse MicroProfile

MicroProfile n'est pas une architecture monolithique, il permet aux développeurs de commencer petit et de construire un micro-service ou réduire les applications Java EE existantes en composants de base



Reactive Stack

Avec Spring, un développeur doit décider de la stack à l'avance Réactif ou impératif

Quarkus unifie le réactif et l'impératif



Similitudes

Spring Initializer : *code.quarkus.io*.

Contrôle des versions des dépendances : Idem via BOM Maven

Quarkus et Spring supportent Kotlin.

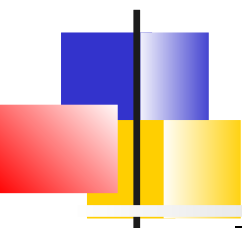
IDES : vscode, IntelliJ, Eclipse

Les applications Quarkus et Spring sont constituées de beans qui exécutent diverses fonctions, telles que l'amorçage d'un serveur HTTP intégré.

1 seul fichier de configuration, profils de configuration

Starter SpringBoot, ~ extensions Quarkus

AutoConfiguration SpringBoot vs Dev Services Quarkus



Différence starter/extension

Il existe cependant une différence fondamentale entre un Spring Boot Starter et une extension Quarkus.

Une extension Quarkus se compose de deux parties distinctes :

- l'augmentation au moment de la construction, appelée **module de déploiement**,
- et le conteneur d'exécution, appelé **module d'exécution**.

La majorité du travail d'une extension est effectuée dans le module de déploiement lorsqu'une application est construite.



Mécanisme

Une extension Quarkus charge et analyse la configuration et le bytecode de l'application compilée et toutes ses dépendances pendant la construction.

À ce stade, l'extension peut lire les fichiers de configuration, analyser les classes pour les annotations, analyser les descripteurs et même générer du code supplémentaire.

Une fois que toutes les métadonnées ont été collectées, l'extension peut prétraiter les actions d'amorçage, telles qu'un framework d'injection de dépendances ou une configuration de endpoint REST.

Le résultat du bootstrap est directement enregistré dans le bytecode et fait partie du package d'application final.



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration applicative, profils

Configuration des logs

Développer des applications natives



Introduction

Quarkus fournit une chaîne d'outils permettant aux développeurs le « live reload » (jusqu'au déploiement d'une application Kubernetes).

- Quarkus CLI
- Plugins Maven ou Gradle

De plus, des plugins et des extensions existent pour les IDEs.

- VSCode
- Eclipse
- Eclipse Che
- IntelliJ



Quarkus CLI

Quarkus CLI permet de créer des projets, de gérer les extensions quarkus, de builder et de lancer les applications en mode dev.

Installation :

<https://quarkus.io/guides/cli-tooling#installing-the-cli>



Commandes

Usage : quarkus [-ehv] [--verbose] [-D=<String=String>]... [COMMAND]

- **create** : Créer un projet
- **build** : Construire le projet
- **dev** : Exécuter le projet en mode dev (Live Coding)
- **extension** : Gérer les extensions
- **registry** : Gérer les registres d'extension



Exemples

```
# Création d'un projet delivery-service
# avec Java/Maven/RestEasy-Reactive
quarkus create app org.formation:delivery-service
# Démarrage du projet en mode dev
cd delivery-service
quarkus dev
# Lister les extensions installables pour le projet
quarkus ext ls -i
# Ajouter un extension
quarkus ext add kubernetes health
```



Maven

Quarkus fournit des plugins Maven permettant d'effectuer également :

- La création de projet
- La gestion des extensions
- Le démarrage du mode dev
- La construction du projet



Exemples

```
# Création d'un projet delivery-service
mvn io.quarkus.platform:quarkus-maven-plugin:2.8.1.Final:create \
    -DprojectGroupId=org.formacion \
    -DprojectArtifactId=delivery-service
# Démarrage du projet en mode dev
cd delivery-service
./mvnw quarkus:dev
# Lister les extensions installables pour le projet
./mvnw quarkus:list-extensions
# Ajouter un extension
./mvnw quarkus:add-extension -Dextensions="hibernate-validator"
```



Mode développement

Le mode Dev permet un déploiement à chaud avec compilation en arrière-plan

- L'actualisation du navigateur déclenche une analyse de l'espace de travail, et si des modifications sont détectées, les fichiers Java sont compilés et l'application est redéployée,
- S'il y a des problèmes avec la compilation ou le déploiement, une page d'erreur vous en informera.



Mode de développement distant

Le mode développement à distance permet d'exécuter Quarkus dans un environnement de conteneur (Kubernetes/OpenShift) et que les modifications apportées à vos fichiers locaux deviennent immédiatement visibles.

=> dev/prod parity !



Mise en place

Pour le mettre en place certaines propriétés doivent être configurées :

```
quarkus.package.type=mutable-jar
quarkus.live-reload.password=changeit
quarkus.live-reload.url=http://my.cluster.host.com:8080
```

L'application est ensuite construite :

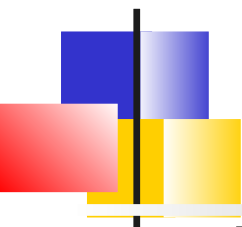
```
./mvnw clean package
```

Avant de démarrer l'application sur l'hôte distant, positionner la variable d'environnement QUARKUS_LAUNCH_DEVMODE

```
QUARKUS_LAUNCH_DEVMODE=true
```

Ensuite, connecter l'agent local à l'hôte distant via :

```
./mvnw quarkus:remote-dev -Dquarkus.live-reload.url=http://my-remote-host:8080
```

pom.xml

Le *pom.xml* typique d'un projet Quarkus contient :

- Des propriétés de version. (en particulier celle de Java et de quarkus)
- Une référence à un BOM permettant de gérer les versions de toutes les dépendances
- Des dépendances sur des extensions
- La configuration des plugins (quarkus, maven-compiler, surefire-plugin)
- Un profil natif pour la génération d'un exécutable natif



IDEs

Une fois le projet Maven ou Gradle généré, on peut l'importer dans son IDE

Des plugins Quarkus sont disponibles pour les différents IDEs.

Ils apportent :

- Des assistants de création de projet
- Une gestion graphique des extensions
- Editeur de propriétés

Tableau comparatif disponible ici :
<https://quarkus.io/guides/ide-tooling>



Tests en continu

Quarkus prend en charge les tests continus, où les tests s'exécutent immédiatement après l'enregistrement des modifications de code.

- Permet d'obtenir un retour instantané sur les modifications de code.

Quarkus détecte quels tests couvrent quel code et utilise ces informations pour n'exécuter que les tests pertinents lorsque le code est modifié.



Mise en place

Lors du démarrage de Quarkus en mode *dev*, il est possible de lancer les tests en appuyant sur la lettre ***r***

Ensuite, toute modification de code relance les tests concernés



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration applicative, profils

Trace, Debug, Test

Développer des applications natives





Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration applicative, profils

Configuration des logs

Développer des applications natives



Introduction

Le modèle de programmation Quarkus est basé sur la spécification Contexts and Dependency Injection (CDI)

Les développeurs écrivent des beans dont le cycle de vie est géré par le framework (Pattern IoC)

Des annotations permettent de déclarer configurer et injecter les beans.

Cette approche permet de déléguer au framework toute la plomberie technique et se concentrer sur les problématiques métier.



Un bean typique

// Annotation déclarant un bean et son cycle de vie

@ApplicationScoped

public class Translator {

// Injection de dépendance d'un autre bean

@Inject

Dictionary dictionary;

// Intercepteur appliquant un cross-cutting concern

@Counted

String translate(String sentence) {

// ...

}

}



Classe de Configuration

L'annotation **@Produce** permet de déclarer une méthode qui instancie un bean

@Dependent

```
public class TracerConfiguration {
```

```
    @Produces
```

```
    public Tracer tracer(Reporter reporter) {  
        return new Tracer(reporter);
```

```
    }
```

```
}
```



Injection de dépendances

Avec CDI, le processus effectuant la correspondance entre un bean et un point d'injection est **type-safe**. (S'appuie sur les types Java)

Chaque bean déclarer un ensemble de types de Beans (Hiérarchie de classes et d'interfaces)

Ensuite, un bean est assignable à un point d'injection si

- le bean a un type correspondant au type requis
- et possède tous les qualificateurs requis

Exactement un seul bean doit être assignable à un point d'injection, sinon la construction échoue :

- `UnsatisfiedResolutionException`. Si aucun bean n'est éligible à l'injection
- `AmbiguousResolutionException`. Si plusieurs beans sont éligibles



Injection par constructeur ou méthodes

```
@ApplicationScoped
```

```
public class Translator {
```

```
    private final TranslatorHelper helper;
```

```
    // Injection par le constructeur, helper is final
```

```
    Translator(TranslatorHelper helper) {
```

```
        this.helper = helper;
```

```
    }
```

```
    // Injection par méthode
```

```
    @Inject
```

```
    void setDeps(Dictionary dic, LocalizationService locService) {
```

```
        / ...
```

```
    }
```

```
}
```



Qualifier

Les ***qualifiers*** sont des annotations qui aident le conteneur à distinguer les beans qui implémentent le même type. Si aucun qualifier n'est précisé à un point d'injection, c'est le qualifier @Default qui est appliqué

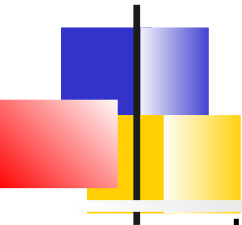


Exemple

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Superior {}
-----
@Superior
@ApplicationScoped
public class SuperiorTranslator extends Translator {

    String translate(String sentence) {
        // ...
    }
}
```

Ce bean serait assignable à *@Inject @Superior Translator* et *@Inject @Superior SuperiorTranslator* mais pas à *@Inject Translator*.



Scopes

Un bean a un scope qui détermine son cycle de vie.

- **@javax.enterprise.context.ApplicationScoped** : Une seule instance de bean est utilisée et partagée entre tous les points d'injection. L'instance est créée en mode lazy lors de l'appel d'une méthode a proxy du bean
La majorité des cas
- **@javax.inject.Singleton** : Une seule instance, créée lors de la résolution d'un point d'injection.
A utiliser avec précaution, car pas de possibilité de mock ni de rechargement
- **@javax.enterprise.context.RequestScoped** : Associé à la requête (http en général)
- **@javax.enterprise.context.Dependent** : Les instances ne sont pas partagées. Le cycle de vie est associé au bean qui l'injecte
- **@javax.enterprise.context.SessionScoped** : Bean associé à la session HTTP



Callback

Une classe de bean peut déclarer des méthodes de cycle de vie **@PostConstruct** et **@PreDestroy**.

```
@ApplicationScoped  
public class Translator {
```

```
    @PostConstruct
```

```
    void init() {  
        // ...  
    }
```

```
    @PreDestroy
```

```
    void destroy() {  
        // ...  
    }
```

```
}
```




Intercepteurs

Les intercepteurs sont utilisés pour séparer les « cross-cutting concern » de la logique métier.

```
@Logged // Annotation associée à l'intercepteur
@Priority(2020)
@Interceptor
public class LoggingInterceptor {

    @Inject
    Logger logger;

    @AroundInvoke
    Object logInvocation(InvocationContext context) {
        // ...log before
        Object ret = context.proceed();
        // ...log after
        return ret;
    }
}
```



Décorateurs

Les décorateurs sont similaires aux intercepteurs, mais parce qu'ils implémentent des interfaces avec la sémantique métier, ils sont capables d'implémenter la logique métier.

```
public interface Account { void withdraw(BigDecimal amount); }
```

```
@Priority(10)
@Decorator
public class LargeTxAccount implements Account {
    @Inject
    @Any // N'importe quel qualifieur
    @Delegate
    Account delegate;

    @Inject
    LogService logService;

    void withdraw(BigDecimal amount) {
        delegate.withdraw(amount);
        if (amount.compareTo(1000) > 0) {
            logService.logWithdrawal(delegate, amount);
        }
    }
}
```



Modèle événementiel

Les beans peuvent produire et consommer des événements pour interagir de manière complètement découplée.

Tout objet Java peut être transmis par l'événement.

Les qualificateurs facultatifs agissent comme des sélecteurs de sujet.



Example

```
class TaskCompleted {  
    // ...  
}  
  
@ApplicationScoped  
class ComplicatedService {  
  
    @Inject  
    Event<TaskCompleted> event;  
  
    void doSomething() {  
        // ...  
        event.fire(new TaskCompleted());  
    }  
}  
  
@ApplicationScoped  
class Logger {  
  
    void onTaskCompleted(@Observes TaskCompleted task) {  
        // ...log the task  
    }  
}
```



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration applicative, profils

Configuration des logs

Développer des applications natives



Introduction

Quarkus (noyau et extensions) et votre application sont tous deux configurés via le même mécanisme.

Basé sur l'API SmallRye Config qui est une implémentation de la spécification MicroProfile Config.



Sources de configuration

Par défaut, Quarkus lit les propriétés de configuration à partir de plusieurs sources (par ordre décroissant) :

- Propriétés système (-D au démarrage)
- Variables d'environnement (Passage en uppercase avec underscore)
- Fichier *.env* dans le répertoire de travail actuel (Même règle que les variables d'environnement)
- Fichier de configuration de l'application Quarkus dans *\$PWD/config/application.properties*
- Fichier de configuration de l'application Quarkus *application.properties* dans le classpath
- Fichier de configuration MicroProfile Config *META-INF/microprofile-config.properties* dans le classpath

La configuration finale est l'agrégation des propriétés définies par toutes ces sources.



Sources additionnelles

Quarkus fournit des extensions supplémentaires qui couvrent d'autres formats de configuration :

- YAML (quarkus-config-yaml)
- HashiCorp Vault (vault)
- Consul (config-consul)
- Spring Cloud (spring-cloud-config-client)



Expression pour les propriétés

Les valeurs de configuration peuvent utiliser des segments d'expression, enveloppés par la séquence **`${ ... }`**.

```
remote.host=quarkus.io
```

```
callable.url=https://${remote.host}/
```



Injection

Pour s'injecter une clé de configuration :

```
// Si clé pas présente, startup fails
@ConfigProperty(name = "greeting.message")
String message;

// Si clé pas présente, valeur par défaut
@ConfigProperty(name = "greeting.suffix", defaultValue="!")
String suffix;

// Si clé pas présente, Optional is Empty
@ConfigProperty(name = "greeting.name")
Optional<String> name;
```



Objets de configuration

Il est possible de regrouper plusieurs propriétés de configuration dans une seule interface qui partagent le même préfixe.

L'annotation ***@io.smallrye.config.ConfigMapping*** nécessite une interface déclarant les différentes propriétés de configuration de l'objet.

// Définition de 2 propriétés server.host et server.port

```
@ConfigMapping(prefix = "server")
interface Server {
    String host();

    int port();
}
```



Enregistrement

Au démarrage de l'application, un ConfigurationMapping peut être enregistré à 2 moments :

- STATIC INIT : Les services Quarkus vont être lancés
- RUNTIME INIT : Les services Quarkus ont été initialisés

On peut indiquer l'annotation **@StaticInitSafe**, si les valeurs des clés de configuration peuvent être résolus en début de boot (c'est généralement le cas)



Usage

Pour utiliser un objet de configuration, il suffit de se l'injecter.

```
class BusinessBean {  
    @Inject  
    Server server;  
  
    public void businessMethod() {  
        String host = server.host();  
    }  
}
```



Groupes imbriqués

Il est possible d'imbriquer les interfaces.

```
@ConfigMapping(prefix = "server")
public interface Server {
    String host();
    int port();
    Log log();

    interface Log {
        // propriété server.log.enabled
        boolean enabled();
        String suffix();
        boolean rotate();
    }
}
```



Collections

Il est possible de mapper des propriétés vers des *List* ou des *Set*

```
@ConfigMapping(prefix = "server")
public interface ServerCollections {
    Set<Environment> environments();

    interface Environment {
        String name();

        List<App> apps();

        interface App {
            String name();
            List<String> services();
            Optional<List<String>> databases();
        }
    }
}
```



Collections (2)

application.properties

server.environments[0].name=dev

server.environments[0].apps[0].name=rest

server.environments[0].apps[0].services=bookstore,registration

server.environments[0].apps[0].databases=pg,h2

server.environments[0].apps[1].name=batch

server.environments[0].apps[1].services=stock,warehouse



Map

Également vers les *Map*

```
@ConfigMapping(prefix = "server")
public interface Server {
    String host();

    int port();

    Map<String, String> form();
}
----
```

server.host=localhost
server.port=8080
server.form.login-page=login.html
server.form.error-page=error.html
server.form.landing-page=index.html



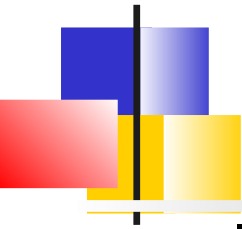
Validation

Une interface de configuration peut combiner des annotations de Bean Validation pour valider les valeurs de configuration

L'extension *quarkus-hibernate-validator* doit être présente

```
@ConfigMapping(prefix = "server")
interface Server {
    @Size(min = 2, max = 20)
    String host();

    @Max(10000)
    int port();
}
```



Les profils

Il est souvent nécessaire de configurer différemment en fonction de l'environnement cible. (intégration, production, etc..)

Les profils de configuration permettent plusieurs configurations dans le même fichier ou des fichiers séparés

Les profils sont ensuite activés via un nom .



Profil dans le nom de la propriété

Pour pouvoir définir des propriétés avec le même nom, chaque propriété doit être précédée d'un %, le nom du profil et un point.

```
quarkus.http.port=9090
```

```
%dev.quarkus.http.port=8181
```

Les profiles dans le fichier .env suivent la syntaxe suivante :

```
_{PROFILE}_CONFIG_KEY=value:
```



Profils par défaut

Par défaut, Quarkus propose trois profils, qui s'activent automatiquement dans certaines conditions :

- **dev** - Activé en mode développement (c'est-à-dire *quarkus:dev*)
- **test** - Activé lors de l'exécution de tests
- **prod** - Le profil par défaut lorsqu'il n'est pas exécuté en mode développement ou test

On peut y ajouter ses profils personnalisés, il suffit de définir une propriété avec un nouveau nom de profil

%staging.quarkus.http.port=9999



Nommage de fichier

Il est également possible de regrouper toutes les configuration d'un profil dans un seul fichier

application-{profile}.properties

```
# application-staging.properties
quarkus.http.port=9190
quarkus.http.test-port=9191
```



Activation du profil

Le profil d'exécution Quarkus est celui défini au moment du build

```
./mvnw package -Pnative -Dquarkus.profile=prod-aws
```

```
./target/my-app-1.0-runner
```

=> Exécution avec le profil *prod-aws*



Beans en fonction du profil

Quarkus ajoute une fonctionnalité que CDI ne prend pas en charge, qui consiste à activer conditionnellement un bean en fonction des profils de build

- **`@io.quarkus.arc.profile.IfBuildProfile`** :
Active le bean si le profil est activé
- **`@io.quarkus.arc.profile.UnlessBuildProfile`** :
Désactive le bean si le profil est activé
- **`@io.quarkus.arc.DefaultBean`** : Le bean est activé si aucun autre Bean de ce type n'est activé



Example

@Dependent

```
public class TracerConfiguration {
```

```
    @Produces
```

```
    @IfBuildProfile("prod")
```

```
    public Tracer realTracer(Reporter reporter, Configuration configuration) {  
        return new RealTracer(reporter, configuration);  
    }
```

```
    @Produces
```

```
    @DefaultBean
```

```
    public Tracer noopTracer() {  
        return new NoopTracer();  
    }
```

```
}
```



Beans en fonction de propriétés

Quarkus ajoute une fonctionnalité que CDI ne prend pas en charge, qui consiste à activer conditionnellement un bean lorsqu'une propriété de build a/n'a pas de valeur spécifique

- ***@io.quarkus.arc.properties.IfBuildProperty*** :
Active si la propriété a une valeur spécifique
- ***@io.quarkus.arc.properties.UnlessBuildProperty*** :
Désactive si la propriété a une valeur spécifique



Example

@Dependent

```
public class TracerConfiguration {
```

```
    @Produces
```

```
    @IfBuildProperty(name = "some.tracer.enabled", stringValue = "true")
```

```
    public Tracer realTracer(Reporter reporter, Configuration configuration) {  
        return new RealTracer(reporter, configuration);  
    }
```

```
    @Produces
```

```
    @DefaultBean
```

```
    public Tracer noopTracer() {  
        return new NoopTracer();  
    }
```

```
}
```



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration applicative, profils

Configuration des logs

Développer des applications natives



Introduction

En interne, Quarkus utilise JBoss Log Manager et la façade JBoss Logging.

- On peut alors utiliser la façade JBoss Logging ou les frameworks supportés (java.util.logging ,SLF4J, Apache Commons Logging)

Toute la configuration peut être effectuée dans *application.properties*.



JBoss Logging

```
import org.jboss.logging.Logger;
...
@Path("/hello")
public class ExampleResource {

    private static final Logger LOG =
        Logger.getLogger(ExampleResource.class);

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



Logging simplifié

Au lieu de déclarer un champ `Logger`, on peut utiliser l'API de logging simplifiée :

```
import io.quarkus.logging.Log;

class MyService {
    public void doSomething() {
        Log.info("Simple!");
    }
}
```



Injection de Logger

Il est également possible de s'injecter une instance de *org.jboss.logging.Logger*

```
import org.jboss.logging.Logger;

@ApplicationScoped
class SimpleBean {

    @Inject
    Logger log;

    @LoggerName("foo")
    Logger fooLog;

    public void ping() {
        log.info("Simple!");
        fooLog.info("Goes to _foo_ logger!");
    }
}
```




Niveaux de logs

Les niveaux de logs utilisés par Quarkus sont :

- OFF : Désactive le logging.
- FATAL
- ERROR
- WARN
- INFO
- DEBUG
- TRACE
- ALL (Niveau spécial pour inclure tous les messages, même les niveaux personnalisés)



Configuration

La configuration s'effectue dans `application.properties`

```
quarkus.log.level=INFO
quarkus.log.category."org.hibernate".level=DEBUG
```

La propriété *min-level* indique le niveau le plus fin autorisé (par défaut DEBUG), elle peut être indiquée au niveau de root ou d'une catégorie

```
quarkus.log.level=INFO
quarkus.log.min-level=TRACE
quarkus.log.category."org.hibernate".level=TRACE
```

Il est également possible de configurer le format et les handlers



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration applicative, profils

Configuration des logs

Développer des applications natives



Introduction

Construire un exécutable natif nécessite d'utiliser une distribution de GraalVM. Il existe trois distributions :

- Oracle GraalVM Community Edition (CE)
- Oracle GraalVM Enterprise Edition (EE)
- Mandrel. (Spécifique Quarkus)

Sans GraalVM, il est possible d'utiliser d'exécuter le build dans un conteneur Docker qui intègre GraalVM.

L'exécutable natif construit contient le code de l'application, les bibliothèques requises, les API Java et une version réduite d'une machine virtuelle.



Construction

Le *pom.xml* contient un profile ***native***

Pour construire :

```
quarkus build -native
```

Ou

```
./mvnw package -Dnative
```

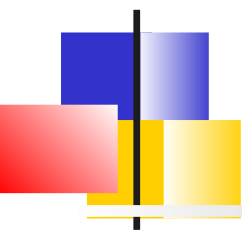
Le build produit l'exécutable :

```
target/<artifactId>-1.0.0-SNAPSHOT-runner
```

De plus, le profil fait en sorte que des tests d'intégration utilisant les endpoint HTTP et annotés par *@QuarkusIntegrationTest* sont exécutés sur l'exécution native.

Ils peuvent également être exécutés sans packager par :

```
./mvnw verify -Pnative
```



Construction d'un container



RESTful API

Introduction

Annotations

Sérialisation JSON

Problématiques RestFul

RestClient



Introduction

Pour les applications RESTful, Quarkus utilise JAX-RS¹ avec comme implémentation par défaut RESTEasy.

Les développeurs mettent au point des classes ressources qui expose des point d'accès HTTP via les annotations JAX-RS

Par défaut, ces ressources communiquent en JSON



Quarkus vs JavaEE

- Il n'est pas nécessaire de définir une classe Application. Quarkus créera et fournira automatiquement une sous-classe Application.
- Une seule application JAX-RS est prise en charge dans une JVM. Quarkus prend en charge la
déploiement d'une seule application JAX-RS,
contrairement à JAX-RS exécuté dans un conteneur de servlet standard, où plusieurs applications JAX-RS peuvent être déployées.
- Toutes les ressources JAX-RS sont traitées comme des beans CDI par défaut et sont définies comme des singletons CDI.



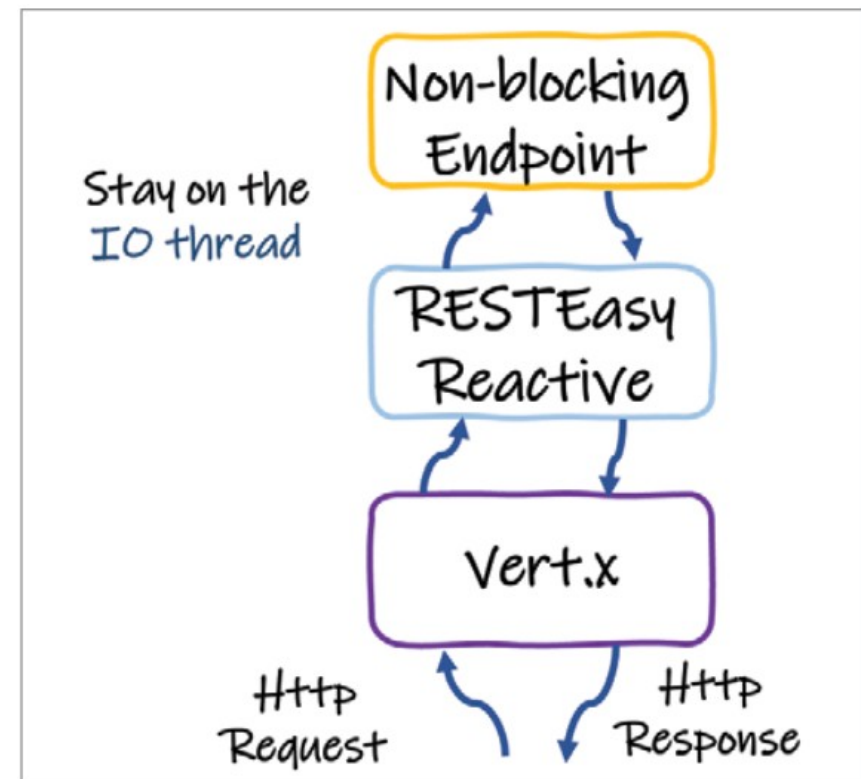
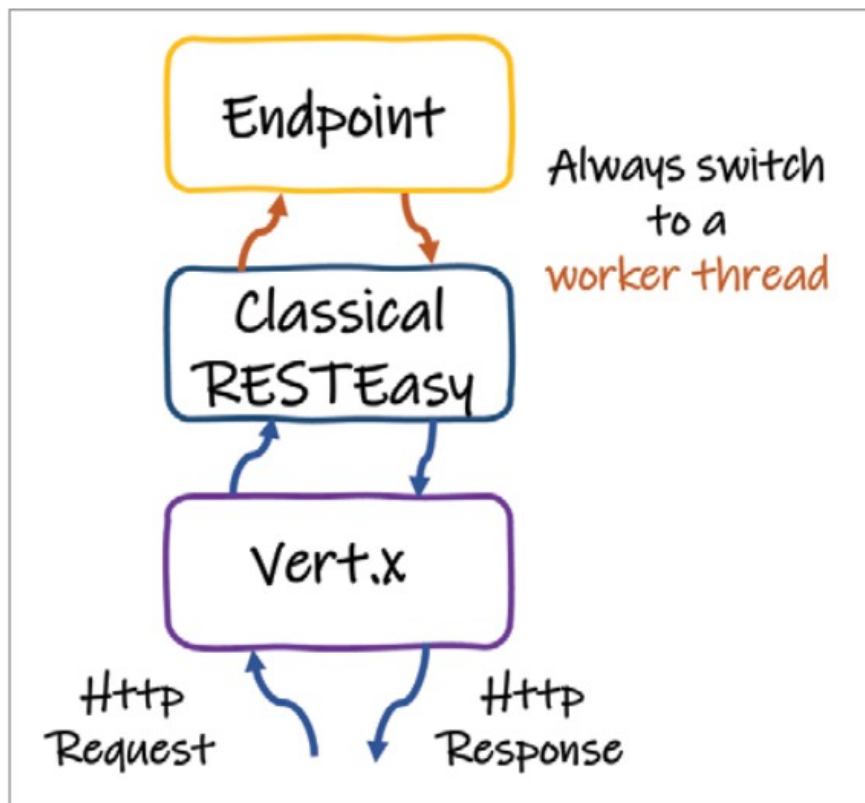
Modèle réactif

L'extension RESTEasy utilise Eclipse Vert.x comme environnement d'exécution et en particulier son modèle de threads event-loop (modèle réactif)

2 extensions RestEASY sont disponibles

- RestEASY Classique : Modèle basé sur des pools de worker threads
- RestEASY Reactif : Basé sur l'évent loop.
Mais à la différence de SpringBoot, les développeurs peuvent mixer du code réactif avec du code non-réactif

Classical vs Reactive





RestEASY Reactive

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy-reactive</artifactId>
</dependency>
```

```
@Path("")
public class Endpoint {

    @GET
    public String hello() {
        return "Hello, World!";
    }
}
```



RESTful API

Introduction
Annotations
Sérialisation JSON
Problématiques RestFul
RestClient



Déclaration des endpoints

Toute classe annotée avec **@Path** peut voir ses méthodes exposées en tant que points de terminaison REST

- L'annotation de classe @Path définit le préfixe URI sous lequel les méthodes de la classe seront exposées.
Il peut être vide ou contenir un préfixe
- Chaque méthode peut à son tour avoir une autre annotation @Path qui s'ajoute au préfixe.



Exemple

Endpoint accessible à */rest/hello*

```
@Path("rest")
```

```
public class Endpoint {
```

```
    @Path("hello")
```

```
    @GET
```

```
    public String hello() {  
        return "Hello, World!";
```

```
    }
```

```
}
```



Path racine

On peut définir le chemin racine pour tous les endpoints de l'application via ***@ApplicationPath***

```
@ApplicationPath("/api")
```

```
public static class MyApplication extends  
    Application {
```

```
}
```




Méthode HTTP

Les méthodes des endpoints sont annotées avec des annotations spécifiant la méthode HTTP

***@GET, @HEAD, @POST, @PUT,
@DELETE, @OPTIONS, @PATCH***



Media Type

La classe peut être annotée par

@Produces ou ***@Consumes***.

Ce qui permet de spécifier un ou plusieurs types de média que le endpoint peut accepter comme corps de requête HTTP ou produire comme corps de réponse HTTP.

Chaque méthode peut surchargée avec ses propres annotations *@Produces* ou *@Consumes*.



Example

```
@Path("negotiated")
public class Endpoint {

    @Produces({MediaType.APPLICATION_JSON, MediaType.TEXT_PLAIN})
    @GET
    public Cheese get() {
        return new Cheese("Morbier");
    }

    @Consumes(MediaType.TEXT_PLAIN)
    @PUT
    public Cheese putString(String cheese) {
        return new Cheese(cheese);
    }

    @Consumes(MediaType.APPLICATION_JSON)
    @PUT
    public Cheese putJson(Cheese cheese) {
        return cheese;
    }
}
```



Paramètres de requêtes

Différentes annotations sont utilisées pour récupérer des données de la requête :

- @RestPath : Une partie de l'URL
- @RestQuery : Un paramètre HTTP
- @RestHeader : Une entête
- @RestCookie : Un cookie
- @RestForm : Un champ de formulaire Web
- @RestMatrix : Un segment de chemin de l'URL

A la différence de JAX-RS, il est en général inutile de préciser le nom du paramètre dans l'annotation, le bon nonnomage de la variable suffit



Example

POST /cheeses;variant=goat/tomme?age=matured HTTP/1.1

Content-Type: application/x-www-form-urlencoded

Cookie: level=hardcore

X-Cheese-Secret-Handshake: fist-bump

smell=strong

```
@Path("/cheeses/{type}")
```

```
@POST
```

```
public String allParams(@RestPath String type,
```

```
                        @RestMatrix String variant,
```

```
                        @RestQuery String age,
```

```
                        @RestCookie String level,
```

```
                        @RestHeader("X-Cheese-Secret-Handshake")
```

```
                        String secretHandshake,
```

```
                        @RestForm String smell) {
```

```
    return type + "/" + variant + "/" + age + "/" + level + "/" + secretHandshake + "/" +  
        smell;
```

```
}
```



Corps de requête

Tout paramètre de méthode sans annotation recevra le corps de la méthode.

Les types supportés sont :

- File, byte[], char[], String, InputStream, Reader
- Tous les types primitifs Java
- Un objet quelconque à partir d'un JSON
- JsonArray, JsonObject, JsonStructure, JsonValue
- Buffer (Vert.x Buffer)



Retourner un corps de réponse

Le type de retour de la méthode et son contenu facultatif seront utilisés pour décider comment le sérialiser dans la réponse HTTP (généralement JSON)

D'autres types sont supportés :

- *Path* : Le contenu d'un fichier spécifié par le Path
- *PathPart* : Contenu partiel d'un fichier spécifié par le Path
- *FilePart* : Le contenu partiel d'un fichier
- *AsyncFile* : *Vert.x AsyncFile*, (complet ou partiel)
- Et les types réactifs : *Uni*, *Multi* ou *CompletionStage*



Retourner la réponse entière

Il est possible de contrôler complètement la réponse avec : ***RestResponse***.

```
@GET
public RestResponse<String> hello() {
    // HTTP OK status avec text/plain
    return ResponseBuilder.ok("Hello, World!", MediaType.TEXT_PLAIN_TYPE)
    // entête
    .header("X-Cheese", "Camembert")
    // Entête Expires
    .expires(Date.from(Instant.now().plus(Duration.ofDays(2))))
    // Envoyer un cookie
    .cookie(new NewCookie("Flavour", "chocolate"))
    // et build
    .build();
}
```




Annotations pour le statuts et les entêtes

Le code statut et les entêtes peuvent également être définies via les annotations ***@ResponseStatus***, ***@ResponseHeader***, ***@Cache*** et ***@NoCache***.

```
@ResponseStatus(201)
@ResponseHeader(name = "X-Cheese", value = "Camembert")
@POST
public String createCheese() {
```



Retours réactifs

Si la méthode exécute une tâche asynchrone ou réactive avant de pouvoir répondre, elle peut renvoyer le type **Uni** ou **Multi**¹

Cela permet de ne pas bloquer la thread event-loop

```
@Path("logs")
public class Endpoint {

    @Inject
    @Channel("log-out")
    Multi<String> logs;

    @GET
    public Multi<String> streamLogs() {
        return logs;
    }
}
```

Quarkus propose également un support pour SSE (Server Side Events)²

1. *Librairie Mutiny*

2. <https://quarkus.io/guides/resteasy-reactive#server-sent-event-sse-support>



Objets du contexte HTTP

Les méthodes peuvent également se faire injecter les objets HTTP en déclarant des arguments des types suivants :

- ***HttpHeaders*** : Toutes les entêtes
- ***ResourceInfo*** ou ***SimpleResourceInfo*** : Informations sur la méthode et la classe du endpoint
- ***SecurityContext*** : L'utilisateur et ses rôles
- ***UriInfo*** : URI courante
- ***HttpRequest***, ***HttpResponse***,
RequestContext, ***Sse*** : Objets bas niveau Vert.x
- ...



RESTful API

Introduction
Annotations
Sérialisation JSON
Problématiques RestFul
RestClient



Sérialisation JSON

Pour obtenir la prise en charge de JSON ,
au lieu d'importer *quarkus-resteasy-reactive*, importer l'un des modules
suivants :

- **io.quarkus:quarkus-resteasy-reactive-jackson**
- **io.quarkus:quarkus-resteasy-reactive-jsonb**

Pour la sérialisation XML, importer :

- **io.quarkus:quarkus-resteasy-reactive-jaxb**



Support Quarkus pour la sérialisation

Quarkus supporte des fonctionnalités avancées pour la sérialisation

- Sérialisation sécurisée, certains champs sont ignorés en fonction des rôles
- Support de @JsonView, (Sérialisation différentes des objets en fonction des cas d'usage)
- Sérialisation complètement personnalisée



Exemple : Sécurisation

```
import io.quarkus.resteasy.reactive.jackson.SecureField;
```

```
public class Person {
```

```
    @SecureField(rolesAllowed = "admin")
```

```
    private final Long id;
```

```
    private final String first;
```

```
    private final String last;
```



Exemple @JsonView

```
public class Views {  
  
    public static class Public { }  
  
    public static class Private extends Public { }  
}  
----  
public class User {  
  
    @JsonView(Views.Private.class)  
    public int id;  
  
    @JsonView(Views.Public.class)  
    public String name;  
}
```




Exemple *@JsonView* (2)

```
@JsonView(Views.Public.class)
```

```
@GET
```

```
@Path("/public")
```

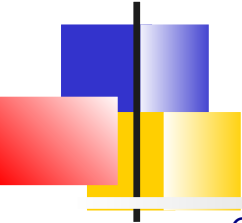
```
public User userPublic() {  
    return testUser();  
}
```

```
@JsonView(Views.Private.class)
```

```
@GET
```

```
@Path("/private")
```

```
public User userPrivate() {  
    return testUser();  
}
```



Exemple : sérialisation personnalisée

```
@CustomSerialization(UnquotedFields.class)
```

```
@GET
```

```
@Path("/invalid-use-of-custom-serializer")
```

```
public User invalidUseOfCustomSerializer() {  
    return testUser();  
}
```

```
----
```

```
public static class UnquotedFields implements BiFunction<ObjectMapper,  
    Type, ObjectWriter> {
```

```
    @Override
```

```
    public ObjectWriter apply(ObjectMapper objectMapper, Type type) {  
        return
```

```
objectMapper.writer().without(JsonWriteFeature.QUOTE_FIELD_NAMES);  
    }
```

```
}
```



RESTful API

Introduction
Annotations
Sérialisation JSON
Problématiques RestFul
RestClient



Filtre CORS

Pour autoriser CORS, il faut tout simplement positionner la propriété :

quarkus.http.cors=true

Si l'on veut restreindre le CORS, on peut en plus utiliser les propriétés suivantes :

- ***quarkus.http.cors.origins***
- ***quarkus.http.cors.methods***
- ***quarkus.http.cors.headers***
-



Threads

RESTEasy Reactive est implémenté à l'aide de deux types de threads principaux :

- Threads de boucle d'événement : qui sont responsables, entre autres, de la lecture des octets de la requête HTTP et de l'écriture des octets dans la réponse HTTP
- Threads de travail : ils sont regroupés et peuvent être utilisés pour décharger des opérations de longue durée

Les threads sont utilisées en fonction de la signature de la méthode. Si une méthode renvoie l'un des types suivants, elle est considérée comme non bloquante et sera exécutée par défaut sur le thread IO :

- `io.smallrye.mutiny.Uni`, `io.smallrye.mutiny.Multi`,
`java.util.concurrent.CompletionStage`,
`org.reactivestreams.Publisher`



Surcharge

On peut surcharger le comportement par défaut en utilisant les annotations **@Blocking** et **@NonBlocking**

@Blocking

@GET

```
public Uni<String> blockingHello() throws InterruptedException {  
    // do a blocking operation  
    Thread.sleep(1000);  
    return Uni.createFrom().item("Yaaaawwwnnnnnnn...");  
}
```

On peut utiliser également les annotations sur la classe principale (sous-classe de `javax.ws.rs.core.Application`) pour positionner le comportement par défaut de toutes les méthodes.



Mapping des exceptions JAX-RS

En cas d'erreur, afin de générer le code HTTP adéquat, on peut utiliser les sous-classes fournies par JAX-RS de ***WebApplicationException***

```
@GET
public String findCheese(String cheese) {
    if(cheese == null)
        // Envoi d'un 400
        throw new BadRequestException();
    if(!cheese.equals("camembert"))
        // Envoi d'un 404
        throw new NotFoundException("Unknown cheese: " + cheese);
    return "Camembert is a very nice cheese";
}
```



Mapping Exception métier

On peut également transformer des exceptions métier à l'aide de l'annotation **@ServerExceptionHandler** sur une méthode transformant son paramètre d'entrée de type exception en une *RestResponse*

```
@ServerExceptionHandler
public RestResponse<String> mapException(UnknownCheeseException x) {
    return RestResponse.status(Response.Status.NOT_FOUND, "Unknown cheese:
" + x.name);
}
```

Si l'annotation est utilisée dans une classe ressource, elle n'a d'effet que pour les exceptions lancées par la ressource

Si elle est définie dans une classe séparée, elle s'applique globalement



OpenAPI

Ajouter l'extension '***quarkus-smallrye-openapi***'

- Une documentation OpenAPI est disponible à *<http://localhost:8080/q/openapi>*
- L'interface swagger-ui est également accessible (par défaut en mode dev et en mode production si *`quarkus.swagger-ui.always-include=true`*) à :
<http://localhost:8080/q/swagger-ui>



Validation

Pour valider les paramètres d'entrée d'un endpoint REST, on peut l'annoter avec les contraintes de Hibernate Validator (*@NotNull*, *@Digits...*) ou avec **@Valid** (qui cascade la validation au bean).

- Si une erreur de validation se produit, un message d'erreur (JSON) et un code retour approprié sont renvoyé par le framework .

L'annotation *@Valid* peut également utilisée sur des méthodes de bean service



Example

```
@Path("/end-point-method-validation")
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Result tryMeEndPointMethodValidation(@Valid Book book) {
    return new Result("Book is valid! It was validated by end point method
        validation.");
}
----
```



```
@ApplicationScoped
public class BookService {

    public void validateBook(@Valid Book book) {
        // your business logic here
    }
}
```



RESTful API

Introduction
Annotations
Sérialisation JSON
Problématiques RestFul
RestClient



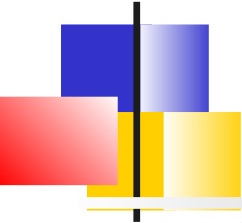
Mise en place

Extensions :

rest-client-reactive-jackson ou ***rest-client-reactive***

Soit :

- Définir des interfaces annotées par ***@RegisterRestClient*** et utilisant les annotations JAX-RX sur ses méthodes
- Créer des clients programmatiquement en utilisant ***RestClientBuilder***



Exemple : Interface

@RegisterRestClient permet à Quarkus de savoir que cette interface est censée être disponible pour l'injection CDI en tant que client REST

Les annotations JAX-RS sont utilisées afin que Quarkus puisse construire la requête correspondante (@Path, @GET, @PathParam, @Produce, @Consume)

Exemple :

```
@Path("/extensions")
@RegisterRestClient
public interface ExtensionsService {

    @GET
    @Path("/stream/{stream}")
    Set<Extension> getByStream(@PathParam("stream") String stream,
    @QueryParam("id") String id);
}
```



Configuration

La définition de la base URL s'effectue dans la configuration

```
quarkus.rest-client."org.acme.rest.client.ExtensionsService".url=  
https://stage.code.quarkus.io/api
```

Ou

```
@RegisterRestClient(configKey="extensions-api")  
public interface ExtensionsService {  
    [...]  
}  
quarkus.rest-client.extensions-api.url=https://  
    stage.code.quarkus.io/api  
quarkus.rest-client.extensions-api.scope=javax.inject.Singleton
```



Usage du Restclient

Pour effectuer l'appel REST, il suffit d'injecter le client et d'appeler la méthode de l'interface

```
@Path("/extension")
public class ExtensionsResource {

    @RestClient
    ExtensionsService extensionsService;

    @GET
    @Path("/id/{id}")
    @Blocking
    public Set<Extension> id(String id) {
        return extensionsService.getById("stream-1", id);
    }
}
```




RestClientBuilder

Il est également possible de créer un client REST
programmatically via ***RestClientBuilder***

En utilisant la même interface :

```
@Path("/extension")
public class ExtensionsResource {

    private final ExtensionsService extensionsService;

    public ExtensionsResource() {
        extensionsService = RestClientBuilder.newBuilder()
            .baseUri(URI.create("https://stage.code.quarkus.io/api"))
            .build(ExtensionsService.class);
    }

    @GET
    @Path("/id/{id}")
    public Set<Extension> id(String id) {
        return extensionsService.getById(id);
    }
}
```



Support pour l'asynchrone

Pour tirer parti de la nature réactive du client, on peut utiliser la version non bloquante qui prend en charge les types *CompletionStage* et *Uni*.

```
@Path("/extensions")
@registerRestClient(configKey = "extensions-api")
public interface ExtensionsService {

    @GET
    Set<Extension> getById(@QueryParam("id") String id);

    @GET
    CompletionStage<Set<Extension>> getByIdAsync(@QueryParam("id") String id);
}
```



Usage

```
Path("/extension")
public class ExtensionsResource {

    @RestClient
    ExtensionsService extensionsService;

    @GET
    @Path("/id/{id}")
    @Blocking
    public Set<Extension> id(String id) {
        return extensionsService.getById(id);
    }

    @GET
    @Path("/id-async/{id}")
    public CompletionStage<Set<Extension>> idAsync(String id) {
        return extensionsService.getByIdAsync(id);
    }
}
```



Entêtes HTTP

Il y a plusieurs façons de spécifier des entêtes personnalisés pour les appels REST :

- Enregistrer un *ClientHeadersFactory* ou *ReactiveClientHeadersFactory* avec l'annotation **@RegisterClientHeaders**
- Spécifier la valeur de l'en-tête avec **@ClientHeaderParam**
- Préciser la valeur du header par **@HeaderParam**



Example

```
@Path("/extensions")
@RegisterRestClient
@RegisterClientHeaders(RequestUUIDHeaderFactory.class)
@ClientHeaderParam(name = "my-header", value = "constant-header-value")
@ClientHeaderParam(name = "computed-header", value = "{org.acme.rest.client.Util.computeHeader}")
public interface ExtensionsService {

    @GET
    @ClientHeaderParam(name = "header-from-properties", value = "${header.value}")
    Set<Extension> getById(@QueryParam("id") String id, @HeaderParam("jaxrs-style-header") String
headerValue);
}

---
@ApplicationScoped
public class RequestUUIDHeaderFactory implements ClientHeadersFactory {

    @Override
    public MultivaluedMap<String, String> update(MultivaluedMap<String, String> incomingHeaders,
MultivaluedMap<String, String> clientOutgoingHeaders) {
        MultivaluedMap<String, String> result = new MultivaluedHashMap<>();
        result.add("X-request-uuid", UUID.randomUUID().toString());
        return result;
    }
}
```



Exceptions

La spécification MicroProfile REST Client introduit l'interface ***ResponseExceptionMapper*** dont le but est de convertir une réponse HTTP en une exception

```
public interface MyResponseExceptionMapper implements
    ResponseExceptionMapper<RuntimeException> {

    RuntimeException toThrowable(Response response) {
        if (response.getStatus() == 500) {
            throw new RuntimeException("Remote service responded with HTTP 500");
        }
        return null;
    }
}
```



Appliquer les mapper

Pour rendre disponible un `ResponseExceptionMapper` à chaque client REST de l'application, la classe doit être annotée avec **`@Provider`**

Si l'on veut la rendre disponible pour un client spécifique, il faut annoter l'interface du REST Client avec :
`@RegisterProvider(MyResponseExceptionMapper.class)`



@ClientExceptionHandler

Un moyen plus simple de convertir les codes d'erreur HTTP en exception est d'utiliser ***@ClientExceptionHandler***.

```
@Path("/extensions")
@registerRestClient
public interface ExtensionsService {
    @GET
    Set<Extension> getById(@QueryParam("id") String id);
    @GET
    CompletionStage<Set<Extension>> getByIdAsync(@QueryParam("id") String id);

    @ClientExceptionHandler
    static RuntimeException toException(Response response) {
        if (response.getStatus() == 500) {
            return new RuntimeException("Remote service responded with HTTP 500");
        }
        return null;
    }
}
```




Persistence

Configuration des sources de données

Hibernate et JPA

Panache

MongoDB



Zero Config Setup (Dev Services)

Lors des tests ou de l'exécution en mode développement, Quarkus peut vous fournir une base de données prête à l'emploi sans aucune configuration.

Il suffit de :

- inclure la bonne extension vers le type de base
- Ne pas déclarer les propriétés jdbc (URL, login, password)

Les bases supportées sont les bases embarquées (H2, HSQL, Derby) et certaines bases démarrant avec Docker (DB2, MariaDB, Microsoft SQL Server, MySQL, Oracle Express Edition, Postgres)



Datasource JDBC

Ajoutez l'extension ***agroal*** plus une parmi *jdbc-db2*, *jdbc-derby*, *jdbc-h2*, *jdbc-mariadb*, *jdbc-mssql*, *jdbc-mysql*, *jdbc-oracle* ou *jdbc-postgresql*.

Puis configurer :

```
quarkus.datasource.username=<your username>
quarkus.datasource.password=<your password>
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/hibernate_orm_test
quarkus.datasource.jdbc.max-size=16
```



Datasource réactive

Ajouter l'extension réactive parmi :
reactive-db2-client, reactive-mssql-client,
reactive-mysql-client, reactive-oracle-
client, ou reactive-pg-client.

Puis configurer :

```
quarkus.datasource.username=<your username>  
quarkus.datasource.password=<your password>  
quarkus.datasource.reactive.url=postgresql:///your_database  
quarkus.datasource.reactive.max-size=20
```



Persistence

Configuration des sources de données

Hibernate et JPA

Panache

MongoDB



Hibernate et JPA

Ajoute des extensions :

- io.quarkus:quarkus-hibernate-orm
- + le driver jdbc

Il suffit ensuite de :

- Spécifier les paramètres de configuration dans application.properties (persistence.xml seulement pour des propriétés avancées)
- Annotez les entités avec @Entity et autres annotations



Principales configuration Hibernate

quarkus.hibernate-orm.log.sql : Affiche les traces SQL

quarkus.hibernate-orm.log.format-sql : Formatte les traces SQL

quarkus.hibernate-orm.database.generation : Si Hibernate génère la base :
none, create, drop-and-create, drop, update, validate

quarkus.hibernate-orm.sql-load-script : Le script a exécuté au démarrage
(import.sql par défaut)

quarkus.hibernate-orm.second-level-caching-enabled : Activer le cache de
2nd niveau

quarkus.hibernate-orm.validate-in-dev-mode : Valide le schéma en mode
dev et affiche des messages de log si erreur

quarkus.hibernate-orm.database.charset : Le charset de la base de données

quarkus.hibernate-orm.jdbc.statement-fetch-size : Nombre de lignes
récupérées à la fois

quarkus.hibernate-orm.jdbc.statement-batch-size : Nombre d'updates
envoyés à la fois



Usage

```
@ApplicationScoped
public class SantaClausService {
    @Inject
    EntityManager em;

    @Transactional
    public void createGift(String giftDescription) {
        Gift gift = new Gift();
        gift.setName(giftDescription);
        em.persist(gift);
    }
}
```




Transactions

On peut définir les transactions :

- de manière déclarative avec `@Transactional`
- par programmation avec `QuarkusTransaction`.
- Ou directement l'API JTA `UserTransaction`



Méthode déclarative

Le moyen le plus simple : utiliser l'annotation **@Transactional** sur une méthode (javax.transaction.Transactional) d'un bean CDI (service ou endpoint REST)

- Si pas d'exception : commit
- Si RuntimeException : rollback

On peut également contrôler comment la transaction est démarré via les attributs de l'annotation :

- **REQUIRED** (défaut) : Se rattache à la transaction existante ou en crée une si il n'y a pas de encore transaction
- **REQUIRES_NEW** : Crée une nouvelle transaction, met en pause la transaction existante
- **MANDATORY** : Echoue si pas de transaction existante
- **SUPPORTS** : Rejoint la transaction existante ou rien si il n'y en a pas
- **NOT_SUPPORTED** : Si une transaction existante la suspend, travaille sans transaction
- **NEVER** : Echoue si une transaction existante, travaille sans transaction

L'annotation **@TransactionConfiguration** permet de faire des configuration avancée, en particulier positionner un timeout



QuarkusTransaction

Les méthodes statiques de QuarkusTransaction peuvent définir les limites des transactions.

2 approches :

- approche fonctionnelle qui permet d'exécuter un lambda dans le cadre d'une transaction
- Approche standard avec des méthodes explicites de début, de validation et de roll-back.



Exemple

```
public void beginExample() {  
    QuarkusTransaction.begin();  
    //do work  
    QuarkusTransaction.commit();  
  
    QuarkusTransaction.begin(QuarkusTransaction.beginOptions()  
        .timeout(10));  
    //do work  
    QuarkusTransaction.rollback();  
}
```



Example (2)

```
public void lambdaExample() {
    QuarkusTransaction.run(() -> {
        //do work
    });

    int result = QuarkusTransaction.call(QuarkusTransaction.runOptions()
        .timeout(10)
        .exceptionHandler((throwable) -> {
            if (throwable instanceof SomeException) {
                return RunOptions.ExceptionResult.COMMIT;
            }
            return RunOptions.ExceptionResult.ROLLBACK;
        })
        .semantic(RunOptions.Semantic.SUSPEND_EXISTING), () -> {
        //do work
        return 0;
    });
}
```



Hibernate Reactif

Il est possible d'utiliser le mode réactif d'Hibernate avec Quarkus.

Ajouter les extensions :

- `io.quarkus:quarkus-hibernate-reactive`
- Les clients réactifs SQL : `quarkus-reactive-pg-client`, `quarkus-reactive-mysql-client`, `quarkus-reactive-mssql-client` et `quarkus-reactive-db2-client`

Ensuite

- Configurer Hibernate
- Annoter les entités



Configuration

```
# datasource configuration
quarkus.datasource.db-kind = postgresql
quarkus.datasource.username = quarkus_test
quarkus.datasource.password = quarkus_test
```

```
quarkus.datasource.reactive.url =  
vertx-reactive:postgresql://localhost/quarkus_test
```

```
quarkus.hibernate-orm.database.generation=drop-and-create
```



Usage

Un objet ***Mutiny.SessionFactory*** est créé par Quarkus à partir de la configuration de la datasource.

```
@ApplicationScoped
public class SantaClausService {
    @Inject
    Mutiny.SessionFactory sf;

    public Uni<Void> createGift(String giftDescription) {
        Gift gift = new Gift();
        gift.setName(giftDescription);
        return sf.withTransaction(session -> session.persist(gift))
    }
}
```




Limitations

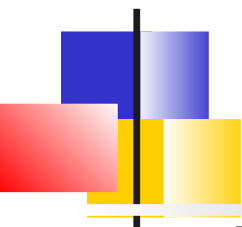
Quelques limitations :

- il n'est pas possible de configurer plusieurs unités de persistance pour le moment
- Pas de configuration via *persistence.xml* possible
- l'intégration avec l'extension Envers n'est pas prise en charge
- la démarcation des transactions ne peut pas être effectuée à l'aide de *javax.transaction.Transactional*



Persistence

Configuration des sources de données
Hibernate et JPA
Panache
MongoDB



Panache

Panache est une librairie spécifique à Quarkus qui simplifie le développement de la couche de persistance basée sur Hibernate. Il est semblable à Spring Data JPA

Panache implémente les patterns **Entity** et **Repository** pour fournir des méthodes pour créer, mettre à jour et supprimer des enregistrements, effectuer des requêtes de base et définir et exécuter ses propres requêtes.

Il est compatible avec Hibernate Réactif



Entity Pattern

Les classes entités étendent ***PanacheEntity*** et profite de toutes les méthodes CRUD. Elles peuvent définir d'autre query en utilisant des méthodes de PanacheEntity

```
@Entity
public class Person extends PanacheEntity {
    public String name;
    public LocalDate birth;
    public Status status;

    public static Uni<Person> findByName(String name){
        return find("name", name).firstResult();
    }

    public static Uni<List<Person>> findAlive(){
        return list("status", Status.Alive);
    }

    public static Uni<Long> deleteStefs(){
        return delete("name", "Stef");
    }
}
```



Usage

// Persister une personne

```
Person person = new Person();
```

```
...
```

```
person.persist();
```

// Supprimer

```
if(person.isPersistent()){
```

```
    person.delete();
```

```
}
```

```
List<Person> allPersons = Person.listAll();
```

```
person = Person.findById(personId);
```

```
Optional<Person> optional = Person.findByIdOptional(personId);
```

// Toutes les personnes vivantes

```
List<Person> livingPersons = Person.list("status", Status.Alive);
```

// Compter toutes les personnes vivantes

```
long countAlive = Person.count("status", Status.Alive);
```

// Supprimer toutes les personnes vivantes

```
Person.delete("status", Status.Alive);
```

// delete by id

```
boolean deleted = Person.deleteById(personId);
```

// Mise à jour

```
Person.update("name = 'Mortal' where status = ?1", Status.Alive);
```



RepositoryPattern

Définir des classes implémentant ***PanacheRepository<Entity>*** et définissant des requêtes supplémentaire

```
@ApplicationScoped
public class PersonRepository implements PanacheRepository<Person> {

    public Uni<Person> findByName(String name){
        return find("name", name).firstResult();
    }

    public Uni<List<Person>> findAlive(){
        return list("status", Status.Alive);
    }

    public Uni<Long> deleteStefs(){
        return delete("name", "Stef");
    }
}
```



Usage

```
@Inject
```

```
PersonRepository personRepository;
```

```
@GET
```

```
public long count(){  
    return personRepository.count();  
}
```



Persistence

Configuration des sources de données
Hibernate et JPA
Panache
MongoDB



Mise en place

Ajout de l'extension 'mongodb-client'

Configuration de la base

Injection de *MongoClient*

Ou utilisation de Panache



Configuration

La propriété principale à configurer est l'URL pour accéder à MongoDB¹

Exemples :

```
# Pour un simple instance sur localhost
quarkus.mongodb.connection-string = mongodb://localhost:27017
```

```
# Pour un ensemble répliqué de 2 noeuds
quarkus.mongodb.connection-string =
  mongodb://mongo1:27017,mongo2:27017
```

1. Toutes les configurations MongoDB peuvent se faire par l'URL.
Voir <https://docs.mongodb.com/manual/reference/connection-string/>



Sans configuration

Si l'URL de connexion n'est pas précisée,
un container docker est démarré dans
les mode dev et test



Usage de MongoClient

```
@ApplicationScoped
public class FruitService {

    @Inject MongoClient mongoClient;

    public List<Fruit> list(){
        List<Fruit> list = new ArrayList<>();
        MongoCursor<Document> cursor = getCollection().find().iterator();

        try {
            while (cursor.hasNext()) {
                Document document = cursor.next();
                Fruit fruit = new Fruit();
                fruit.setName(document.getString("name"));
                list.add(fruit);
            }
        } finally { cursor.close(); }
        return list;
    }

    public void add(Fruit fruit){
        Document document = new Document()
            .append("name", fruit.getName())
            getCollection().insertOne(document);
    }

    private MongoCollection getCollection(){
        return mongoClient.getDatabase("fruit").getCollection("fruit");
    }
}
```



Panache (Entity Pattern)

```
public class Person extends PanacheMongoEntity {  
    public String name;  
    public LocalDate birth;  
    public Status status;  
  
    public static Person findByName(String name){  
        return find("name", name).firstResult();  
    }  
  
    public static List<Person> findAlive(){  
        return list("status", Status.Alive);  
    }  
  
    public static void deleteLoics(){  
        delete("name", "Loïc");  
    }  
}
```



Messaging

Support pour le messaging
Quarkus et Apache Kafka



Introduction

Quarkus supporte de nombreuses alternatives pour le messaging

- EventBus
- JMS
- AMQP
- Kafka
- RabbitMQ



EventBus

Quarkus permet à différents beans d'interagir à l'aide d'événements asynchrones. 3 types d'interaction sont supportés :

- point à point : Un consommateur (éventuellement répliqué)
- pub/sub : Publier un message vers plusieurs consommateurs
- request/esponse : Envoi du message et attente d'une réponse. Le destinataire peut répondre au message de manière asynchrone

Nécessite l'extension vertx

Possibilité de s'échanger des Objets grâce aux codecs par défaut de Vert.x Event Bus



Consommateur

L'annotation **@ConsumeEvent** permet d'annoter la méthode à exécuter lors de la réception du message

Elle prend comme attributs :

- **value** : le nom de l'adresse virtuelle
- **blocking** : Si l'exécution du code est bloquante

Le paramètre d'entrée est soit :

- Message et on a accès à l'intégralité du message Vert.x (address, body, ...)
- Soit la payload

La valeur de retour de la méthode est utilisée comme réponse au message.

Si *void*, on implémente une interaction fire-and-forget

Si la méthode envoie une exception, celle-ci est peut être gérée par un handler côté producteur ou par le handler par défaut
`io.vertx.core.Vertx#exceptionHandler()`



Example

```
@ApplicationScoped
public class GreetingService {

    @ConsumeEvent(value = "blocking-consumer", blocking = true)
    public String consume(String name) {
        return name.toUpperCase();
    }
}
---
```

```
@ApplicationScoped
public class GreetingService {

    @ConsumeEvent
    public CompletionStage<String> consume(String name) {
        // return a CompletionStage completed when the processing is finished.
        // You can also fail the CompletionStage explicitly
    }

    @ConsumeEvent
    public Uni<String> process(String name) {
        // return an Uni completed when the processing is finished.
        // You can also fail the Uni explicitly
    }
}
```



Producteur

Le producteur utilise le bean

io.vertx.mutiny.core.eventbus.EventBus pour
envoyer des messages

Cet objet propose des méthodes pour :

- Envoyer d'un message à une adresse spécifique
`bus.sendAndForget("greeting", name)`
- Publier un message à une adresse spécifique - tous les consommateurs reçoivent les messages.
`bus.publish("greeting", name)`
- Envoyer un message et attendre une réponse
`Uni<String> response =
bus.<String>request("address", "hello, how are
you?").onItem().transform(Message::body);`



Example

```
@Path("/async")
public class EventResource {

    @Inject
    EventBus bus;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("{name}")
    public Uni<String> greeting(String name) {
        return bus.<String>request("greeting", name)
            .onItem().transform(Message::body);
    }
}
```



JMS

Une application Quarkus peut utiliser JMS via

- le client Apache Qpid JMS AMQP,
quarkus-qpid-jms
- ou le client Apache ActiveMQ Artemis JMS
quarkus-artemis-jms

Le standard JMS provenant de JakartaEE permet l'échange de messages entre processus Java via un broker. (Point 2 Point ou PubSub).

Quelques implémentations :

- Qpid JMS : Un bridge vers AMQP
- OpenMQ : Implémentation de référence
- Serveurs Jakarta EE



Exemple : Producteur

```
@ApplicationScoped
public class PriceProducer implements Runnable {

    @Inject
    ConnectionFactory connectionFactory;

    private final Random random = new Random();
    private final ScheduledExecutorService scheduler = Executors.newSingleThreadScheduledExecutor();

    void onStart(@Observes StartupEvent ev) {
        scheduler.scheduleWithFixedDelay(this, 0L, 5L, TimeUnit.SECONDS);
    }

    void onStop(@Observes ShutdownEvent ev) {
        scheduler.shutdown();
    }

    @Override
    public void run() {
        try (JMSContext context = connectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE)) {
            context.createProducer().send(context.createQueue("prices"),
Integer.toString(random.nextInt(100)));
        }
    }
}
```



Exemple : Consommateur

```
@ApplicationScoped
public class PriceConsumer implements Runnable {
    @Inject
    ConnectionFactory connectionFactory;

    private final ExecutorService scheduler = Executors.newSingleThreadExecutor();
    private volatile String lastPrice;

    public String getLastPrice() { return lastPrice; }

    void onStart(@Observes StartupEvent ev) { scheduler.submit(this); }

    void onStop(@Observes ShutdownEvent ev) { scheduler.shutdown(); }

    @Override
    public void run() {
        try (JMSContext context = connectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE)) {
            JMSConsumer consumer = context.createConsumer(context.createQueue("prices"));
            while (true) {
                Message message = consumer.receive();
                if (message == null) return;
                lastPrice = message.getBody(String.class);
            }
        } catch (JMSEException e) {
            throw new RuntimeException(e);
        }
    }
}
```



Kafka

Apache Kafka est une plate-forme de streaming d'événements distribués.

- Couramment utilisé pour les pipelines de données hautes performances, l'analyse de flux, l'intégration de données et les applications critiques.

Semblable à un message broker, il permet :

- De publier et s'abonner à des flux d'événements, appelés **records**.
- stocker des flux d'enregistrements de manière durable et fiable dans des **topics**.
- traiter les flux d'enregistrements **au fur et à mesure qu'ils se produisent ou rétrospectivement**.



Quarkus et Kafka

Quarkus supporte Apache Kafka via le framework ***SmallRye Reactive Messaging***.

Extension : *smallrye-reactive-messaging-kafka*

Basé sur la spécification ***Eclipse MicroProfile Reactive Messaging 2.0***, il propose un modèle de programmation flexible reliant CDI et event-driven.



Smallrye Reactive Messaging

Smallrye Reactive Messaging supporte différents brokers
(*Apache Kafka, AMQP, Apache Camel, JMS, MQTT, etc.*)

=> Il utilise donc un vocabulaire générique :

- Les applications échangent des **messages** : Un message contient les données utiles (payload) et des méta-données. Equivalent à record dans Kafka
- Les messages transitent dans des **canaux** (channels). Les applications se connectent aux canaux. Equivalent à topic
- Channels sont connecté aux système de messagerie sous-jacent via des **connecteurs (connector)**. Les connecteurs sont configurés pour mapper les messages entrants sur un canal spécifique et collecter les messages sortants envoyés à un canal spécifique.
Chaque connecteur est dédié à une technologie de messagerie spécifique et est identifié par un mot-clé prédéfini



Dev Services

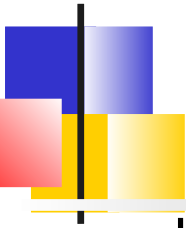
Si une extension liée à Kafka est présente , Dev Services démarre automatiquement un broker Kafka en mode *dev* et *test*.

=> L'application est configurée automatiquement.

Nécessite Docker

Quelques points de configuration du Dev Services :

- ***quarkus.kafka.devservices.enabled*** : true/false
- Si ***kafka.bootstrap.servers*** est configuré, Dev Services n'est pas démarré
- ***quarkus.kafka.devservices.shared*** : true/false. Indique si le mécanisme de partage automatique du broker (entre application producteur et consommateur) est activé
- ***quarkus.kafka.devservices.port*** : Fixer le port d'écoute. Par défaut aléatoire
- ***quarkus.kafka.devservices.image-name*** : Le nom de l'image. Quarkus supporte Redpanda et Strimzi
- ***quarkus.kafka.devservices.topic-partitions.<topic>*** : Permet d'initialiser des topics avec un nombre de partitions



Configuration minimale Kafka

Une configuration minimale pourrait être :

```
%prod.kafka.bootstrap.servers=kafka:9092
```

```
mp.messaging.incoming.prices.connector=smallrye-kafka
```

- Configure dans le profil de *prod*, l'adresse d'un broker kafka faisant partie d'un cluster.
On peut indiquer plusieurs adresses
- Configuration du connecteur Kafka gérant le canal *prices* (par défaut le canal a le même nom que le topic Kafka)
Si un seul connecteur est dans le classpath, cette configuration est optionnelle

Sur un channel, de nombreuses options de configuration sont disponibles selon la syntaxe :

```
mp.messaging.incoming.your-channel-name.attribute=value
```



Réception de message

La réception de message s'effectue en annotant une méthode avec **@Incoming** et en spécifiant le nom du topic.

La méthode peut récupérer via son argument :

- La charge utile (payload)
- Le message complet avec Message
- Le record avec ConsumerRecord ou Record

Il est également possible de se faire injecter un Multi représentant le stream de message



Example

```
@ApplicationScoped
public class PriceConsumer {
    @Incoming("prices")
    public void consume(double price) { // process your price. }
    @Incoming("prices")
    public CompletionStage<Void> consume(Message<Double> msg) {
        var metadata = msg.getMetadata(IncomingKafkaRecordMetadata.class).orElseThrow();
        double price = msg.getPayload();
        // Ack manuel (commit l'offset)
        return msg.ack();
    }
    @Incoming("prices")
    public void consume(ConsumerRecord<String, Double> record) {
        String key = record.key();
        String value = record.value();
        String topic = record.topic();
        int partition = record.partition();
    }
    @Incoming("prices")
    public void consume(Record<String, Double> record) {
        String key = record.key();
        String value = record.value();
    }
}
```



Exemple : Injection de channel

```
@Path("/prices")
public class PriceResource {

    @Inject
    @Channel("prices")
    Multi<Double> prices;

    @GET
    @Path("/prices")
    @Produces(MediaType.SERVER_SENT_EVENTS)
    @RestStreamElementType(MediaType.TEXT_PLAIN)
    public Multi<Double> stream() {
        return prices;
    }
}
```

Les autres types possibles sont :

```
@Inject @Channel("prices") Multi<Double> streamOfPayloads;
@Inject @Channel("prices") Multi<Message<Double>> streamOfMessages;
@Inject @Channel("prices") Publisher<Double> publisherOfPayloads;
@Inject @Channel("prices") Publisher<Message<Double>> publisherOfMessages;
```



Traitement bloquant

L'appel de la méthode de réception est effectuée par une thread d'I/O, si l'on effectue un traitement bloquant, il faut le signaler au framework via :

- ***io.smallrye.reactive.messaging.annotations.Blocking***
- ***io.smallrye.common.annotation.Blocking***
- ***@Transactional*** a le même effet

```
@ApplicationScoped
public class PriceStorage {
    @Incoming("prices")
    @Transactional
    public void store(int priceInUsd) {
        Price price = new Price();
        price.value = priceInUsd;
        price.persist();
    }
}
```




Stratégies de ack

Tous les messages reçus par un consommateur doivent être acquittés.

- Si la méthode reçoit un *Record* ou la payload, le message sera acquitté au retour de la méthode
- Si la méthode renvoie un autre flux réactif ou *CompletionStage*, le message sera acquitté lorsque le message en aval sera acquitté
- Si la méthode reçoit un *Message*, l'acquittement est manuel

On peut surcharger le comportement par défaut en utilisant l'annotation **@Acknowledgment** pour :

- Accuser réception du message à l'arrivée
@Acknowledgment(Acknowledgment.Strategy.PRE_PROCESSING)
- Ou ne pas accuser réception du message du tout sur la méthode consommateur
@Acknowledgment(Acknowledgment.Strategy.NONE)



Stratégie de commit

Lorsqu'un message est acquitté, le connecteur invoque une stratégie de commit.

3 stratégies sont possibles :

- ***throttled*** : Commit à intervalle régulier.
Garantie *at-least-once* même si le canal effectue un traitement asynchrone.
- ***latest*** : Commit du dernier ack.
Garantie *at-least-once* si le canal effectue un traitement synchrone
- ***ignore*** : Délègue le commit au client Kafka sous-jacent.
Ne garantie pas *at-least-once*

Attribut de configuration :

`mp.messaging.incoming.<channel>.commit-strategy`



Stratégies de gestion d'erreur

Si le message n'est pas acquitté, une stratégie de gestion d'erreur est appliquée.

La propriété de configuration failure-strategy peut prendre 3 valeurs :

- **fail** : Faire échouer l'application, plus aucun enregistrement ne sera traité (stratégie par défaut).
L'offset de l'enregistrement qui n'a pas été traité correctement n'est pas committé.
- **ignore** : l'échec est loggé, mais le traitement continue.
L'offset de l'enregistrement qui n'a pas été traité correctement est committé.
- **dead-letter-queue** : l'offset de l'enregistrement qui n'a pas été traité correctement est committé, mais l'enregistrement est écrit dans un topic spécifique de Kafka.
D'autres configurations sont possible sur le topic dead-letter



Groupe de consommateurs

Avec Kafka, un groupe de consommateurs est un ensemble de consommateurs qui coopèrent pour consommer les données d'un topic.

- Un topic est divisé en partitions.
- Les partitions d'un topic sont attribuées parmi les consommateurs du groupe
Chaque partition est affectée à un seul consommateur d'un groupe. Un consommateur peut être affecté à plusieurs partitions si le nombre de partitions est supérieur au nombre de consommateurs dans le groupe.



Patterns

1. Une thread unique d'une application unique
 - Comportement par défaut
 - L'ID du groupe est le nom de l'application (***quarkus.application.name***) ou défini par ***kafka.group.id***
2. Plusieurs threads d'une même application
 - La propriété ***mp.messaging.incoming.\$channel.partitions*** fixe le nombre de threads
Si elle dépasse le nombre de partition du topic, certaines threads ne seront pas affectées
3. Plusieurs applications consommatrice ayant le même groupe
 - La propriété ***mp.messaging.incoming.\$channel.group.id*** est alors identique pour toutes les applications
 - Elles peuvent également être multi-thread



Traitement rétrospectif

Une des fonctionnalités appréciées de Kafka est de pouvoir recevoir les messages à posteriori.

Il suffit de

- Choisir identifiant de groupe de consommateurs qui n'est utilisé par aucune autre application.
- De positionner la propriété :
auto.offset.reset = earliest



Configuration pour l'envoi

La configuration des canaux sortants du connecteur Kafka est similaire à celle des canaux entrants.

Exemple :

```
%prod.kafka.bootstrap.servers=kafka:9092  
mp.messaging.outgoing.prices-out.connector=smallrye-kafka  
mp.messaging.outgoing.prices-out.topic=prices
```



Envoi de message

Via l'annotation ***@Outgoing***, une méthode peut publier un message vers un *topic*

```
@ApplicationScoped
public class KafkaPriceProducer {

    private final Random random = new Random();

    @Outgoing("prices-out")
    public Multi<Double> generate() {
        // Build an infinite stream of random prices
        // It emits a price every second
        return Multi.createFrom().ticks().every(Duration.ofSeconds(1))
            .map(x -> random.nextDouble());
    }
}
```




Types d'envoi

Comme pour la réception, il est possible également d'envoyer des *Record* ou *Message*

```
@Outgoing("out")
public Multi<Record<String, Double>> generate() {
    return Multi.createFrom().ticks().every(Duration.ofSeconds(1))
        .map(x -> Record.of("my-key", random.nextDouble()));
}
```

```
@Outgoing("generated-price")
public Multi<Message<Double>> generate() {
    return Multi.createFrom().ticks().every(Duration.ofSeconds(1))
        .map(x -> Message.of(random.nextDouble()))
        .addMetadata(OutgoingKafkaRecordMetadata.<String>builder()
            .withKey("my-key")
            .withTopic("my-key-prices")
            .withHeaders(new RecordHeaders().add("my-header", "value".getBytes()))
            .build());
}
```



Types d'envoi (2)

Les méthodes d'envoi peuvent également ne produire qu'un seul message :

```
@Outgoing("prices-out") T generate(); // T excluding void
@Outgoing("prices-out") Message<T> generate();
@Outgoing("prices-out") Uni<T> generate();
@Outgoing("prices-out") Uni<Message<T>> generate();
@Outgoing("prices-out") CompletionStage<T> generate();
@Outgoing("prices-out") CompletionStage<Message<T>> generate();
```



Envoi avec *Emitter*

L'autre façon d'envoyer des messages est de se faire injecter par le framework un bean ***Emitter***.

L'envoi retourne un *CompletionStage*, terminé lorsque le message est acquitté.

```
@Path("/prices")
public class PriceResource {

    @Inject
    @Channel("price-create")
    Emitter<Double> priceEmitter;

    @POST
    @Consumes(MediaType.TEXT_PLAIN)
    public void addPrice(Double price) {
        // Exception si nack
        CompletionStage<Void> ack = priceEmitter.send(price);
    }
}
```



Envoi *Message*

Emitter peut utiliser le type **Message**. Cela permet de traiter les cas *ack* et *nack* différemment

```
@Path("/prices")
public class PriceResource {

    @Inject @Channel("price-create") Emitter<Double> priceEmitter;

    @POST
    @Consumes(MediaType.TEXT_PLAIN)
    public void addPrice(Double price) {
        priceEmitter.send(Message.of(price)
            .withAck(() -> {
                // Called when the message is acked
                return CompletableFuture.completedFuture(null);
            })
            .withNack(throwable -> {
                // Called when the message is nacked
                return CompletableFuture.completedFuture(null);
            }));
    }
}
```



Processor

Un ***processor***¹ dans les architecture *event-driven* est un micro-service qui lit un *topic* en entrée, effectue un traitement et écrit sur un *topic* de sortie.

Il peuvent être facilement mis en place avec Quarkus

```
@ApplicationScoped
public class PriceProcessor {

    private static final double CONVERSION_RATE = 0.88;

    @Incoming("price-in")
    @Outgoing("price-out")
    public double process(double price) {
        return price * CONVERSION_RATE;
    }
    // Version asynchrone
    @Incoming("price-in")
    @Outgoing("price-out")
    public Multi<Double> process(Multi<Integer> prices) {
        return prices.filter(p -> p > 100).map(p -> p * CONVERSION_RATE);
    }
}
```



Sécurité

Introduction
Authentication
OpenId



Introduction

Quarkus Security Manager est construit sur 3 interfaces :

- ***HttpAuthenticationMechanism*** responsable d'extraire les informations d'identification d'authentification de la requête HTTP
Il délègue à :
- ***IdentityProvider*** pour convertir les crédeniels en informations d'identification
- ***SecurityIdentity*** : Contenant le principal et les rôles du client

Quarkus Security est hautement personnalisable. On peut fournir ses propres *HttpAuthenticationMechanisms*, *IdentityProviders* et *SecurityidentityAugmentors*.



Authentication

Quarkus supporte différents mécanismes d'authentification :

- Basic et Form HTTP-based authentication. (Transmission d'un logi/mot de passe dans une entête http ou dans un POST de formulaire)
- TLS mutuel : Basé sur des certificats TLS/SSL (client et serveur)
- OpenID Connect : Délégation de l'authentification à un fournisseur de jeton. (Adaptation de oAuth2)
Plusieurs extensions sont disponibles selon les cas d'usage
- LDAP

Les mécanismes d'authentification peuvent être combinés



Autorisation

Quarkus permet de mettre des contraintes d'accès sur les endpoints web

Cela s'effectue :

- Via la configuration
- Via les annotations *@RolesAllowed*, *@DenyAll*, *@PermitAll*



Autorisation Web

Si la sécurité est activée, toutes les requêtes HTTP feront l'objet d'une vérification des autorisations

2 types de vérifications sont effectuées

- Les autorisations définies dans la configuration
- Les autorisations spécifiées par les annotations



Exemple configuration

Déclaration d'une role-policy : *role-policy1*

quarkus.http.auth.policy.role-policy1.roles-allowed=user,admin

Ensemble de path sous le nom roles1

quarkus.http.auth.permission.roles1.paths=/roles-secured/*,/other/*,/api/*

Les path roles1 sont accessible par les rôles user et admin

quarkus.http.auth.permission.roles1.policy=role-policy1

Ensemble de path sous le nom permit1

quarkus.http.auth.permission.permit1.paths=/public/*

permit1 limité aux méthodes GET

quarkus.http.auth.permission.permit1.methods=GET

Accès à tout le monde aux URLs de permit1

quarkus.http.auth.permission.permit1.policy=permit

Accès refusé à tout le monde aux URLs de deny1

quarkus.http.auth.permission.deny1.paths=/forbidden

quarkus.http.auth.permission.deny1.policy=deny



Priorité des contraintes

Correspondance de plusieurs chemins :
le chemin le plus long l'emporte

Correspondance de plusieurs chemins :
la méthode la plus spécifique l'emporte

Correspondance de plusieurs chemins et
méthodes : les permissions s'ajoutent



Autorisation via annotations

```
@Path("subject")
public class SubjectExposingResource {

    @GET
    @Path("secured")
    @RolesAllowed("Tester")
    public String getSubjectSecured(@Context SecurityContext sec) {
        Principal user = sec.getUserPrincipal();
        return user != null ? user.getName() : "anonymous";
    }

    @GET
    @Path("unsecured")
    @PermitAll
    public String getSubjectUnsecured(@Context SecurityContext sec) {
        Principal user = sec.getUserPrincipal();
        return user != null ? user.getName() : "anonymous";
    }

    @GET
    @Path("denied")
    @DenyAll
    public String getSubjectDenied(@Context SecurityContext sec) {
        Principal user = sec.getUserPrincipal();
        return user != null ? user.getName() : "anonymous";
    }
}
```



Sécurité

Introduction
Authentication HTTP/S
OpenId



Activation SSL

Les mécanismes d'authentification transmettent des informations sensibles via le réseau. Il est nécessaire d'activer SSL

Pour activer SSL, il suffit de fournir dans la configuration :

- Un fichier certificat et une clé
- Un accès à un keystore

Exemple keystore :

```
quarkus.http.ssl.certificate.key-store-file=/path/to/keystore
quarkus.http.ssl.certificate.key-store-password=your-password
```

En général, on désactive également le port HTTP avec la propriété ***quarkus.http.insecure-requests***

- *enabled*
- *redirect* : Redirection http vers https
- *disabled*



Authentication HTTP

Basic Authentication :

- *quarkus.http.auth.basic=true*

Form-based

- *quarkus.http.auth.form.enabled=true*
- *quarkus.http.auth.form.login-page=...*
- *quarkus.http.auth.form.landing-page*
- ...

A noter que Quarkus n'utilise pas la session pour stocker l'utilisateur authentifié mais un cookie spécifique crypté avec la clé *quarkus.http.auth.session.encryption-key*

Il faut ensuite définir où sont stockés les utilisateurs avec une extension qui fournit un *IdentityProvider*



Configuration TLS mutuel

Une fois SSL activé, on peut mettre en place l'authentification TLS du client, en ajoutant un *truststore* à l'application :

```
quarkus.http.ssl.certificate.key-store-file=server-keystore.jks
quarkus.http.ssl.certificate.key-store-password=the_key_store_secret
quarkus.http.ssl.certificate.trust-store-file=server-truststore.jks
quarkus.http.ssl.certificate.trust-store-password=the_trust_store_secret
quarkus.http.ssl.client-auth=required
```



Identity Provider

Les mécanismes de base supportés pour le stockage des utilisateurs et de leurs rôles sont :

- Fichier properties : Dev/Test
 - Fichiers spécifique
 - Embarqué dans application.properties
- BD : Configuration JPA ou JDBC
- LDAP



Fichier *.properties*

Extension : ***quarkus-elytron-security-properties-file***

Fichier spécifique :

```
quarkus.security.users.file.enabled=true
quarkus.security.users.file.users=test-users.properties
quarkus.security.users.file.roles=test-roles.properties
quarkus.security.users.file.realm-name=MyRealm
quarkus.security.users.file.plain-text=true
```

test-users.properties :

```
scott=jb0ss
jdoe=p4ssw0rd
stuart=test
noadmin=n0Adm1n
```

test-role.properties

```
scott=Admin,admin,Tester,user
jdoe=NoRolesUser
stuart=admin,user
noadmin=user
```



Exemple embarqué

Dans *application.properties* :

```
quarkus.security.users.embedded.enabled=true
quarkus.security.users.embedded.plain-text=true
quarkus.security.users.embedded.users.scott=jb0ss
quarkus.security.users.embedded.users.stuart=test
quarkus.security.users.embedded.users.jdoe=p4ssw0rd
quarkus.security.users.embedded.users.noadmin=n0Adm1n
quarkus.security.users.embedded.roles.scott=Admin,admin,Tester,user
quarkus.security.users.embedded.roles.stuart=admin,user
quarkus.security.users.embedded.roles.jdoe=NoRolesUser
quarkus.security.users.embedded.roles.noadmin=user
```



Sécurité avec JPA

Extension : ***quarkus-security-jpa***

```
@Entity
@Table(name = "test_user")
@UserDefinition
public class User extends PanacheEntity {
    @Username
    public String username;
    @Password
    public String password;
    @Roles // Liste de rôles séparés par virgule
    public String role;

    public static void add(String username, String password, String role) {
        User user = new User();
        user.username = username;
        user.password = BcryptUtil.bcryptHash(password);
        user.role = role;
        user.persist();
    }
}
```



Sécurité avec JDBC

Extension : ***quarkus-elytron-security-jdbc***

Configuration :

```
quarkus.security.jdbc.enabled=true
quarkus.security.jdbc.principal-query.sql=SELECT u.password, u.role FROM
    test_user u WHERE u.username=?
quarkus.security.jdbc.principal-query.clear-password-mapper.enabled=true
quarkus.security.jdbc.principal-query.clear-password-mapper.password-index=1
# Autres propriétés comme algorithme de hash
quarkus.security.jdbc.principal-query.attribute-mappings.0.index=2
quarkus.security.jdbc.principal-query.attribute-mappings.0.to=groups
```



Sécurité avec LDAP

Extension : ***elytron-security-ldap***

Configuration

```
quarkus.security.ldap.enabled=true
```

```
quarkus.security.ldap.dir-context.principal=uid=tool,ou=accounts,o=YourCompany,c=DE
```

```
quarkus.security.ldap.dir-context.url=ldaps://ldap.server.local
```

```
quarkus.security.ldap.dir-context.password=PASSWORD
```

```
quarkus.security.ldap.identity-mapping.rdn-identifiant=uid
```

```
quarkus.security.ldap.identity-mapping.search-base-dn=ou=users,ou=tool,o=YourCompany,c=DE
```

```
quarkus.security.ldap.identity-mapping.attribute-mappings."0".from=cn
```

```
quarkus.security.ldap.identity-mapping.attribute-mappings."0".to=groups
```

```
quarkus.security.ldap.identity-mapping.attribute-mappings."0".filter=(member=uid={0})
```

```
quarkus.security.ldap.identity-mapping.attribute-mappings."0".filter-base-  
dn=ou=roles,ou=tool,o=YourCompany,c=DE
```



Sécurité

Introduction
Authentication HTTP/S
OpenId



Introduction

OpenID et OAuth2 peuvent être utilisés dans différents cas d'usage

- Authentification d'une application web
- Protection de ressources REST
- Authentification de micro-service

Dans tous ces cas, on s'appuie sur un serveur d'autorisation fournissant des jetons (d'authentification, d'accès ou de rafraîchissement)

Le client (Application web, micro-service ou autre) doit s'enregistrer auprès du fournisseur et doit utiliser un mécanisme d'obtention du jeton en fonction du grant type



Extensions

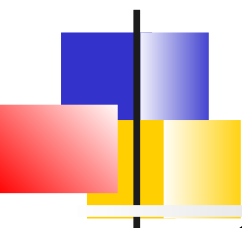
Différentes extensions Quarkus peuvent être utilisées en fonction des usages :

- ***quarkus-oidc*** : fournit un adaptateur OpenID supportant :
 - Un mécanisme Bearer Token qui extrait le token de l'entête HTTP Authorization
 - Authorization Code Flow : Redirection de l'utilisateur vers le fournisseur de jeton pour retourner un code d'autorisation permettant à l'application d'obtenir un jeton d'accès et de rafraîchissement
Les jetons sont au format JWT et peuvent être validés via un clé JWK
- ***quarkus-oidc-client*** : Permet d'obtenir des tokens d'accès et de rafraîchissement auprès de fournisseur supportant les grant type : *client-credentials*, *password* et *refresh_token*
- ***quarkus-oidc-client-filter*** : dépend de *quarkus-oidc-client* qui positionne le jeton obtenu par oidc-client dans l'entête d'Authorization pour faire des appels REST vers des backend
- ***quarkus-oidc-token-propagation*** : dépend de *quarkus-oidc* qui positionne le jeton obtenu par oidc dans l'entête d'Authorization. Ce filter est appliqué sur les RestClient pour propager un jeton d'accès aux services en aval



Extensions (2)

- ***quarkus-smallrye-jwt*** : fournit une implémentation de Microprofile JWT 1.1.1 et pour vérifier les jetons JWT signés et chiffrés et les représenter sous la forme *org.eclipse.microprofile.jwt.JsonWebToken*. C'est une alternative au mécanisme de Bearer Token Authentication de *quarkus-oidc*
- ***quarkus-elytron-security-oauth2*** : fournit une alternative au mécanisme *Bearer Token Authentication quarkus-oidc*. Il est basé sur Elytron



DevServices et UI KeyCloak

Quarkus propose une auto-configuration pour KeyCloak, la configuration est visible dans la DevUI

Activé si

- *quarkus-oidc*
- dev et test mode
- La propriété `quarkus.oidc.auth-server-url` n'est pas configuré

Le service démarre un container Keycloak et l'initialise en important ou créant un realm.

Le realm peut-être spécifié par :

`quarkus.keycloak.devservices.realm-path=quarkus-realm.json`

De plus, la Dev UI (/q/dev) permet d'acquérir les jetons de Keycloak et de tester l'application Quarkus.



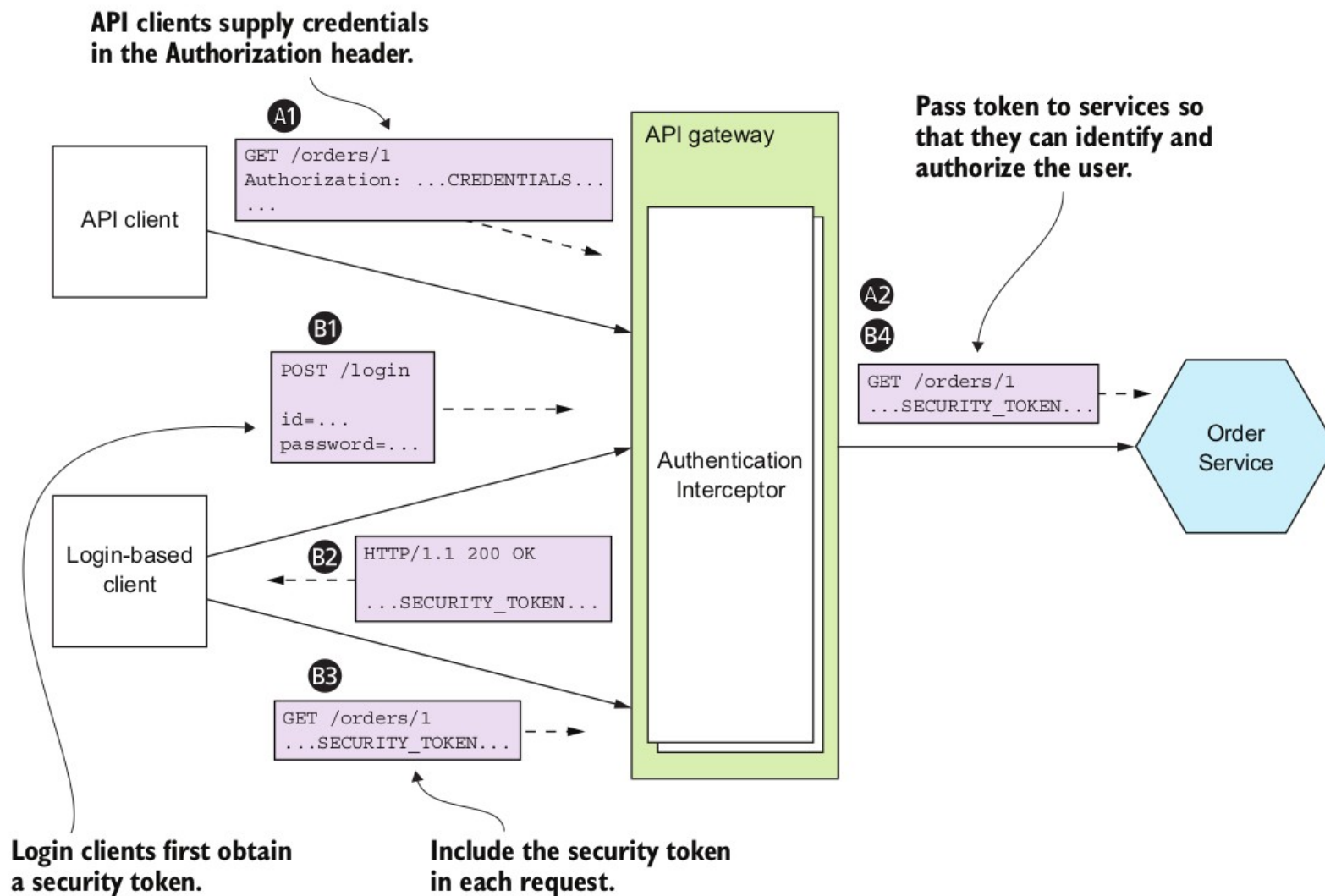
Sécurité micro-services

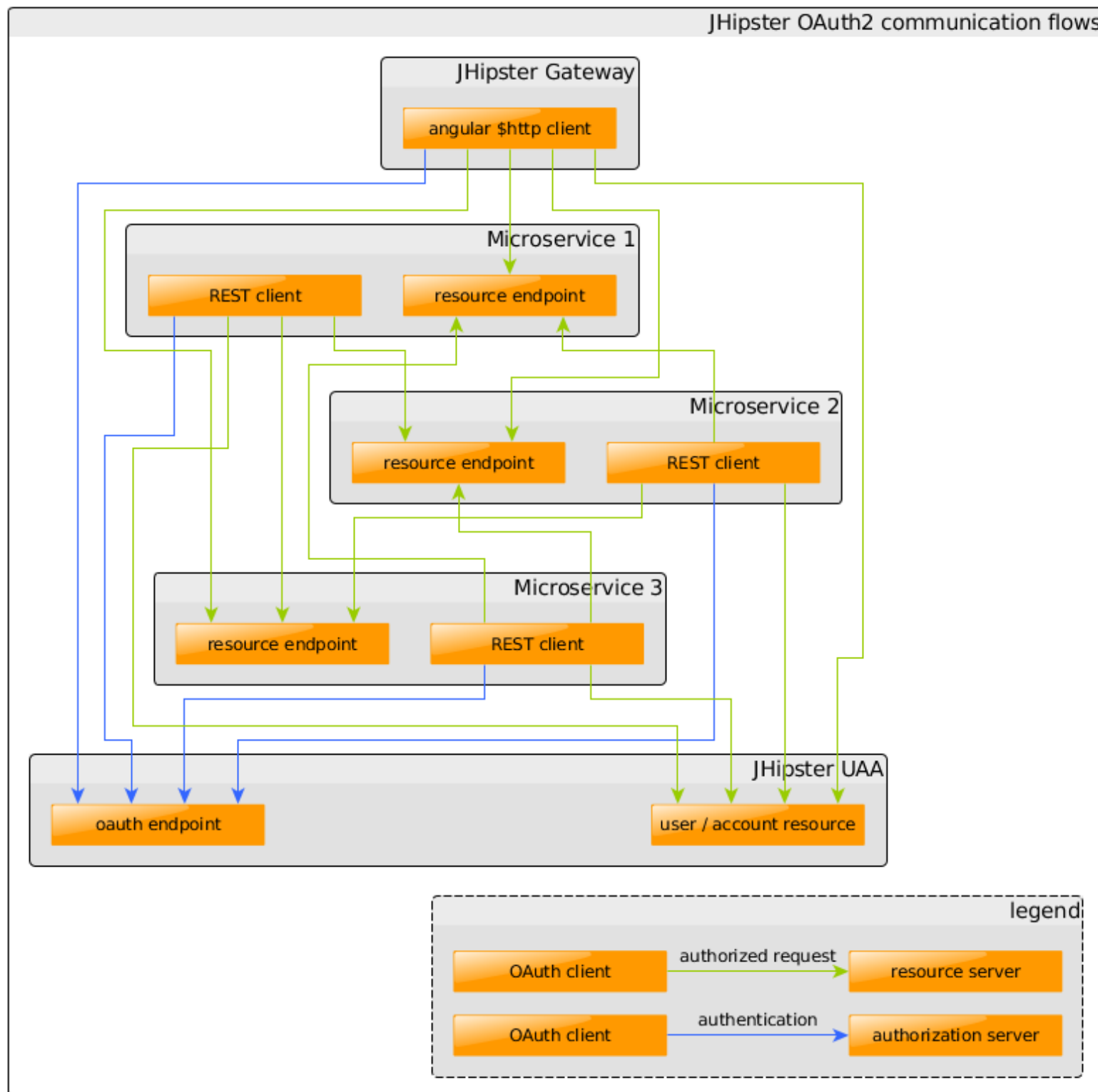
Plusieurs approches pour sécuriser une architecture micro-services :

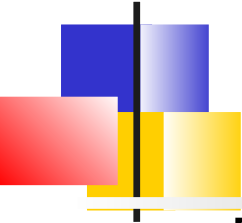
- N'implémenter la sécurité qu'au niveau de la gateway proxy. Les micro-services back-end ne sont protégés que par le firewall
- **Access token Pattern**¹ : La gateway passe un jeton contenant l'information sur le user (identité et rôles) Chaque micro-service a alors sa propre politique de sécurité.
- Chaque micro-service a sa propre politique de sécurité et chaque micro-service demande son propre jeton pour effectuer ses appels REST vers ses dépendances

1. <http://microservices.io/patterns/security/access-token.html>

Access Token Pattern







Protection des services via le mécanisme bearer token

Dans ce scénario,

- 1) un utilisateur essaie d'accéder à une ressource protégée
- 2) Une application (framework Javascript par exemple) lui propose un formulaire permettant de saisir son login mot de passe
- 3) L'application Javascript utilise les mots de passe utilisateur pour obtenir un jeton d'accès (grant type password)
- 4) Elle essaie ensuite d'accéder à la ressource en utilisant son jeton d'accès

Ce scénario pourrait s'appliquer à la gateway d'une architecture micro-service

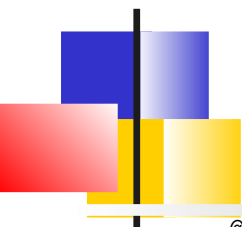


Mise en place

Extension *oidc*

Configuration

```
%prod.quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus  
# La configuration de dev utilise DevServices  
quarkus.oidc.client-id=backend-service  
quarkus.oidc.client-secret=secret
```



Exemple Service REST

```
@Path("/api/users")
public class UsersResource {

    @Inject
    SecurityIdentity securityIdentity;

    @GET
    @Path("/me")
    @RolesAllowed("user")
    @NoCache
    public User me() {
        return new User(securityIdentity);
    }

    public static class User {

        private final String userName;

        User(SecurityIdentity securityIdentity) {
            this.userName = securityIdentity.getPrincipal().getName();
        }

        public String getUserName() {
            return userName;
        }
    }
}
```



Test

Les jetons peuvent alors être obtenu
via :

```
curl --insecure -X POST  
  http://localhost:34735/realms/quarkus/protocol/openid-connect/token \  
  --user quarkus-app:secret \  
  -H 'content-type: application/x-www-form-urlencoded' \  
  -d 'username=alice&password=alice&grant_type=password'
```

Ou en utilisant la DevUI qui propose un
lien vers la page de login keycloak
(authorization code grant type)



Usage du jeton d'accès

Le Jeton d'accès est alors utilisé pour effectuer des requêtes vers le service rest :

```
curl -v -X GET \  
  http://localhost:8080/api/users/me \  
  -H "Authorization: Bearer "$access_token
```



Client REST utilisant les jetons OAuth2

Un client REST voulant accéder à une ressource protégée par OAuth2 doit utiliser un jeton.

2 scénarios

- Un client obtient son propre jeton et l'utilise dans ses appels rest :
oidc-client + oidc-client-reactive-filter ou oidc-client-filter
un micro-service accédant à un autre micro-service
- Le token courant (obtenu par oidc) et propagé
oidc-token-propagation ou oidc-token-propagation-reactive
Cas de la gateway par exemple



OidClient

L'extension ***quarkus-oidc-client*** fournit un *OidcClient* réactif qui peut être utilisé pour acquérir et actualiser des jetons à l'aide de SmallRye Mutiny Uni et Vert.x WebClient.

OidcClient est initialisé au build avec l'**URL du fournisseur de jeton** (qui peut être découverte automatiquement ou configurée manuellement)

Il utilise ce endpoint pour acquérir des jetons d'accès à l'aide des grant type ***client_credentials*** (défaut) ou ***password*** et actualiser les jetons à l'aide du grant type ***refresh_token***.



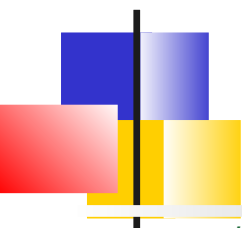
Configuration du endpoint

Découverte automatique

```
quarkus.oidc-client.auth-server-url=  
http://localhost:8180/auth/realms/quarkus
```

Découverte manuelle

```
quarkus.oidc-client.discovery-enabled=false  
quarkus.oidc-client.token-path=http://localhost:8180/auth/  
realms/quarkus/protocol/openid-connect/tokens
```



Configuration du grant type

grant type client_credentials

```
quarkus.oidc-client.auth-server-url=http://localhost:8180/auth/realms/quarkus/  
quarkus.oidc-client.client-id=quarkus-app  
quarkus.oidc-client.credentials.secret=secret
```

grant type password

```
quarkus.oidc-client.client-id=quarkus-app  
quarkus.oidc-client.credentials.secret=secret  
quarkus.oidc-client.grant.type=password  
quarkus.oidc-client.grant-options.password.username=alice  
quarkus.oidc-client.grant-options.password.password=alice
```




Utilisation dans le RestClient

Extensions : ***quarkus-oidc-client-reactive-filter*** ou ***quarkus-oidc-client-filter***

Les extensions fournissent ***OidcClientRequestReactiveFilter*** ou ***OidcClientRequestFilter*** qui peuvent être appliqués à une interface RestClient

```
@RegisterRestClient
@RegisterProvider(OidcClientRequestReactiveFilter.class)
@Path("/")
public interface ProtectedResourceService {

    @GET
    Uni<String> getUsername();
}
```



Propagation de jeton

L'extension ***quarkus-oidc-token-propagation*** fournit 2 implémentations de *ClientRequestFilter* qui simplifient la propagation des informations d'authentification

- ***AccessTokenRequestFilter*** propage le jeton Bearer présent dans la requête en cours ou le jeton acquis d'un code d'autorisation
- ***JsonWebTokenRequestFilter*** fournit la même fonctionnalité, mais fournit en plus la prise en charge des jetons JWT

Cette extension est typiquement utilisée :

- Pour propager le jeton venant d'être obtenu par l'Authorization Code Flow.
Ex : requête initiale Gateway
- Pour propager le jeton présent dans le Bearer
le même jeton circule dans toute l'architecture micro-services



Enregistrement du filtre AccessToken

```
@RegisterRestClient
@AccessToken
@Path("/")
public interface ProtectedResourceService {
```

```
    @GET
    String getUsername();
}
```

Ou

```
@RegisterRestClient
@RegisterProvider(AccessTokenRequestFilter.class)
@Path("/")
public interface ProtectedResourceService {
```

```
    @GET
    String getUsername();
}
```

Ou automatiquement si configuration suivante :

```
quarkus.oidc-token-propagation.register-filter=true
quarkus.oidc-token-propagation.json-web-token=false
```



Enregistrement JsonWebTokenFilter

`@RegisterRestClient`

`@JsonWebToken`

`@Path("/")`

`public interface ProtectedResourceService {`

`@GET`

`String getUsername();`

`}`

Ou

`@RegisterRestClient`

`@RegisterProvider(JsonWebTokenRequestFilter.class)`

`@Path("/")`

`public interface ProtectedResourceService {`

`@GET`

`String getUsername();`

`}`

Ou automatiquement si ces 2 propriétés sont positionnées à true

`quarkus.oidc-token-propagation.register-filter`

`quarkus.oidc-token-propagation.json-web-token`















Annexes

oAuth2



Rôles du protocole

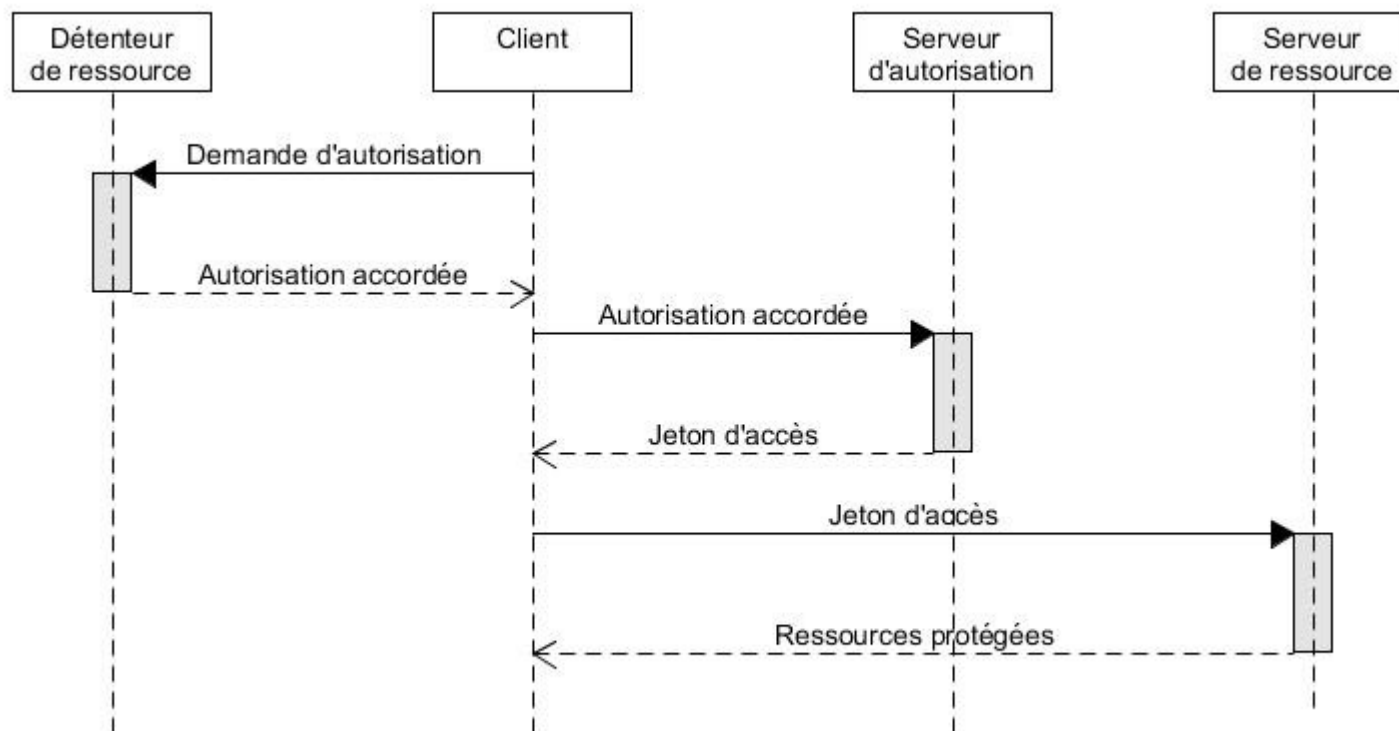
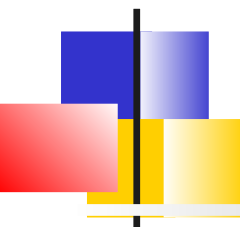
Le **Client** est l'application qui essaie d'accéder au compte utilisateur. Elle a besoin d'obtenir une permission de l'utilisateur pour le faire.

Le **serveur de ressources** est l'API utilisée pour accéder aux ressources protégées

Le **serveur d'autorisation** est le serveur qui autorise un client à accéder aux ressources en lui fournissant un jeton. Il peut demander l'approbation de l'utilisateur

L'utilisateur est la personne qui donne accès à certaines parties de son compte

Rq: Un participant du protocole peut jouer plusieurs rôles





Scénario

1. Pré-enregistrer le client auprès du service d'autorisation (=> client ID et un secret)
2. Obtenir l'autorisation de l'utilisateur.
(4 types de grant)
3. Obtention du token (date d'expiration)
4. Appel de l'API pour obtenir les informations voulues en utilisant le token
5. Validation du token par le serveur de ressource



Enregistrement du client

Le protocole ne définit pas comment l'enregistrement du client doit se faire mais définit les paramètres d'échange.

Le client doit fournir :

- **Application Name**: Le nom de l'application
- **Redirect URLs**: Les URLs du client pour recevoir le code d'autorisation et le jeton d'accès
- **Grant Types** : Les types d'autorisations utilisables par le client
- **Scopes** : paramètre utilisé pour limiter les droits d'accès d'un client
- **Javascript Origin** (optionnel): Le host autorisé à accéder aux ressources via *XMLHttpRequest*

Le serveur répond avec :

- **Client Id**:
- **Client Secret**: Clé devant rester confidentielle

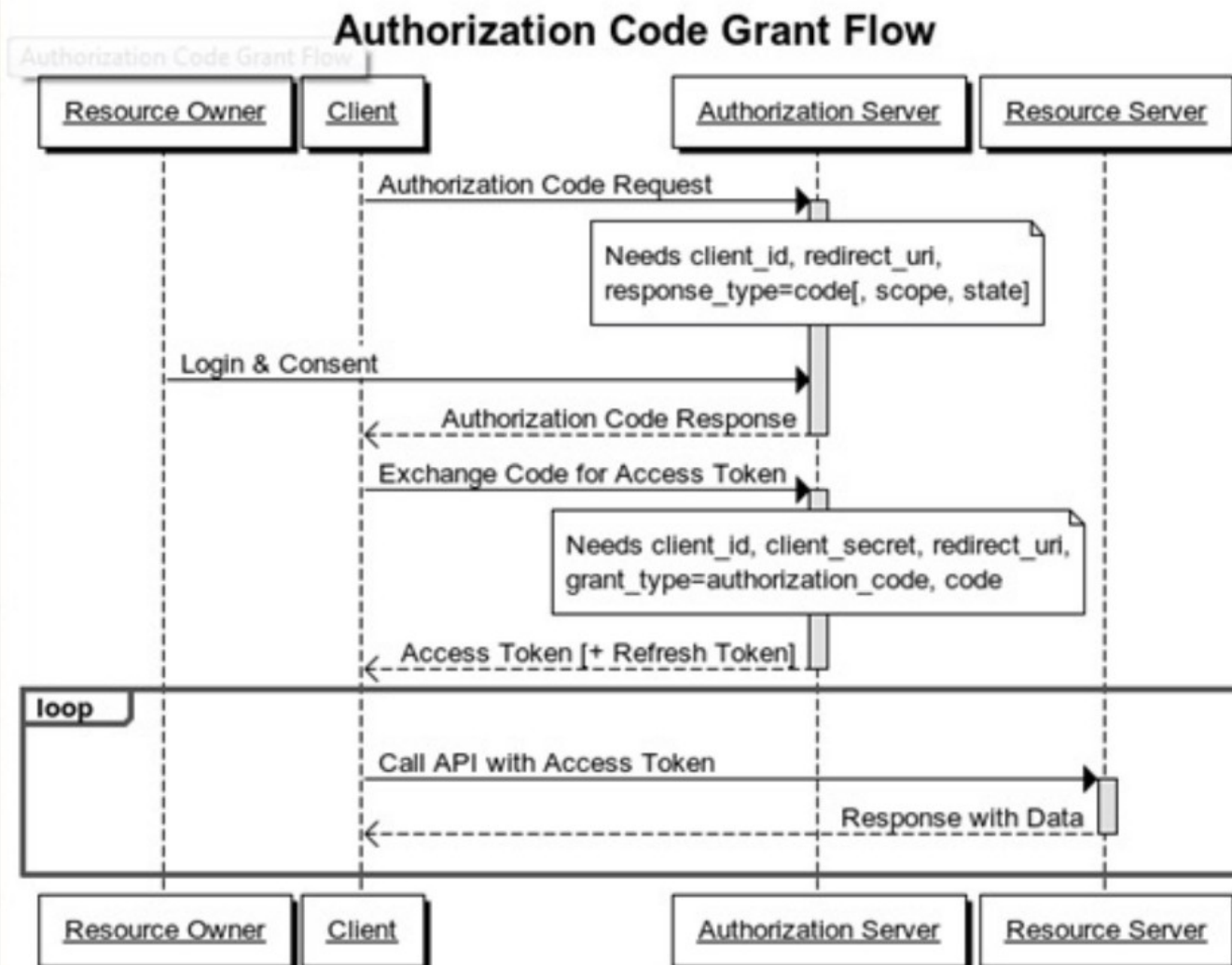


OAuth2 Grant Type

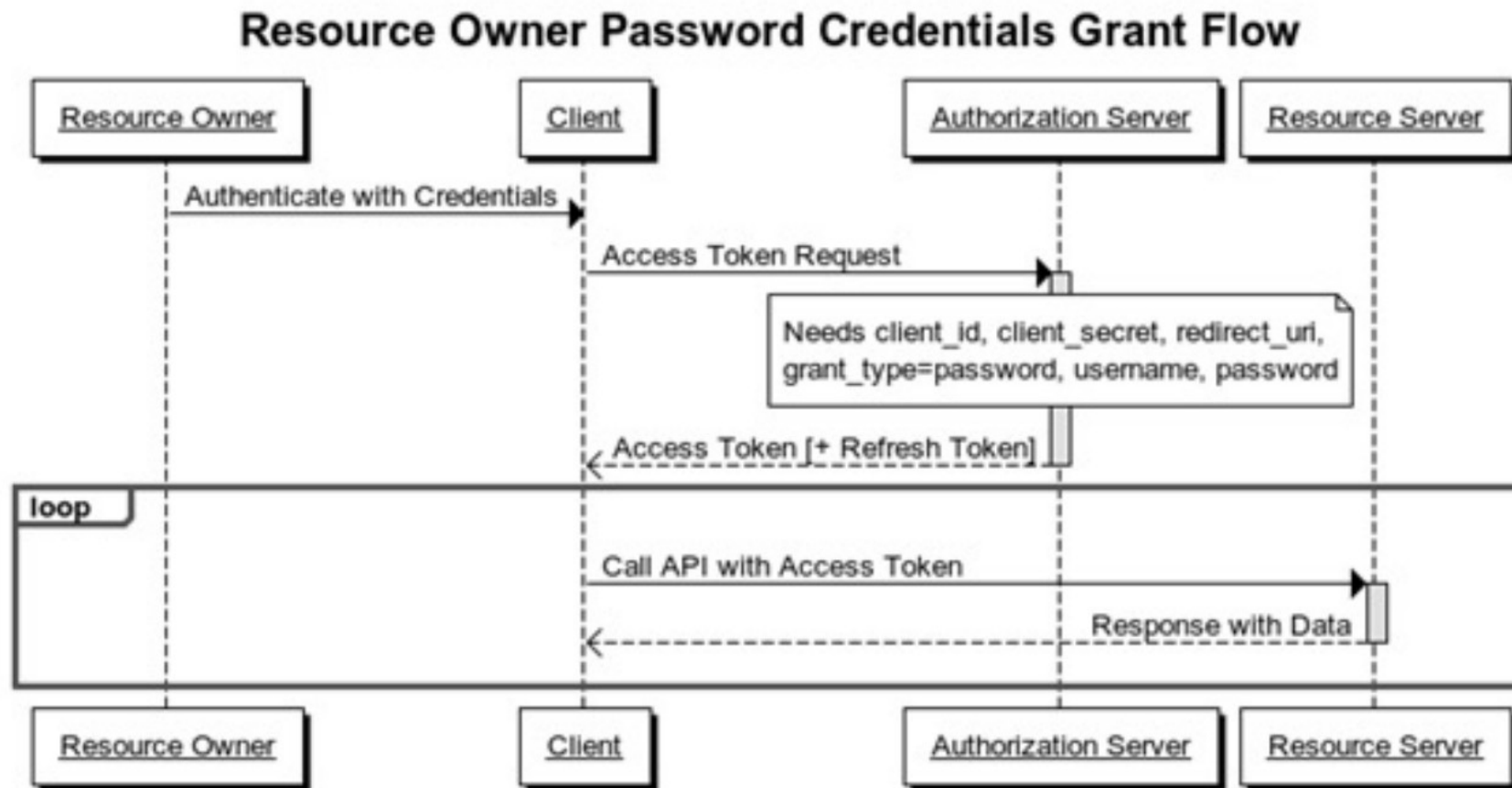
Différents moyens afin que l'utilisateur donne son accord : les **grant types**

- ***authorization code*** : Approbation de l'utilisateur sur le serveur d'autorisation et échange d'un code d'autorisation avec le client
- ***implicit*** : Jeton fourni directement. Certains serveurs interdisent de mode
- ***password*** : Le client fournit les créidentiels de l'utilisateur
- ***client credentials*** : Les créidentiels client suffise

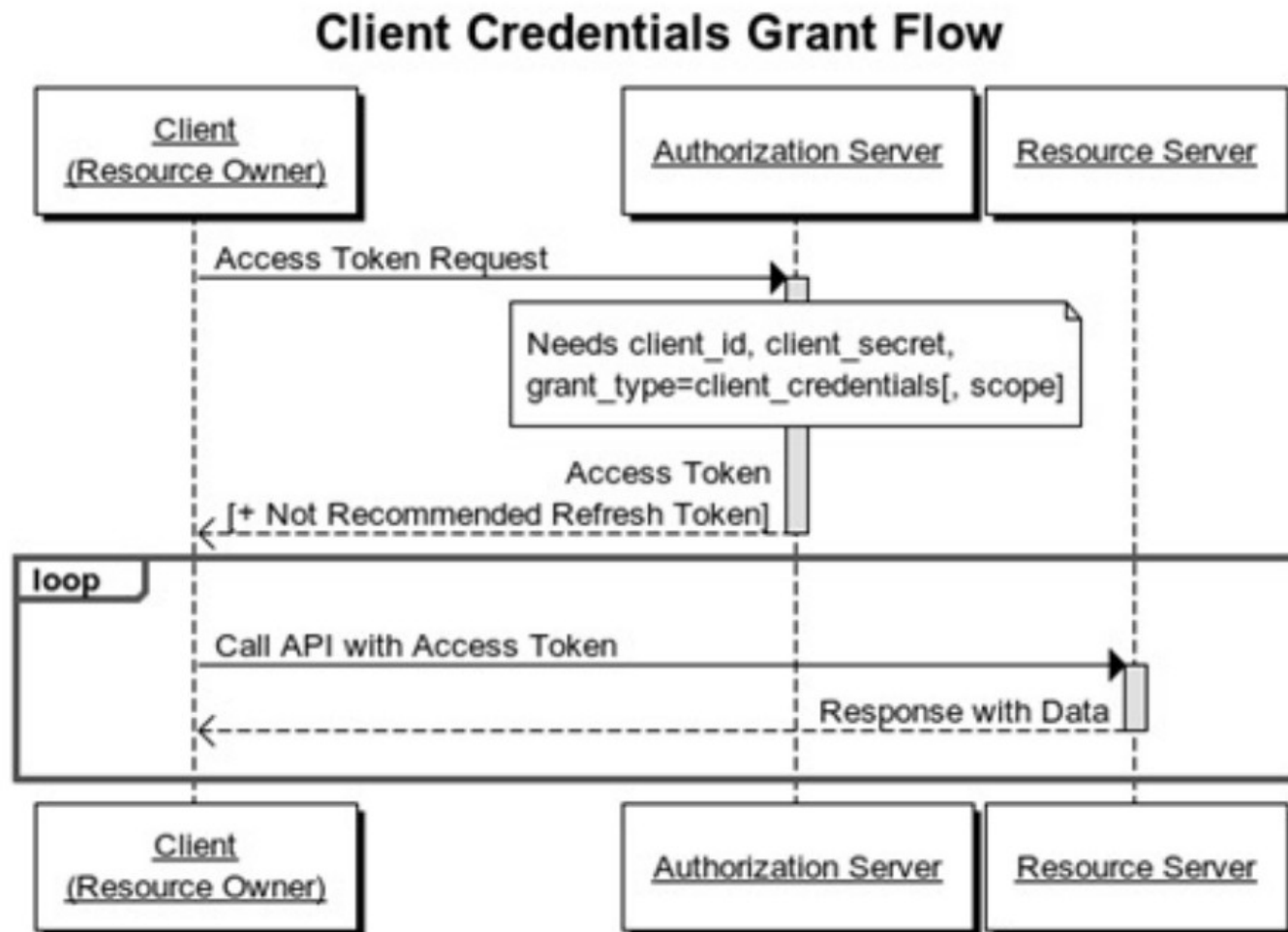
Authorization Code



Password Grant



Client Credentials





Tokens

Les Tokens sont des chaînes de caractères aléatoire générées par le serveur d'autorisation

Les jetons sont ensuite présents dans les requêtes HTTP et contiennent des informations sensibles => HTTPS

Il y a 2 types de token

- Le **jeton d'accès**: Il a une durée de vie limité.
- Le **Refresh Token**: Délivré avec le jeton d'accès. Il est renvoyer au serveur d'autorisation pour renouveler le jeton d'accès lorsqu'il a expiré



Usage du jeton

Le jeton est passé à travers 2 moyens :

- Les paramètres HTTP. (Les jetons apparaissent dans les traces du serveur)
- ***L'entête d'Authorization***

```
GET /profile HTTP/1.1
```

```
Host: api.example.com
```

```
Authorization: Bearer MzJmNDc3M2VjMmQzN
```

<http://www.bubblecode.net/en/2016/01/22/understanding-oauth2/>



Validation du jeton

Lors de la réception du jeton, le serveur de ressource doit valider l'authenticité du jeton et extraire ses informations différentes techniques sont possibles

- Appel REST vers le serveur d'autorisation
- Utilisation d'un support persistant partagé (ex. JdbcStore)
- Utilisation de JWT et validation via clé privé ou clé publique



JWT

JSON Web Token (JWT) est un standard ouvert défini dans la RFC 75191.

Il permet l'échange sécurisé de jetons (tokens) entre plusieurs parties.

La sécurité consiste en la vérification de l'intégrité des données à l'aide d'une signature numérique. (HMAC ou RSA).

Dans le cadre d'une application REST SpringBoot, le jeton contient les informations d'authentification d'un user :
Subject + Rôles

Différentes implémentations existent en Java (*io.jsonwebtoken*, ...) ou le starter ***spring-security-oauth2-jose***