



Microservices avec Quarkus

David THIBAU - 2024

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, profils, traces
 - Construire des applications natives
 - Tests
- **Quarkus et la persistance**
 - Hibernate et JPA
 - Panache
 - BD Reactive
 - MongoDB
- **API Restful avec Quarkus**
 - Introduction
 - Annotations JAX-RS
 - Problématiques RestFul
- **Interactions RPC**
 - RestClient
 - SOAP Client
- **Messaging**
 - Support pour le messaging
 - Intégration avec Kafka
- **Sécurité**
 - Architecture de la sécurité
 - Authentification HTTP/S
 - OpenID, oAuth2, JWT
- **Container**
 - Construction d'image
 - Déploiement vers Kubernetes
- **Observabilité**
 - Health
 - Tracing distribué
 - Métriques
 - Centralisation des traces



Introduction

Architectures micro services

Infrastructures de déploiement

Quarkus

Quarkus vs SpringBoot



DevOps et micro-services

Avec DevOps, une nouvelle architecture de systèmes visant à améliorer la rapidité des déploiements des retours utilisateur est apparu : les « **micro-services** »

C'est le même objectif visé que l'approche *DevOps* : « *Déployer plus souvent* »



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 stacks : Utilisation des stacks technologiques les plus appropriées pour une fonctionnalité donnée

Isolation des fautes : Un dysfonctionnement peut être plus facilement localisé et isolé.

Equipe DevOps autonome : Équipe réduite *Full-stack*, Communication renforcée

=> Favorise le partage et les montées en compétences



Caractéristiques

Design piloté par le métier : La décomposition en micro-services est pilotée par le métier (voir *Evans's DDD approach*)

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

Une interface explicitement publiée : Chaque service publie son API qui peut être consommée, mockée, validée

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

Communication légère : Les interactions entre services s'appuie sur des techno simple 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 nécessite que la localisation des services soit dynamique => Service de discovery

Monitoring : Les services sont surveillés en permanence. Des traces sont collectées et agrégées en un point central de surveillance

Résilience : Les services dépendants peuvent être en erreur. Un service doit être résilients aux erreurs de ses dépendances.

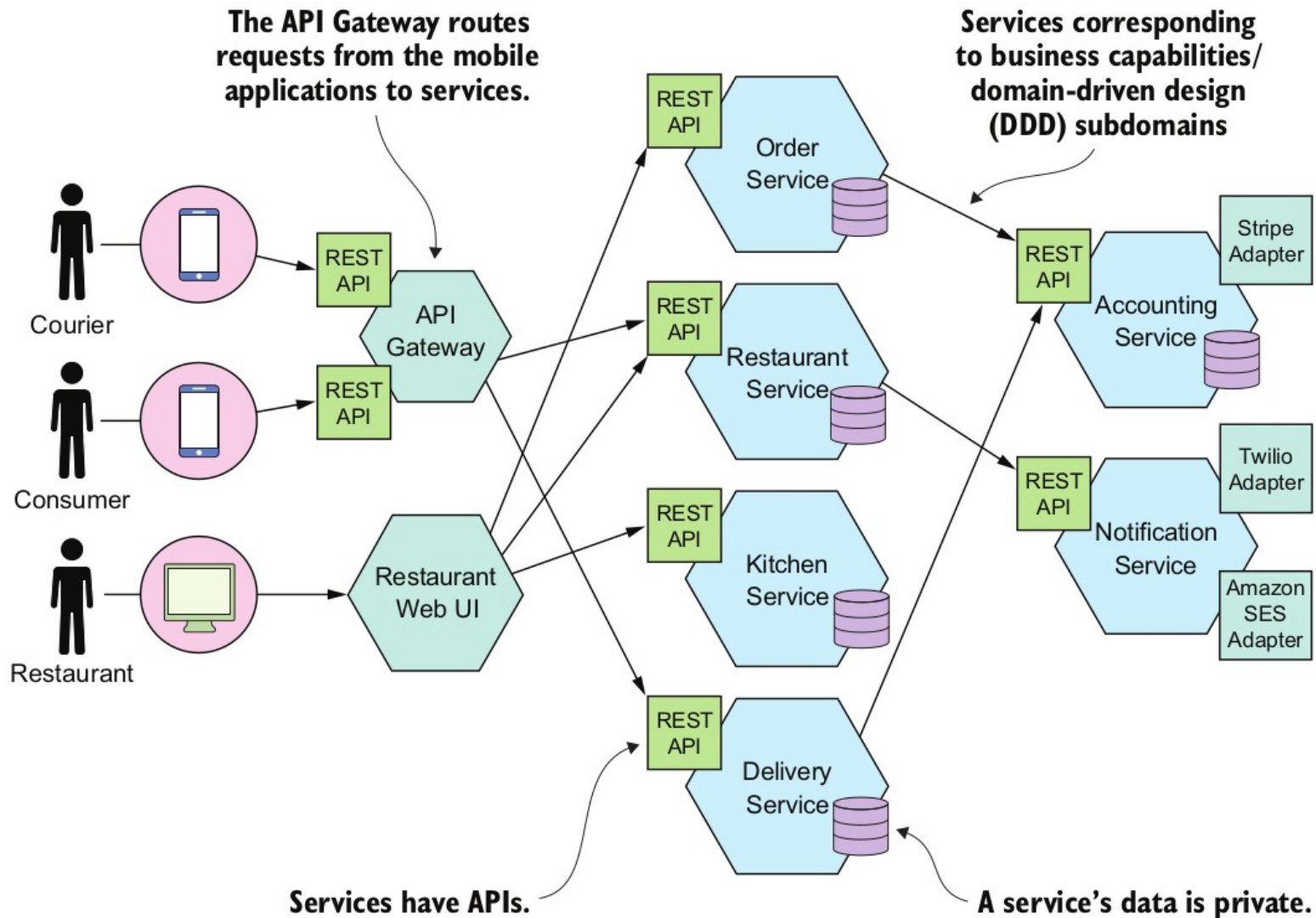
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.



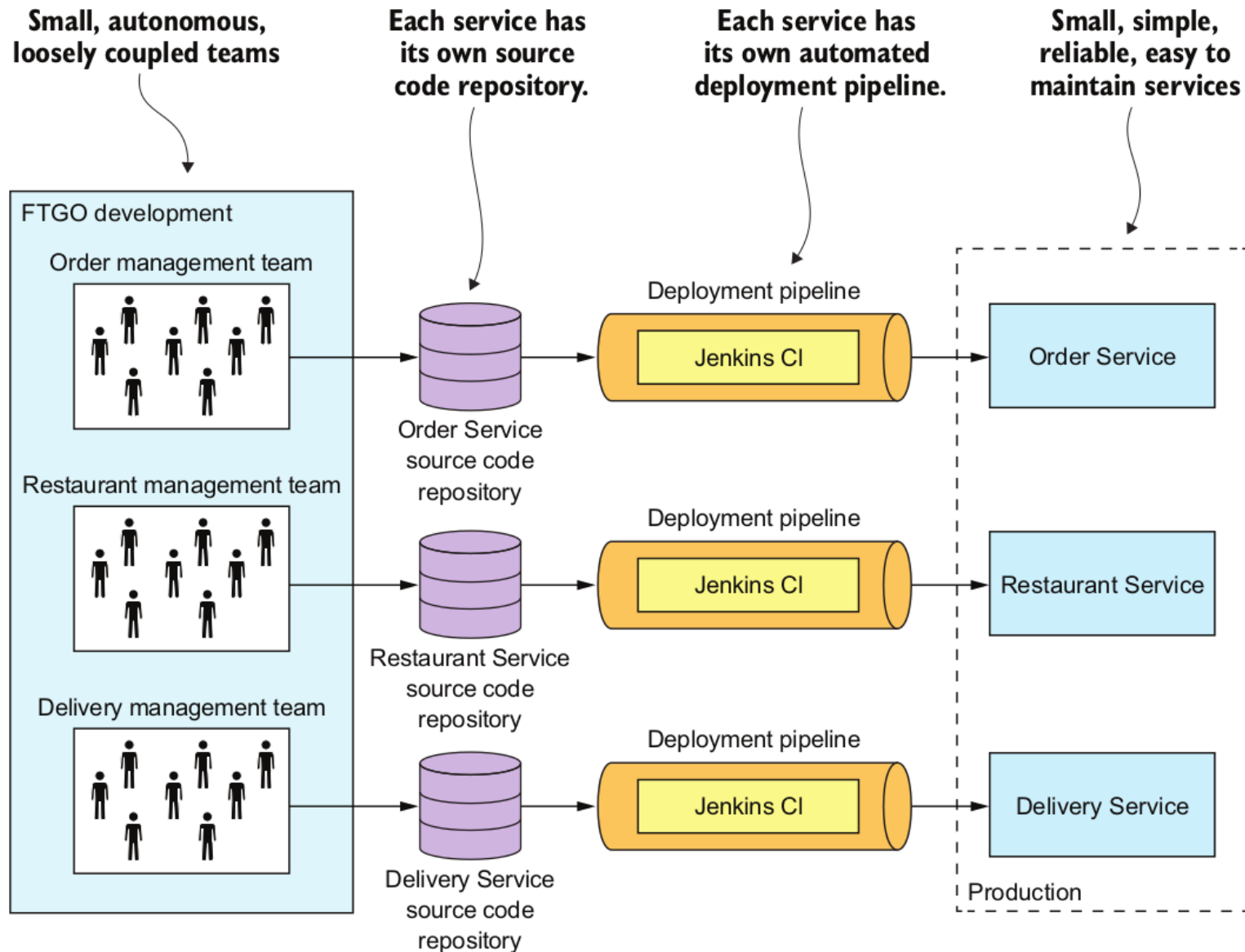
Inconvénients et difficultés

- 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

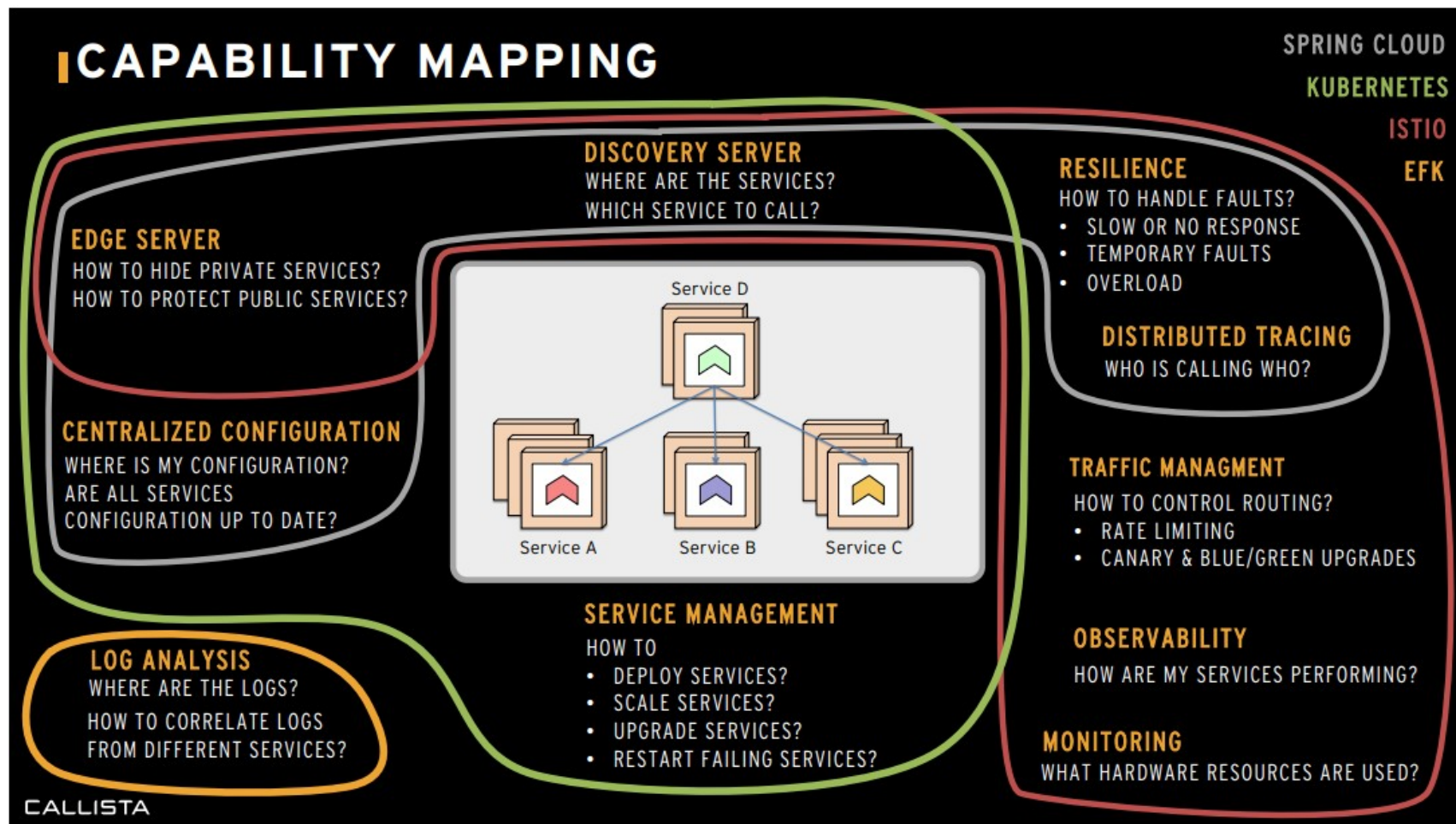


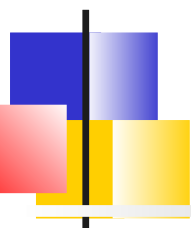


Introduction

Architectures micro services
Infrastructures de déploiement
Quarkus
Quarkus vs SpringBoot

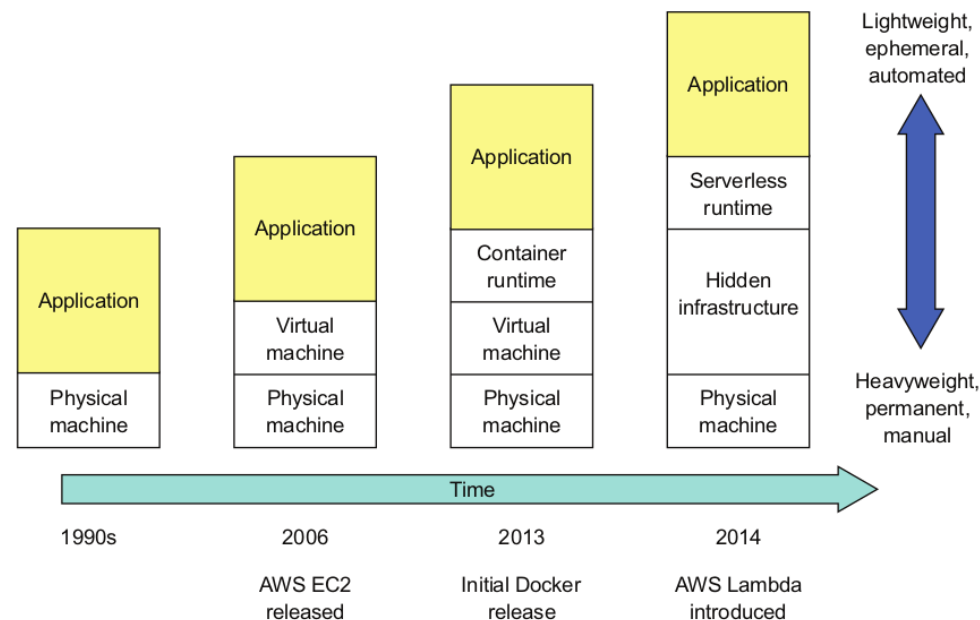
Capability Mapping





Infrastructure de déploiement

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





Service comme Container

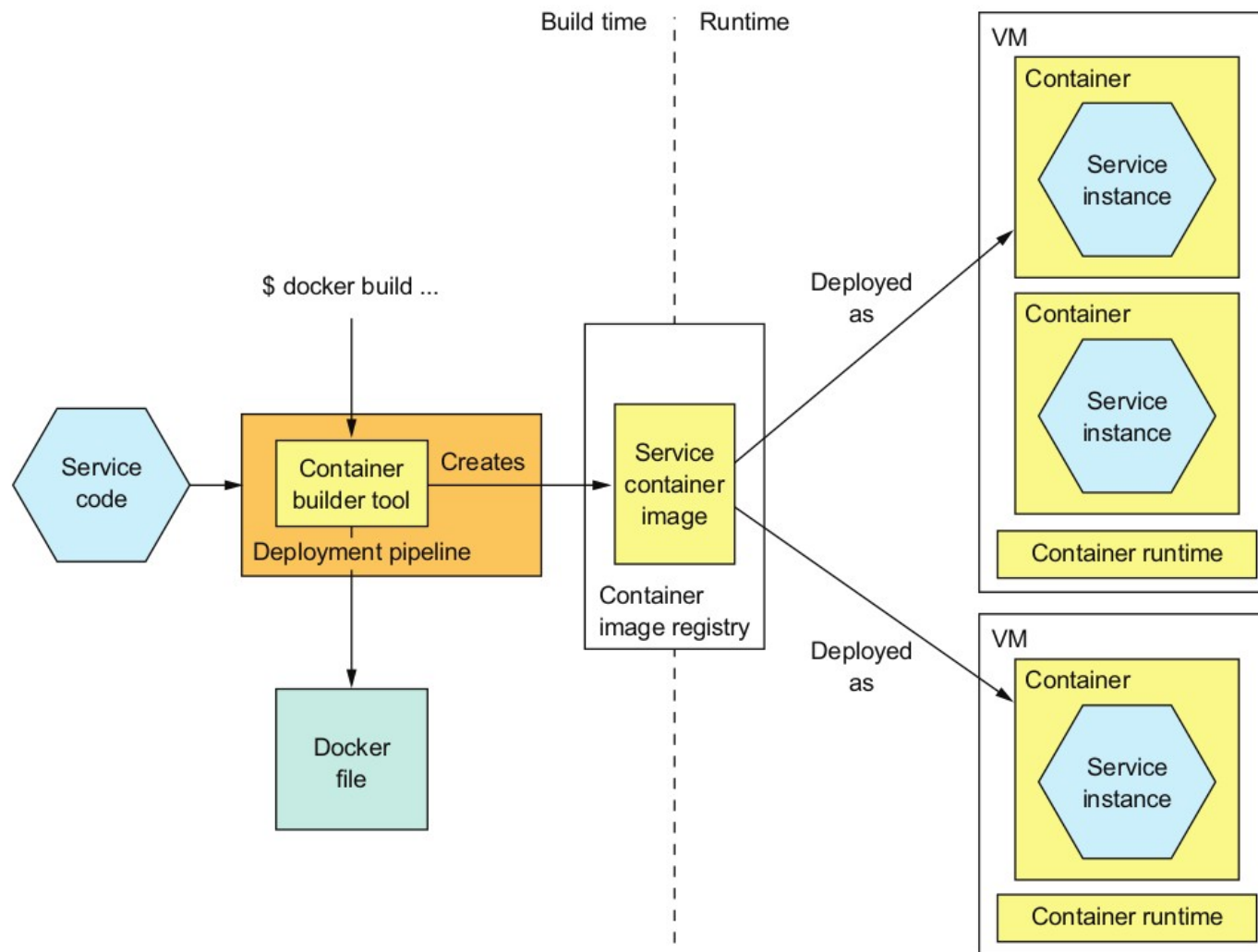
Deploy a service as a container

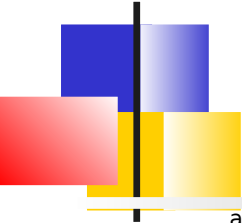
Pattern¹ : Déployer les services packagés comme des images de conteneur. Chaque service est un conteneur

Le packaging en image fait partie de la pipeline de déploiement

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

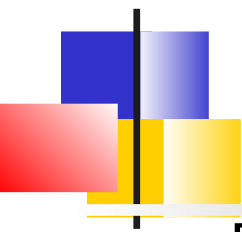
Pipeline CI/CD





Manifeste Kubernetes

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: delivery-service
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      name: delivery-service
  template:
    spec:
      containers:
        - name: delivery-service
          env:
            - name: POSTGRES_USER
              valueFrom:
                configMapKeyRef:
                  name: postgres-config
                  key: POSTGRES_USER
            - name: POSTGRES_PASSWORD
              valueFrom:
                configMapKeyRef:
                  name: postgres-config
                  key: POSTGRES_PASSWORD
          image: dthibau/delivery-service:1.0
```



Bénéfices / Inconvénients

Bénéfices

Encapsulation de la pile technologique. Déploiements immuables

Les outils de build ou les frameworks prennent en charge la construction des images du conteneur.

Les instances de service sont isolées.

Les ressources des instances de service sont limitées.

Les orchestrateurs de conteneurs offrent des services requis par les micro-services

Inconvénients

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

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*.

Objectifs :

- Faire de Java la plate-forme leader dans les environnements Kubernetes et serverless
- Offrir un framework capable de traiter le plus large éventail d'architectures d'applications distribuées.

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, Dev/Prod parity
- Sélection des meilleures librairies Java
- Programmation réactive ou impérative

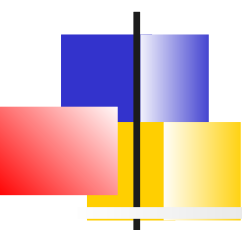


GraalVM

GraalVM compile les applications Java dans des binaires autonomes (qui s'exécute sans JVM).

Ces binaires sont plus petits, démarrent jusqu'à 100 fois plus rapidement, offrent des performances optimales période de chauffe et utilisent moins de mémoire et de processeur.

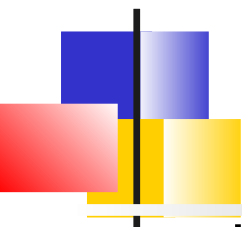
Elles offrent également une surface d'attaque réduite en excluant les classes inutilisées, en évitant la réflexion



Exécutable GraalVM Natif

Quarkus a un support pour créer des exécutables
GraalVM Native

- 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
- La cible native apporte quelques contraintes sur le développement Java



Build-time

L'idée centrale de Quarkus est de faire durant la compilation ce que les autres frameworks font à l'exécution

- 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

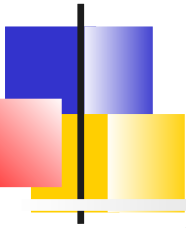


Support Kubernetes, Serverless

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 sondes utilisés par Kubernetes
- ConfigMaps et Secret pour la configuration applicative
- Live coding, le code est directement déployé sur le cluster Kubernetes de développement



Modèles de programmation

Quarkus fournit du support pour les modèles de programmation modernes

- **API Synchrones** : RestFul avec JAX-RS, GraphQL, MicroProfile Rest Client
- **Modèle réactif** avec Vert.x
- **Support de persistance** : *JPA, NoSQL*
- **Architecture Event-driven** pour Kafka ou AMQP
- **ServerLess / FaaS (Functions As A Service)**
avec Funqy permettant d'être indépendant du fournisseur d'infrastructure ou avec des extensions dédiées comme *Quarkus Amazon Lamda*



Expérience développeur

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

Configuration unifiée : Un unique fichier de configuration

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

Dev UI : Une interface utilisateur pour les développeurs. 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. grâce au démarrage de conteneur.

Test en continu : A chaque changement de code, les tests dépendants sont ré-exécutés

CLI : Un outil pour gérer les extensions et exécuter des commandes de build

Développement Remote : Code live dans un environnement Kubernetes.



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 coeur :
Eclipse Vert.x, Apache Camel et Hibernate



Eclipse MicroProfile

Eclipse MicroProfile est une
spécification définissant les APIs
nécessaires dans une architecture
micro-services

La spécification fait partie de Jakarta EE.

Quarkus est une des implémentations de
ce standard.



Reactive Stack

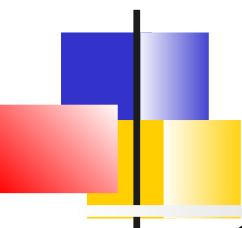
Quarkus unifie le réactif et l'impératif.
On peut mixer dans le même projet les
2 modes de programmation

Spring dans ses versions récentes peut
également mixer les 2 modes de
programmation.



Similitudes

- Spring Initializer \Leftrightarrow *code.quarkus.io*.
- *Starter SpringBoot*, \Leftrightarrow *Extensions Quarkus*
- Contrôle des versions des dépendances : Idem via BOM Maven
- IoC et Injection de dépendances : Les applications Quarkus et Spring sont constituées de beans qui exécutent diverses fonctions, telles que l'amorçage d'un serveur HTTP intégré.
- AutoConfiguration SpringBoot vs Dev Services Quarkus
DevServices va plus loin en proposant des images Docker des services d'appui
- 1 seul fichier de configuration, notion de profils de configuration
- Quarkus et Spring supportent Kotlin.
- IDEs : vscode, IntelliJ, Eclipse



Différence starter/extension

Comme dans SpringBoot, les extensions servent généralement à intégrer un framework tiers en exposant leurs composants principaux sous forme de beans CDI

Il existe cependant une différence fondamentale avec un Starter Spring Boot.

Une extension Quarkus est composée de deux parties distinctes :

- Un **module de déploiement**, pour le traitement de la configuration et la génération du code d'exécution. Il intervient au moment de la compilation.
- Un **module d'exécution** contenant l'implémentation de l'extension

La majorité du travail d'une extension est effectuée dans le module de déploiement lorsqu'une application est construite. C'est à ce moment que les beans d'intégration sont créés.



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration, profils, traces

Construire des applications natives

Tests



Introduction

Quarkus fournit une chaîne d'outils permettant aux développeurs le « *live reload* ».

- *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, 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=<key=value>] . . . [COMMAND]

- **create** : Créer un projet, un cli, une extension
- **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 l'extension rest
quarkus create app org.formation:delivery-service --
    extension=quarkus-rest
# 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 quarkus-kubernetes quarkus-smallrye-health
```



Maven

Quarkus fournit des plugins Maven permettant d'effectuer les mêmes opérations que Quarkus CLI :

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



Exemples Maven

```
# 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"
```



pom.xml

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

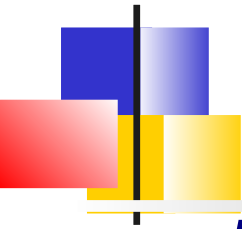
- Des propriétés de version. (en particulier celle de Java et de quarkus)
- Une référence à un BOM permettant de gérer les versions de toutes les dépendances
- Des dépendances sur des extensions
- La configuration des plugins (quarkus, maven-compiler, surefire-plugin)
- Un profil natif pour la génération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



Mode développement

Le mode Dev permet un déploiement à chaud avec compilation en arrière-plan lors de l'actualisation du navigateur

- Sur un reload, 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.



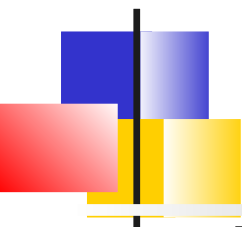
DevUI

Dev UI est un outil web permettant d'obtenir des informations sur votre application exécutée en mode Dev et éventuellement changer son état.

Son interface dépend des extensions utilisées

Voici quelques exemple de fonctionnalités :

- Lister les clés/valeurs de configuration
- Lister tous les beans CDI et leurs caractéristiques
- Lister les beans inutilisés lors du build
- Lister les entités JPA et leur mapping vers les tables BD
- Les tâches schedulées
- Les rapports de tests
- ...



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
- Éditeur de propriétés avec complétion

Tableau comparatif disponible ici :

<https://quarkus.io/guides/ide-tooling>



Démarrage dans l'IDE

Afin de démarrer une application Quarkus dans l'IDE, on s'appuie sur les plugins.

Le démarrage dans l'IDE permet l'utilisation du debugger de l'IDE

Par exemple pour Eclipse avec le plugin Quarkus

- *Run → Run Configurations ...*
- *QuarkusApplication → New Configuration*
- Sélectionner le projet et éventuellement le profil

Pour VSCode et l'extension Quarkus de RedHat

- Commande : *Quarkus: Debug current Quarkus project.*



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration, profils, traces

Construire des applications natives

Tests



Extensions cœur automatiquement incluses

Configuration : MicroProfile Configuration
via SmallRye

Propriétés fixées par application.properties, argument
de démarrage ou variable d'environnement

Logging : Gestion des traces

ArC : Injection de dépendances CDI,
io.quarkus:quarkus-arc



Extensions Web

Les extensions web sont orientées API REST :

- ***Netty, Undertow Servlet, Websockets*** : Couche de base
- ***RestEasy JAX-RS*** : Implémentation réactive de JAX-RS
- ***RestEasy JSON-B***, Jackson : Idem avec librairie de sérialisation
- ***SmallRye GraphQL*** : Service *GraphQL*
- ***Rest, GraphQL Client***: Client REST ou GraphQL
- ***SmallRye OpenAPI*** : Documentation de l'API
- ***Hibernate Validator*** : Validation des données
- ***CXF*** : Services SOAP



Extensions persistance

Agroal : Pools de connexions BD

JDBC Driver H2, MariaDB, MySql, PostgreSQL, SQL Server, Derby : Drivers JDBC

Hibernate ORM : Accès via JPA. Impératif ou réactif

Hibernate ORM with Panache : Facilitation de la couche JPA.
Impératif ou réactif

Hibernate Search + Elasticsearch : Indexation des entités dans des index Elasticsearch

Flyway, liquibase : Migration de base de données

InfiniSpan, Client Embedeed : Cache distribué

Reactive MySQL, Postgres, MongoDB, Neo4J, Amazon DB :
Clients réactif pour les différents Data Store

Narayana JTA, STM : Gestionnaire de transaction JTA



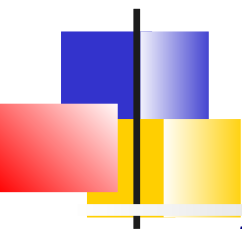
Messaging

SmallRye Reactive Messaging : Messagerie asynchrone pour *ReactiveStream*

SmallRye Reactive Messaging Kafka, AMQP, MQTT Connectors : Reactive Stream avec les différents message brokers

Apache Kafka Client, Kafka Streams : Client Kafka et KafkaStream

Artemis Core, JMS : Utilisation de Active MQ (natif ou JMS)



Programmation réactive

Eclipse Vert.x : Boite à outils réactive

SmallRye Reactive Streams Operators :
Opérateurs pour Reactive Streams

SmallRye Reactive Type Converters :
Conversion de type pour différentes bibliothèques
réactives

SmallRye Context Propagation : Propagation
de contexte

Reactive PostgreSQL client : Client réactif pour
Postgres



Cloud

Kubernetes : Génération de ressources
Kubernetes

Kubernetes Client : Interactions avec
un cluster Kubernetes

AWS Lambda : Support pour AWS
Lambda

SmallRye Health : Sondes de liveness,
readiness, etc.



Micro-services

SmallRye Fault Tolerance :

Implémentation de pattern de résilience (CircuitBreaker, Retry, etc.)

SmallRye Metrics : Extraction de métriques

SmallRye OpenTracing : Tracing de requêtes avec Jaeger



Sécurité

OpenID Connect : Obtention de jetons auprès de provider OpenID comme Keycloak

Vault : Stockage des crédits dans HashiCorp Vault

Elytron Security : Sécurisation des services

SmallRye JWT : Sécurisation avec jetons JWT

Elytron Security OAuth 2.0 : Sécurisation avec des jetons OAuth2

Elytron Security JDBC Realm : Intégration de realm JDBC



Divers

Mailer : Envoi d'emails

Scheduler - tasks : Planifier des jobs

Apache Tika : Extraire des textes des documents bureautiques

JGit : Accès aux dépôts Git

Compatibilité Spring :

Quarkus Extension for Spring DI API : Injection de dépendance avec annotations Spring

Quarkus Extension for Spring Web API : Annotations REST Spring

Quarkus Extension for Spring Data JPA API : Utilisation de SpringData

Autres langages :

Kotlin

Scala



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration, profils, traces

Construire des applications natives

Tests



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.
- L'extension Quarkus concernée est ***Arc***



Un bean typique

// Annotation déclarant un bean et son cycle de vie

@ApplicationScoped

public class Translator {

// Injection de dépendance d'un autre bean

// Attention, le qualifier private n'est pas recommandé dans ce cas

@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 déclare 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
```

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

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

```
    }
```

```
    // Injection par méthode
```

```
    @Inject
```

```
    void setDepts(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



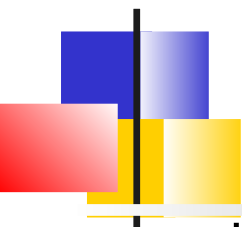
Exemple

```
@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 appliquer transversalement des « cross-cutting concern » .

La mise en place d'un intercepteur nécessite :

- La définition d'une annotation permettant de binder la classe intercepteur
- La classe intercepteur qui définit une ou plusieurs méthodes d'interception grâce aux annotations de *jakarta.interceptor*.
- Appliquer l'annotation sur les beans concernés



Intercepteurs

Définition de l'annotation

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

Classe d'interception

```
@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(); // exécution de la chaîne d'interception
        // ...log after
        return ret;
    }
}
```



Décorateurs

Les décorateurs sont similaires aux intercepteurs, mais parce qu'ils implémentent des interfaces métier, ils implémentent de 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 écouter des événements pour interagir de manière complètement découplée.

Tout objet Java peut être transmis par l'événement.

Quarkus génère des événements :

- Démarrage/ arrêt de l'application



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  
    }  
}
```



Évènements prédéfinis

Quarkus génère des événements de démarrage et d'arrêt de l'application.

On peut écouter ces événements pour effectuer du code personnalisé

```
@ApplicationScoped
public class AppLifecycleBean {

    private static final Logger LOGGER = Logger.getLogger("ListenerBean");

    void onStart(@Observes StartupEvent ev) {
        LOGGER.info("The application is starting...");
    }

    void onStop(@Observes ShutdownEvent ev) {
        LOGGER.info("The application is stopping...");
    }

}
```




@Startup

Il est également possible d'initialiser un bean dès le démarrage de l'application via **@Startup**

```
@Startup
@ApplicationScoped
public class EagerAppBean {

    private final String name;

    EagerAppBean(NameGenerator generator) {
        this.name = generator.createName();
    }
}
```



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration, profils, traces

Construire des applications natives

Tests



Introduction

Quarkus (noyau et extensions) et votre application utilisent le même mécanisme pour la configuration.

Basé sur l'API ***SmallRye Config*** : une implémentation de la spécification *MicroProfile Config*.



Sources de configuration

Les propriétés de configuration peuvent être définies 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 « *ops* »** dans *\$PWD/config/application.properties*
- **Fichier de configuration « *dev* »** *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** (*quarkus-vault*)
- **Consul** (*quarkus-config-consul*)
- **Spring Cloud Config** (*quarkus-spring-cloud-config-client*)
- **Kubernetes ConfigMap** et **Secrets** : (*quarkus-kubernetes-config*)



Expression pour les propriétés

Les valeurs de configuration peuvent utiliser des expression spécifiées par la séquence **`${ ... }`**.

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

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



Injection

@ConfigProperty permet de 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 **@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();
}
```




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



Valeurs par défaut

L'annotation **@WithDefault** permet de fixer une valeur par défaut pour une propriété.

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

    @WithDefault("8080")
    int port();

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



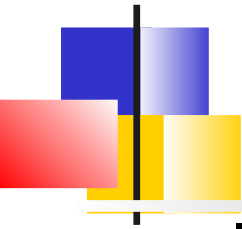
Validation

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 au moment du build



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 .

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

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

Les profils 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 -Dquarkus.profile=staging
```

```
java -jar ./target/quarkus-app/quarkus-run.jar
```

=> Exécution avec le profil *staging*



Beans en fonction du profil

Quarkus ajoute une fonctionnalité à CDI qui permet d'activer différents beans en fonction des profils

- **@IfBuildProfile** : Active le bean si le profil est activé
- **@UnlessBuildProfile** : Désactive le bean si le profil est activé
- **@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 permet également d'activer des beans en fonction des propriétés de configuration

- **@IfBuildProperty** : Active si la propriété a une valeur spécifique
- **@UnlessBuildProperty** : Désactive si la propriété a une valeur spécifique



Exemple

@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();  
    }
```

```
}
```




Gestion des traces

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

- On peut alors utiliser directement 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, profils, traces

Construire des applications natives

Tests



Introduction

Construire un exécutable natif nécessite d'utiliser une distribution de GraalVM. 3 distributions existent :

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

La construction de l'artefact natif nécessite également un **environnement de développement C** (*GCC*, et les entêtes *glibc* et *zlib*)

Pour s'épargner l'installation de GraalVM, Quarkus permet d'exécuter l'artefact natif dans un conteneur Docker (qui intègre *GraalVM*).



Installation GraalVM

2 options :

- Téléchargement du binaire
- Utilisation de *sdkman*, *homebrew*, ou *scoop*

Positionnement de ***GRAALVM_HOME***

Puis :

```
export JAVA_HOME=${GRAALVM_HOME}
export PATH=${GRAALVM_HOME}/bin:$PATH
```



Construction

Un environnement de compilation C doit être disponible.

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

Pour construire :

```
quarkus build --native
```

Ou

```
./mvnw package -Pnative
```

Le build produit l'exécutable :

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



Alternatives sans GraalVM

Création exécutable Linux

Dans ce cas, l'image est construite à l'intérieur d'un container :

```
./mvnw package -Dnative  
-Dquarkus.native.container-build=true
```

Création d'un container contenant l'exécutable Linux

*Nécessite l'extension **container-image-docker***

```
./mvnw package -Pnative  
-Dquarkus.native.container-build=true  
-Dquarkus.container-image.build=true
```



Support des exécutables natives

GraalVM impose certaines contraintes :

- Gestion des ressources :
GraalVM n'inclura aucune des ressources qui se trouvent sur le classpath dans l'exécutable natif qu'il crée. Les ressources destinées à faire partie de l'exécutable natif doivent être configurées explicitement
- Attention à la reflection :
*Les classes non explicitement référencées ne sont pas incluses dans le package natif
Il faut alors utiliser @RegisterForReflection pour les inclure*
- Attention au blocs *static* servant à l'initialisation.
*Nécessite alors de passer des arguments de build spéciaux
quarkus.native.additional-build-args=--initialize-at-run-time=com.example.SomeClass\\,org.acme.SomeOtherClass*
- Attention au génération de Proxy lors de l'exécution,
Cela doit être fait au buildTime



Développer avec Quarkus

IDEs et outils

Les extensions Quarkus

CDI

Configuration, profils, traces

Construire des applications natives

Tests



Introduction

Le support de *Quarkus* pour les tests se concentre sur :

- L'approche TDD et les tests continus
- Les tests unitaires en se reposant sur CDI
 - Injection des classes à tester
 - Annotations pour appliquer des aspects durant les tests
 - Support pour l'isolation et le Mocking
- Les tests d'intégration
 - Tests d'intégration sur l'artefact exécuté
 - Si appli web, support pour *RestAssured*



Tests en continu

Quarkus permet les **tests continus**, i.e les tests s'exécutent immédiatement après l'enregistrement des modifications de code.

- Cela 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.



Tests continus

Après avoir lancé l'application en mode *dev*, on obtient dans la console

--

Tests paused, press [r] to resume, [h] for more options>

En appuyant sur **r**, les tests s'exécutent

Toute modification de code provoquera alors la ré-exécution des tests concernés

On peut également provoquer les tests continus via la commande :

```
mvn quarkus:test
```




Tests unitaires

L'extension impliquée dans les tests unitaire est ***quarkus-junit5***

Elle fournit l'annotation ***@QuarkusTest*** qui contrôle l'initialisation du framework lors des tests

L'assistant de création de projet configure également le plugin maven *surefire* afin qu'il soit compatible avec JUnit5



Injection des classes de test

L'annotation ***@Inject*** est utilisée pour injecter la classe à tester

```
@QuarkusTest
public class GreetingServiceTest {

    @Inject
    GreetingService service;

    @Test
    public void testGreetingService() {
        Assertions.assertEquals("hello Quarkus",
            service.greeting("Quarkus"));
    }
}
```



Intercepteurs

Il est possible d'appliquer des intercepteurs durant les tests, par exemple *@Transactional* sur une classe ou une méthode de test.

Une annotation intéressante pour les tests d'intégration est ***@TestTransaction*** :

- La méthode est exécutée dans le contexte de transaction et peut donc effectuer des opérations de persistance
- A la fin de la méthode, un roll-back est effectué permettant de retrouver l'état initial de la base



Support pour le Mock

Lors des tests, il peut être utile de remplacer un bean par un Mock

CDI définit les annotations *@Alternative* et *@Priority* qui permet de définir un bean d'un type existant et de lui affecter une priorité.

L'annotation **@Mock** de quarkus est un stéréotype équivalent à :

- *@Alternative*
- *@Priority(1)*
- *@Dependent*



Exemple

```
@ApplicationScoped
public class ExternalService {

    public String service() {
        return "external";
    }
}
```

Et dans ***src/test/java***

@Mock

```
@ApplicationScoped
public class MockExternalService extends ExternalService {

    @Override
    public String service() {
        return "mock";
    }
}
```



@InjectMock

L'extension ***quarkus-junit5-mockito*** apporte l'annotation ***@InjectMock*** qui permet facilement de s'injecter un Mock dont la classe à tester est dépendante

Le mock peut alors être configuré avant l'exécution du test

La variante *@InjectSpy* permet de s'injecter un Spy



Exemple

```
@QuarkusTest
public class MockTestCase {

    @InjectMock
    MockableBean1 mockableBean1;

    @InjectMock
    MockableBean2 mockableBean2;

    @BeforeEach
    public void setup() {
        Mockito.when(mockableBean1.greet("Stuart")).thenReturn("A mock for Stuart");
    }

    @Test
    public void aTest() {
        Mockito.when(mockableBean2.greet("Stuart")).thenReturn("Bonjour Stuart");
        Assertions.assertEquals("A mock for Stuart", mockableBean1.greet("Stuart"));
        Assertions.assertEquals("Bonjour Stuart", mockableBean2.greet("Stuart"));
    }
}
```



Test d'intégration

L'annotation **@QuarkusIntegrationTest** permet de tester les artefacts produits par le build.

Il peut donc servir à tester :

- Le jar produit
- L'image native
- Le container

Attention l'injection durant les tests n'est plus possible

Typiquement, ces tests sont exécutés par le plugin Maven **fail-safe** dédié aux tests d'intégration après avoir démarré l'artefact.

Ils doivent respecter la règle de nommage *IT.java



Test d'intégration et profil natifs

La configuration du profil native configure le plug-in ***failsafe-maven*** afin qu'il exécute les tests d'intégration en indiquant l'emplacement de l'exécutable natif

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <executions><execution>
    <goals>
      <goal>integration-test</goal>
      <goal>verify</goal>
    </goals>
    <configuration><systemPropertyVariables>
<native.image.path>${project.build.directory}/${project.build.finalName}-runner</native.image.path>
    </systemPropertyVariables></configuration>
  </execution></executions>
</plugin>
```

=> ./mvnw verify -Pnative



Tests d'intégration http

L'extension ***rest-assured*** facilite l'écriture des tests d'intégration http

// L'application est démarrée sur le port 8081 lors de l'exécution du test

@QuarkusTest

```
public class GreetingResourceTest {

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

    @Test
    public void testGreetingEndpoint() {
        String uuid = UUID.randomUUID().toString();
        given()
            .pathParam("name", uuid)
            .when().get("/hello/greeting/{name}")
            .then()
                .statusCode(200)
                .body(is("hello " + uuid));
    }
}
```



Injection d'URL

Quarkus propose des annotations permettant d'injecter

- Des chemins d'accès à des ressources statiques sous la forme d'URI
@TestHTTPResource
- Des chemins d'accès à des ressources via
@TestHTTPEndpoint



Exemple

```
@QuarkusTest
public class StaticContentTest {

    //L'url est composé du chemin d'accès de GreetingResource + "sayHello"
    @TestHTTPEndpoint(GreetingResource.class)
    @TestHTTPResource("sayHello")
    URL url;

    @Test
    public void testIndexHtml() throws IOException {
        try (InputStream in = url.openStream()) {
            String contents = new String(in.readAllBytes(),
StandardCharsets.UTF_8);
            Assertions.assertEquals("hello", contents);
        }
    }
}
```



Persistence

Hibernate et JPA

Panache

BD Reactive

MongoDB



Zero Config Setup (Dev Services)

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

Il suffit de :

- Inclure la bonne extension vers le driver jdbc
- 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, MS SQL Server, MySQL, Oracle Express Edition, Postgres)



Datasource JDBC

Si l'on désire se connecter à une base pré-installée

Juste 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
```



Hibernate et JPA

Ajout 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 via 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 configurations avancées, 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;
    });
}
```



Persistence

Hibernate et JPA
Panache
BD Reactive
MongoDB



Panache

Panache est une librairie spécifique Quarkus visant à simplifier le développement de la couche de persistance basée sur Hibernate¹.

Panache propose d'implémenter la persistance via 2 les patterns **Entity** ou **Repository**.

Il est compatible avec Hibernate Réactif

1. Similaire à Spring Data JPA



Entity Pattern

Les classes entités étendent ***PanacheEntity***

- Profitent de toutes les méthodes CRUD
- Peuvent définir d'autres requêtes via les méthodes de *PanacheEntity*.
- Bénéficie d'un ID prédéfini de type *Long*, qui est mappé à la clé primaire de la table.

=> Pas besoin de créer un DAO (Data Access Object) ou un repository distinct.



Exemple *PanacheEntity*

@Entity

```
public class Person extends PanacheEntity {  
    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 Long deleteStefs(){  
        return delete("name", "Stef");  
    }  
}
```



Usage

// Persister une personne

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

// Supprimer

```
if(person.isPersistent()){  
    person.delete();  
}
```

// Requêtes basiques

```
List<Person> allPersons = Person.listAll();  
person = Person.findById(personId);  
Optional<Person> optional = Person.findByIdOptional(personId);
```



Usage (2)

// Toutes les personnes vivantes

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

```
List<Person> entities = Person.list("FROM Person p WHERE p.name = ?1", 'Dupont');
```

// 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>*** , bénéficiant des méthodes CRUD et définissant des requêtes supplémentaires

```
@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

Hibernate et JPA
Panache
BD Reactive
MongoDB



Introduction

Le modèle réactif est un modèle basé sur les I/O non bloquant et un modèle d'exécution des threads différents.

- Il permet une meilleure montée en charge

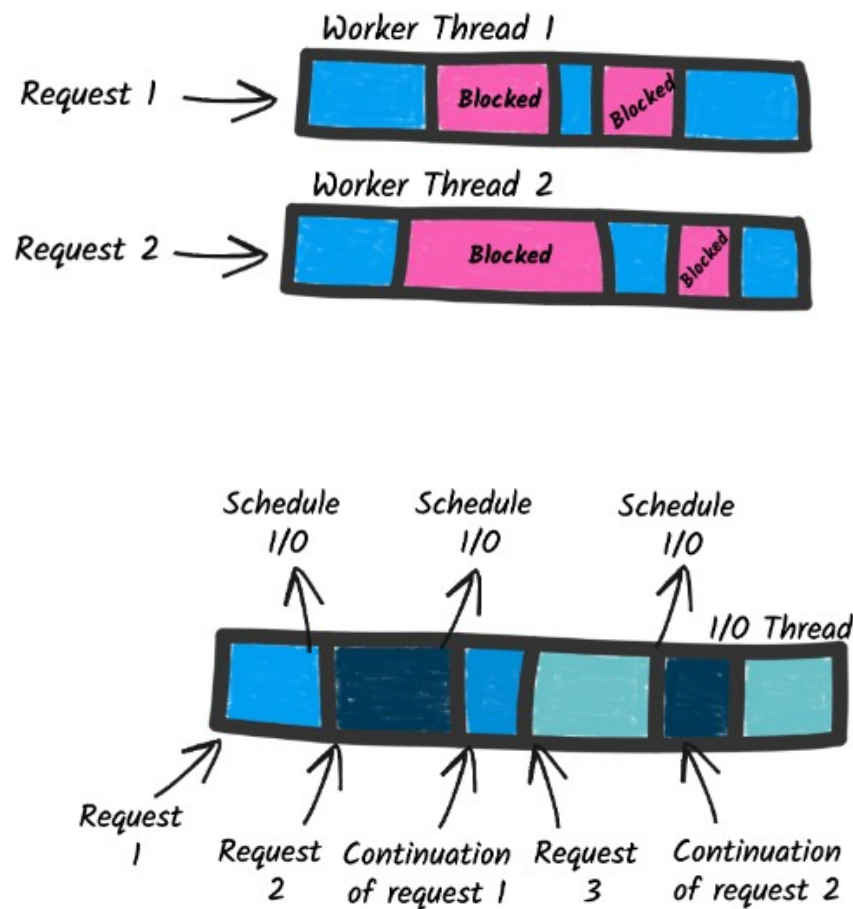
Le modèle de développement consiste à construire son code à partir de **flux** de données

- Des composants produisent des flux (Observable)
- D'autres s'abonnent à des flux, les transforme et produisent d'autre flux

Reactive Streams définit un standard pour le traitement asynchrone de flux d'événements offrant une fonctionnalité de **non-blocking back pressure**¹

1. Les subscribers peuvent contrôler la cadence d'émission des Observables

Imperative vs Reactive





Vert.x et Mutiny

Dans quarkus, le modèle de thread est fourni par Vert.x

2 possibilité pour écrire du code réactif avec Quarkus :

- Utilisation de Mutiny
- Coroutines et Kotlin



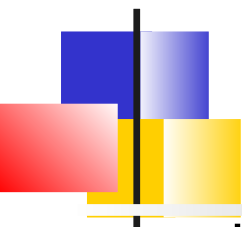
Interfaces Reactive Streams

```
public interface Publisher<T> {  
    public void subscribe(Subscriber<? super T> s);  
}
```

```
public interface Subscription {  
    public void request(long n);  
    public void cancel();  
}
```

```
public interface Subscriber<T> {  
    public void onSubscribe(Subscription s);  
    public void onNext(T t);  
    public void onError(Throwable t);  
    public void onComplete();  
}
```

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {  
}
```



Mutiny

L'utilisation d'extension réactive permet l'utilisation des types de la librairie **Mutiny** :

- **Uni<T>** est un flux d'objet T qui n'émet qu'un item ou un échec
- **Multi<T>** est un flux de données qui émet 0..n items, un échec ou un événement de fin

Ce sont des implémentations des interfaces de *ReactiveStream* (*Publisher*)

Les opérations sur ces types sont :

- *subscribe()*
- Application des opérateurs ReactiveStream (*map*, *filter*, *flatMap*, *reduce*, ...)

L'API *Mutiny* fournit des méthodes statiques pour la construction de ces flux



Méthodes statiques Uni

// Création + pipeline d'opérateurs

```
Uni.createFrom().item("hello")  
    .onItem().transform(item -> item + " mutiny")  
    .onItem().transform(String::toUpperCase)  
    .subscribe().with(  
        item -> System.out.println(">> " + item));
```

// Création d'un événement échec

```
Uni<Integer> failed1 = Uni.createFrom().failure(new  
    Exception("boom"));
```

// Création d'un événement Void

```
Uni<Void> uni = Uni.createFrom().nullItem();
```

// Création à partir d'un CompletionStage

```
Uni<String> uni = Uni.createFrom().completionStage(stage);
```



Méthodes statiques Multi

// Création + pipeline d'opérateurs

```
Multi.createFrom().items(1, 2, 3, 4, 5)
    .onItem().transform(i -> i * 2)
    .select().first(3)
    .onFailure().recoverWithItem(0)
    .subscribe().with(System.out::println);
```

// A partir d'un Iterable

```
Multi<Integer> multiFromIterable =
    Multi.createFrom().iterable(Arrays.asList(1, 2, 3, 4, 5));
```

// Un événement échec

```
Multi<Integer> failed1 = Multi.createFrom().failure(new
    Exception("boom"));
```

// Vide

```
Multi<String> multi = Multi.createFrom().empty();
```



invoke et call

En dehors des opérateurs (map, filter, find, ...), Mutiny propose 2 méthodes permettant d'agir sur les événements *onItem*, *onFailure*, *onCompletion*.

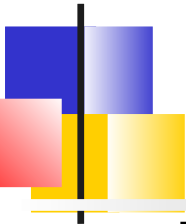
En utilisant ces 2 méthodes, le flux en aval reçoit l'événement d'origine.

- ***invoke*** : Traitement synchrone, retourne void :
L'événement n'est pas propagé tant que la lambda n'est pas terminée.

```
Multi<String> m = multi.onItem()  
    .invoke(i -> System.out.println("Received item: " + i));
```

- ***call*** : Traitement asynchrone, retourne *Uni<T>*

```
multi.onItem().call(i ->  
    Uni.createFrom().voidItem()  
    .onItem().delayIt().by(Duration.ofSeconds(1)));
```

transform et *transformToUni*

Mutiny propose également 2 méthodes permettant de transformer l'événement originel en un autre événement :

- ***transform(Function<I, O> function)*** :
Transforme l'événement en un autre événement à l'aide d'une fonction synchrone.
L'aval reçoit le résultat de la fonction (ou un échec si la transformation a échoué).
- ***transformToUni(Function<I, Uni<O>> function)*** : Transforme l'événement en un autre événement à l'aide d'une fonction asynchrone.



Examples

// Transformation synchrone

```
Multi<String> someMulti = Multi.createFrom().items("a", "b", "c");
someMulti
    .onItem().transform(i -> i.toUpperCase())
    .subscribe().with(
        item -> System.out.println(item)); // Print A B C
```

// Transformation asynchrone

```
Uni<String> uni = Uni.createFrom().item("Cameron");
uni
    .onItem().transformToUni(name -> invokeRemoteGreetingService(name))
    .subscribe().with(
        item -> System.out.println(item), // Print "Hello Cameron",
        fail -> fail.printStackTrace()); // Print the failure stack trace
```



Persistence Reactive

Quarkus propose plusieurs support de persistance réactifs :

- Hibernate Reactive
- Hibernate Reactive avec Panache
- Clients BD réactifs : PostgreSQL, MySQL, Microsoft SQL Server, DB2, Oracle
- MongoDB
- Mongo avec Panache
- Cassandra
- Redis



Reactive BD

Pour profiter d'appels réactifs vers une base de données relationnelles :

- Extension ***hibernate-reactif*** ou ***hibernate-panache-reactif***
- Extension d'un **driver reactif**
- Définition d'une **datasource réactive**
- Utiliser ***Uni*** et ***Multi*** dans les couches DAO



Datasource réactive

Des extensions spécifiques sont fournies pour le modèle réactif

Les propriétés sont légèrement différentes :

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



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*



Exemple Hibernate Reactif

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

    public Uni<List<Fruit>> findAll() {
        return sf.withTransaction((s,t) -> s.createQuery("from Gift", Gift.class).getResultList());
    }
    public Uni<Gift> getSingle(Integer id) {
        return sf.withTransaction((s,t) -> s.find(Gift.class, id));
    }
    public Uni<Gift> update(Integer id, Gift gift) {
        return sf.withTransaction((s,t) -> s.find(Gift.class, id)
            .onItem().ifNull().failWith(new EntityNotFoundException("Gift not found"))
            // If entity exists then update it
            .invoke(entity -> entity.setName(fruit.getName())));
    }

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



Exemple Panache Reactif

```
import io.quarkus.hibernate.reactive.panache.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

Attention, *@Transactional* n'est pas supportée par Hibernate Reactive, il faut utiliser **@WithTransaction** et la méthode annotée doit renvoyer un Uni.

@WithTransaction

```
public Uni<Void> create(Person person){
    // Here we use the persistAndFlush() shorthand method on a Panache
    repository to persist to database then flush the changes.
    return person.persistAndFlush()
        .onFailure(PersistenceException.class)
        .recoverWithItem(() -> {
            LOG.error("Unable to create the parameter", pe);
            //in case of error, I save it to disk
            diskPersister.save(person);
            return null;
        });
}
```



Persistence

Hibernate et JPA
Panache
BD Reactive
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*

La seule configuration nécessaire est
alors la base Mongo

`quarkus.mongodb.database`



Usage de MongoClient

```
@ApplicationScoped
public class FruitService {

    @Inject MongoClient mongoClient;

    public List<Fruit> list(){
        List<Fruit> list = new ArrayList<>();
        MongoClient 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)

```
// On peut utiliser PanacheReactiveMongoEntity
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");
    }
}
```



RESTful API

Introduction

Annotations

Problématiques RestFul



Introduction

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

Les développeurs mettent au point des classes ressources qui exposent des points 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 fournit automatiquement une sous-classe *Application* qui initialise le contexte CDI et démarre un serveur
 - Une seule application *JAX-RS* est prise en charge dans la JVM démarrée par Quarkus.
A la différence, d'un déploiement *.war* sur un serveur.
- Toutes les ressources *JAX-RS* sont traitées comme des beans CDI singletons.



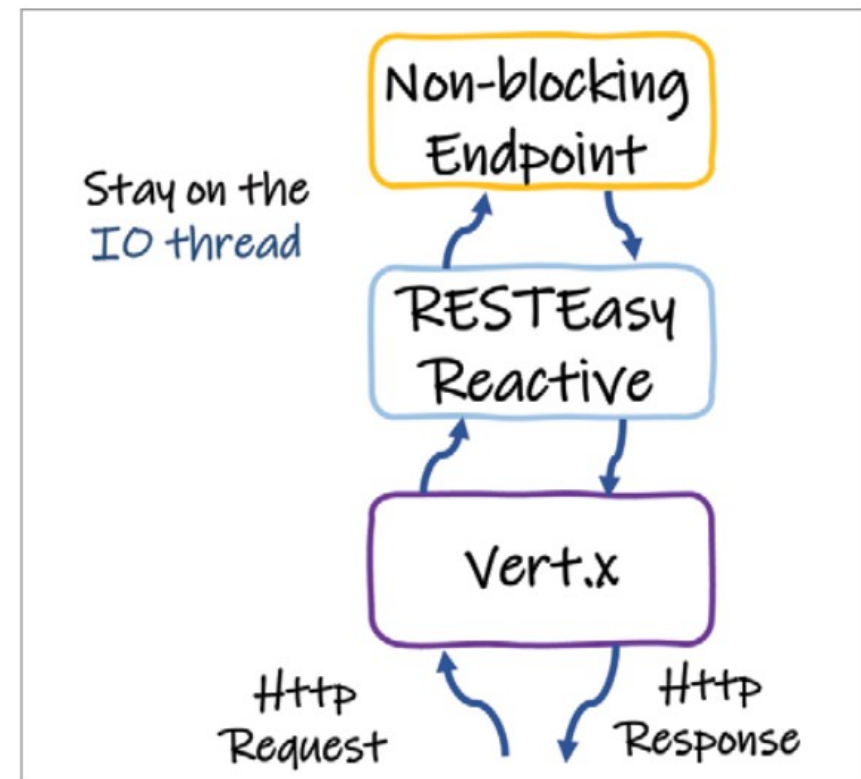
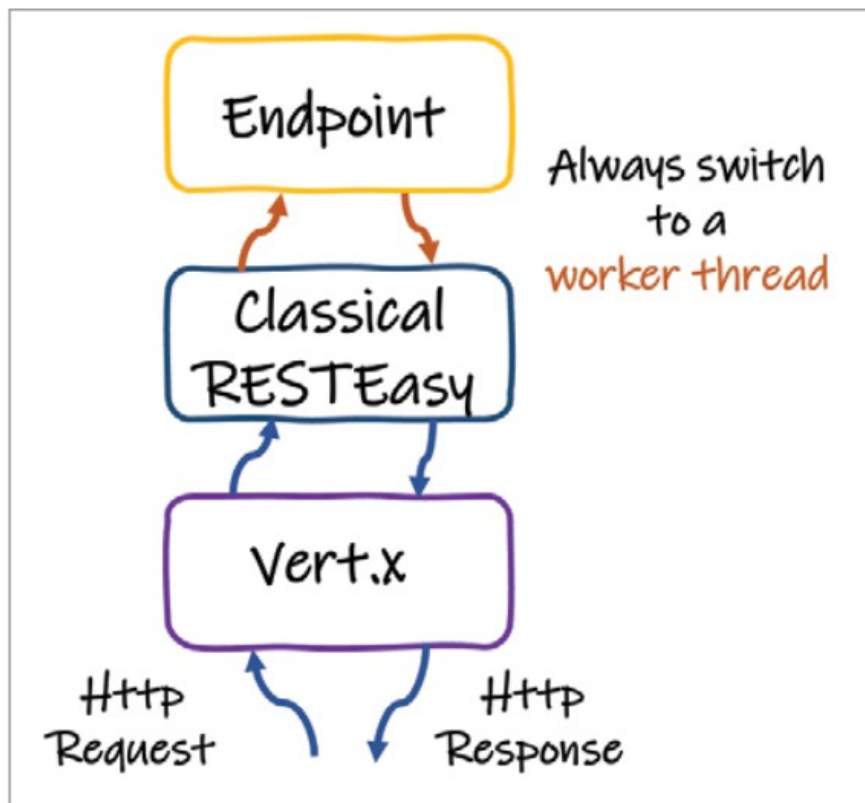
Modèle réactif

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.
Cependant, les développeurs peuvent mixer du code réactif avec du code non-réactif

Classical vs Reactive





RestEASY Reactive

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-rest</artifactId>
</dependency>
```

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

    @GET
    public Uni<String> hello() {
        return Uni.createFrom().item("Hello, World!");
    }
}
```




RESTful API

Introduction
Annotations
Problématiques RestFul



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 nommage 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)

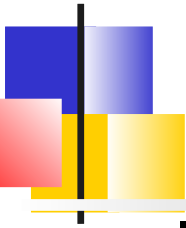


Retourner un corps de réponse

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 elle doit renvoyer le type **Uni** ou **Multi**

Cela permet de ne pas bloquer la thread event-loop

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

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

    @GET
    public Uni<List<Order>> findAll() {
        return logRepository.findAll().list();
    }

    @GET
    @Produces(MediaType.SERVER_SENT_EVENTS) // SSE Server side events1
    public Multi<String> streamLogs() {
        return logs;
    }
}
```

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



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** : 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 Event-loop :

- *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.



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

Problématiques RestFul



Sérialisation JSON

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

- **io.quarkus:quarkus-rest-jackson**
- **io.quarkus:quarkus-rest-jsonb**

Pour la sérialisation XML, importer :

- **io.quarkus:quarkus-rest-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

```
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")
```

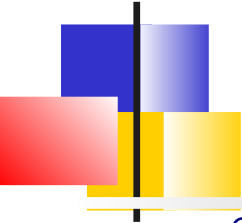
```
public User userPublic() {  
    return testUser();  
}
```

```
@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);  
    }
```

```
}
```




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***
-



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 à ***<http://localhost:8080/q/swagger-ui>***
 - par défaut en mode *dev*
 - Ou si *quarkus.swagger-ui.always-include=true*



Validation

Pour valider les paramètres d'entrée d'un endpoint REST, on peut les annoter avec les contraintes de *Hibernate Validator* (*@NotNull*, *@Digits...*) ou avec **@Valid** (qui cascade la validation sur l'objet¹).

- Si une erreur de validation se produit, un message d'erreur (JSON) et un code retour 400 sont renvoyés.

@Valid peut également être utilisée sur des méthodes de bean service.

1. Attention, il faut explicitement cascader la validation sur les objets embarqués avec *@Valid*



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
    }
}
```



Interactions RPC

RestClient
SOAP Client



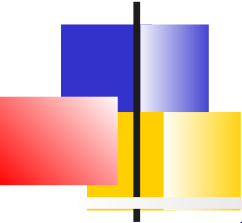
Mise en place

Quarkus propose une extension pour faciliter la mise en place d'un client REST.

En particulier : ***rest-client-jackson***

Ensuite 2 alternatives :

- 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 instruit Quarkus de créer un bean implémentant l'interface

Les annotations *JAX-RS* sont utilisées pour préciser 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 l'URL de base 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
```



Paramètre de requêtes

2 façons de préciser des paramètres de requête :

- Utiliser **@QueryParam** ou **@RestQuery**¹ et l'argument de méthode
- Utiliser **@ClientQueryParam**.
Et la valeur peut être
 - une constante,
 - une propriété de configuration
 - La valeur de retour d'une méthode.

1. @RestQuery est équivalent à @QueryParam mais n'a pas besoin de spécifier l'attribut name



Exemples *@QueryParam*, *@RestQuery*

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


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

    @GET
    Set<Extension> getByName(@RestQuery String name);

    @GET
    Set<Extension> getByFilter(@RestQuery Map<String, String> filter);

    @GET
    Set<Extension> getByFilters(@RestQuery MultivaluedMap<String, String> filters);

}
```



Exemples avec *@ClientQueryParam*

```
@ClientQueryParam(name = "my-param", value = "${my.property-value}")
```

```
public interface Client {
```

```
    @GET
```

```
    String getWithParam();
```

```
    @GET
```

```
    @ClientQueryParam(name = "some-other-param", value = "other")
```

```
    String getWithOtherParam();
```

```
    @GET
```

```
    @ClientQueryParam(name = "param-from-method", value = "{with-param}")
```

```
    String getFromMethod();
```

```
    default String withParam(String name) {  
        if ("param-from-method".equals(name)) {  
            return "test";
```

```
        }
```

```
        throw new IllegalArgumentException();
```

```
    }
```

```
}
```



Usage du *RestClient*

Pour effectuer l'appel REST, il suffit d'injecter le client via l'annotation *@RestClient* 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
    Uni<Set<Extension>> getByIdAsync(@QueryParam("id") String id);
}
```




Usage

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

    @RestClient
    ExtensionsService extensionsService;

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

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



Support pour SSE

La production d'événements SSE est possible :

```
@Path("/sse")
@registerRestClient(configKey = "some-api")
public interface SseClient {
    @GET
    @Produces(MediaType.SERVER_SENT_EVENTS)
    Multi<String> get();
}
```



Entêtes HTTP

Plusieurs façons pour spécifier des entêtes :

- 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;
    }
}
```



Fault-tolerance

Quarkus fournit ***SmallRye Fault Tolerance***, une implémentation de la spécification *MicroProfile Fault Tolerance*.

L'implémentation fournit les annotations ***@Timeout***, ***@Fallback***, ***@Retry*** et ***@CircuitBreaker*** permettant aux applications d'être résilient aux défaillances des services sur lesquelles elle s'appuie.



Exemple

```
@Path("/api/fruit")
@registerRestClient
public interface FruityViceService {

    @GET
    @Path("/{name}")
    @Produces(MediaType.APPLICATION_JSON)
    @Retry(maxRetries = 3, delay = 2000)
    @Fallback(FruityViceFallback.class)
    FruityVice getFruitByName(@PathParam("name") String name);

    public static class FruityViceFallback implements FallbackHandler<FruityVice> {

        private static final FruityVice EMPTY_FRUITY_VICE =
            FruityVice.of("empty", Nutritions.of(0.0, 0.0));

        @Override
        public FruityVice handle(ExecutionContext context) {
            return EMPTY_FRUITY_VICE;
        }

    }

}
```



Interactions RPC

RestClient
SOAP Client



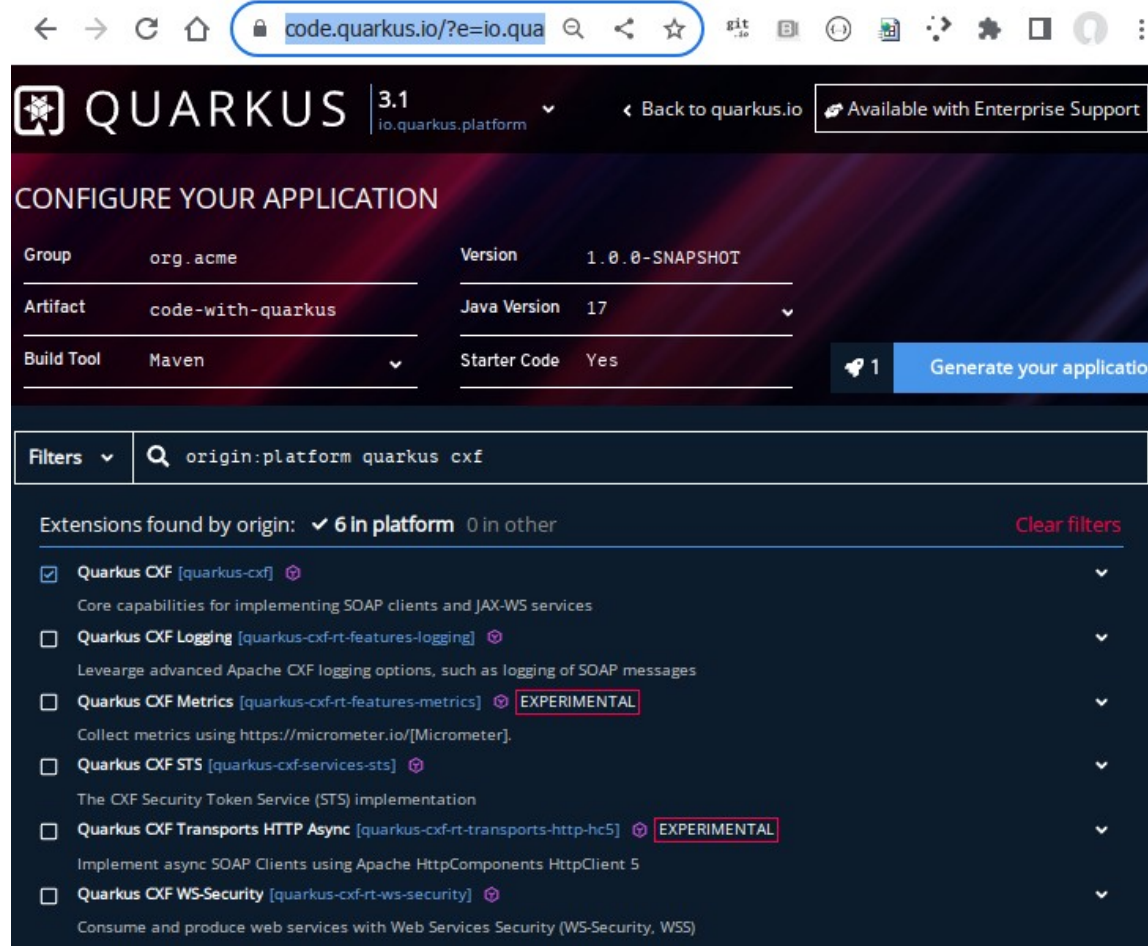
Introduction

L'extension ***io.quarkiverse.cxf:quarkus-cxf*** permet d'implémenter et de consommer des services SOAP en utilisant Apache CXF

D'autres extensions sont également disponibles :

- CXF Metrics
- CXF Security Token Service (STS)
- Async SOAP Client
- CXF WS-Security

Extensions



The screenshot shows the Quarkus code.quarkus.io website. The browser address bar displays `code.quarkus.io/?e=io.qua`. The page header includes the Quarkus logo, version 3.1, and a link to `io.quarkus.platform`. The main section is titled "CONFIGURE YOUR APPLICATION" and contains a form with the following fields:

Field	Value
Group	org.acme
Artifact	code-with-quarkus
Build Tool	Maven
Version	1.0.0-SNAPSHOT
Java Version	17
Starter Code	Yes

Below the configuration form, there is a search bar with the text "origin:platform quarkus cxf". The search results show a list of extensions found by origin:

- ☒ **Quarkus CXF** [quarkus-cxf] ⓘ
Core capabilities for implementing SOAP clients and JAX-WS services
- ☐ **Quarkus CXF Logging** [quarkus-cxf-rt-features-logging] ⓘ
Leverage advanced Apache CXF logging options, such as logging of SOAP messages
- ☐ **Quarkus CXF Metrics** [quarkus-cxf-rt-features-metrics] ⓘ **EXPERIMENTAL**
Collect metrics using https://micrometer.io/[Micrometer].
- ☐ **Quarkus CXF STS** [quarkus-cxf-services-sts] ⓘ
The CXF Security Token Service (STS) implementation
- ☐ **Quarkus CXF Transports HTTP Async** [quarkus-cxf-rt-transports-http-hc5] ⓘ **EXPERIMENTAL**
Implement async SOAP Clients using Apache HttpComponents HttpClient 5
- ☐ **Quarkus CXF WS-Security** [quarkus-cxf-rt-ws-security] ⓘ
Consume and produce web services with Web Services Security (WS-Security, WSS)



Étapes de mise en place

Un Service SOAP nécessite :

- Une interface du service
- Une implémentation

Apache CXF génère le WSDL en conséquence



Exemple

```
@WebService(name = "HelloService", serviceName = "HelloService")  
public interface HelloService {
```

```
    @WebMethod  
    String hello(String text);
```

```
}
```

```
@WebService(serviceName = "HelloService")  
public class HelloServiceImpl implements HelloService {
```

```
    @WebMethod  
    @Override  
    public String hello(String text) {  
        return "Hello " + text + "!";  
    }
```

```
}
```

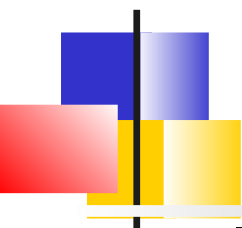


SOAP Client

Le client SOAP nécessite l'interface SEI (*Service Endpoint Interface*) et les classes du modèles.

Plusieurs façons de les obtenir :

- Écrire à la main
- Copier depuis le projet Web Service, s'il est écrit en Java
- Disposer d'un artefact Maven contenant les classes du modèle
- Générer les classes de modèles à partir de WSDL (le plus simple)



Génération à partir du wsdl

L'objectif ***generate-code*** doit être présent dans la configuration de ***quarkus-maven-plugin***

Le fichier WSDL doit être placé dans ***src/main/resources*** et doit être suffixé par ***.wsdl***

La propriété ***quarkus.cxf.codegen.wsdl2java.includes*** doit indiquer l'emplacement des wsdl

Ex : *quarkus.cxf.codegen.wsdl2java.includes = wsdl/*.wsdl*

Les classes sont alors générées dans :
target/generated-sources/wsdl2java



Usage du client

Certaines propriétés doivent être définies dans *application.properties*

```
cxf.it.calculator.baseUri=http://localhost:8082
quarkus.cxf.client.myCalculator.wsdl =
    ${cxf.it.calculator.baseUri}/calculator-ws/CalculatorService?wsdl
quarkus.cxf.client.myCalculator.client-endpoint-url =
    ${cxf.it.calculator.baseUri}/calculator-ws/CalculatorService
quarkus.cxf.client.myCalculator.service-interface =
    org.jboss.eap.quickstarts.wscalculator.calculator.CalculatorService
```

L'annotation `@CXFClient` permet ensuite de s'injecter le SEI



Exemple

```
@Path("/cxf/calculator-client")
public class CxfClientRestResource {

    @CXFClient("myCalculator")
    CalculatorService myCalculator;

    @GET
    @Path("/add")
    @Produces(MediaType.TEXT_PLAIN)
    public int add(@QueryParam("a") int a, @QueryParam("b") int b) {
        return myCalculator.add(a, b);
    }
}
```



Messaging

Support pour le messaging
Reactive Messaging avec Kafka



Introduction

Quarkus supporte de nombreuses alternatives pour le messaging

- ***EventBus*** : Messaging à l'intérieur de la JVM
- ***JMS*** : Standard Jakarta EE
- ***AMQP*** : Performance, implémentation JMS
- ***Kafka*** : Clustering, Performance, Scalability, Fault-tolerance,
- ***RabbitMQ*** : Simple



EventBus

Quarkus Event Bus 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/response*** : 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

@ConsumeEvent indique 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** : Encapsule Adress, body, ...
- Soit juste la payload

La valeur de retour de la méthode est utilisée comme réponse au message. Si *void*, interaction *fire-and-forget*

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



Exemple

```
@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) {
        // retourne un CompletionStage qui se termine lorsque le traitement du msg est terminé.
        // On peut faire échouer le CompletionStage explicitement
    }

    @ConsumeEvent
    public Uni<String> process(String name) {

    }
}
```



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
- le client *Apache ActiveMQ Artemis JMS*
quarkus-artemis-jms

Le standard JMS de JakartaEE, permet l'échange de messages entre processus Java via un broker. (P2P ou PubSub).

Quelques implémentations :

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



Exemple : Producteur JMS

```
@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); }

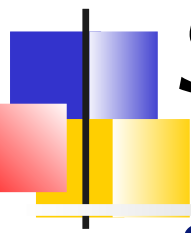
    void onStop(@Observes ShutdownEvent ev) { scheduler.shutdown(); }

    @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);
        }
    }
}
```



Messaging

Support pour le messaging
Reactive Messaging avec Kafka



Smallrye Reactive Messaging

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

C'est une implémentation de **Eclipse MicroProfile Reactive Messaging 2.0** :

- Les applications échangent des **messages** contenant la payload et des méta-données.
- Les messages transitent dans des **canaux** (channels). Les applications se connectent aux canaux.
- Les canaux sont connecté aux système de messagerie sous-jacent via des **connecteurs (connector)**.
Chaque connecteur est dédié à une technologie de messagerie spécifique et sont packagés dans des extensions



Kafka

Apache Kafka a de nombreux cas d'usage :

- Message Broker
- Bufferisation d'événements
- Architecture Event-drive
- Journal d'évènements

Ses fonctionnalités de bases sont :

- Publier des événements vers des topics
- Stocker les évènements de manière durable et fiable dans des **topics** sous la forme de **record**.
- S'abonner aux topics. Et offrir des garanties de livraisons des messages malgré des défaillances
- S'abonner rétroactivement .



Quarkus et Kafka

Quarkus supporte Apache Kafka via :

- *SmallRye Reactive Messaging.*
smallrye-reactive-messaging-kafka
- L'API native Kafka
quarkus-kafka-client
- Kafka Streams (Traitement d'évènements via l'API Kafka Streams)
quarkus-kafka-streams
- Kafka via Camel
camel-quarkus-kafka
- Kafka via Camel sur AWS
camel-quarkus-aws2-msk

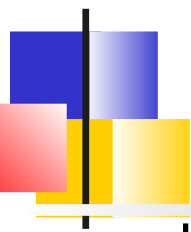


Dev Services

Si une extension liée à Kafka est présente , Dev Services démarre automatiquement un container Kafka prêt à l'emploi (profils *dev* et *test*).

Dev Services peut être configuré :

- **`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 d'un connecteur Kafka gérant le canal *prices* (donc le topic Kafka)
Si un seul connecteur est dans le classpath, cette configuration est optionnelle

–

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

```
mp.messaging.incoming.<nom-du-canal>.attribute=value
```

```
mp.messaging.outcoming.<nom-du-canal>.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



Exemple

```
@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")
    @RestStreamElementType(MediaType.TEXT_PLAIN) // produit MediaType.SERVER_SENT_EVENTS
    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;
```



Sérialiseurs

Des sérialiseurs et des sérialiseurs sont utilisés par Kafka pour stocker les objets Java.

En utilisant l'annotation `@Channel`, on profite de configurations automatiques.

- Le nom du canal est associé à un *topic* du même nom
- Des sérialiseurs sont déduits pour les types de base et pour les classes du domaine si l'on a Jackson ou JSON-B dans le classpath

Le mécanisme d'auto-détection n'est pas applicable lors d'une sérialisation Avro et l'utilisation de registre de Schéma



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. Cela indique au framework que le traitement du message a réussi

- 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
`msg.ack()`;

On peut surcharger le comportement par défaut via
@Acknowledgment



Stratégie de commit

Lorsqu'un message est acquitté, le connecteur invoque une stratégie de commit qui permet de déplacer l'offset géré par Kafka.

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*
- **checkpoint** (expérimental) : L'offset est stocké localement dans un *state store*

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 indiquée par la propriété de configuration ***failure-strategy*** est appliqué

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 et partitions

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

Un topic est divisé en partitions. (Répartition des données sur les différents nœuds du cluster)

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 :

Les méthodes ne prennent pas d'arguments

```
@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

Architecture de la sécurité

Authentication HTTP/S

OpenId Connect

Patterns micro-services



Introduction

Quarkus Security est construit sur 3 interfaces :

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

Quarkus Security fournit de nombreuses implémentations pour s'adapter aux technologies standard de la sécurité

On peut fournir ses propres implémentations.

(*HttpAuthenticationMechanisms*, *IdentityProviders* et *SecurityIdentityAugmentors*)



Authentication Support Quarkus

Quarkus supporte différents mécanismes d'authentification :

- **Basic et Form HTTP-based authentication.**
(Transmission d'un user/password dans une entête http ou dans un POST de formulaire)
- **TLS mutuel** : Basé sur des certificats TLS/SSL (client et serveur)
- **OpenID Connect** ou **SmallRye JWT** : Délégation de l'authentification à un fournisseur de jeton.
Plusieurs extensions sont disponibles selon les cas d'usage
- **WebAuthn**: Basé sur des paires de clé privé/publiques

Les mécanismes d'authentification peuvent être combinés



Autorisation

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

- la configuration
- 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

Architecture de la sécurité

Authentication HTTP/S

OpenId Connect

Patterns micro-services



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*



Identity Provider

Extensions *IdentityProvider* :

- Fichier *.properties*
elytron-security-properties-file
- Base de donnée :
quarkus-elytron-security-jdbc
quarkus-security-jpa
quarkus-security-jpa-reactive
- LDAP :
quarkus-elytron-security-ldap
- Spring Security :
quarkus-spring-security



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
```



Activation SSL

Pour activer SSL :

- Stocker un fichier certificat et une clé dans un keystore
- Fournir l'accès au keystore dans la configuration

Exemple :

```
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*



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
```



Sécurité

Architecture de la sécurité
Authentification HTTP/S
OpenId Connect
Patterns micro-services



Introduction

OpenID et *oAuth2* peuvent être utilisés dans différents cas d'usage

- Authentification d'un utilisateur ou d'un service
- Protection de ressources REST (*oAuth2*)

On s'appuie alors sur un fournisseur de jetons (Authentification, Accès, Rafraîchissement) comme Keycloak

Le fournisseur de jeton stocke

- Les utilisateurs et leurs rôles (SSO, etc.)
- Les applications clientes et leur configuration pour obtenir des jetons (*grant_type* et autre)



Extensions

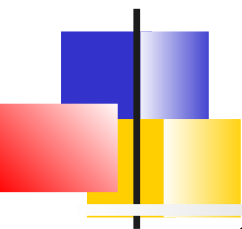
Différentes extensions Quarkus sont utilisées en fonction des usages :

- ***quarkus-oidc*** : fournit un adaptateur OpenID supportant :
 - ***Authorization Code Flow*** : Redirection de l'utilisateur vers le fournisseur de jeton pour retourner un code d'autorisation permettant à l'application d'obtenir les jetons
Les jetons sont au format JWT et peuvent être validés via un clé JWK
 - ***L'extraction du jeton*** de l'entête HTTP *Authorization*
- ***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*.
Permet d'appliquer un filtre sur RestClient pour ajouter le jeton obtenu avec quarkus-oidc-client
- ***quarkus-oidc-token-propagation*** : dépend de *quarkus-oidc*.
Permet d'appliquer un filtre sur RestClient pour propager le jeton obtenu avec quarkus-oidc



Extensions (2)

- ***quarkus-smallrye-jwt*** : fournit une implémentation de Microprofile JWT 1.1.1 pour vérifier les jetons JWT signés et chiffrés.
C'est une alternative à *quarkus-oidc* qui utilise également des jetons au format microprofile-jwt
- ***quarkus-elytron-security-oauth2*** : fournit une alternative au mécanisme de *quarkus-oidc*. Il est capable de gérer des jetons opaques



DevServices et 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 permet d'acquérir les jetons de Keycloak et de tester l'application Quarkus. (Remplacement d'une application SPA)



OpenId Connect

Avec la seule extension de *quarkus-oidc*, il est possible de profiter de la connexion OpenId et de récupérer un jeton au format microprofile-jwt.

C'est le scénario d'un service monolithique et d'une SPA



Sécurité

Architecture de la sécurité
Authentication HTTP/S
OpenId Connect
Patterns micro-services



Sécurité micro-services

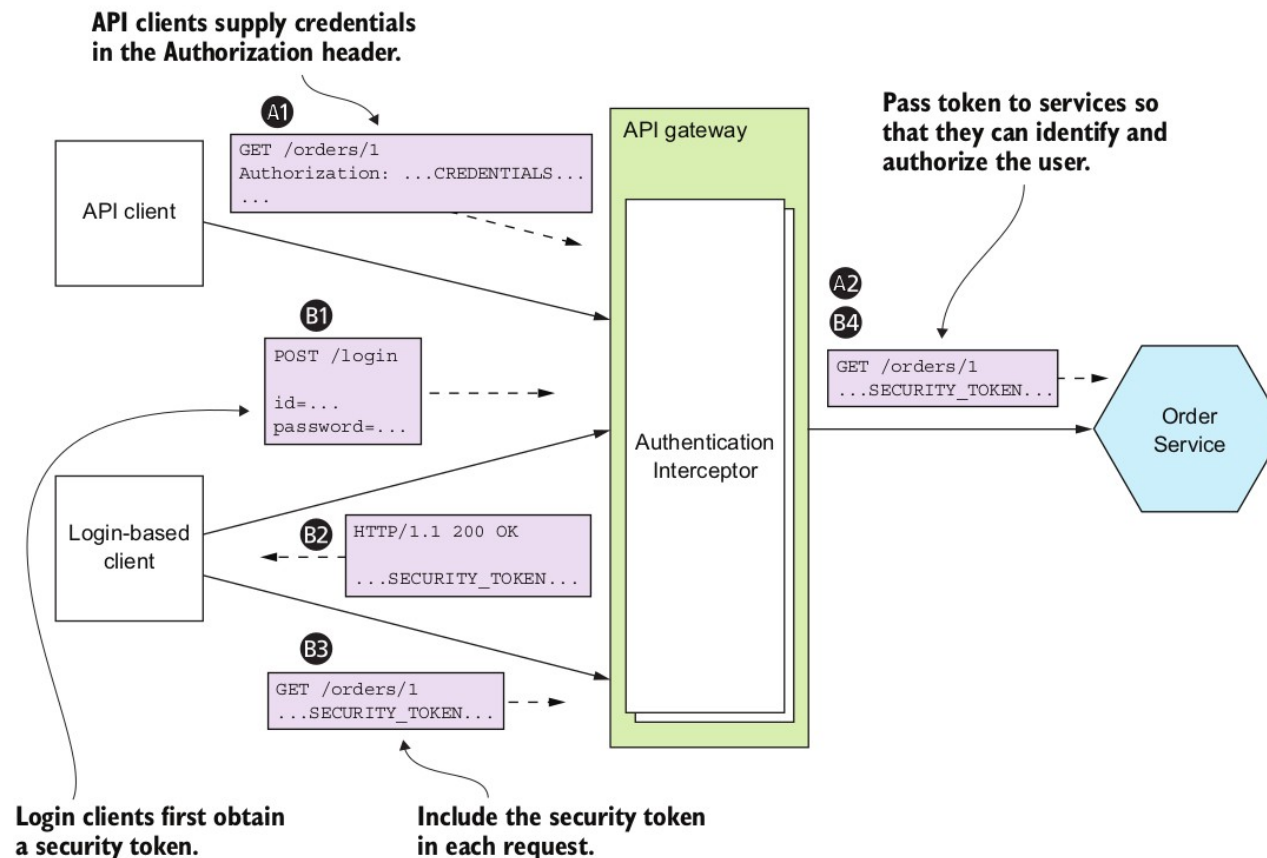
Plusieurs approches pour sécuriser une architecture micro-services :

- 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

L'API gateway est le point d'entrée unique pour les demandes des clients. Il authentifie les requêtes et les transmet à d'autres services, qui peuvent à leur tour appeler d'autres services.





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 se concentre sur JWT

Cette extension est typiquement utilisée :

- Pour propager le jeton venant d'être obtenu par l'Authorization Code Flow.
Ex : requête initiale Gateway



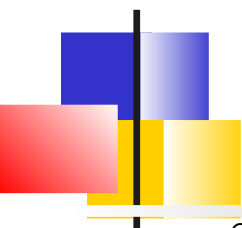
Enregistrement du filtre AccessToken

```
@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

JsonWebTokenRequestFilter

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

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

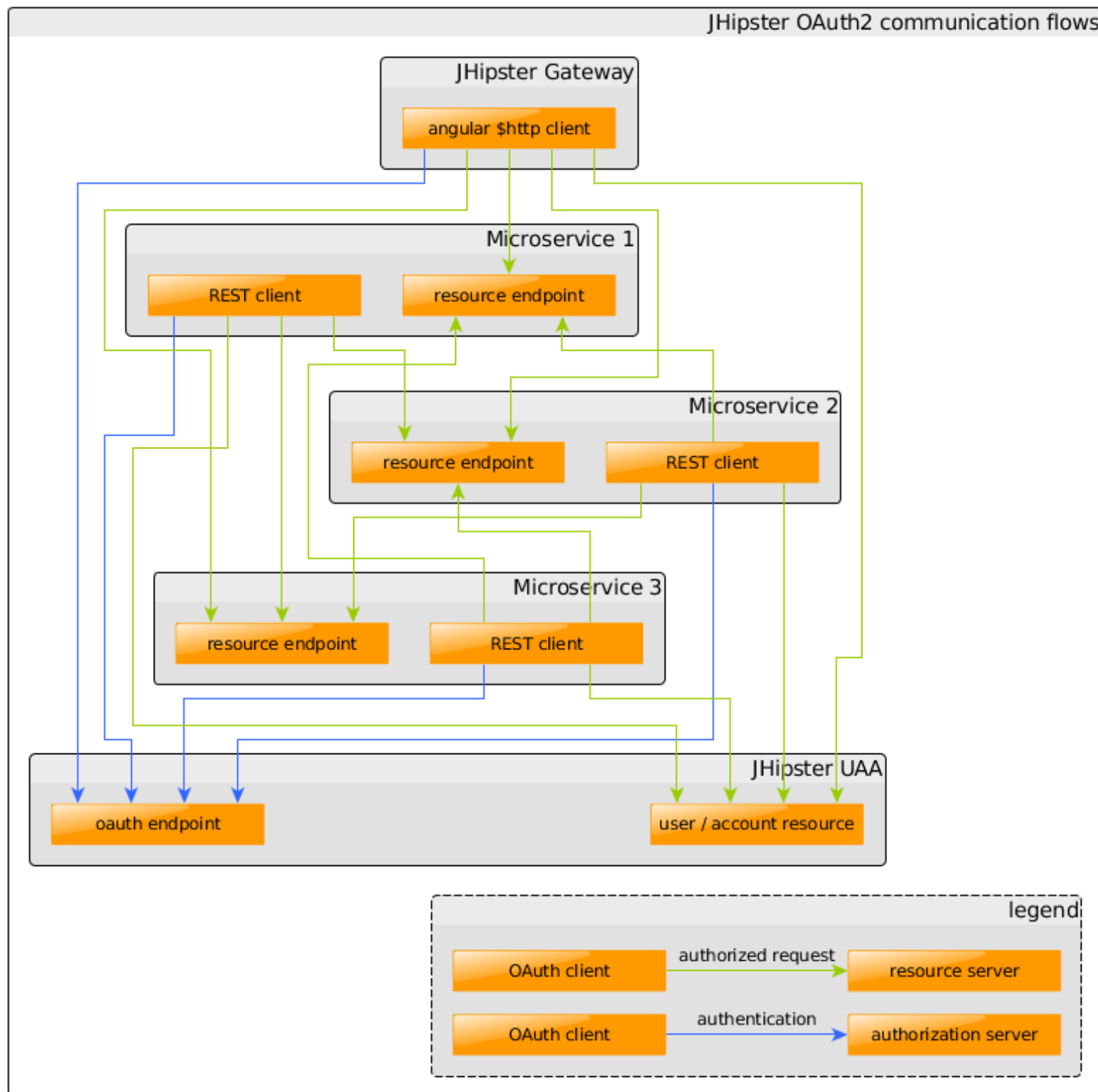
Ou


```
@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
```





Oidc-Client

L'extension ***quarkus-oidc-client*** fournit un *OidcClient* réactif qui peut être utilisé pour acquérir des jetons à l'aide des grant type ***client_credentials*** (défaut) ou ***password*** et actualiser les jetons à l'aide du grant type ***refresh_token***.

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



Configurations du endpoint

Découverte automatique

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

Configuration manuelle

```
quarkus.oidc-client.discovery-enabled=false  
quarkus.oidc-client.token-path=  
    http://localhost:8180/realms/quarkus/protocol/openid-connect/token  
quarkus.oidc-client.grant.type=password  
quarkus.oidc-client.grant-options.password.username=alice  
quarkus.oidc-client.grant-options.password.password=alice
```



Utilisation directe du OidcClient

```
@Path("/service")
public class OidcClientResource {

    @Inject
    OidcClient client;

    volatile Tokens currentTokens;

    @PostConstruct
    public void init() {
        currentTokens = client.getTokens().await().indefinitely();
    }

    @GET
    public String getResponse() {

        Tokens tokens = currentTokens;
        if (tokens.isAccessTokenExpired()) {
            tokens = client.refreshTokens(tokens.getRefreshToken()).await().indefinitely();
            currentTokens = tokens;
        }
        // Use tokens.getAccessToken() to configure MP RestClient Authorization header/etc
    }
}
```

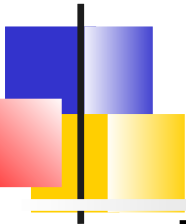


Accès au jeton

```
@Path("/service")
public class OidcClientResource {

    @Inject Tokens tokens;

    @GET
    public String getResponse() {
        // Get the access token, which might have been refreshed.
        String accessToken = tokens.getAccessToken();
        // Use the access token to configure MP RestClient
        Authorization header/etc
    }
}
```



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();
}
```




Container

Construction d'images
Déploiement vers Kubernetes



Introduction

Quarkus fournit des extensions pour construire (et pousser) des images de conteneurs. Actuellement, il prend en charge :

- **Jib : *container-image-jib***
Image optimisée nécessitant pas d'installation de docker.
Cache des librairies quarkus
- **Docker : *container-image-docker***
Image construite à partir des Dockerfiles générés
- **OpenShift: *container-image-openshift***
Téléchargement de l'artefact et construction sur le cluster
- **BuildPack : *container-image-buildpack***
Construction à partir des sources en utilisant la technologie build-pack



Commandes

Construction :

```
quarkus build -Dquarkus.container-image.build=true  
./mvnw clean package -Dquarkus.container-image.build=true
```

Push

```
quarkus build -Dquarkus.container-image.push=true  
./mvnw clean package -Dquarkus.container-image.push=true
```



Container

Construction d'images
Déploiement vers Kubernetes



Introduction

Quarkus offre la possibilité de générer automatiquement des ressources Kubernetes basées sur des valeurs par défaut et une configuration fournie par l'utilisateur

Supporte *Kubernetes*, *OpenShift*, *Knative*

Capable de déployer les service après avoir créer le container et l'avoir pousser dans un registre



Exemple jib

Extensions : ***kubernetes,jib***

```
quarkus create app org.acme:kubernetes-quickstart \
  --extension=resteasy-reactive,kubernetes,jib
quarkus build
```

=> Création de *kubernetes.json* et *kubernetes.yml* dans le répertoire *target/kubernetes* contenant une ressource *Deployment* et une ressource *Service*

Ils utilisent une image par défaut

yourDockerUsername/<artifactId>:<version>

qui peut être personnalisée par :

quarkus.container-image.group, *quarkus.container-image.name* et *quarkus.container-image.tag*



Autres configuration

quarkus.kubernetes.namespace :

Spécification du namespace

quarkus.container-image.registry :

L'adresse du registre

***quarkus.kubernetes.name*, *quarkus.kubernetes.version* et
*quarkus.kubernetes.part-of :***

Les labels recommandés

quarkus.kubernetes.labels.foo :

Un label personnalisé

quarkus.kubernetes.annotations.foo

Annotation personnalisée

quarkus.kubernetes.replicas

Le nombre de réplique



Variables d'environnement

Kubernetes propose plusieurs façons de définir les variables d'environnement :

- paires clé/valeur
- importer toutes les valeurs d'un Secret ou d'un *ConfigMap*
- interpoler une seule valeur identifiée par un champ donné dans un *Secret* ou *ConfigMap*
- interpoler une valeur à partir d'un champ au sein de la même ressource



Exemples

Clé valeur

```
quarkus.kubernetes.env.vars.my-env-var=foobar
```

A partir d'un secret

```
quarkus.kubernetes.env.secrets=my-secret,my-other-secret
```

Une valeur d'un secret

```
quarkus.kubernetes.env.mapping.foo.from-secret=my-secret
```

```
quarkus.kubernetes.env.mapping.foo.with-key=keyName
```

Toutes les valeurs d'une ConfigMap

```
quarkus.kubernetes.env.configmaps=my-config-map,another-config-map
```

Une valeur d'une ConfigMap

```
quarkus.kubernetes.env.mapping.foo.from-configmap=my-configmap
```

```
quarkus.kubernetes.env.mapping.foo.with-key=keyName
```

A partir d'un champ de la ressource

```
quarkus.kubernetes.env.fields.foo=metadata.name
```



Montage de volume

Montage du volume my-volume vers le chemin /where/to/mount

`quarkus.kubernetes.mounts.my-volume.path=/where/to/mount`

Volume secret

`quarkus.kubernetes.secret-volumes.my-volume.secret-name=my-secret`

Volume ConfigMap

`quarkus.kubernetes.config-map-volumes.my-volume.config-map-name=my-secret`



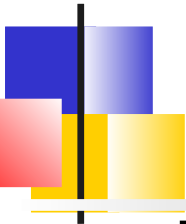
Configuration applicative

La configuration applicative externe (SmallRye Config) s'effectue généralement en

- Définissant un volume
- Montant le volume
- Créer une variable d'environnement
SMALLRYE_CONFIG_LOCATIONS

Quarkus fournit des raccourcis avec 2 propriétés de configuration :

```
quarkus.kubernetes.app-secret=<nom du secret contenant la config>  
quarkus.kubernetes.app-config-map=<nom du ConfigMap contenant la config>
```



Readiness et Liveness probes

Nécessite l'extension :

quarkus-smallrye-health

Les valeurs des sondes sont déterminées par les propriétés :

quarkus.smallrye-health.root-path,
quarkus.smallrye-health.liveness-path et
quarkus.smallrye-health.readiness-path

D'autres propriétés peuvent être configurées :

`quarkus.kubernetes.readiness-probe.initial-delay=20s`

`quarkus.kubernetes.readiness-probe.period=45s`



Extensions pour les distributions Kubernetes

Quarkus fournit également des extensions pour des distributions Kubernetes de développement :

- ***quarkus-minikube : minikube***
- ***quarkus-kind : kind***



Mise en place Développement distant

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
```



Observabilité

HealthCheck

OpenTracing

Métriques

Centralisation des traces



Introduction

Quarkus utilise ***SmallRye Health***, une implémentation de la spécification *MicroProfile Health* qui permet de fournir des informations sur l'état de l'application.

Typiquement, les sondes utilisées par Kubernetes.



Mise en place

Extension : ***smallrye-health***

Les endpoints REST suivants sont alors exposés :

- ***/q/health/live*** : L'application s'exécute
- ***/q/health/ready*** : L'application est prête à répondre aux requêtes
- ***/q/health/started*** : L'application est démarré
- ***/q/health*** : Agrège toutes les vérifications de santé implémentées dans l'application



Création de Health Check

Les tests de santé sont implémentés par des beans implémentant l'interface **HealthCheck** et annotés avec :

- *@Readiness*
- *@Liveness*
- Ou *@Startup*

L'interface définit une méthode :

```
public HealthCheckResponse call()
```

Certaines extensions fournissent déjà leur « health check » : Exemple *agroal* pour les sources de données



Example

@Liveness

@ApplicationScoped

public class DataHealthCheck implements **HealthCheck** {

@Override

public HealthCheckResponse call() {

 return HealthCheckResponse.named("Health check with data")

 .up()

 .withData("foo", "fooValue")

 .withData("bar", "barValue")

 .build();

 }

}



Observabilité

HealthCheck

OpenTracing

Métriques

Centralisation des traces



Tracing distribué

Le tracing distribué consiste à être capable de suivre le cheminement des requêtes traversant plusieurs micro-services.

Quarkus propose 2 extensions basées sur :

- *OpenTelemetry*
- *OpenTracing*



Exemple *OpenTracing*

Extension : ***quarkus-opentelemetry-exporter-otlp***

Démarrage d'un serveur de collecte des traces (*jaeger* / *zipkin*)

Configuration (Exemple OpenTelemetry):

```
quarkus.application.name=myservice
quarkus.opentelemetry.enabled=true
quarkus.opentelemetry.tracer.exporter.otlp.endpoint=http://localhost:4317
quarkus.opentelemetry.tracer.exporter.otlp.headers=Authorization=Bearer my_secret
# Autorise la compatibilité zipkin
quarkus.jaeger.zipkin.compatibility-mode=true
```



Tracing

Les appels REST sont alors automatiquement tracés. On peut ajouter des tracing supplémentaire :

- Appel d'une méthode service :
l'annoter avec **@Traced**
- Jdbc :
extension **opentracing-jdbc**
- Kafka :
extension **opentracing-kafka-client**
- MongoDB :
extension **opentracing-mongo-common**



Observabilité

HealthCheck

OpenTracing

Métriques

Centralisation des traces



Agrégation de métriques

Quarkus offre du support pour :

- ***Micrometer/Prometheus***

Permet le développement de métriques personnalisés et l'utilisation de métriques fournies par certaines extensions *Quarkus*
Méthode recommandée

Extension :

micrometer-registry-prometheus

- ***SmallRye Metrics*** une implémentation de la spécification *MicroProfileMetrics*

Extension :

smallrye-metrics



Micrometer

Micrometer est une librairie qui fournit un mécanisme d'enregistrement pour les métriques et les types de métriques de base (*Counter, Gauge, Timer, etc.*).

Une application (ou une bibliothèque) peut enregistrer un type de metrique avec un ***MeterRegistry***.

Ces métriques sont ensuite exploitées par un système de monitoring comme Prometheus



Extensions

Quarkus fournit donc :

- ***quarkus-micrometer*** qui fournit l'intégration à Micrometer
- ***micrometer-registry-prometheus*** qui intègre Prometheus

D'autres extensions intégrant d'autres systèmes de monitoring existent :

micrometer-registry-azure-monitor, micrometer-registry-datadog, micrometer-registry-graphite, micrometer-registry-influx, micrometer-registry-jmx, ...



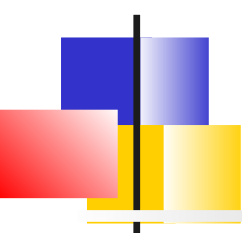
Métriques automatiquement générés

L'extension *Micrometer* génère automatiquement des métriques selon les conventions de nommage de *Prometheus* :

- *http_server_requests_seconds_count*
- *http_server_requests_seconds_sum*
- *http_server_requests_seconds_max*

Des étiquettes permettant des agrégations sont ajoutées : la méthode HTTP (GET, POST, etc.), le code de statut (200, 302, 404, etc.)

Les métriques concernant la JVM sont également générés automatiquement



Extensions instrumentalisées

La plupart des extensions Quarkus peuvent être instrumentalisées par Micrometer.

Pour cela, il suffit d'activer les métriques dans la configuration.

Par exemple :

```
quarkus.datasource.metrics.enabled=true
```



Métrique personnalisé

La création de son propre métrique nécessite :

- Définir un nom respectant les conventions `MicroMeter`
- Définir les tags utilisés pour les agrégations
- S'injecter `MeterRegistry`
- Y définir une Jauge, un compteur, une valeur résumé, un Timer (*Gauge, Counter, Summaries, Timers*)



Exemple Jauge

```
@Path("/example")
@Produces("text/plain")
public class ExampleResource {

    private final MeterRegistry registry;
    LinkedList<Long> list = new LinkedList<>();

    // Création de la gauge
    ExampleResource(MeterRegistry registry) {
        this.registry = registry;
        registry.gaugeCollectionSize("example.list.size", Tags.empty(), list);
    }

    @GET
    @Path("gauge/{number}")
    public Long checkListSize(long number) {
        if (number == 2 || number % 2 == 0) {
            list.add(number);
        } else {
            try {
                number = list.removeFirst();
            } catch (NoSuchElementException nse) { number = 0; }
        }
        return number;
    }
}
```



Exemple (2)

```
curl http://localhost:8080/example/gauge/1
curl http://localhost:8080/example/gauge/2
curl http://localhost:8080/example/gauge/4
curl http://localhost:8080/q/metrics
curl http://localhost:8080/example/gauge/6
curl http://localhost:8080/example/gauge/5
curl http://localhost:8080/example/gauge/7
curl http://localhost:8080/q/metrics
```

Les URLs */q/metrics* affichent (au format texte) la valeur de *example_list_size*

Typiquement ces URLs sont exécutées par Prometheus qui connecté à un Grafana permet d'afficher de beaux graphiques



Observabilité

HealthCheck
OpenTracing
Métriques

Centralisation des traces



Centralisation des logs

L'extension ***logging-gelf*** permet de configurer un handler pour envoyer les logs sur un endpoint UDP sur un port particulier.

```
quarkus.log.handler.gelf.enabled=true  
quarkus.log.handler.gelf.host=localhost  
quarkus.log.handler.gelf.port=12201
```

Il suffit alors de s'équiper d'une solution comme *ElasticStack*, *Graylog* ou *FluentD* pour injecter les traces dans une pipeline de transformation ou de normalisation qui les stocke dans des index offrant des fonctionnalités de recherche



Annexes

OAuth2

Concept Kubernetes
Compléments sur les ressources
Kubernetes



Rôles du protocole

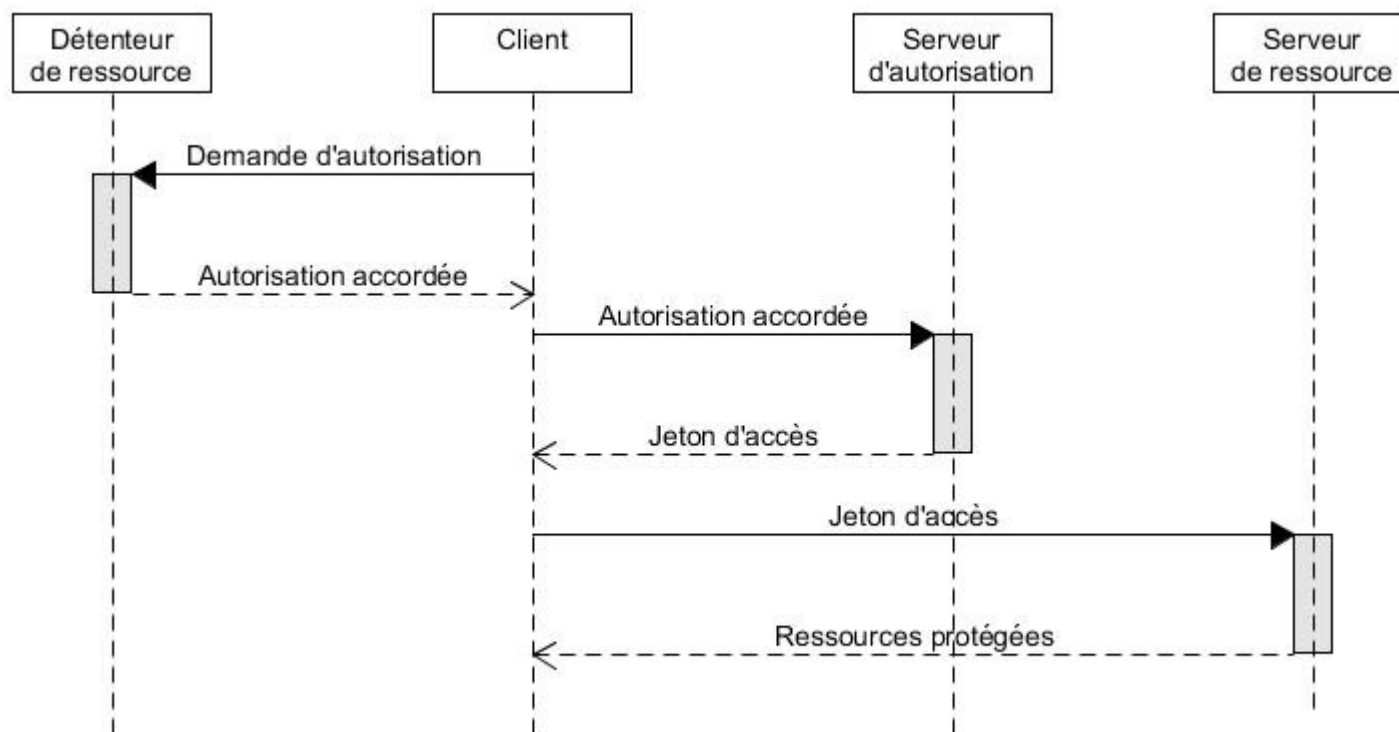
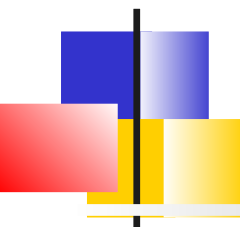
Le **Client** est l'application qui essaie d'accéder au compte utilisateur. Elle a besoin d'obtenir une permission de l'utilisateur pour le faire.

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

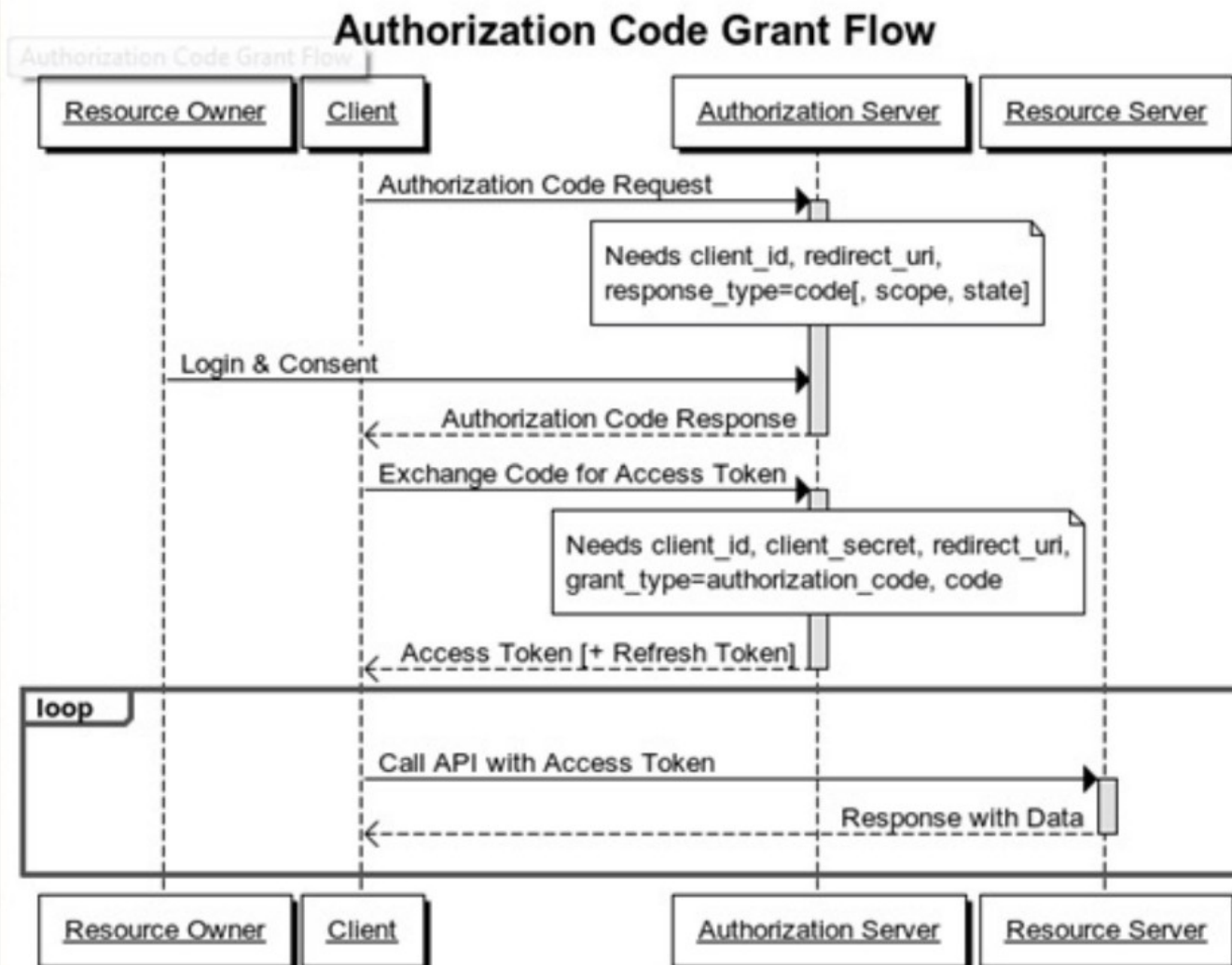


OAuth2 Grant Type

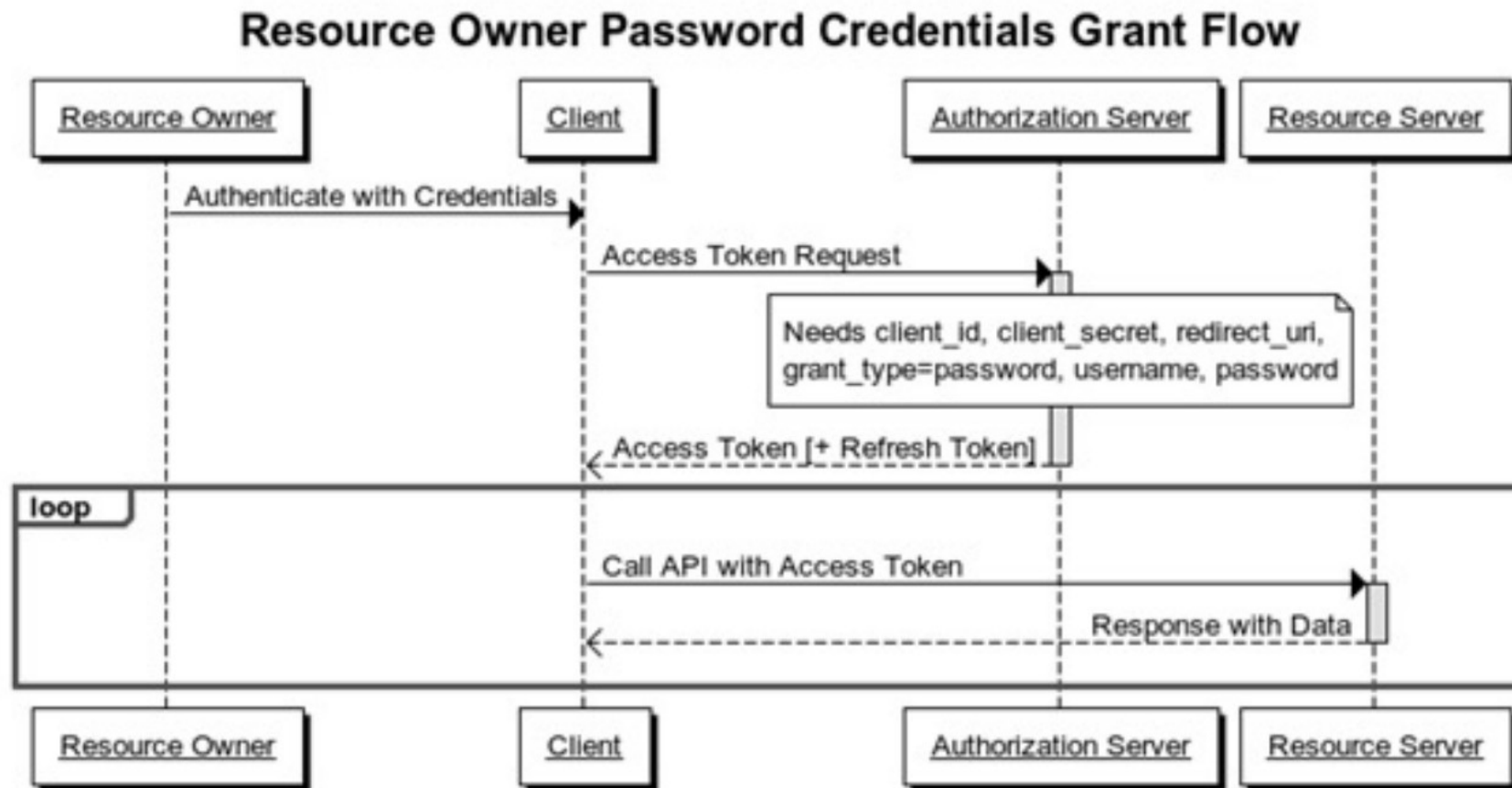
Différents moyens afin que l'utilisateur donne son accord : les **grant types**

- ***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



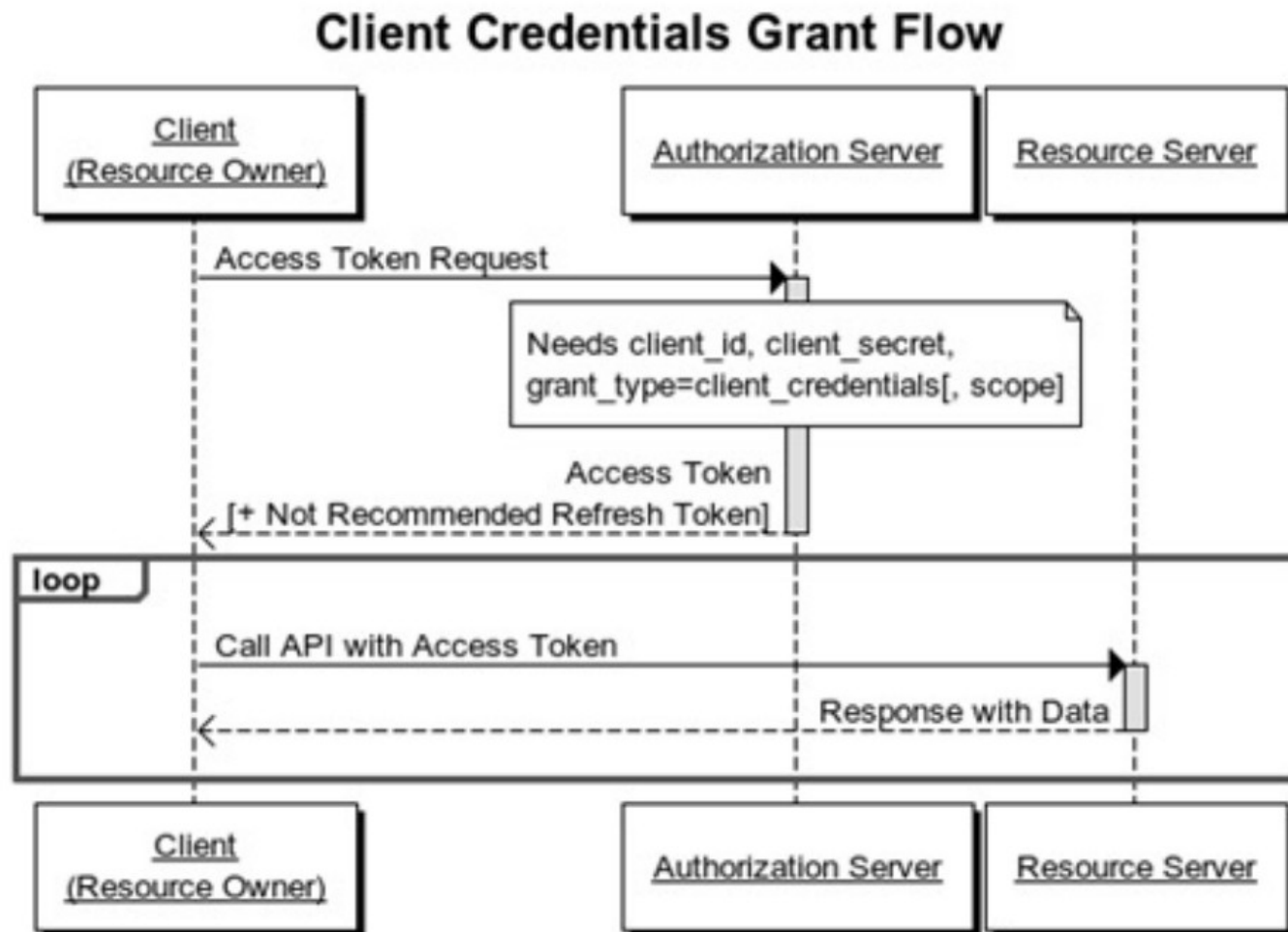


OpenAPI

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>

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***



Annexes

OAuth2

Concept Kubernetes

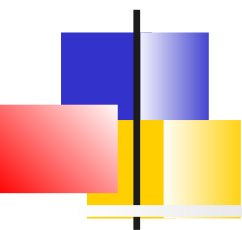
Compléments sur les ressources
Kubernetes



Auto-correctif

Kubernetes va TOUJOURS essayer de diriger le cluster vers son état désiré.

- **Moi**: «Je veux que 3 instances de Redis toujours en fonctionnement.»
- **Kubernetes**: «OK, je vais m'assurer qu'il y a toujours 3 instances en cours d'exécution. »
- **Kubernetes**: «Oh regarde, il y en a un qui est mort. Je vais essayer d'en créer un nouveau. »

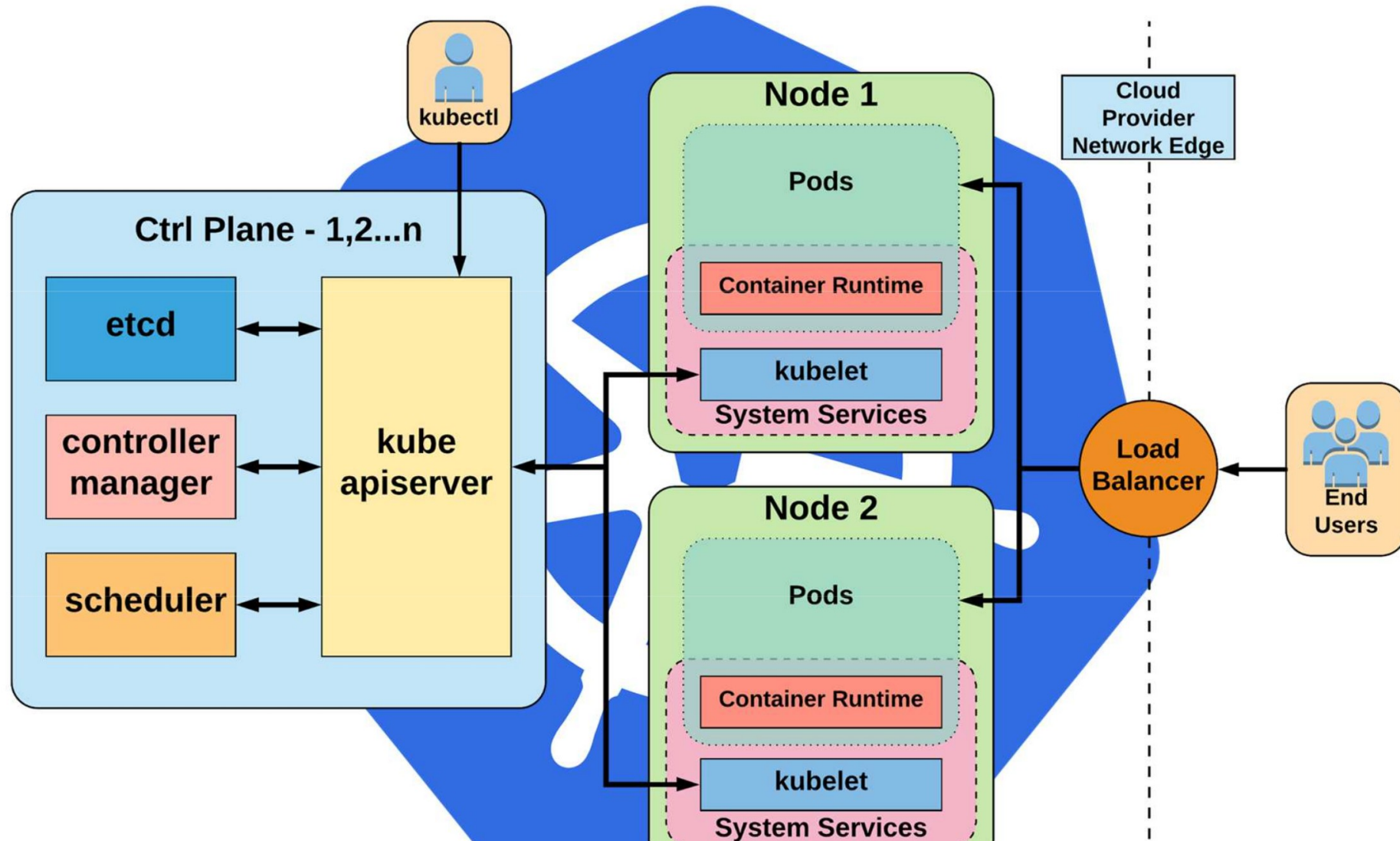


Fonctionnalités applicatives

- Scaling automatique
- Déploiements Blue/Green
- Démarrage de jobs planifiés
- Gestion d'application Stateless et Stateful
- Méthodes natives pour la découverte de services
- Intégration et support d'applications fournies par des tiers (*Helm*)

pod = 1 ou plusieurs conteneurs co-localisés

Architecture cluster





API

L'interaction se fait par une API Rest très riche.

L'API est très cohérente et tous les appels suivent le même format

Format:

`/apis/<group>/<version>/<resource>`

Examples:

`/apis/apps/v1/deployments`

`/apis/batch/v1beta1/cronjobs`

L'outil ***kubectl*** et le format ***yaml*** sont les plus appropriés pour effectuer les requêtes REST

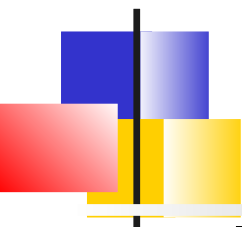


Principes

L'API est une API Rest, elle permet principalement des opérations CRUD sur des **ressources**

En particulier, le client *kubectl* propose les commandes :

- **create** : Créer une ressource
- **get** : Récupérer une ressource
- **edit/set** : Mise à jour d'une ressource
- **delete** : Suppression d'une ressource



Ressources applicatives

Les principales ressources d'une application sont :

- **deployment** : Un déploiement, les déploiements font référence à des *ReplicaSet*, ils peuvent être historisés
- **replicaSet** : Ils définissent le nombre d'instances maximales pour une image de conteneur applicative
- **pod** : Ce sont des conteneurs qui s'exécutent, ils sont distribués sur les nœuds par le scheduler de *Kubernetes*
- **service** : Ce sont des point d'accès stable à un service applicatif

pod

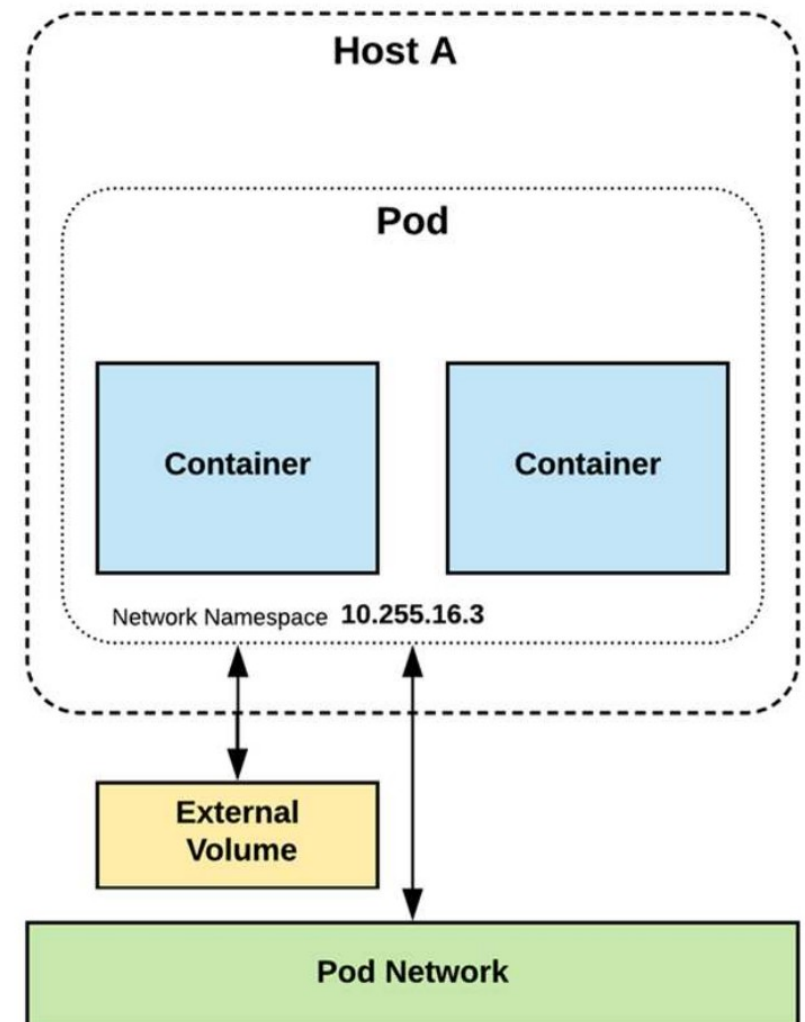
Un **pod** est la plus petite unité de travail

Un *pod* regroupe un ou plusieurs conteneurs qui partagent :

- Une adresse réseau
- Les mêmes volumes

Les pods sont éphémères. Ils disparaissent lorsqu'ils :

- Sont terminés
- Ont échoués
- Sont expulsés par manque de ressources





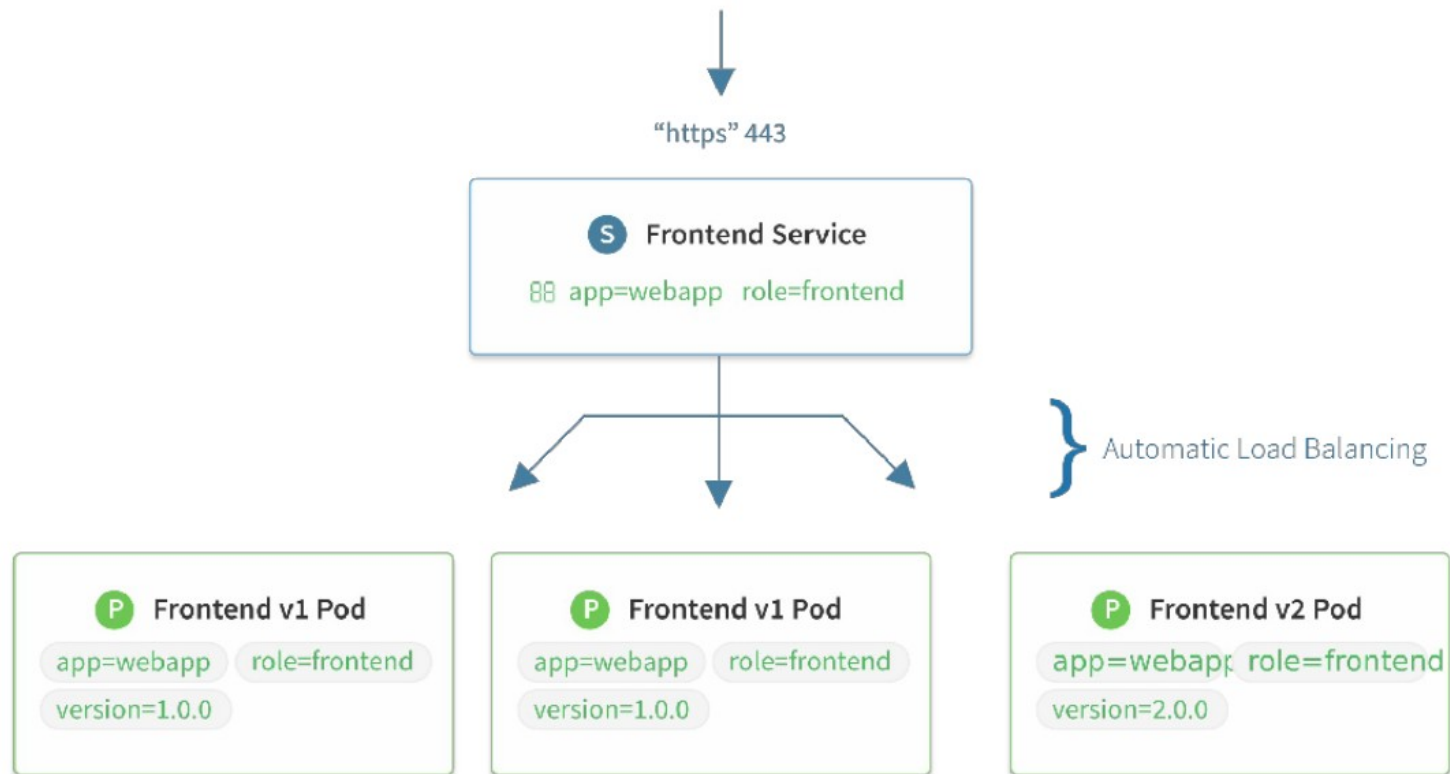
Services

Un **service** est une méthode unifiée d'accès aux charges de travail exposées des *Pods*.

Ressource durable. Les services ne sont pas éphémères :

- IP statique du cluster
- Nom DNS statique (unique à l'intérieur d'un espace de nom)

Service





Ressource *deployment*

Exemple description d'un déploiement:

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  spec:
    containers:
      - image: dthibau/annuaire
        name: annuaire
```

A partir de ce type de fichier *.yaml*, on peut créer la ressource via :

```
kubectl create -f ./my-manifest.yaml
```



Exemple service

Un service nommé *my-service* qui représente tous les pods ayant le **label *app=MyApp*** et qui mappe son port 80 vers le port 80 des pods

```
kind: Service
  apiVersion: v1
  metadata:
    name: my-service
  spec:
    selector:
      app: MyApp
    ports:
      - protocol: TCP
        port: 80
        targetPort: 80
```



Type de service

Un service peut avoir plusieurs types :

- **ClusterIP (défaut)** : Expose le service sur une IP interne au cluster. Le service n'est pas accessible de l'extérieur
- **NodePort** : Expose le service sur un port statique créé automatiquement sur chaque nœud du cluster. Le service est accessible de l'extérieur via `<ClusterIP>:<NodePort>`
- **LoadBalancer** : Expose le service en externe à l'aide de l'équilibreur de charge d'un fournisseur de cloud.
- **ExternalName** : Mappe le service au contenu du champ `externalName` (par exemple `foo.bar.example.com`),



Commandes *kubectl*

get : Afficher 1 ou plusieurs ressources

describe : Afficher les détails sur une ou plusieurs ressources

create : Crée une ressource à partir d'un fichier ou de stdin.

set : Mettre à jour des attributs sur une ressource

edit : Éditer une ressource

delete : Supprimer des ressources

logs : Afficher les logs d'un container

expose : Exposer un déploiement en tant que service

execute : Exécuter une image particulière sur le cluster

attach : S'attacher à un container qui s'exécute

exec : Exécuter une commande dans un container

port-forward : Forward un ou plusieurs ports d'un pod

cp : Copier des fichiers entre conteneurs

auth : Inspecter les autorisations

...



Exemples

Affiche les paramètres fusionnés de *kubeconfig*

kubectl config view

Liste tous les services du namespace par défaut

kubectl get services

Liste tous les pods de tous les namespaces

kubectl get pods --all-namespaces

Description complète d'un pod

kubectl describe pods my-pod

Supprime les pods et services ayant le noms "baz"

kubectl delete pod,service baz

Affiche les logs du pod (stdout)

kubectl logs my-pod

S'attacher à un conteneur en cours d'exécution

kubectl attach my-pod -i

Exécute une commande dans un pod existant (un seul conteneur)

kubectl exec my-pod -- ls /

Visualiser la consommation mémoire et CPU des pods

kubectl top pod

Écoute le port 5000 de la machine locale et forward vers le port 6000 de my-pod

kubectl port-forward my-pod 5000:6000



La commande *apply*

Dans la pratique, la commande ***apply*** avec en paramètre un fichier *.yaml* décrivant la ressource est la plus adaptée pour des déploiements via *kubectl* :

- Elle peut créer ou modifier la ressource
- Les fichiers *.yaml* décrivant les ressources à déployer sont committés, versionnés dans le dépôt des sources

```
kubectl apply -f ./my-manifest.yaml
```




Déploiement

La ressource **deployment** permet de manipuler un ensemble de *Replicaset* (*ensemble de conteneurs répliqués*)

Les principales actions que l'on peut faire sur un déploiement sont :

- Le **rollout**: Création/Mise à jour entraînant la création des pods en arrière-plan
- Le **rollback**: Permet de revenir à une ancienne version des *ReplicaSets*
- La **scalabilité** horizontale : Permet de mettre en échelle l'application horizontalement
- La mise en pause
- La suppression de vieilles versions



Commandes de déploiement *kubectl*

Mettre à jour une image dans un déploiement existant

Enregistrer la mise à jour

```
kubectl set image deployment/nginx-deployment  
nginx=nginx:1.9.1 -record
```

Regarder le statut d'un rollout

```
kubectl rollout status deployment/nginx-deployment
```

Obtenir l'historique des révisions

```
kubectl rollout history deployment/nginx-deployment
```

Roll-back sur la version précédente

```
kubectl rollout undo deployment/nginx-deployment
```

Scaling

```
kubectl scale deployment/nginx-deployment --replicas=10
```



Scheduler et Workload

Les actions de l'API sont souvent asynchrones

Pour *Kubernetes*, ces ordres sont considérés comme des **workloads** à exécuter via le scheduler.

Les *workload* sont visibles via l'API, elles comportent 2 blocs de données :

- ***spec*** : La spécification de la ressource
- ***status*** : Est géré par *Kubernetes* et décrit l'état actuel de l'objet et son historique.



Autres ressources du cluster

ClusterRole : Rôle avec permissions sur l'API

VolumePersistent : Système de stockage

PersistentVolumeClaims : Demande d'usage d'un volume persistant

ConfigMaps : Stockage clé-valeur pour la configuration

Secrets : Stockage de créidentiels



Namespace

Kubernetes prend en charge plusieurs clusters virtuels soutenus par le même cluster physique.

Ces clusters virtuels sont appelés **espaces de noms**.

- Les noms des ressources doivent être uniques dans un espace de noms, mais pas entre les espaces de noms.
- Chaque ressource *Kubernetes* ne peut être que dans un seul *namespace*

Les *namespaces* sont généralement utilisés dans des clusters utilisés par différentes équipes



Labels et sélecteurs

Les **labels** sont des paires clé / valeur attachées à des objets, tels que des pods, des services, des déploiements

Ils sont utilisés pour organiser et sélectionner des sous-ensembles d'objets.

Les **sélecteurs** permettent de rechercher des objets ayant des labels spécifiques.

Il y a 2 types de sélecteurs: égalité ou ensemble.

- Ils sont utilisés par les opérations *LIST* et *WATCH* de l'API
- Les services et les ReplicaSet utilisent les labels et les sélecteurs pour sélectionner les pods



Annotations

Les **annotations** (*metadata*) permettent d'attacher des métadonnées arbitraires non identifiables à des objets.

- Les clients tels que les outils et les bibliothèques peuvent récupérer ces métadonnées.



Écosystème *Kubernetes*

De nombreux outils peuvent compléter une installation cœur de Kubernetes :

- **CoreDNS** : Permet de déclarer dans un DNS interne les services (qui deviennent accessibles via leur nom)
- **Helm** : Système de gestion de package permettant d'automatiser l'installation d'autres outils (ressources *Kubernetes*)
- **Prometheus** : Monitoring du cluster, généralement associé à Grafana
- **Ingress** : Permettant d'exposer les services à l'extérieur du cluster
- **Istio** : Maillage de service (services mesh), gère les communications inter-pods



Distribution Kubernetes

Kubernetes est disponible en OpenSource mais une installation nécessite encore beaucoup d'expertise ... et beaucoup de ressources

Kubernetes est donc proposé par les acteurs du cloud

- Amazon Elastic Container Service for Kubernetes
- Azure Kubernetes Services
- Google Kubernetes Engine
- Digital Ocean
- ...

Il est également disponible en version « dev » mono-nœud : *microk8s*, *minikube*, *kind*

Des versions en ligne comme : <https://labs.play-with-k8s.com/>

L'outil *Rancher* permet de gérer graphiquement plusieurs installation

Terraform permet de provisionner des cluster (et services) as Code



Annexes

OAuth2

Concept Kubernetes

***Compléments sur les ressources
Kubernetes***



Limites Mémoire et CPU des pods

Il est possible de positionner les limites mémoires et CPU des containers d'une ressource de déploiement

Les exploitants peuvent également définir ces limites au niveau d'un namespace.

containers:

- name: memory-demo-ctr

image: polinux/stress

resources:

limits: #Max

memory: "200Mi"

cpu : "1"

requests: #Min

memory: "100Mi"

cpu : "0.5"



QoS

Il est possible d'affecter des classes de Qualité de Service (QoS) aux pods.

Kubernetes utilisent ces QoS pour programmer ou supprimer des pods

3 classes existent :

- **Guaranteed** : Tous les containers du pod ont définis la mémoire et le CPU à une unique valeur (request = limits)
- **Burstable** : La condition Guaranteed n'est pas respecté mais au moins 1 des container spécifie une limite mémoire ou CPU
- **BestEffort** : Les containers n'ont pas défini de limite mémoire et CPU



Exemple Guaranteed

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

```
kubectl get pod qos-demo --namespace=qos-example --output=yaml
```



Conséquences

Le scheduler attribue des *Pods guaranteed* uniquement aux nœuds qui disposent de suffisamment de ressources

Le scheduler ne pourra pas garantir que les *Pods Burstable* soient placés sur des nœuds disposant de suffisamment de ressources.

Il n'est pas garanti que les *Pods BestEffort* soient placés sur des nœuds disposant de suffisamment de ressources pour eux.

N'ayant pas de limites, ils peuvent créer des problèmes pour les autres pods



Utilisation de volumes

Un volume permet de monter de nouveaux répertoires dans les container d'un pod.

Un volume est conservé pendant tout la vie du pod (même si ces containers redémarrent)

Le volume est persistant si il survit à un redémarrage du pod (ou même du nœud)



Utilisation de volume

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

=> Un nouveau répertoire vide est disponible dans le conteneur Redis, il peut y lire et y écrire, d'autres conteneurs du pod pourraient monter cet espace de stockage sur d'autres répertoires



Volume persistant

L'utilisation d'un volume persistant nécessite :

- 1) Un administrateur crée un **PersistentVolume** correspondant à un stockage physique.
- 2) Le développeur crée un **PersistentVolumeClaim** qui est automatiquement associé à un *PersistentVolume*.
- 3) Un pod utilise le *PersistentVolumeClaim* pour le stockage



PersistentVolume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```



PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```



Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```



ReplicaSet

Un ReplicaSet (ensemble de réplicas en français) est utilisé pour garantir la disponibilité d'un certain nombre identique de Pods

Un ReplicaSet est défini :

- un selecteur qui identifie les Pods
- un nombre de replicas
- et un modèle de Pod.



Example

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
```



Horizontal Pod Autoscaler

Un ReplicaSet peut également être une cible pour un **Horizontal Pod Autoscalers** (HPA).

L'HPA permet le scaling automatique

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-scaler
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: frontend
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Voir également : *kubectl autoscale deployment foo --min=2 --max=10*



Sondes Kubernetes

Le kubelet utilise des sondes et prend des actions en conséquences :

- **Liveness** est utilisé pour redémarrer un conteneur. Par exemple, lorsqu'une application est en cours d'exécution, mais incapable de progresser (deadlock).
- **Readiness** est utilisé pour savoir si un conteneur est prêt à accepter du trafic. Lorsqu'un pod n'est pas prêt, il est supprimé des équilibres de charge d'un service.
- **Startup** détermine quand un conteneur a démarré. Les vérifications *liveness* et *readiness* sont désactivées tant que cette sonde n'est pas correcte.