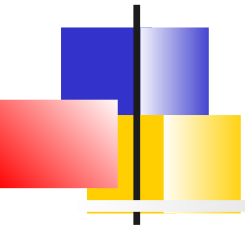
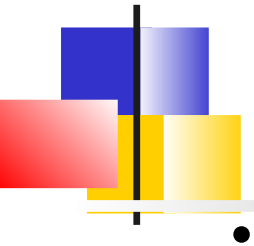


Eclipse Micro-profile
spécifications pour les microservices

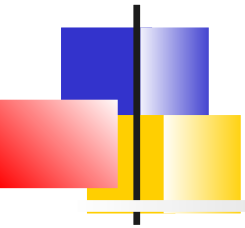
David THIBAU – 2022

david.thibau@gmail.com



Agenda

- **Introduction**
  - Architectures micro services
  - Services techniques pour les micro-services
  - API Micro-Profile
- **APIs Jakarta EE**
  - CDI
  - JAX-RX
  - JSON-P, JSON-B
  - MicroProfile Config
  - MicroProfile OpenAPI
- **APIs pour la distribution**
  - RestClient
  - Fault Tolerance
  - Reactive Messaging
- **APIs pour l'observabilité**
  - Health
  - Metrics
  - OpenTracing
- **Sécurité**
  - Rappels OpenID Connect, oAuth2, JWT
  - Spécification JWT

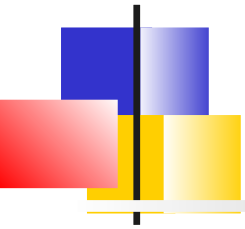


Introduction

Architectures micro services

Services techniques

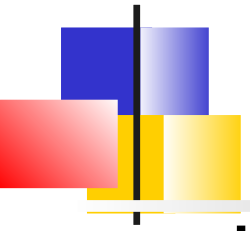
Eclipse MicroProfile



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

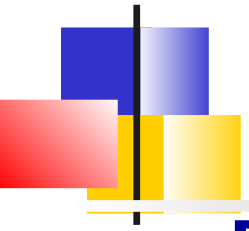
Maintenance facilitée : Les services sont plus petits
=> Corrections, évolutions plus rapide

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

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

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

=> Favorise le partage et les montées en compétences



Caractéristiques

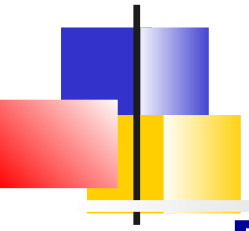
Design piloté par le métier : La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)

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

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

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

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



Contraintes

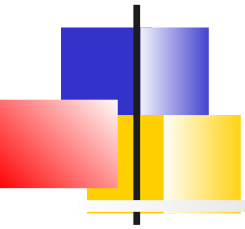
Réplication : Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)

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

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

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

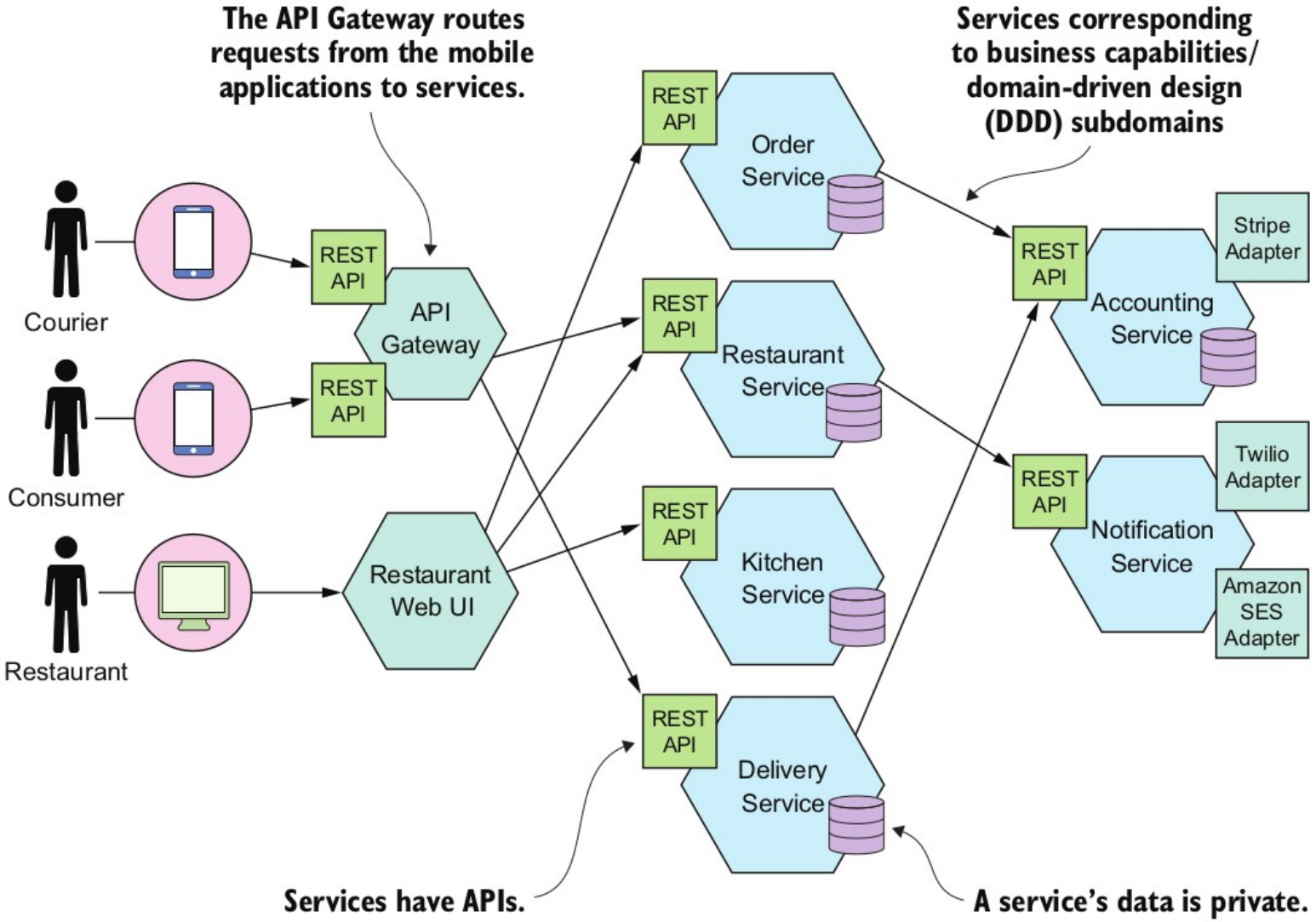
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.



Inconvénients et difficultés

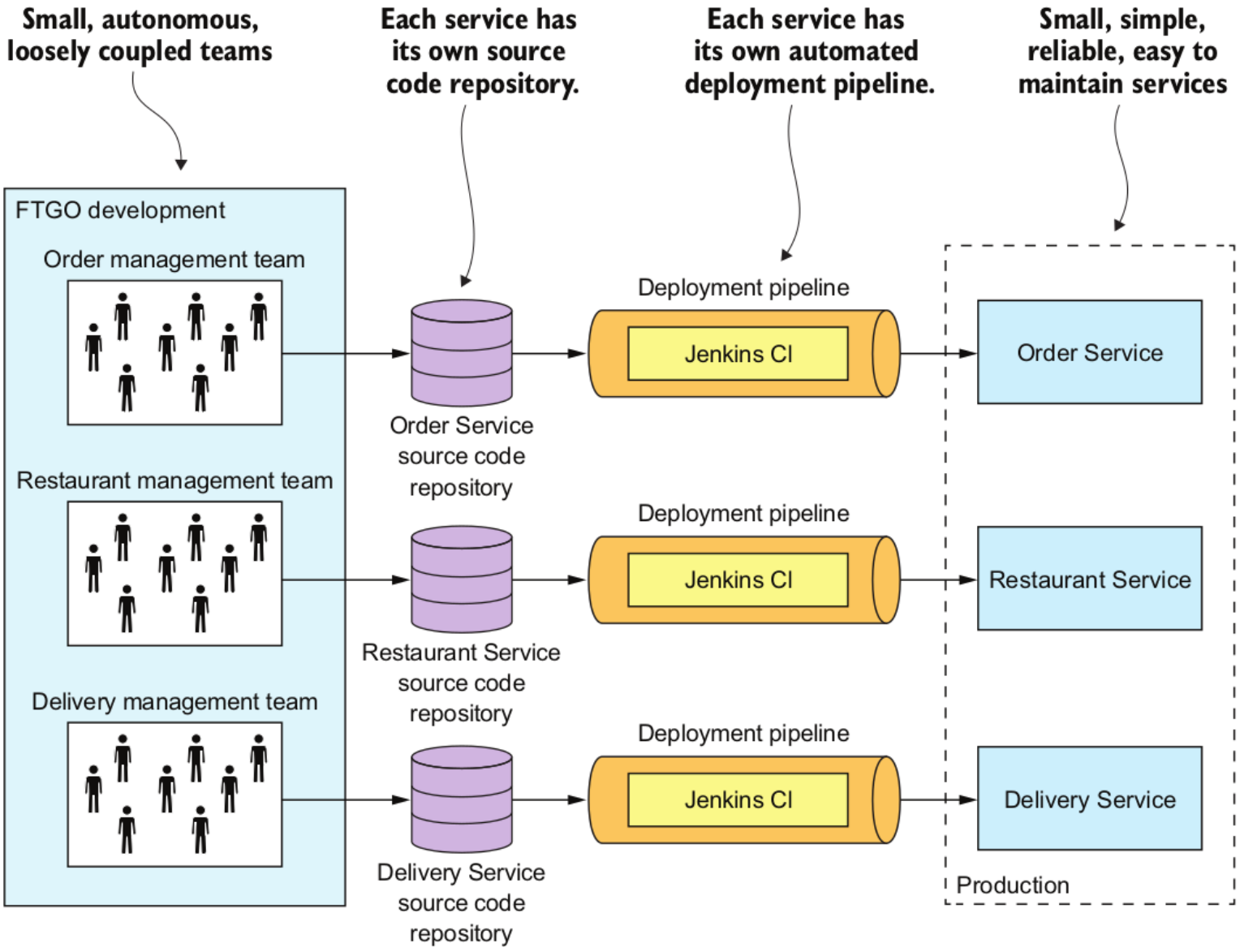
- Trouver la bonne décomposition est difficile. Une mauvaise décomposition peut entraîner des couplages entre les micro-services
- Le côté distribué fait que le système complet est plus difficile à tester, déployer
- Le déploiement de fonctionnalités qui touche plusieurs services est plus délicat
- La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

Une architecture micro-service

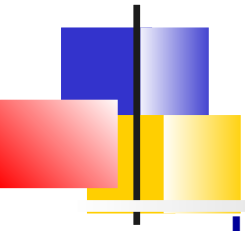


Microservices Pattern, Chris Richardson, 2017

Organisation DevOps



Les 12 facteurs de réussite



I. Outil de scm : Unique source de vérité

II. Dépendances: Déclarer explicitement et isoler les dépendances du code source

III. **Configuration** : Configuration externalisée (séparée) du code

IV. Services d'appui (backend) : Les services d'appui sont des ressources attachées, possibilité de switcher sans modification de code

V. Build, **release**, **run** : Distinguer clairement les phases de build, release et deploy. Permet la coexistence de différentes releases en production

VI. Processus : Exécute l'application comme un ou plusieurs processus stateless.

VII. Port binding : Application est autonome (pas de déploiement sur un serveur). Elle expose juste un port TCP

VIII. Concurrence : Montée en charge grâce au modèle de processus

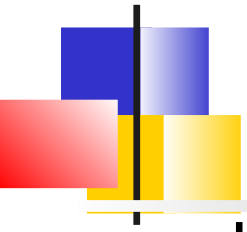
IX. Disposability : Renforce la robustesse avec des démarrages et arrêts rapides

X. Dev/prod parity : Garder les environnements de développement, de pré-production et de production aussi similaires que possible

XI. Logs : Traiter les traces comme un flux d'événements

XII. Processus d'Admin : Considérer les tâches d'administration comme un processus parmi d'autres

https://12factor.net/



Patterns micro-service

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

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

Problèmes à résoudre et design patterns

Patterns applicatifs

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

https://microservices.io/patterns/

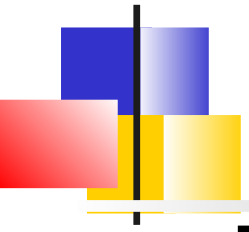
Problèmes à résoudre et design patterns

Patterns infrastructure applicative

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

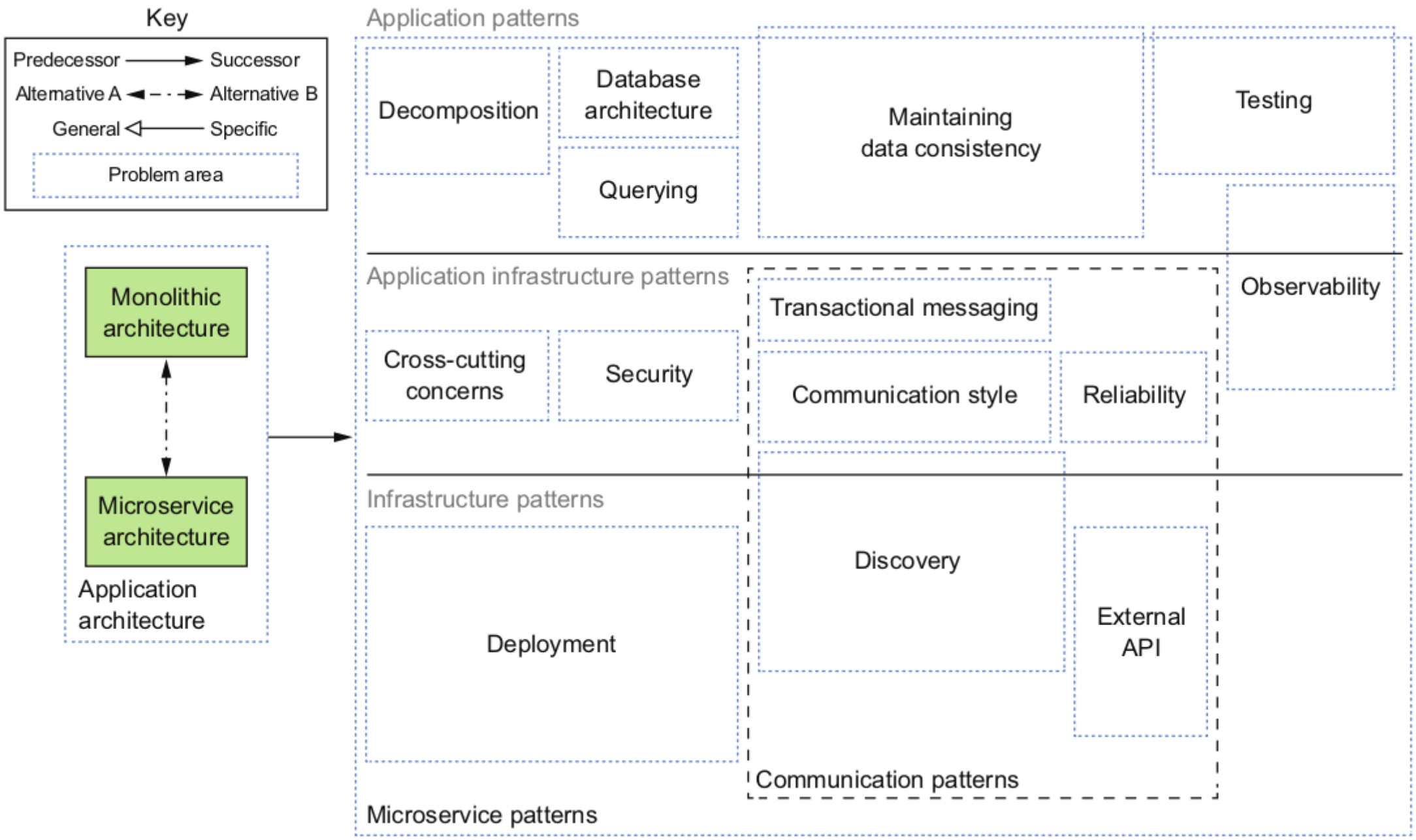
https://microservices.io/patterns/

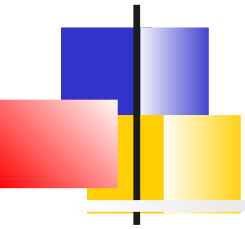
Patterns et problèmes à résoudre



Pattern d’infrastructure

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


Key
Predecessor
Successor
Alternative A
Alternative B
General
Specific
Problem area
Application patterns
Decomposition
Database architecture
Querying
Maintaining data consistency
Testing
Observability
Application infrastructure patterns
Transactional messaging
Cross-cutting concerns
Security
Communication style
Reliability
Infrastructure patterns
Discovery
Deployment
External API
Communication patterns
Microservice patterns
Monolithic architecture
Microservice architecture
Application architecture

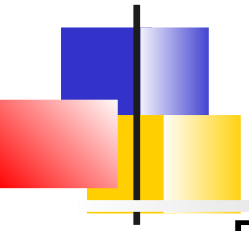


Introduction

Architectures micro-services

Services techniques

Eclipse MicroProfile



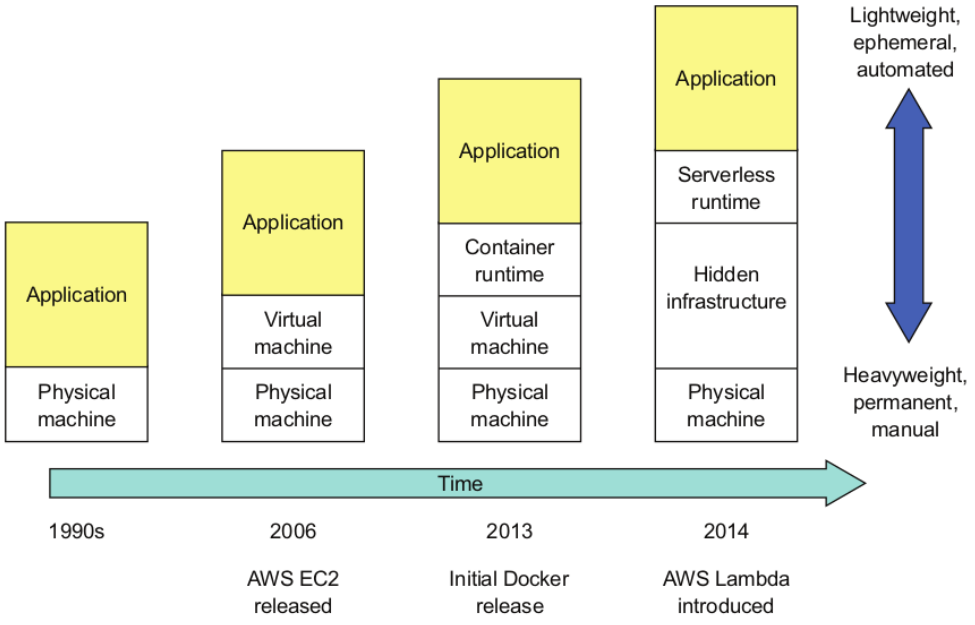
Services Transverses

De nombreux services transverses peuvent être fournis par un framework ou une infrastructure

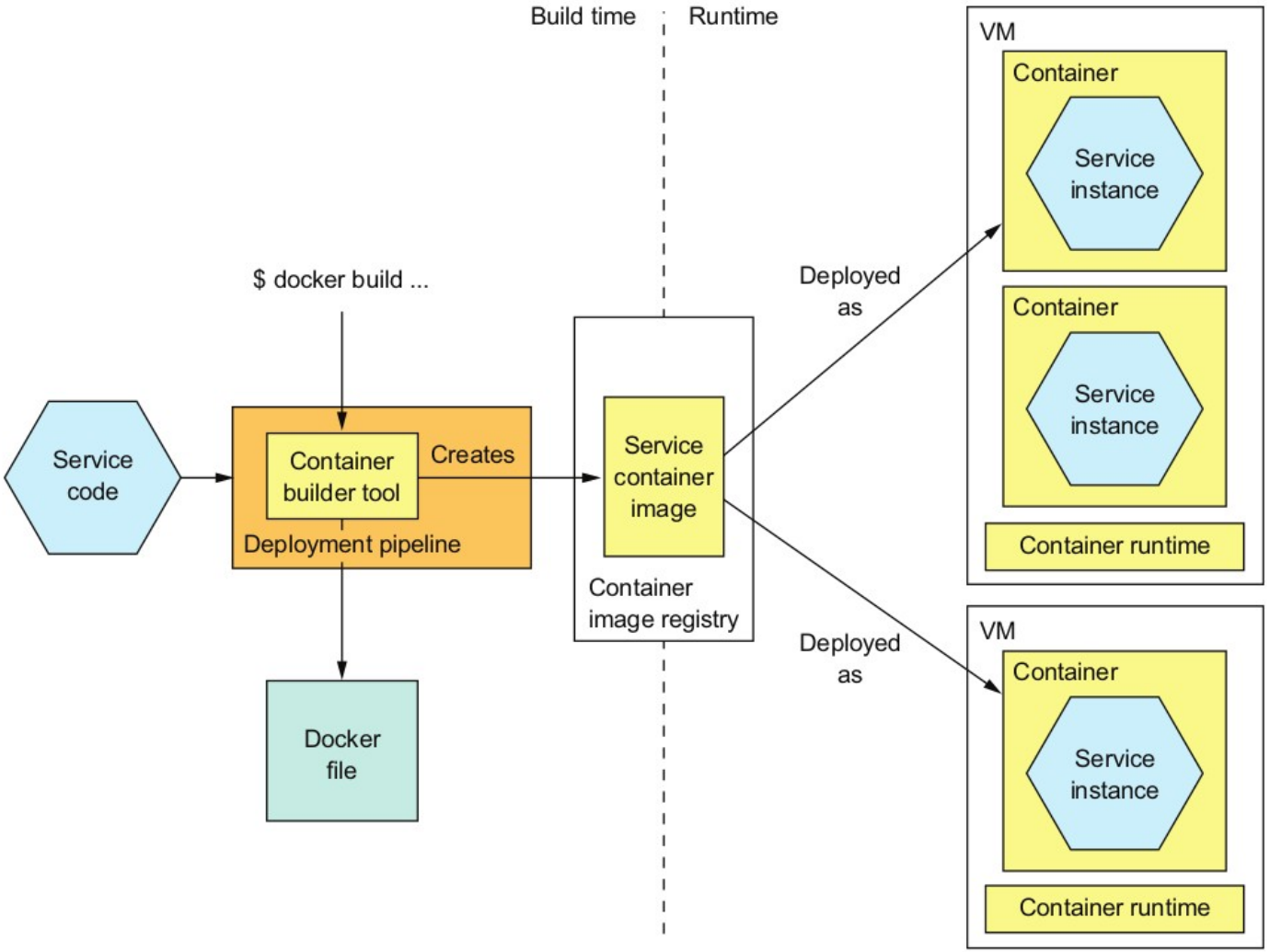
- Service **de discovery** permettant à un micro-service de s'enregistrer et de localiser ses micro-services dépendants
- Service de centralisation de **configuration** facilitant la configuration et l'administration des micro-services
- Service de **monitoring et d'observabilité** agrégeant les métriques de surveillance en un point central
- Services liés à la **sécurité** offrant des fonctionnalités de SSO et de gestion des autorisation (oAuth2)
- Support pour la répartition de charge, le fail-over, la résilience aux fautes, l'implémentation de pattern comme le gateway, SAGA, CQRS, ...

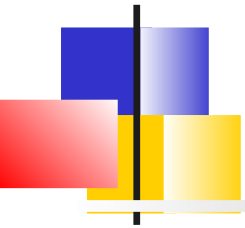
Infrastructure de déploiement

Même si plusieurs alternatives peuvent être envisagées, l'utilisation d'orchestrateur de container facilitent les déploiements.



Déploiement



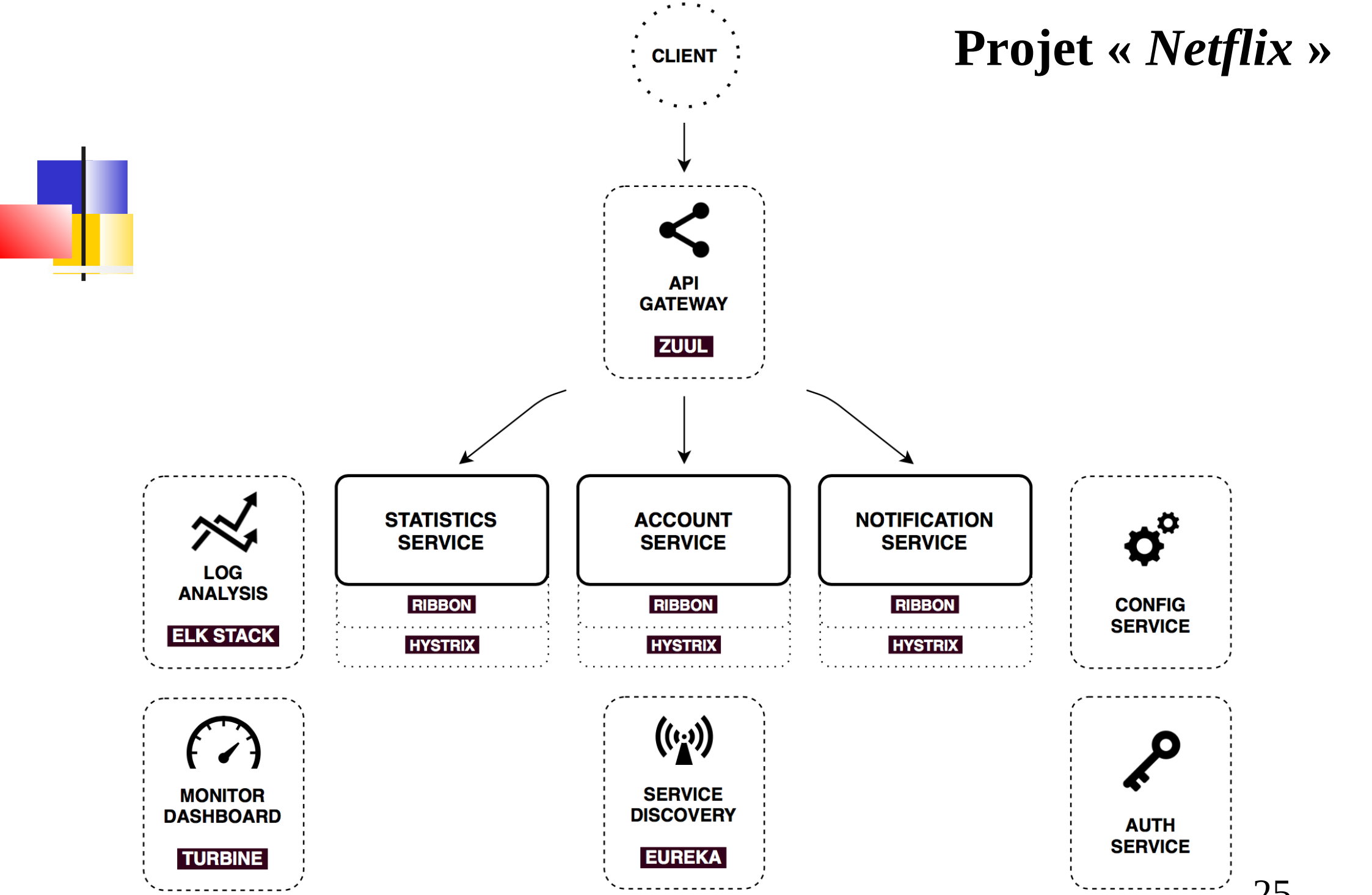


Services techniques vs Infra

Qui fournit les services techniques ?

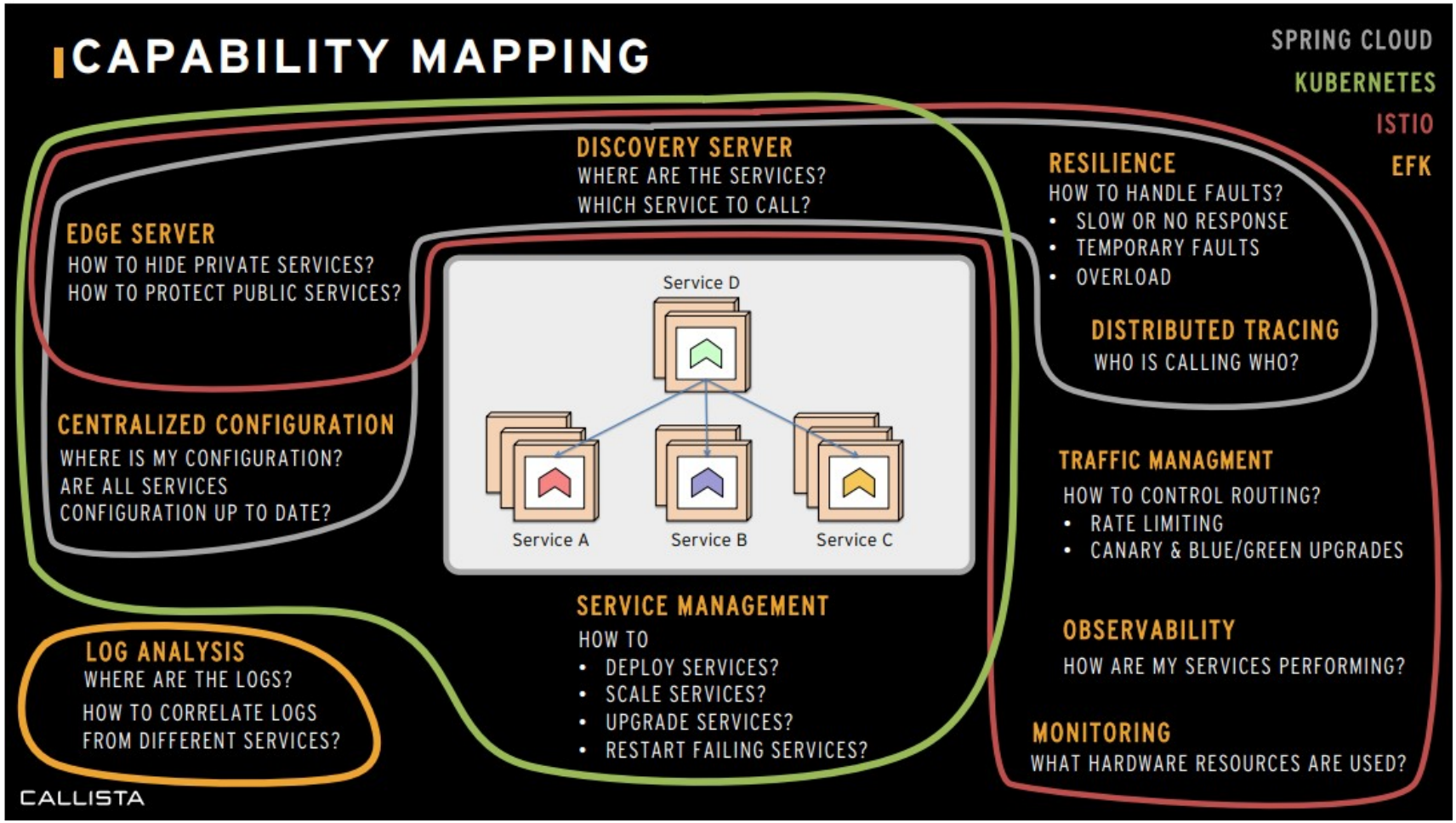
- Dans les premières architectures, c'est le software => Exemple framework Netflix intégré dans SpringCloud
- Actuellement, de nombreux services techniques migrent vers l'infrastructure :
  - Discovery, Config, Répartition de charge offert nativement par Kubernetes
  - Résilience, Sécurité, Monitoring : Add-on Kubernetes comme le service mesh Istio

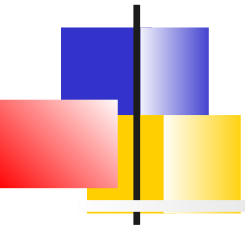
Projet « *Netflix* »



https://www.jfokus.se/jfokus20-preso/Getting-the-best-out-of-Spring-Cloud--Kubernetes-and-Istio.pdf

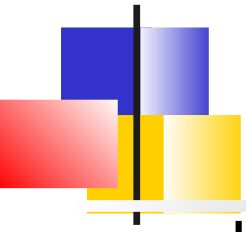
Capability Mapping
framework vs infra





Introduction

Architectures micro-services
Services techniques
Eclipse MicroProfile

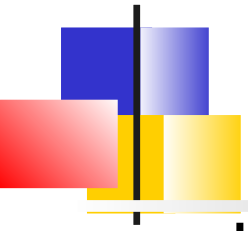


MicroProfile Rationale

Le projet MicroProfile® vise à optimiser Jakarta EE pour l'architecture des microservices.

L'objectif est d'itérer et d'innover en cycles courts pour proposer de nouvelles API et fonctionnalités communes, obtenir l'approbation de la communauté, publier et répéter.

Les APIs sont naturellement inspirées des expériences des différents frameworks Java (Jakarta EE ou non): SpringCloud, Quarkus, Micronaut



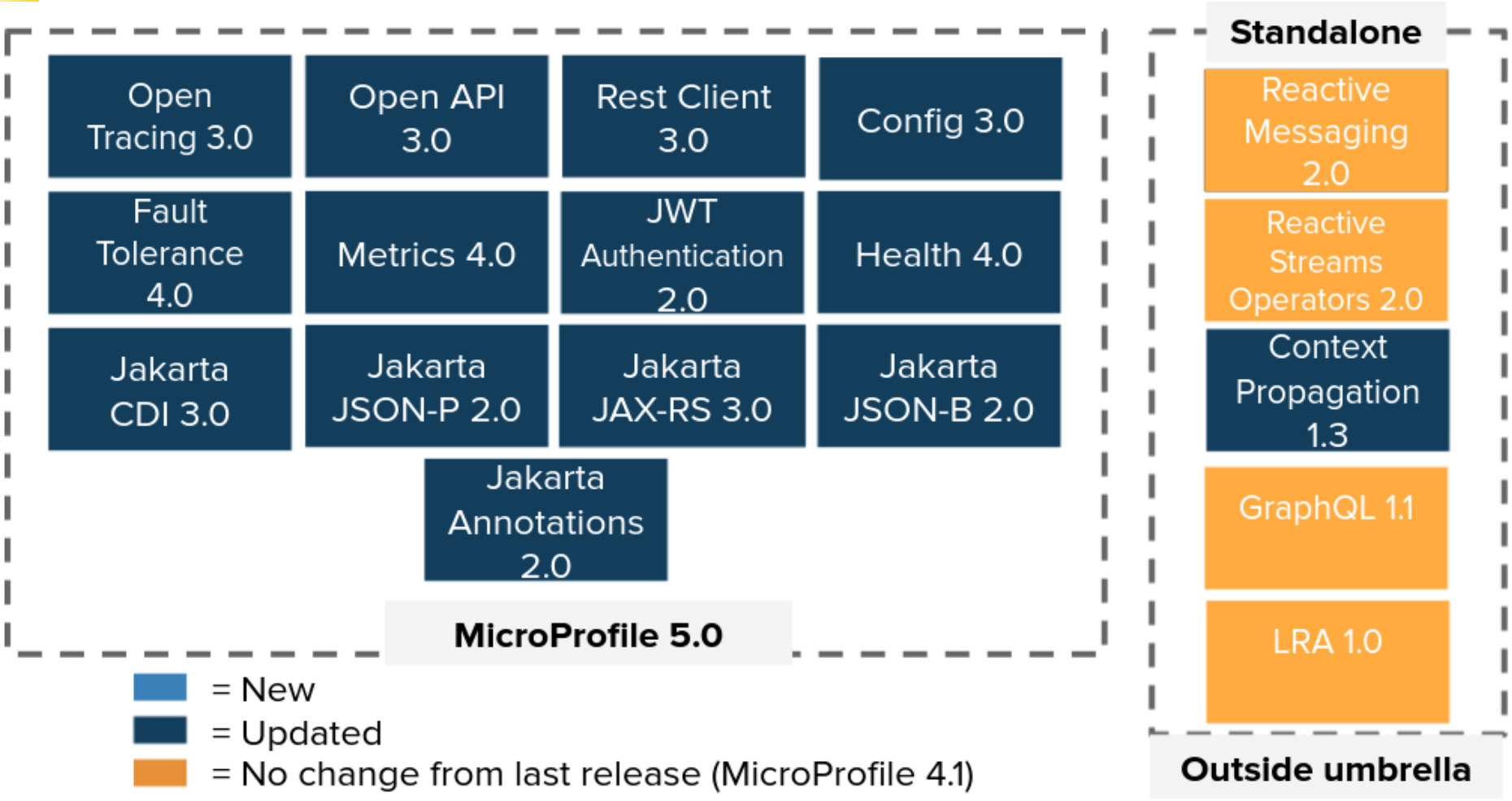
Historique

- Le 1er octobre 2020, MicroProfile est devenu membre du groupe de travail Eclipse
  - La charte définit la vision et la portée du groupe de travail, sa gouvernance, ses membres, etc.
  - Les spécifications des composants MicroProfile suivent un processus de spécification
- MicroProfile 4.1 est la première version livrée avec une implémentation compatible déclarée
- Microprofile 5.0 est sorti en décembre 2021

Membres du WorkingGroup

Implémentations

APIs


Open Tracing 3.0
Open API 3.0
Rest Client 3.0
Config 3.0
Fault Tolerance 4.0
Metrics 4.0
JWT Authentication 2.0
Health 4.0
Jakarta CDI 3.0
Jakarta JSON-P 2.0
Jakarta JAX-RS 3.0
Jakarta JSON-B 2.0
Jakarta Annotations 2.0
MicroProfile 5.0
Standalone
Reactive Messaging 2.0
Reactive Streams Operators 2.0
Context Propagation 1.3
GraphQL 1.1
LRA 1.0
Outside umbrella
= New
= Updated
= No change from last release (MicroProfile 4.1)

APIs Jakarta

Jakarta CDI 3.0 : Contexte Dependency Injection

Jakarta JAX-RS 3.0 : Interface de programmation Java permettant de créer des services Web avec une architecture REST

Jakarta JSON-P 2.0 : fournit des API portables pour analyser, générer, transformer et interroger des documents JSON.

Jakarta JSON-B 2.0 : Framework pour la conversion d'objets Java(R) vers et depuis des documents JSON

Jakarta Annotations 2.0 : Collection d'annotations qui permettent un style de programmation déclaratif .

MicroProfile (1)

MicroProfile Config 3.0 : Fournit un système facile à utiliser et flexible pour la configuration des applications

MicroProfile Fault Tolerance 4.0 : Définit des API faciles à utiliser et flexibles pour créer des applications résilientes

MicroProfile Health 4.0 : Exposer la disponibilité d'un environnement d'exécution MicroProfile à la plate-forme sous-jacente

MicroProfile JWT RBAC 2.0 : Utilisation des jetons Web JSON (JWT) basés sur OpenID Connect (OIDC) pour le contrôle d'accès basé sur les rôles (RBAC) des points de terminaison de microservice.

MicroProfile (2)

MicroProfile Metrics 4.0 : Définir des métriques d'application personnalisées et exposer des métriques de plate-forme sur un point de terminaison standard à l'aide de formats standard

MicroProfile OpenTracing 3.0 : Suivre le cheminement des requêtes entre les différents micro-services

MicroProfile OpenAPI 3.0 : Fournit des interfaces Java et des modèles de programmation pour produire nativement des documents OpenAPI v3 à partir d'applications JAX-RS

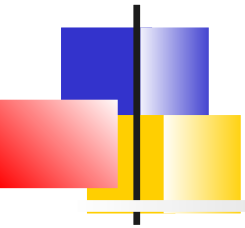
MicroProfile Rest Client 3.0 : Client REST typesafe défini en tant qu'interfaces Java

Reactive Programming

MicroProfile Reactive Streams Operators : Un ensemble d'opérateurs pour créer de nouveaux flux réactifs, traiter les données en transit et les consommer en toute simplicité

Microprofile Reactive Messaging : Définit un modèle de développement pour déclarer les beans CDI produisant, consommant et traitant des messages. S'appuie sur les Reactive Streams Operators et le CDI

MicroProfile Context Propagation : API pour propager des contextes à travers des unités de travail indépendantes des threads



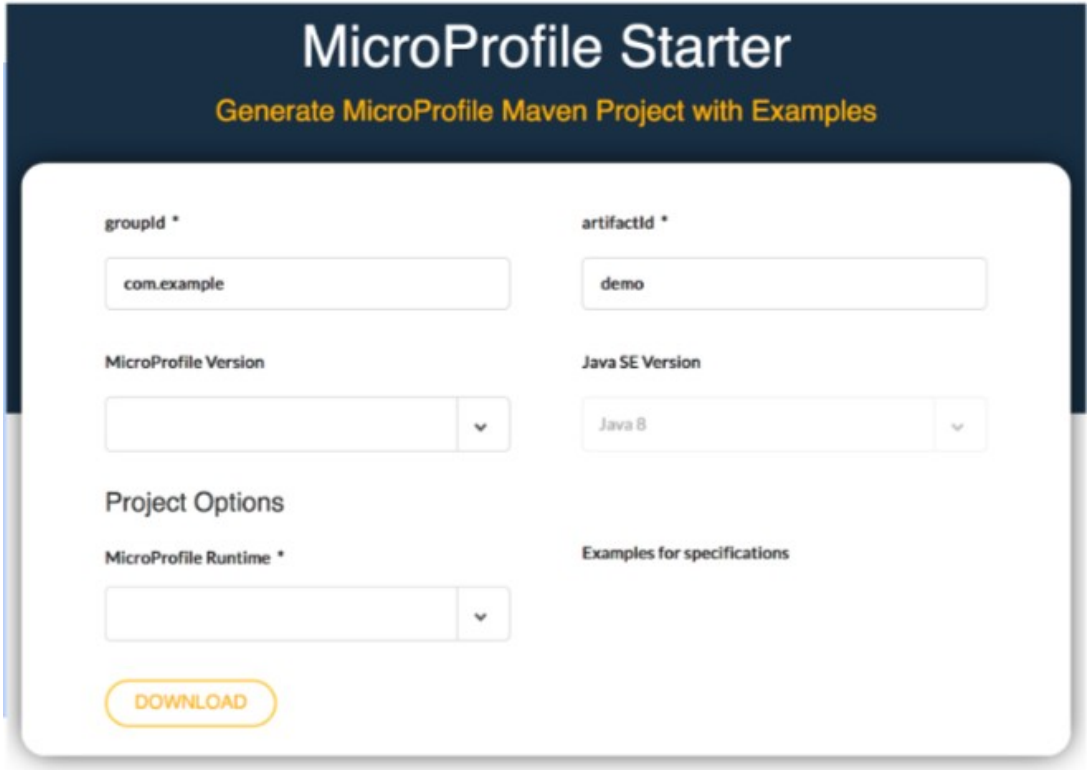
Autres

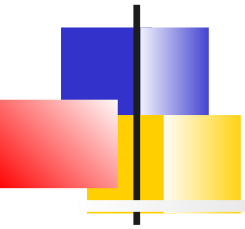
GraphQL 1.1 : Une alternative à REST permettant à un client HTTP de définir les données qu'il veut

LRA 1.0 (Long Running Actions) : Annotations et API permettant pour coordonner les activités de longue durée tout en maintenant un couplage lâche

start.microprofile.io

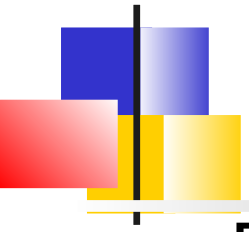
- Générer des projets MicroProfile
- Plug-in Visual Studio Code, Intellij
- Outils de ligne de commande





APIs JakartaEE

CDI et Jakarta Annotations
MicroProfile Config
JAX-RS
JSON-P, JSON-B
MicroProfile OpenAPI



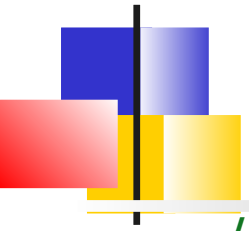
Introduction

Eclipse MicroProfile présuppose de bénéficier d'une API d'injection de dépendances

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

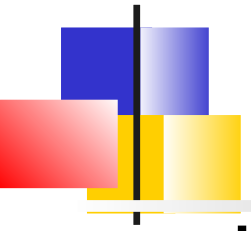
Des annotations :

- permettent de configurer le conteneur de beans (déclarer de beans et injection de dépendance)
- Permettent également de déclarer des aspects implémentant des services techniques



Un bean typique

```
// Annotation déclarant un bean et son cycle de vie
@ApplicationScoped
public class Translator {

    // Injection de dépendance d'un autre bean
    @Inject
    Dictionary dictionary;

    // Intercepteur appliquant un cross-cutting concern
    @Counted
    String translate(String sentence) {
      // ...
    }
}
```



Classe de Configuration

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

```
@Dependent
public class TracerConfiguration {

    @Produces
    public Tracer tracer(Reporter reporter) {
        return new Tracer(reporter);
    }

}
```

Injection de dépendances

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

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

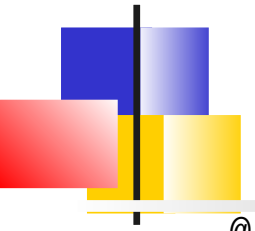
Ensuite, un bean est assignable à un point d'injection si

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

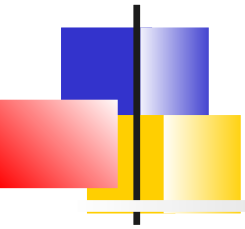
Exactement un seul bean doit être assignable à un point d'injection, sinon la construction échoue :

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

Injection par constructeur ou méthodes



```
@ApplicationScoped
public class Translator {

    private final TranslatorHelper helper;

    // Injection par le constructeur, helper is final
    Translator(TranslatorHelper helper) {
       this.helper = helper;
    }

   // Injection par méthode
    @Inject
    void setDeps(Dictionary dic, LocalizationService locService) {
      / ...
    }
}
```



Qualifier

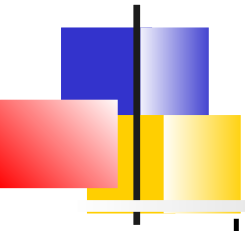
Les ***qualifiers*** sont des annotations qui aident le conteneur à distinguer les beans qui implémentent le même type.

Si aucun qualifier n'est précisé à un point d'injection, c'est le qualifier *@Default* qui est appliqué

Exemple

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

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

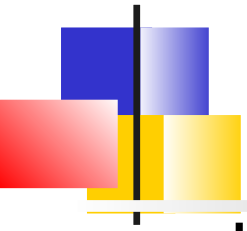
Ce bean serait assignable à *@Inject @Superior Translator* et *@Inject @Superior SuperiorTranslator* mais pas à *@Inject Translator*.



Scopes

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

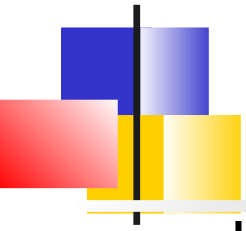
- ***@javax.enterprise.context.ApplicationScoped*** : Une seule instance de bean est utilisée et partagée entre tous les points d'injection. L'instance est créée en mode lazy lors de l'appel d'une méthode a proxy du bean
  La majorité des cas
- ***@javax.inject.Singleton*** : Une seule instance, créée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



Callback

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

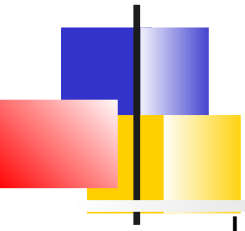
```
@ApplicationScoped
public class Translator {

    @PostConstruct
    void init() {
       // ...
    }

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



Intercepteurs

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

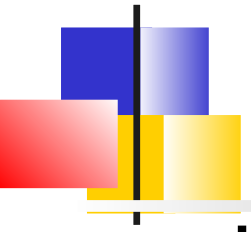
```
@Logged // Annotation associée à l'intercepteur
@Priority(2020)
@Interceptor
public class LoggingInterceptor {

   @Inject
   Logger logger;

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



Décorateurs

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

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

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

   @Inject
   LogService logService;

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

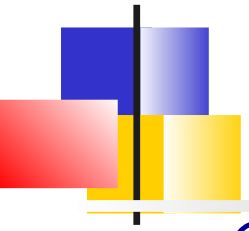


Modèle événementiel

- Les beans peuvent produire et consommer des événements pour interagir de manière complètement découplée.
- Tout objet Java peut être transmis par l’événement.
- Les qualificateurs facultatifs agissent comme des sélecteurs de sujet.

Exemple

```
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
   }
}
```



Jakarta Annotations

@Generated : Utilisée pour marqué du code généré.

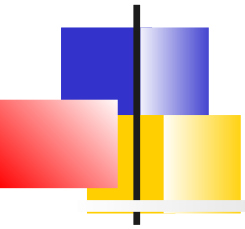
@Resource, ***@Resources*** : Utilisées pour déclarer une ou plusieurs références à des ressources provoquant une injection.

@PostConstruct, ***@PreDestroy*** : Méthodes de call-back

@Priority : Définit les priorités (exemple intercepteurs, décorateurs)

@RolesAllowed, ***@PermitAll***, ***@DenyAll***, ***@RunAs*** : : ACLs via rôles, Rôle d'exécution de l'application

@DataSourceDefinition, ***@DataSourceDefinitions*** : Utilisées pour définir une DataSource à enregistrer dans JNDI.



APIs JakartaEE

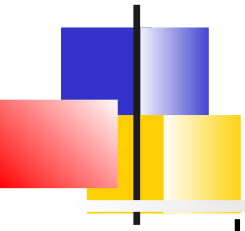
CDI et Jakarta Annotations

MicroProfile Config

JAX-RS

JSON-P, JSON-B

MicroProfile OpenAPI



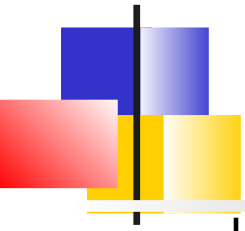
Rationale

La majorité des applications doivent être configurées en fonction d'un environnement d'exécution.
=> Il doit être possible de modifier les données de configuration depuis l'extérieur d'une application afin que l'application elle-même n'ait pas besoin d'être repackagée.

Les données de configuration peuvent provenir de différents emplacements et dans différents formats: les ***ConfigSources***.

Certaines sources de configuration peuvent changer de manière **dynamique**.
=> Les valeurs modifiées doivent être introduites dans le client sans qu'il soit nécessaire de redémarrer l'application.

Inspiration et implémentation



Inspirations

DeltaSpike Config
(http://deltaspike.apache.org/documentation/configuration.html) et
(https://github.com/struberg/javaConfig/)

Apache Tamaya (http://tamaya.incubator.apache.org/)

Implémentations

Apache Geronimo Config
(https://svn.apache.org/repos/asf/geronimo/components/config/trunk)

WebSphere Liberty (https://developer.ibm.com/wasdev/)

Payara Server et Payara Micro
(https://docs.payara.fish/documentation/microprofile/config.html)

WildFly & Thorntail (https://github.com/smallrye/smallrye-config)

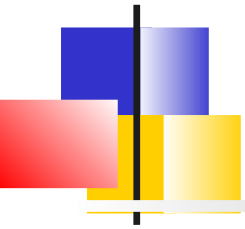
microBean™ MicroProfile Config

ConfigSources

Par défaut, il existe 3 *ConfigSources* :
Dans l'ordre de précédence :

- **Propriétés JVM :**
  System.getProperties()
- **Environnement OS** :
  System.getenv()
- **Fichier** .**properties** dans le classpath :
  META-INF/microprofile-config.properties

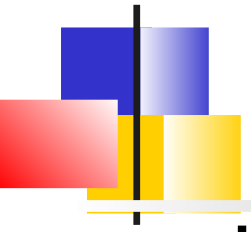
D'autres *ConfigSources* personnalisées peuvent s'enregistrer.
Ex : BD, fichier .yml, ...



Exemple Quarkus : Sources additionnelles

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

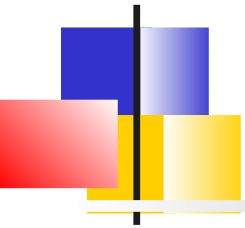
- YAML (quarkus-config-yaml)
- HashiCorp Vault (vault)
- Consul (config-consul)
- Spring Cloud (spring-cloud-config-client)



Expression pour les propriétés

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

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



Injection

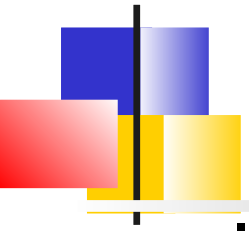
@ConfigProperty permet de s'injecter une clé de configuration :

```
// 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;
```

Extensions courantes à la spécification

Par exemple smallrye-config permet d'encapsuler dans une classe plusieurs propriétés de configuration et d'appliquer de la validation de surface via *javax.validation*

```
// Définition de 2 propriétés server.host et server.port
@ConfigMapping(prefix = "server")
interface Server {
    @URL
    String host();

    @Min(1025)
    int port();
}
```

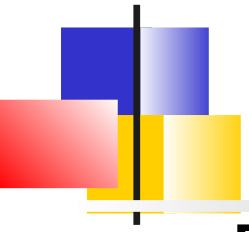


Les profils

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

Les profils de configuration permettent de spécifier dans le (ou les) fichiers .properties différentes configurations qui seront activées lors de l’activation d’un profil.

Profil dans le nom de la propriété

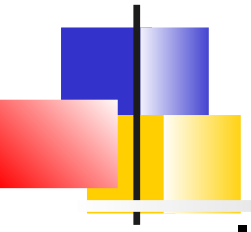


Pour pouvoir définir des propriétés avec le même nom, chaque propriété doit être précédée d'un **%**, le nom du profil et **.**

```
quarkus.http.port=9090
%dev.quarkus.http.port=8181
```

Les profils dans le fichier *.env* suivent la syntaxe suivante :

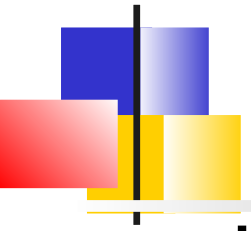
```
_{PROFILE}_CONFIG_KEY=value:
```



Nommage de fichier

- Il est souvent possible de regrouper toutes les configuration d'un profil dans un fichier séparé ***application-{profile}.properties***

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



Activation du profil

Le profil d'exécution est activé par la propriété ***mp.config.profile***

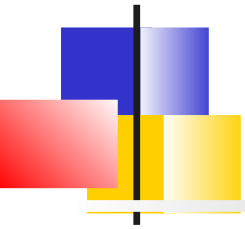
```
java -jar myapp.jar -Dmp.config.profile=testing
```

Beans en fonction du profil

Dans un contexte CDI, il est généralement possible de conditionner l'instanciation d'un bean à l'activation du profil.

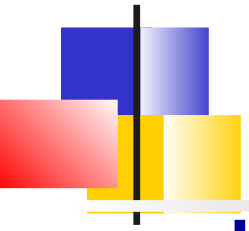
Par exemple, Quarkus propose les annotations suivantes :

- ***@io.quarkus.arc.profile.IfBuildProfile*** : Active le bean si le profil est activé
- ***@io.quarkus.arc.profile.UnlessBuildProfile*** : Désactive le bean si le profil est activé
- ***@io.quarkus.arc.properties.IfBuildProperty*** : Active si la propriété a une valeur spécifique
- ***@io.quarkus.arc.properties.UnlessBuildProperty*** : Désactive si la propriété a une valeur spécifique



APIs JakartaEE

CDI et Jakarta Annotations
MicroProfile Config
JAX-RS
JSON-P, JSON-B
MicroProfile OpenAPI



Introduction

JAX-RS : Java API for RESTful Web Services est une interface qui fait partie (depuis longtemps) de JakartaEE

Son objectif : Bâtir facilement des architectures RestFul

Implémentations :

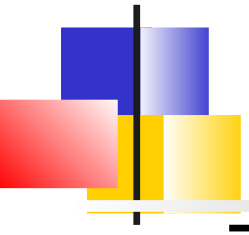
Apache CXF : https://cxf.apache.org/

Jersey (Implémentation de référence) : https://jersey.github.io/

RESTEasy (RedHat, Jboss) : https://resteasy.dev/

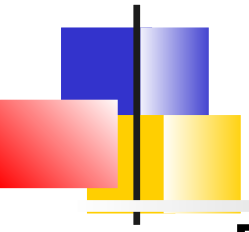
Apache TomEE : https://tomee.apache.org/

...



Déclaration des endpoints

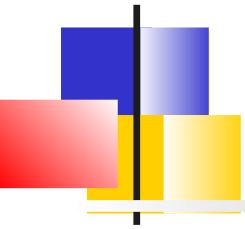
- Toute classe annotée avec ***@Path*** peut voir ses méthodes exposées en tant que points de terminaison REST
  - L'annotation de classe *@Path* définit le préfixe URI sous lequel les méthodes de la classe seront exposées.
    Il peut être vide ou contenir un préfixe
  - Chaque méthode peut à son tour avoir une autre annotation *@Path* qui s'ajoute au préfixe.



Exemple

Endpoint accessible à */rest/hello*

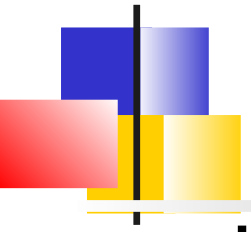
```
@Path("rest")
public class Endpoint {

    @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***

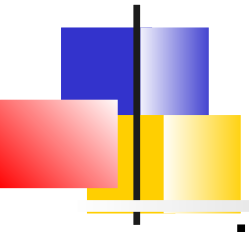
```
@ApplicationPath("/api")
public static class MyApplication extends
  Application {


}
```



Méthode HTTP

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

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



Media Type

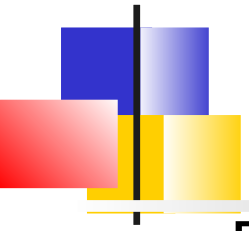
La classe peut être annotée par ***@Produces*** ou ***@Consumes.***

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*.

Exemple

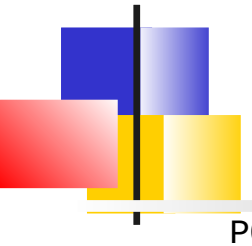
```
@Path("negotiated")
public class Endpoint {

    @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

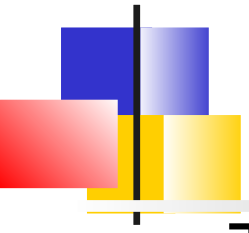
Différentes annotations sont utilisées pour récupérer des données de la requête :

- ***@PathParam*** : Une partie de l'URL
- ***@QueryParameter*** : Un paramètre HTTP
- ***@HeaderParam*** : Une entête
- ***@CookieParam*** : Un cookie
- ***@FormParam*** : Un champ de formulaire Web
- ***@MatrixParam*** : Un segment de chemin de l'URL
- ***@Provider*** : Extensions JAX-RS, permettant des convertisseurs, des gestionnaires d'exceptions ou des fournisseurs de contexte spécialisés.



Exemple

```
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(@PathParam("type") String type,
                        @MatrixParam("variant") String variant,
                        @QueryParameter("age") String age,
                        @CookieParam("level") String level,
                        @HeaderParam("X-Cheese-Secret-Handshake")
                        String secretHandshake,
                        @FormParam("smell") String smell) {
return type + "/" + variant + "/" + age + "/" + level + "/" + secretHandshake + "/" +
  smell;
}
```

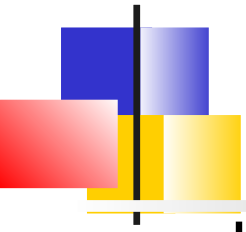


Corps de requête

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)

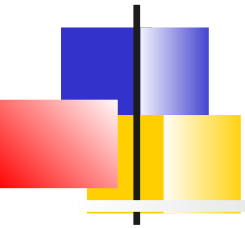
D'autre type sont supportés :

- *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 : ***Response***.

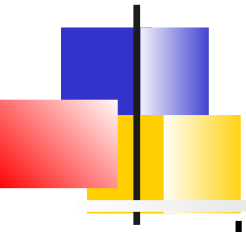
```
@GET
    public Response<String> hello() {
    // HTTP OK status avec text/plain
    return Response.ok("Hello, World!", MediaType.TEXT_PLAIN_TYPE)
    // entête
    .header("X-Cheese", "Camembert")
    // Entête Expires
    .expires(Date.from(Instant.now().plus(Duration.ofDays(2))))
    // Envoyer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



@Provider

@Provider : Permet d'étendre les capacités du runtime JAX-RS. La spécification définit 3 types de Providers :

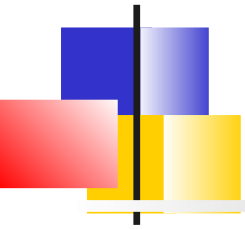
- ***Entité*** : Ils contrôlent le mapping entre Java et un MimeType (XML, JSON, CSV ..)
- ***Context*** : Ils contrôlent le contexte au quel les ressources peuvent accéder en utilisant l'annotation *@Context*
- ***Exception*** : Ils contrôlent le mapping entre les exceptions Java et une instance de *Response* JAX-RS.



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
- ***HttpServerRequest***, ***HttpServerResponse***, ***ServerRequestContext***, ***Sse*** : Objets bas niveau Vert.x
- ...



APIs JakartaEE

CDI et Jakarta Annotations
MicroProfile Config
JAX-RS
JSON-P, JSON-B
MicroProfile OpenAPI

Introduction JSON-P

JSON-P permet de parser et générer un document JSON

Deux API sont proposées :

- *Streaming API* : API de bas niveau qui permet la consommation et la production d'un document JSON en utilisant un flux d'événements, de manière similaire à l'API StAX
- *Object Model API* : API de plus haut niveau qui utilise des objets, de manière similaire à DOM. Cette API utilise l'API Streaming.

Principales classes

Json : Fabrique qui permet de créer des instances de certains types ou fabriques de l'API (Parser, Builder, Generator, Writer, Reader)

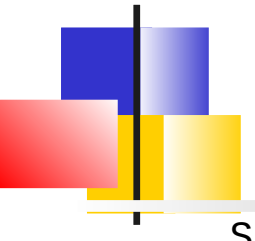
JsonReader : Créer un modèle objets à partir d'une représentation Json des données

JsonObjectBuilder, ***JsonArrayBuilder*** : Créer un modèle objets ou un tableau en lui ajoutant des éléments

JsonWriter : Envoyer dans un flux une représentation Json d'un modèle objet

JsonValue, ***JsonStructure***, ***JsonObject***, ***JsonArray***, ***JsonString***, ***JsonNumber*** : Encapsule les types de données d'un élément JSON.

JsonException : Exception pouvant être levée lors du traitement de la représentation JSON



Exemple

```
String document = "[{\n"
                  +
  "\"nom\":\"nom1\", \"prenom\": \"prenom1\", \"taille\":175\n"
                  + "},\n"
                  + "{\n"
                  +
  "\"nom\":\"nom2\", \"prenom\": \"prenom2\",\"taille\": 183\n"
                  + "}\n"
                  + "]";
 try (JsonReader reader = Json.createReader(new StringReader(document))) {
     JsonArray array = reader.readArray();
     JsonObject obj = array.getJsonObject(1);
     String nom = obj.getJsonString("nom").getString();
 }
```

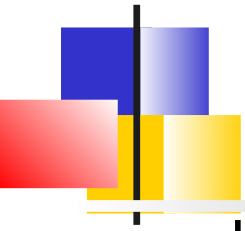
JSON-B

JSON-B propose une API standard pour réaliser le binding entre des documents JSON et des objets Java

- Équivalent à : Jackson, Genson, Gson, ...

L'API JSON-B permet de réaliser des opérations de sérialisation et de désérialisation entre des documents JSON et des objets Java

Les conversions de JSON-B se font avec un comportement par défaut mais il est possible de les personnaliser.



Comportement par défaut

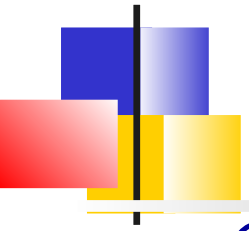
Le mapping par défaut ne requiert ni configuration ni annotation particulière.

Ce mapping par défaut comporte des règles pour la sérialisation/désérialisation pour les principaux types :

- Les types primitifs au travers de leur wrapper
- Certains types du JDK
- Les dates
- Les tableaux
- Les collections
- Les énumérations
- Des classes de JSON-P
- Les classes qui encapsulent des données des types précédents

Exemple

```
// Sérialisation
Personne pers = new Personne();
pers.nom = "nom";
pers.prenom = "prenom";
pers.taille = 175;
pers.adulte = true;
pers.dateNaissance = LocalDate.of(1985, Month.AUGUST, 11);

Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(pers);


// Désérialisation
Jsonb jsonb = JsonbBuilder.create();
 Personne pers =
   jsonb.fromJson("{\"nom\":\"nom1\",\"prenom\":\"prenom1\",\"taille\":18
   3}", Personne.class);
```



Annotations JSON-B

@JsonbProperty : Changer le nom par défaut d'un champ JSON

@JsonbPropertyOrder : Ordre de sérialisation des propriétés

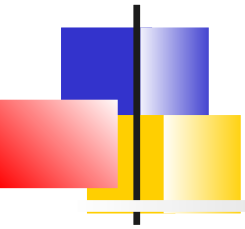
@JsonbTransient : Ignorer une propriété

@JsonbNillable : Sérialisation des propriétés nulles

@JsonbCreator : Si l'objet n'a pas de constructeur par défaut

@JsonbDateFormat et ***@JsonbNumberFormat*** : Contrôler le format des dates et des nombres

Classes ***JsonbAdapter*** et ***JsonbSerializer*** : Permet d'adapter la sérialisation sans annotations

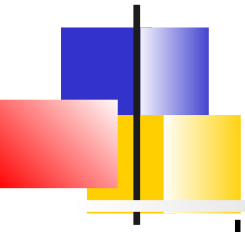


APIs JakartaEE

CDI et Jakarta Annotations
MicroProfile Config
JAX-RS
JSON-P, JSON-B
MicroProfile OpenAPI

Rationale

MicroProfile OpenAPI, vise à fournir un ensemble d'interfaces Java et de modèles de programmation permettant aux développeurs Java de produire nativement des documents OpenAPI v3 à partir de leurs applications JAX-RS.



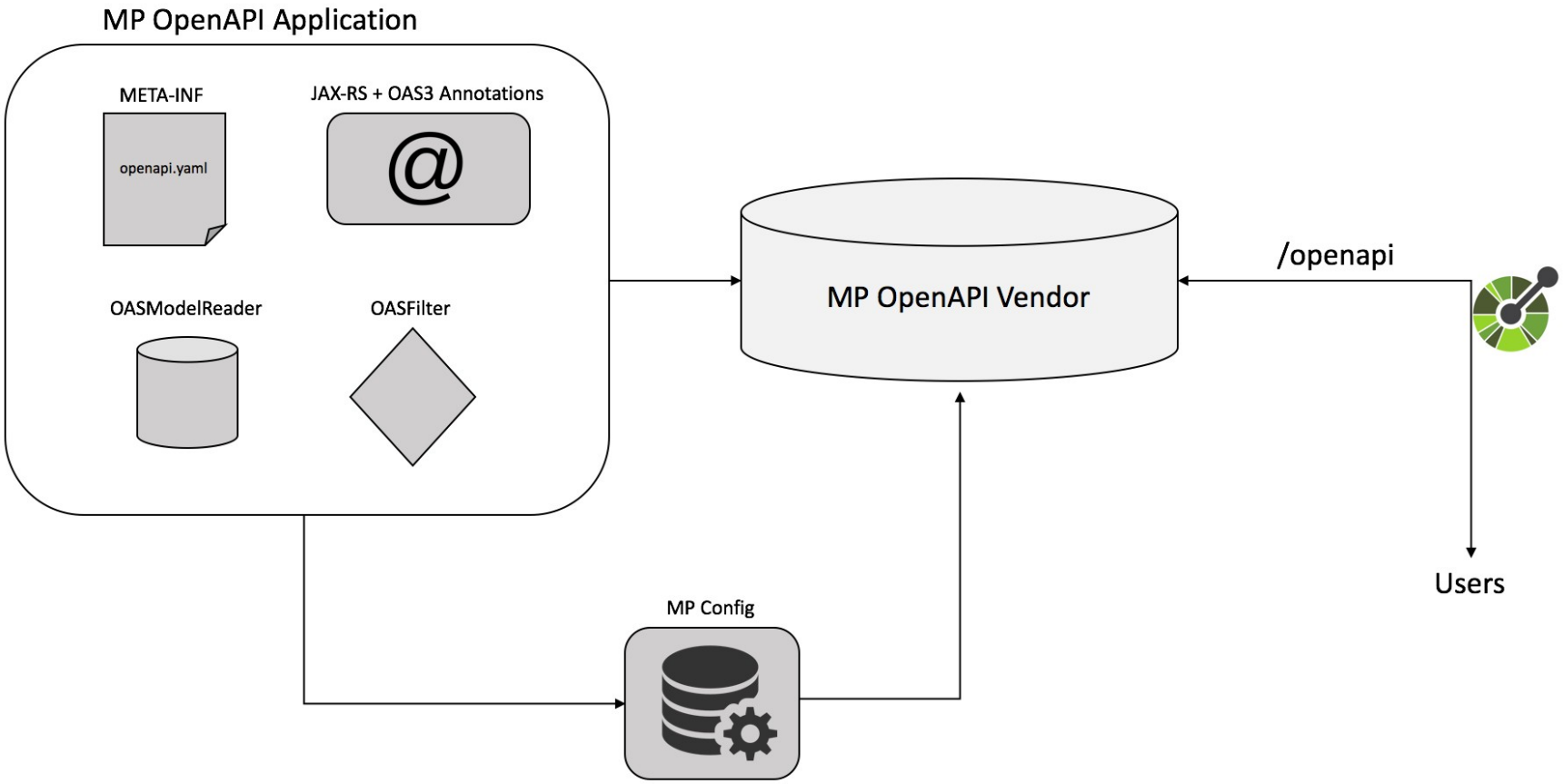
Principes

La spécification exige que les fournisseurs produisent un document OpenAPI valide à partir d'applications pures JAX-RS 2.0.
=> Aucun effort supplémentaire pour avoir une première version de la spécification OpenAPI

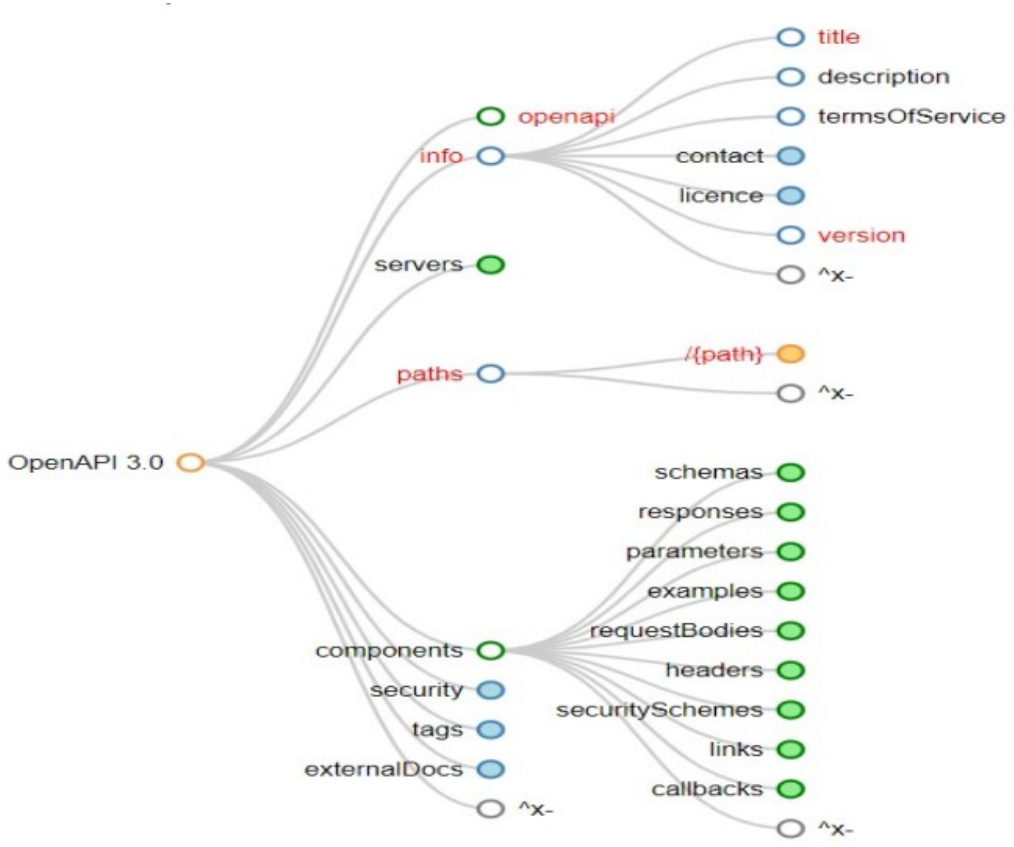
Le développeur peut alors :

- Compléter les annotations JAX-RS avec les annotations OpenAPI.
- Prendre la sortie initiale de */openapi* comme point de départ pour documenter l'API via des fichiers statiques.
- Utilisez le modèle de programmation pour fournir une arborescence de modèles OpenAPI.

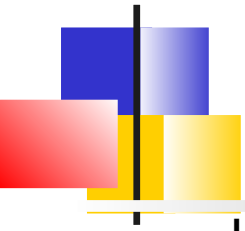
Architecture



OpenAPI Map



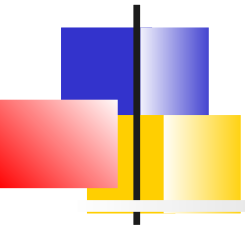
http://openapi-map.apihandyman.io



Format de description

L'*OpenAPI Specification* définit plusieurs objets :

- ***openapi*** : numéro de version de l'*OpenAPI Specification*
- ***info*** : Métadonnées sur l'API (description, licence, contact, etc.)
- ***servers***: Liste des environnements avec les informations de connexion (URL, etc.)
- ***paths*** : tous les chemins relatifs aux endpoints et opérations disponibles pour l'API
- ***components*** : Ensemble d'objets réutilisables pour différents aspects de la spécification (schemas, responses, parameters, headers, etc.)
- ***security*** : tableau des mécanismes de sécurité qui peuvent être utilisés au travers de l'API pour exécuter les opérations
- ***tags*** : tableau de tags utilisés pour ajouter des métadonnées
- ***externalDocs*** : référencement de ressource / documentation externe additionnelle
- ***^x-*** : balise / préfixe pour ajouter des propriétés personnalisées



Exemple

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: Sample API
  description: A sample API to illustrate OpenAPI concepts
paths:
  /list:
    get:
      description: Returns a list of stuff
      responses:
        '200':
          description: Successful response
```

Annotations MicroProfile

Beaucoup de ces annotations OpenAPI v3 ont été dérivées de la bibliothèque Swagger Core.

Quelques principales :

@OpenAPIDefinition : Méta-données générales

@Schema, ***@SchemaProperty*** : permet de définir les types de données d'entrée et de sortie.

@Operation : Typiquement une méthode HTTP et un chemin spécifiqueth.

@Parameter, ***@Parameters*** : Paramètres d'une opération

@RequestBody, ***@RequestBodySchema*** : Décrit le corps d'une requête

@APIResponses, ***@APIResponse***, ***@APIResponseSchema*** : Décrit les réponses d'une opération

@ExampleObject : Illustre un exemple

Exemple

```
@GET
@Path("/{username}")
@Operation(summary = "Get user by user name")
@APIResponse(description = "The user",
             content = @Content(mediaType = "application/json",
                                schema = @Schema(implementation = User.class))),
@APIResponse(responseCode = "400", description = "User not found")
public Response getUserByName(
        @Parameter(description = "The name that needs to be fetched. Use user1 for
   testing. ", required = true) @PathParam("username") String username)
 {...}
```

Propriétés de configuration

mp.openapi.scan.disable : Le scan est désactivé. (Utilisation d'un fichier statique ou environnement de production)

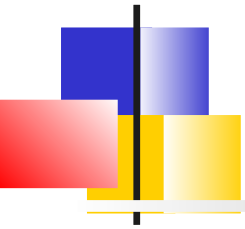
mp.openapi.scan.packages, ***mp.openapi.scan.classes***, ***mp.openapi.scan.exclude.packages***, ***mp.openapi.scan.exclude.classes*** : Limité le scan

mp.openapi.servers : Propriété de configuration pour spécifier la liste des serveurs globaux

mp.openapi.servers.path. : Préfixe de la propriété de configuration pour spécifier une autre liste de serveurs pour traiter toutes les opérations d'un chemin particulier

mp.openapi.schema. : Préfixe de la propriété de configuration pour spécifier un schéma pour une classe spécifique, au format JSON.

```
mp.openapi.schema.java.util.Date = { \
  "name": "EpochMillis" \
  "type": "number", \
  "format": "int64", \
  "description": "Milliseconds since January 1, 1970, 00:00:00 GMT" \
}
```



APIs pour la distribution

RestClient
Fault-tolerance
Reactive Messaging

Introduction

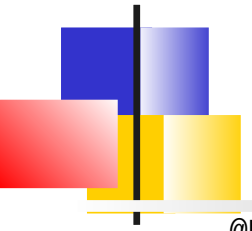
MicroProfile Rest Client fournit une approche **type-safe** pour appeler des services RESTful .

- Autant que possible, la spécification utilise les annotations de JAX-RS pour des soucis de cohérence et de réutilisation.

Le développeur définit des interfaces fixant les types d'entrées et de retour et les annotent avec JAX-RS.

L'implémentation RestClient génère alors les beans effectuant les appels REST et les dé/sérialisation d'objets

MP RestClient est liée aux spécifications JAX-RS, CDI, MP Config et MP Fault-tolerance

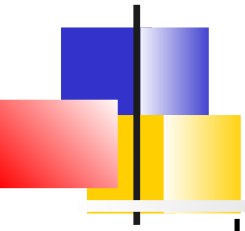


Exemple

```
@Path("/movies")
 public interface MovieReviewService {
     @GET
     Set<Movie> getAllMovies();

     @GET
     @Path("/{movieId}/reviews")
     Set<Review> getAllReviews( @PathParam("movieId") String movieId );

     @GET
     @Path("/{movieId}/reviews/{reviewId}")
     Review getReview( @PathParam("movieId") String movieId, @PathParam("reviewId") String
  reviewId );

     @POST
     @Path("/{movieId}/reviews")
     String submitReview( @PathParam("movieId") String movieId, Review review );

     @PUT
     @Path("/{movieId}/reviews/{reviewId}")
     Review updateReview( @PathParam("movieId") String movieId, @PathParam("reviewId") String
  reviewId, Review review );
 }
```

Utilisation

```
// Sans CDI, l'API fournit des méthodes builder
// pour instancier les beans
URI apiUri = new URI("http://localhost:9080/movieReviewService");
MovieReviewService reviewSvc = RestClientBuilder.newBuilder()
            .baseUri(apiUri)
            .build(MovieReviewService.class);

Review review = new Review(3, "This was a delightful comedy, but not
   terribly realistic.");
reviewSvc.submitReview( movieId, review );
```



Implémentations

L'implémentation de *RestClient* fait en général partie des implémentations de JAX-RS

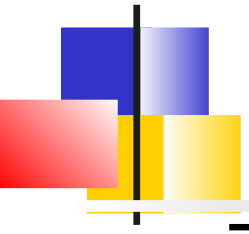
Apache CXF
(http://cxf.apache.org/download.html)

Open Liberty
(https://openliberty.io/blog/2018/01/31/mpRestClient.html)

Thorntail
(https://github.com/thorntail/thorntail/tree/master/fractions/microprofile/microprofile-restclient)

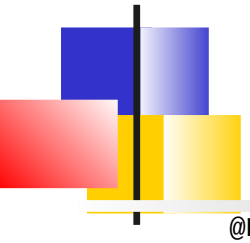
RESTEasy
(https://resteasy.github.io)

Jersey
(https://github.com/eclipse-ee4j/jersey)



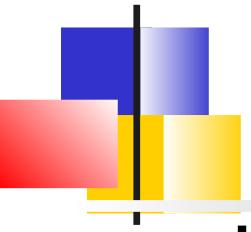
Annotations

- Toutes les annotations JAX-RS sont supportées :
  - *@Path, @GET, @POST, @PUT, ...*
  - *@PathParam, @QueryParameter, @HeaderParam, @CookieParam, ...*
  - *@Produces, @Consumes*
- ***@ClientHeaderParam*** permet de spécifier une entête sans influencer la signature de la méthode



Exemple *@ClientHeaderParam*

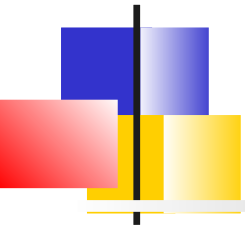
```
@Path("/somePath")
public interface MyClient {

    @POST
    @ClientHeaderParam(name="X-Http-Method-Override", value="PUT")
    Response sentPUTviaPOST(MyEntity entity);

    @POST
    @ClientHeaderParam(name="X-Request-ID", value="{generateRequestId}")
    Response postWithRequestId(MyEntity entity);

    @GET

 @ClientHeaderParam(name="CustomHeader",value="{pkg.MyHeaderGenerator.generateCustomHeader}",required=false)
    Response getWithoutCustomHeader();

    default String generateRequestId() { return UUID.randomUUID().toString(); }
}

public class MyHeaderGenerator {
    public static String generateCustomHeader(String headerName) {
        if ("CustomHeader".equals(headerName)) { throw UnsupportedOperationException(); }
        return "SomeValue";
    }
}
```



Usage

Les implémentations de *RestClient* doivent supporter :

- Les approches CDI.
  L'interface est alors annotée avec ***@RegisterRestClient*** et un attribut qui indique la **base Uri** afin de provoquer l'enregistrement du bean
- La construction explicite par l'API *RestClientBuilder*



Exemple CDI

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


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

Configuration (Quarkus)

- La définition de la base URL s’effectue dans la configuration

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

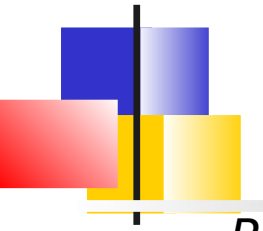
- Ou

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

Usage du Restclient, CDI

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

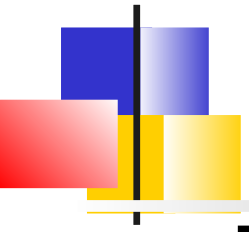
```
@Path("/extension")
public class ExtensionsResource {

    @RestClient
    ExtensionsService extensionsService;

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



Providers

RestClientBuilder étend l'interface *Configurable* de JAX-RS, permettant à un utilisateur d'enregistrer des *Providers* personnalisés lors de la construction.

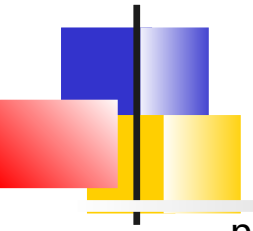
- ***ClientReponseFilterClientReponseFilter*** : Filtres appelés lorsqu'une réponse est reçue.
- ***ClientRequestFilterClientRequestFilter*** : Filtres appelés lorsqu'une requête est adressée.
- ***MessageBodyReader*** : Extraction de l'entité à partir de la réponse
- ***MessageBodyWriter*** : Conversion de l'entité dans le corps de la requête POST ou PUT
- ***ParamConverter*** : Conversion d'un paramètre de requête ou de réponse
- ***ReaderInterceptor*** : Listener des lectures
- ***WriterInterceptor*** : Listener des écritures
- ***ResponseExceptionMapper*** : Spécifique à MicroProfile Rest Client. Convertit une réponse en un Throwable.



@RegisterProvider

Dans un contexte CDI, les interfaces peuvent utiliser l'annotation ***@RegisterProvider***

```
@RegisterRestClient
@RegisterProvider(MyFilter.class)
public interface MyRestClient1 { /* ... */ }
```

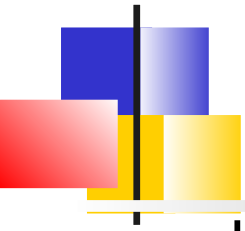


Exemple *ResponseExceptionMapper*

```
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;
    }
}
```

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

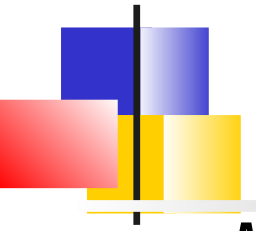
Si l'on veut la rendre disponible pour un client spécifique, il faut annoter l'interface du REST Client avec :
@RegisterProvider(MyResponseExceptionMapper.class)



@ClientExceptionMapper

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

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

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



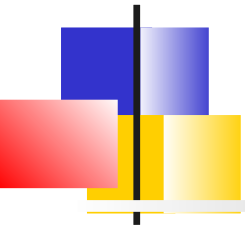
Divers

MP RestClient permet de configurer SSL :

- Utilisation trustore JVM ou configurer un spécifique.
- Vérificateur de hostname
- KeyStore (pour l'identification Client)

MP Rest Client a du support pour SSE (Server Side Events)

```
@GET
@Path("ssePath")
@Produces(MediaType.SERVER_SENT_EVENTS)
Publisher<InboundSseEvent> getEvents();
```

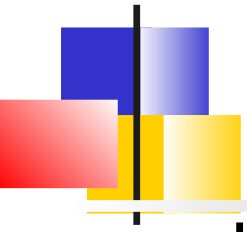


APIs pour la distribution

RestClient

Fault-tolerance

Reactive Messaging



Introduction

Les stratégies de tolérances aux pannes sont indispensables dans les architectures fortement distribuées telles que les micro-services.

Les défaillances des services peuvent être de toute sorte :

- Défaillance réseau
- Surcharge
- Crash
- Redémarrage
- Montée de version avec interruption de service

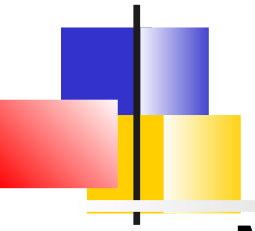
Patterns

Différents patterns permettent d'appliquer des stratégies de tolérance aux pannes :

- ***Timeout*** : Définir une durée maximale d'exécution
- ***Retry*** : Autoriser plusieurs tentatives d'exécution
- ***Bulkhead***: Limiter les exécutions concurrentes afin que les défaillances ne puissent pas surcharger l'ensemble du système
- ***CircuitBreaker*** : Disjoncter un service défaillant
- ***Fallback*** : Fournir une solution alternative en cas d'échec de l'exécution

MP Fault Tolerance fournit une annotation pour chacune des stratégie. L'anotation peut être placée sur les méthodes des beans CDI ou des interfaces RestClient.

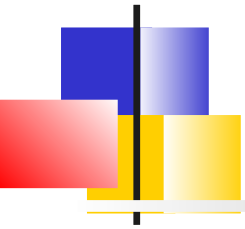
L'appel de la méthode est alors intercepté et la ou les stratégies de tolérance aux pannes correspondantes sont appliquées.



Autres spéc.

MP Fault Tolerance est liée à :

- CDI
- Aux intercepteurs
- *MP Config* : Configuration des intercepteurs. Ex *maxRetry*
- MP Metrics : Surveillance des relations entre micro-services



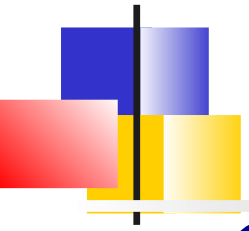
@Asynchronous

@Asynchronous signifie que l'exécution de la méthode se fera sur un thread séparé qui doit avoir le contexte (sécurité par exemple).

Le retour de la méthode est alors *Future*

```
@Asynchronous
public Future<Connection> serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return CompletableFuture.completedFuture(conn);
}
```

L'annotation peut être utilisée avec *@Timeout*, *@Fallback*, *@Bulkhead* et *@Retry*



@Timeout

@Timeout empêche d'attendre indéfiniment.
Il est recommandé qu'un appel de micro-service soit associé à un délai d'expiration.

```
@Timeout(400) // timeout is 400ms
public Connection serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return conn;
}
```

Lorsque le timeout survient, une *TimeoutException* est lancée.

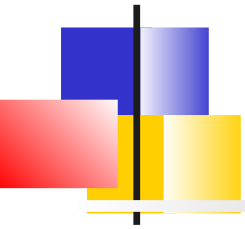
L'annotation peut être utilisée avec *@Fallback*, *@CircuitBreaker*, *@Asynchronous*, *@Bulkhead* et *@Retry*.

@Retry

Afin de récupérer d'un problème transitoire de réseau, ***@Retry*** peut être utilisée pour invoquer à nouveau la même opération.

L'annotation permet de configurer :

- ***maxRetries*** : le nombre maximal de tentatives
- ***delay*** et ***delayUnit*** : délais entre chaque nouvelle tentative
- ***maxDuration*** et ***durationUnit*** : durée maximale pendant laquelle effectuer la nouvelle tentative.
- ***jitter*** et ***jitterDelayUnit*** : la plage aléatoire des délais des tentatives
- ***retryOn*** : spécifiez les échecs à réessayer
- ***abortOn*** : spécifiez les échecs à abandonner

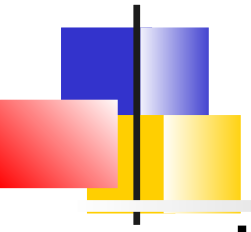


Usage

@Retry se positionne sur une méthode ou sur une classe.

L'annotation peut être utilisée avec *@Fallback, @CircuitBreaker, @Asynchronous, @Bulkhead* et *@Timeout.*

- Un *@Fallback* peut être spécifié et il sera invoqué après l’essai de toutes les tentatives.
- Si *@Retry* est utilisé avec *@Timeout*, une nouvelle tentative ne sera déclenchée que si *TimeoutException* ou l'une de ses super classes sont définies dans l'attribut *retryOn*.



Fallback

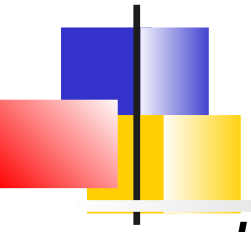
Les fallbacks sont invoqués que les stratégies *@Retry* ou *@CircuitBreaker* ont échoués

L'annotation ***@Fallback*** sur une méthode peut spécifier :

- Une classe *FallbackHandler* class
- La méthode de fallback

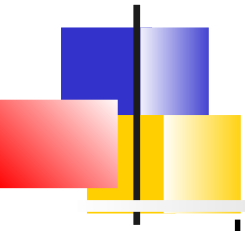
Exemples

```
// En cas d'erreur StringFallbackHandler.handle(ExecutionContext context) est exécutée
@Retry(maxRetries = 1)
   @Fallback(StringFallbackHandler.class)
   public String serviceA() {
       counterForInvokingServiceA++;
       return nameService();
   }


----
@Retry(maxRetries = 2)
   @Fallback(fallbackMethod= "fallbackForServiceB")
   public String serviceB() {
       counterForInvokingServiceB++;
      return nameService();
   }


   private String fallbackForServiceB() {
       return "myFallback";
}
```



Circuit Breaker Pattern

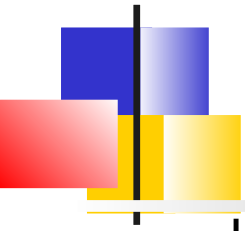
Lorsqu'un service en appelle un autre de manière synchrone, il est toujours possible que l'autre service ne soit pas disponible ou présente une latence si élevée qu'il est essentiellement inutilisable. Des ressources précieuses telles que des threads peuvent être consommées dans l'appelant en attendant que l'autre service réponde. Cela pourrait conduire à l'épuisement des ressources, ce qui rendrait le service appelant incapable de traiter d'autres demandes. La défaillance d'un service peut potentiellement se répercuter sur d'autres services dans l'application.[1]



Etats du circuit

Le circuit peut prendre alors 3 états :

- **Fermé** : Conditions normales.
  Si une panne se produit, le disjoncteur enregistre l'événement. Les paramètres ***requestVolumeThreshold*** et ***failureRatio*** déterminent les conditions d'ouverture du circuit.
- **Ouvert** : Lorsque le circuit est ouvert, les appels au service fonctionnant sous le disjoncteur échouent immédiatement. Un paramètre ***delay*** détermine le moment où le circuit passe à l'état semi-ouvert.
- **Semi-ouvert** : En état semi-ouvert, les tentatives d'exécutions sont autorisées. Par défaut, une tentative est autorisés.
  - Si elle échoue, le circuit reviendra à l'état ouvert.
  - Le paramètre ***successThreshold*** permet de configurer le nombre de tentatives réussies avant que le circuit puisse être fermé.



Usage

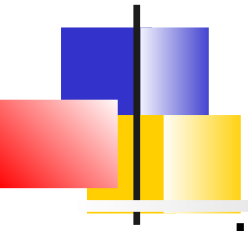
Une méthode ou une classe peut être annotée avec ***@CircuitBreaker***.

```
@CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 4,
                failureRatio=0.75, delay = 1000)
public Connection serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return conn;
}
```

Lorsque le circuit est ouvert, une *CircuitBreakerOpenException* est lancée.

L'annotation peut être utilisée avec *@Timeout, @Fallback, @Asynchronous, @Bulkhead* et *@Retry*.

- Un *@Fallback* sera invoqué si *CircuitBreakerOpenException* est lancée.
- *@Timeout* permet de faire échouer une opération si elle prend trop de temps.



BulkHead Pattern

Le pattern *BulkHead* consiste à empêcher les défauts d'une partie du système de se répercuter sur l'ensemble du système.

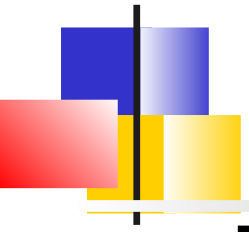
L'implémentation consiste à limiter le nombre de requêtes simultanées accédant à une instance.

2 approches pour l’implémentation :

- Sémaphores
- Pool de threads pour les méthodes asynchrones

Exemples

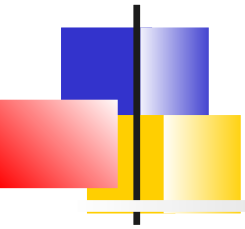
```
@Bulkhead(5) // maximum 5 concurrent requests allowed
public Connection serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return conn;
}
----
// maximum 5 concurrent requests, maximum 8 requests in the waiting queue
@Asynchronous
@Bulkhead(value = 5, waitingTaskQueue = 8)
public Future<Connection> serviceA() {
   Connection conn = null;
   counterForInvokingServiceA++;
   conn = connectionService();
   return CompletableFuture.completedFuture(conn);
}
```



Intégration avec MP Metrics

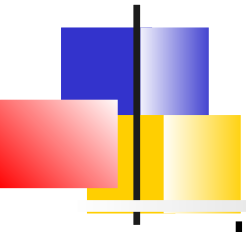
De nombreux métriques associés à la tolérance aux fautes sont définies :

- Nombre de tentatives, réussies, échouées
- Nombre de timeout
- Total par états des circuits
- Nombre d’exécution concurrentes
- ....



APIs pour la distribution

RestClient
Fault-tolerance
Reactive Messaging



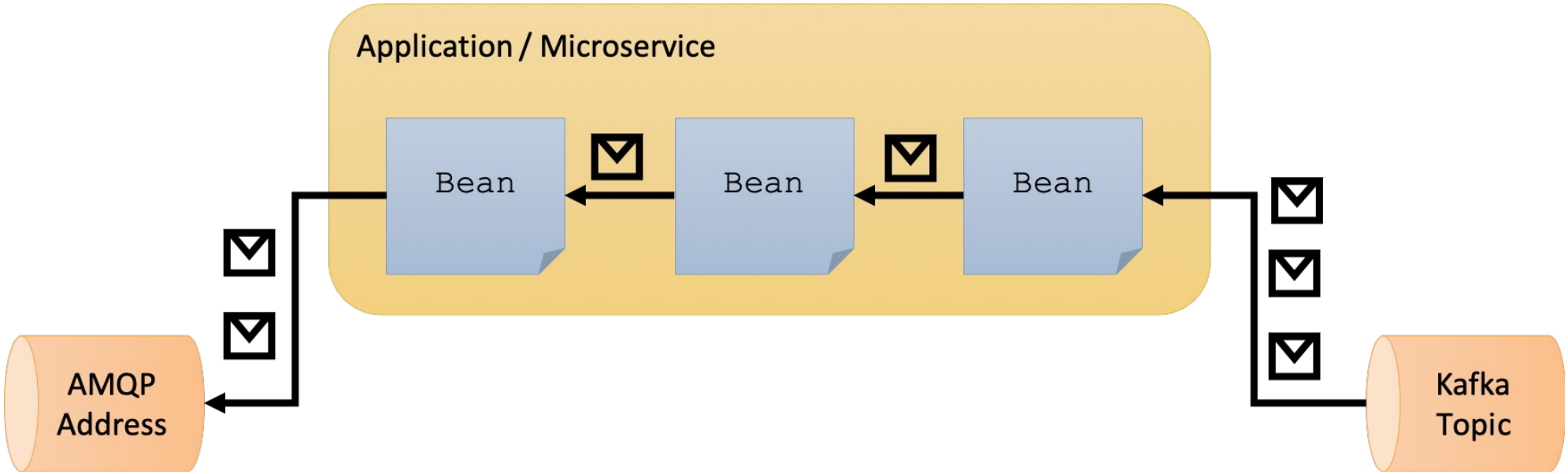
Rationale

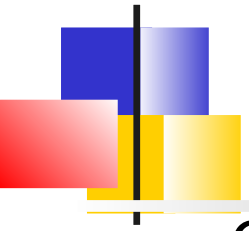
La communication asynchrone permet le découplage : temporel et des emplacements.

Les micro-services interagissent alors à l'aide de messages. Le découplage permet :

- Une meilleure gestion des pannes. Garantie de livraison des messages offer par les messages broker
- Une meilleure scalabilité, le système peut décider d'augmenter ou de réduire certains des microservices.
- Une meilleure évolutivité. La possibilité d'introduire de nouvelles fonctionnalités plus facilement.

Architecture



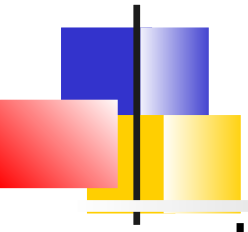


MP Reactive Messaging

Objectif : Fournir un support pour le messaging asynchrone basé sur Reactive Streams et indépendamment du message Broker.

Concepts :

- Les applications échangent des ***messages*** : Payload et méta-données.
  Les messages sont acquittés après traitement
- Les messages transitent dans des ***canaux*** (channels).
  Les applications se connectent aux canaux.
- Certains canaux sont internes à l'application d'autres sont associés à des système de messagerie sous-jacent via des ***connecteurs (connector)***.



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 :

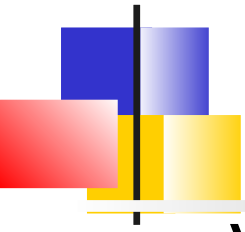
- La charge utile (payload)
- Le message complet avec Message

Ou alors elle peut retourner :

- Un *Subscriber* (ReactiveStream)
- Un *CompletionStage*

Exemples

```
@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 CompletionStage<Void> consume(double price) {

  }
  @Incoming("prices")
  public Subscriber<Message<Double>> consume() {

  }
  @Incoming("prices")
  public Subscriber<Double> consume() {

  }
}
```

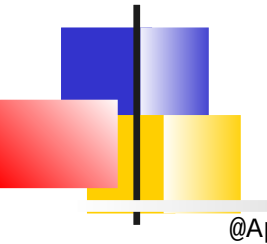


Envoi de message

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

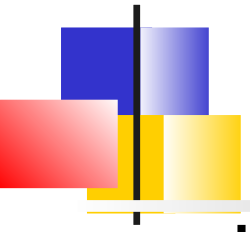
Le retour de la méthode peut être :

- La payload
- Le Message
- Un *Publisher* encapsulant la payload ou le Message
- Un *CompletionStage* encapsulant la payload ou le Message



Exemples

```
@ApplicationScoped
public class PriceConsumer {
  @Outgoing("prices")
  public Message<Double> produce() {  // process your price. }
  @Outgoing("prices")
  public Double produce() {

  }
  @Outgoing("prices")
  public CompletionStage<Message<Double>> produce() {

  }
  @Outgoing("prices")
  public CompletionStage<Double> produce() {

  }
  @Outgoing("prices")
  public Publisher<Message<Double>> produce() {

  }
  @Outgoing("prices")
  public Publisher<Double> produce() {

  }
}
```



Envoi avec *Emitter*

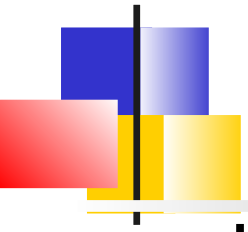
La signature des méthodes de @Outgoing ne permet pas le passage de paramètre.

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

Le bean propose la méthode send() qui retourne un *CompletionStage*, terminé lorsque le message est acquitté.

Envoi avec *Emitter*

```
@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);
    }
}
```



Processeur

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.

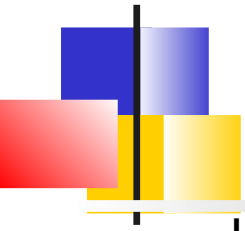
Les méthodes processeurs sont annotées avec les 2 annotations, *@Incoming* et *@Outgoing*.

Ils combinent les possibilités de signatures des 2 annotations

1. A la différence d'une *Source* et d'un *Sink*

Exemple

```
@ApplicationScoped
public class PriceProcessor {

    private static final double CONVERSION_RATE = 0.88;

    @Incoming("price-in")
    @Outgoing("price-out")
    public double process(double price) { }

    // Version asynchrone
    @Incoming("price-in")
    @Outgoing("price-out")
    public Publisher<Message<Double>> process(Message<Double> prices) {

    }

}
```



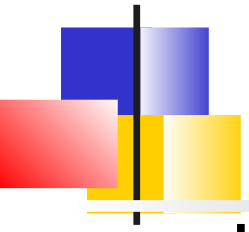
Acquittement de messages

Les messages sont acquittés soit explicitement, soit implicitement par l'implémentation.

Chaque signature de méthode peut implémenter différentes politiques d'accusé de réception par défaut.

Pour contrôler les acquittements, on utilise l'annotation ***@Acknowledgment***

- ***NONE*** - aucun acquittement n'est effectué
- ***MANUEL*** - l'utilisateur est responsable de l'accusé de réception, en appelant la méthode Message#ack(),
- ***PRE_PROCESSING*** - l'implémentation accuse la réception du message avant l'exécution de la méthode
- ***POST_PROCESSING*** - l'implémentation accuse réception du message une fois :
  - la méthode ou le traitement se termine si la méthode n'émet pas de données
  - lorsque les données émises sont acquittées



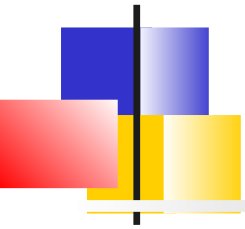
Connecteurs

Les **connecteurs** sont responsable d'associer un canal à un *sink* ou *source* de messages spécifique à une technologie

La configuration s'effectue via *MP Config*

Elle suit la structure suivante :

```
mp.messaging.incoming.[channel-name].[attribute]=[value]
mp.messaging.outgoing.[channel-name].[attribute]=[value]
mp.messaging.connector.[connector-name].[attribute]=[value]

Exemples :
mp.messaging.outgoing.prices-out.connector=smallrye-kafka
mp.messaging.outgoing.prices-out.topic=prices
```

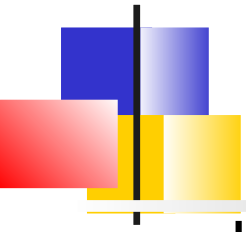


Observabilité

MP Health

MP OpenTracing

MP Metrics

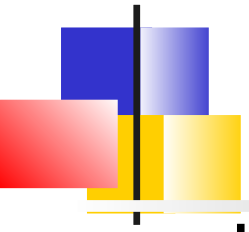


Motivation

Les health-checks sont utilisés pour sonder l'état d'une application *MicroProfile* à partir d'une autre machine (i.e., le contrôleur kubernetes)

Dans ce scénario, les vérifications d’état sont utilisées pour déterminer si un service doit être supprimé (arrêté) et éventuellement remplacé par une autre instance (saine).

Il n'est pas destiné (bien qu'il puisse être utilisé) comme solution de surveillance pour les opérateurs humains.

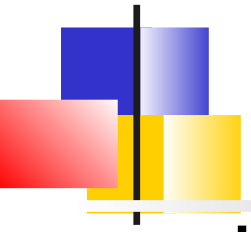


Principes

L'architecture ***MicroProfile Health Check*** se compose de 2 points de terminaison :

- ***/health/ready*** : Le service est prêt à l'emploi
- ***/health/live*** : Le service est vivant (il progresse).

Ces points de terminaison sont associés à des procédures de vérifications via les annotations ***@Liveness*** et ***@Readiness***.



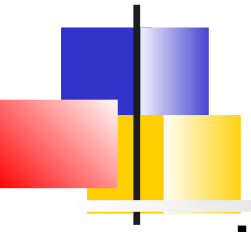
MP Health

La spécification comporte 2 parties :

- Un protocole de vérification et santé avec un format des données
- Une API Java permettant d'implémenter les procédures de vérification.

Implémentation :

SmallRye Health

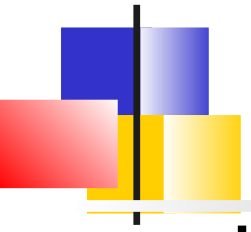


HealthCheck

L'API principale pour fournir des procédures de vérification est l'interface ***HealthCheck***

```
@FunctionalInterface
public interface HealthCheck {

    HealthCheckResponse call();
}
```

Cela s’applique aussi bien à du *Liveness* que du *Readiness*

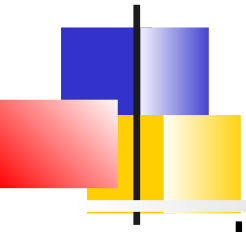


HealthCheckResponse

La valeur de retour de l'interface fonctionnelle est ***HealthCheckResponse***

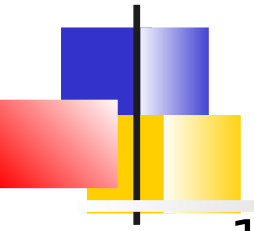
```
public abstract class HealthCheckResponse {
    public enum State { UP, DOWN }
    public abstract String getName();
    public abstract State getState();
    public abstract Optional<Map<String, Object>> getData();

    [...]
}
```

Exemple avec données arbitraires



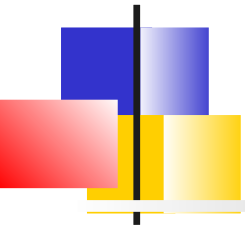
Les méthodes statiques de *HealthCheckResponse* permet d'avoir accès à un *HealthCheckResponseBuilder*

```
public class CheckDiskspace implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.named("diskspace")
                .withData("free", "780mb")
                .up()
                .build();
    }
}
```



Protocole

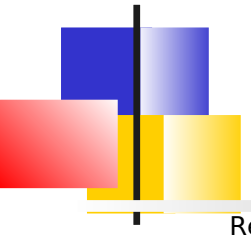
1. Le consommateur invoque la vérification de santé d'un producteur via l'un des protocoles pris en charge
2. Le producteur applique des contraintes de sécurité sur l'invocation (c'est-à-dire l'authentification)
3. Le producteur exécute un ensemble de procédures de vérification de l'état
4. Le producteur détermine l'état général
5. L'état est mappé sur le protocole (c'est-à-dire les codes d'état HTTP)
6. La charge utile est écrite dans le flux de réponse
7. Le consommateur lit la réponse
8. Le consommateur détermine l'état général



HTTP

- Concernant HTTP et des endpoints REST, la spécification définit
  - Les codes retours
    - 200 : UP
    - 500, 503 : DOWN
  - La charge utile JSON

Exemples



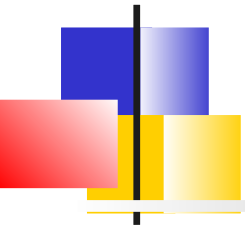
Retour 200

```
{
  "status": "UP",
  "checks": [
    {
      "name": "myCheck",
      "status": "UP",
      "data": {
        "key": "value",
        "foo": "bar"
      }   }  ]  }
```

—

Retour 500 ou 503

```
{
  "status": "DOWN",
  "checks": [
    {
      "name": "example.health.FirstCheck",
      "status": "DOWN",
      "data": {
        "rootCause": "timed out waiting for available connection"
      }
    },
    {
      "name": "secondCheck",
      "status": "UP"
    }  ]  }
```

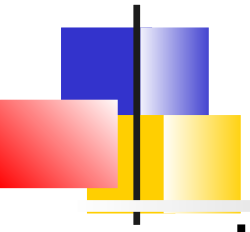


Observabilité

MP Health

MP OpenTracing

MP Metrics

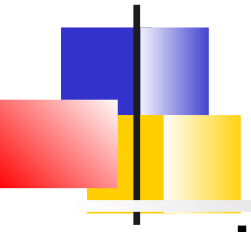


Tracing distribué

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

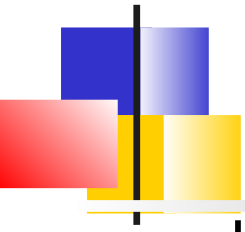
Chaque service doit être instrumenté pour consigner les messages avec un ID de corrélation qui peut avoir été propagé à partir d'un service en amont.

Un autre service stocke et agrègent les traces



MP OpenTracing

La spécification a pour but de faciliter l'instrumentation de services avec une fonction de traçage distribué, étant donné un système de traçage distribué existant (Zipkin, Jaeger).



Tracing

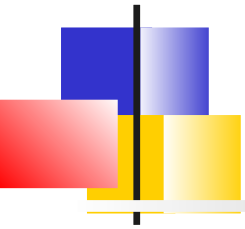
La spécification impose que tous les appels REST soient automatiquement tracés sans intervention des développeurs.

Des spans supplémentaires peuvent être définies en annotant des méthodes avec ***@Traced***

Des librairies connexes sont disponibles pour définir d'autres types de span

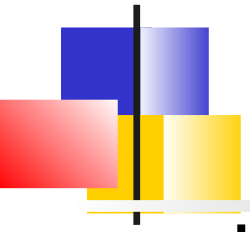
Eco-système Quarkus :

- Jdbc : extension ***opentracing-jdbc***
- Kafka : extension ***opentracing-kafka-client***
- MongoDB : extension ***opentracing-mongo-common***



Observabilité

MP Health
MP OpenTracing
MP Metrics

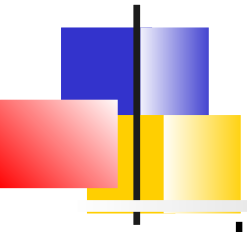


Introduction

Les architecture micro-services nécessitent du monitoring et JMX n’est plus adapté.

La spécification ***MP Metrics*** vise à fournir :

- un moyen unifié afin que des services puissent exporter les données de monitoring vers les agents de gestion
- une API Java permettant d’exposer leurs données de télémétrie.



Scopes

La spécification organise les métriques en scope :

- **Base** : Les métriques que chaque implémentation doit fournir, exposés sur :
  /metrics/base
- **Vendor** : Des métriques spécifiques à l'implémentation, exposés sur :
  /metrics/vendor
- **Applicatif** : Des métriques définis via l'API, exposés sur :
  /metrics/application

Les métriques sont taggés permettant le filtre et l'agrégation et des méta-données y sont associées

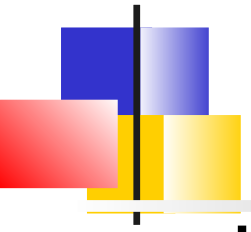
MetricRegistry

MetricRegistry stocke les métriques et les informations de métadonnées.

Il existe une instance *MetricRegistry* pour chacun des scopes.

Les métriques peuvent être ajoutées ou extraites du registre soit à l'aide de l'annotation ***@Metric***, soit directement via l'objet *MetricRegistry*.

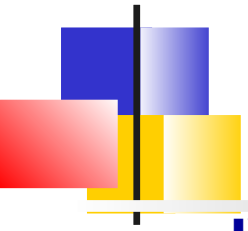
Chaque métrique a un ID unique



API Rest

Les données sont exposées sous le chemin de base ***/metrics*** dans 2 formats de données :

- Format JSON : en-tête *HTTP Accept : application/json*.
- Format de texte OpenMetrics : en-tête *HTTP Accept* : *text/plain* ou lorsqu'aucun type de média n'est demandé



Métrique requis

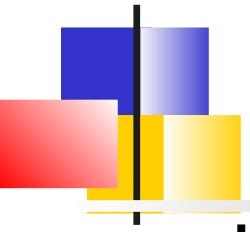
JVM : *UsedHeapMemory, CommittedHeapMemory, MaxHeapMemory, GCCount, GCTime, JVM Uptime*

Threads : *ThreadCount, DaemonThreadCount, PeakThreadCount, ActiveThreads, PoolSize*

ClassLoading : *LoadedClassCount, TotalLoadedClassCount, UnloadedClassCount*

OS : *AvailableProcessors, SystemLoadAverage, ProcessCpuLoad, ProcessCpuTime*

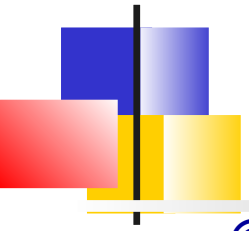
REST (Optionnel) : *RESTRequests*



Métriques applicatifs

La façon la plus simple de déclarer un métriques applicatif est d'utiliser des annotations.
Les métriques sont alors automatiquement enregistrés et exposés par l'implémentation de MP Metrics

Il est également possible de travailler directement sur le *MetricRegistry*



Annotations

@Counted : Compte les invocations de l'objet annoté

@Gauge : Indique une jauge, qui échantillonne la valeur de l'objet annoté. On doit fournir une unité

@ConcurrentGauge : Indique une jauge qui compte les invocations parallèles de l'objet

@Metered : Désigne un compteur, qui suit la fréquence des appels de l'objet annoté.
Unité par défaut MetricUnits.PER_SECOND

@Timed : Trace la durée de l'objet annoté. Utilisé pour des sttaistiques de durée et de de débit.
Unité par défaut MetricUnits.NANOSECONDS

@SimplyTimed : Trace les invocations et leur durée.
Unité par défaut MetricUnits.NANOSECONDS

@Metric : Renseigne les méta-données associées à la jauge, au compteur, etc..

Exemple

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

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



    @GET
    @Path("business")
    @Counted
    @Metric(name="countBusiness")
    public Long doBusiness() {

    }
}
```

Exemple

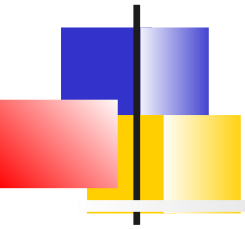
```
@Path("/example")
@Produces("text/plain")
public class ExampleResource {

    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
```

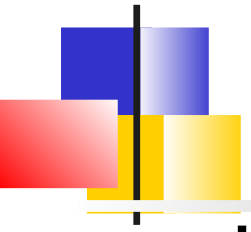
Les URLs */metrics* affichent (au format texte) la valeur de *example_list_size*

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



Sécurité

MP JWT



Introduction

L'objectif de la spécification ***JWT RBAC Security (MP-JWT)*** est la définition du format requis du JWT utilisé comme pour l'authentification et l'autorisation

JOSE

JavaScript Object Signing and Encryption (JOSE) est un groupe de travail qui a fournit plusieurs spécifications liées à la sécurité et JSON

En particulier :

- ***JWT*** : Un jeton au format JSON qui contient des entêtes et des revendications (key/value).
  Les revendications sont utilisées pour appliquer des ACLs

Les revendications peuvent être représentées via :

- ***JSON Web Signature*** (***JWS***) : Dans ce cas, elles peuvent être lues car simplement encodé en base 64. Cependant, un signature permet de vérifier leur authenticité
- ***JSON Web Encryption*** (***JWE***) : Dans ce cas, les revendications sont cryptées et nécessite un décryptage pour appliquer les ACLs.
  On parle de jeton opaque

Jetons de sécurité

Dans le cadre d’une architecture microservices RESTful, les jetons de sécurité offrent un moyen très léger et interopérable de propager les identités sur les différents services :

- Les services n'ont pas besoin de stocker d'état sur les clients ou les utilisateurs
- Les services peuvent vérifier la validité du jeton si le jeton suit un format bien connu. Sinon, les services peuvent invoquer un service séparé.
- Les services peuvent identifier l'appelant en examinant le jeton. Si le jeton suit un format bien connu, les services sont capables d'introspecter le jeton par eux-mêmes, localement. Sinon, les services peuvent invoquer un service séparé.
- Les services peuvent appliquer des politiques d'autorisation basées sur n'importe quelle information contenue dans un jeton de sécurité
- Prise en charge de la délégation et de l'emprunt d'identité

Rôles du protocole

L’ **émetteur de jeton** à la suite de la vérification réussie d'une identité (authentification). Les émetteurs sont généralement liés à des fournisseurs d'identité.

Le **Client** est l’application qui utilise le jeton.

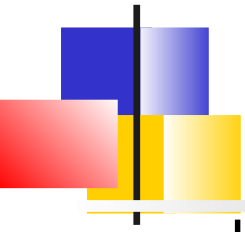
Le **sujet :** Entité à laquelle les informations d'un jeton font référence.

Le **serveur de ressources** est l’API qui consomme le jeton afin de vérifier si il permet d’accéder aux ressources

Etapes pour l’authentification

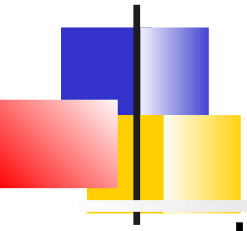
Indépendamment du format de jeton ou du protocole utilisé, l'authentification repose sur les étapes suivantes :

- Extraire le jeton de sécurité de la requête (en-tête d'autorisation).
- Effectuer des vérifications de validation par rapport au jeton. Vérifications de signature, de cryptage et d'expiration.
- Introspecter le jeton et extraire des informations sur le sujet. L'objectif est d'obtenir toutes les informations nécessaires sur le sujet à partir du jeton.
- Créer un contexte de sécurité pour le sujet



Avantages de JWT

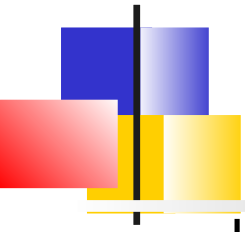
- Les jetons peuvent être validée localement par chaque service
- Compte tenu de sa nature JSON, le traitement des jetons JWT est léger.
- Facilite la prise en charge de différents types de mécanismes de contrôle d'accès tels que ABAC, RBAC, context-based, etc.
- Sécurité au niveau des messages utilisant la signature et le cryptage tels que définis par les normes JWS et JWE
- Les parties peuvent convenir d'un ensemble spécifique de revendications (claims) afin d'échanger à la fois des informations d'authentification et d'autorisation.
- Largement adopté par différentes solutions SSO et des standards comme OpenID Connect



Inter-opérabilité

L'utilité maximale du *MP-JWT* en tant que format de jeton dépend de l'accord entre les fournisseurs d'identité et les fournisseurs de services.

Les fournisseurs d'identité - responsables de l'émission des jetons - doivent pouvoir émettre des jetons en utilisant le format MP-JWT afin que les fournisseurs de services puissent introspecter le jeton et recueillir des informations sur un sujet.



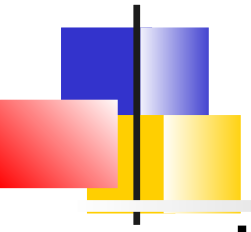
Exigences

Les exigences pour le MicroProfile JWT sont donc :

- Être utilisable comme jeton d'authentification.
- Être utilisable en tant que jeton d'autorisation (rôles au niveau de l'application Java EE)
- Peut être mappé à *IdentityStore* de la *JSR375*.
- Peut prendre en charge les revendications standard supplémentaires décrites dans les attributions *IANA JWT* ainsi que les revendications non standard.

A cet effet, 2 nouveau *claims* ont été ajoutés :

- **"*upn*"**: Une revendication lisible par l'homme qui identifie de manière unique le sujet ou l'utilisateur principal du jeton,
- "***groups***": Une liste de nom de groupes qui seront associés aux rôles applicatifs.



Entêtes et claims

La spécification définit donc les entêtes et les claims constituant le jetons.

Elle définit :

- Les entêtes et claims requis
- Les recommandés

Entêtes

Entêtes requises :

- ***alg*** : Identifie l'algorithme cryptographique utilisé pour sécurisé JWT : RSASSA-PKCS1-v1_5, ECDSA, RSAES
- ***enc*** : identifie l'algorithme cryptographique utilisé pour chiffrer les revendications ou les jetons JWT imbriqués

Entêtes recommandées :

- ***typ*** : Égal à « JWT », Voir RFC7519, Section 5.1
- ***kid*** : un indice indiquant quelle clé a été utilisée pour sécuriser le JWT. Voir RFC7515, Section-4.1

Revendications (claims)

Requises :

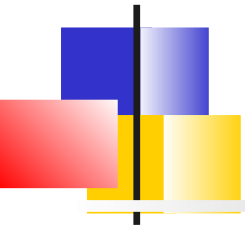
- ***iss*** : Identifie le fournisseur de jeton
- ***iat*** : Identifie la date de création du jeton
- ***exp*** : Identifie la date d'expiration
- ***upn*** : Le nom du user dans *java.security.Principal*

Recommandées

- ***sub*** : Le subject associé à upn
- ***jti*** : Identifiant unique du jeton contenant le namespace du fournisseur
- ***aud*** : Identifie les endpoints de MP JWT

Exemple minimal

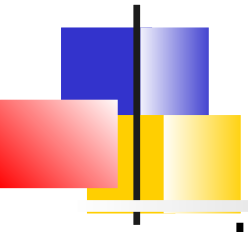
```
{
     "typ": "JWT",
    "alg": "RS256",
    "kid": "abc-1234567890"
}
{
       "iss": "https://server.example.com",
       "jti": "a-123",
       "exp": 1311281970,
       "iat": 1311280970,
       "sub": "24400320",
       "upn": "jdoe@server.example.com",
       "groups": ["red-group", "green-group", "admin-group", "admin"],
}
```



Classe Claim

La classe utilitaire ***org.eclipse.microprofile.jwt.Claims*** encapsule une énumération de toutes les revendications standard liées à JWT, ainsi qu'une description et le type Java requis.

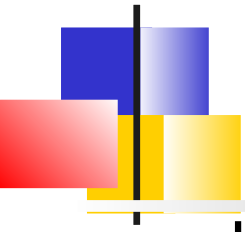
```
public enum Claims {

    iss("Issuer", String.class),
    sub("Subject", String.class),
    exp("Expiration Time", Long.class),
    iat("Issued At Time", Long.class),
    jti("JWT ID", String.class),
    ….
```



Configuration de l’authentification

L’ annotation ***org.eclipse.microprofile.auth.LoginConfig*** fournissant les mêmes informations que l'élément *login-config* de *web.xml*, permet de marquer une application JAX-RS comme nécessitant MicroProfile JWT RBAC

```
@LoginConfig(authMethod = "MP-JWT", realmName = "TCK-MP-JWT")
@ApplicationPath("/")
public class TCKApplication extends Application {
}
```

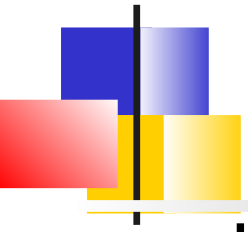


Injection du Jeton

Une implémentation MP-JWT doit prendre en charge l'injection de l'appelant actuellement authentifié en tant que ***JsonWebToken***.

Le bean a une portée *@RequestScoped* même si le bean externe englobant est *@ApplicationScoped*

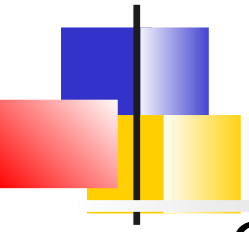
```
@Path("/endp")
@DenyAll
@ApplicationScoped
public class RolesEndpoint {

    @Inject
    private JsonWebToken callerPrincipal;
```



Injection de claims

Les revendications doivent pouvoir également être injecté avec le qualifier ***org.eclipse.microprofile.jwt.Claim*** :

```
@ApplicationScoped
public class MyEndpoint {
    @Inject
    @Claim(value="exp", standard=Claims.iat)
    private Long timeClaim;
...
}
```

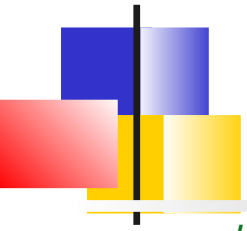


Jetons signés

Généralement, les jetons JWT sont signés[1]
La vérification de la signature s'effectue alos via une clé publique
Les implémentations MP-JWT doivent pouvoir récupérer les clés publiques via une requête https[2]

Le jeton signé peut également être chiffré (On parle alors de jeton imbriqué interne).[3]
Dans ce cas, il devra d'abord être déchiffré.

1. Spécification JSON Web Signature (JWS)
2. Format JSON Web Key (JWK) et JSON Web Key Set (JWKS)
3. JSON Web Encryption(JWE)



Configuration

```
# Configuration clé publique pour validation
##############################################
# la clé sous forme de String
mp.jwt.verify.publickey

# L'emplacement, peut être une URL
mp.jwt.verify.publickey.location

# L'algorithme
mp.jwt.verify.publickey.algorithm

# Configuration clé privé pour décryptage
##############################################
# Attention pas d'URI ici
mp.jwt.decrypt.key.location
```

Vérification fournisseur et audience

MP JWT définit également la vérification des revendications concernant :

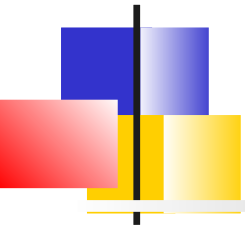
- ***iss*** : Le fournisseur de jeton
- **aud** : Les audiences.

Ces vérifications interviennent après la validation et le décryptage éventuel.

Elles sont configurées par :

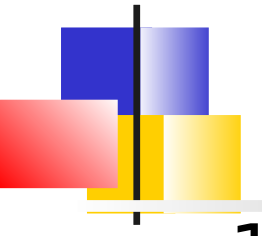
mp.jwt.verify.issuer : Indique le fournisseur attendu

mp.jwt.verify.audiences : Indique les audiences supportées



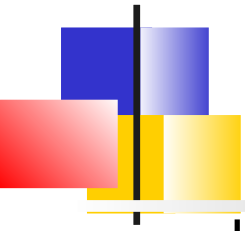
Annexes

oAuth2



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

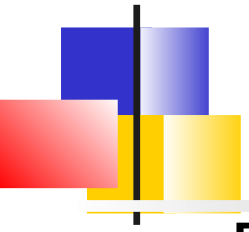
Le protocole ne définit pas comment l'enregistrement du client doit se faire mais définit les paramètres d'échange.

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

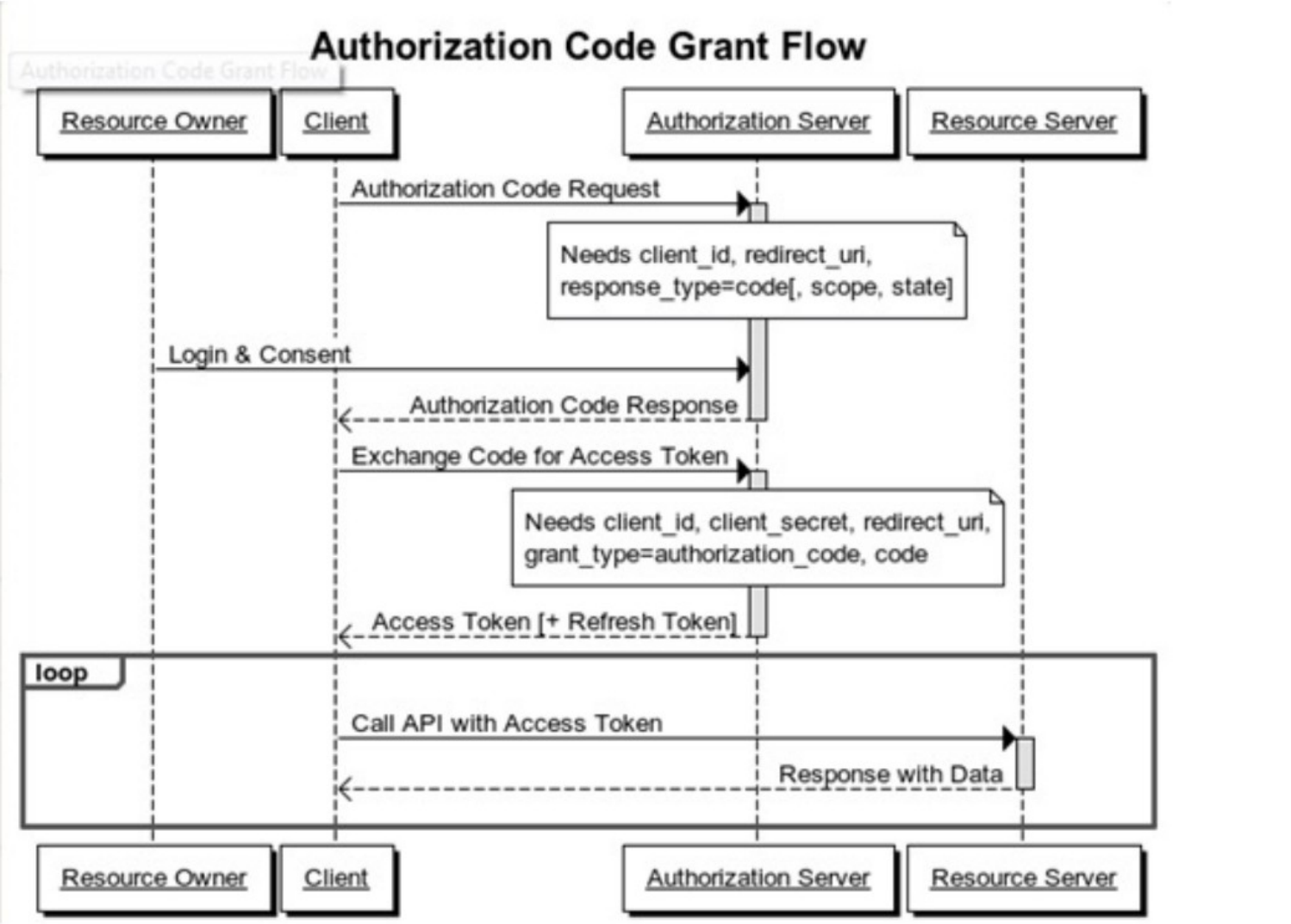


OAuth2 Grant Type

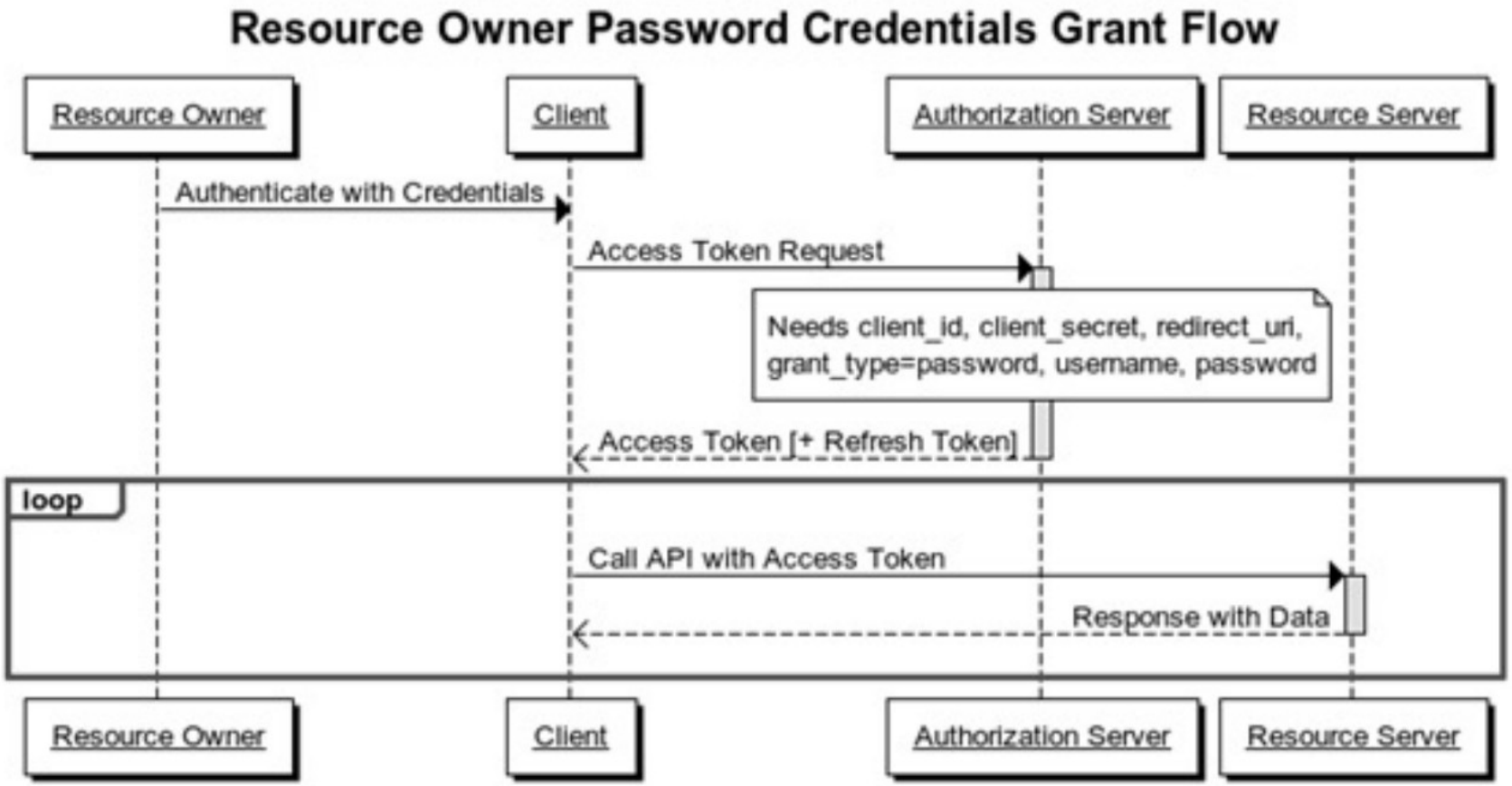
Différents moyens afin que l'utilisateur donne son accord : les **grant types**

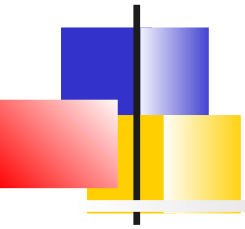
- ***authorization code*** : Approbation de l'utilisateur sur le serveur d'autorisation et échange d'un code d'autorisation avec le client
- ***implicit*** : Jeton fourni directement. Certains serveurs interdisent de mode
- ***password :*** Le client fournit les crédentiels de l'utilisateur
- ***client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

Authorization Code



Password Grant

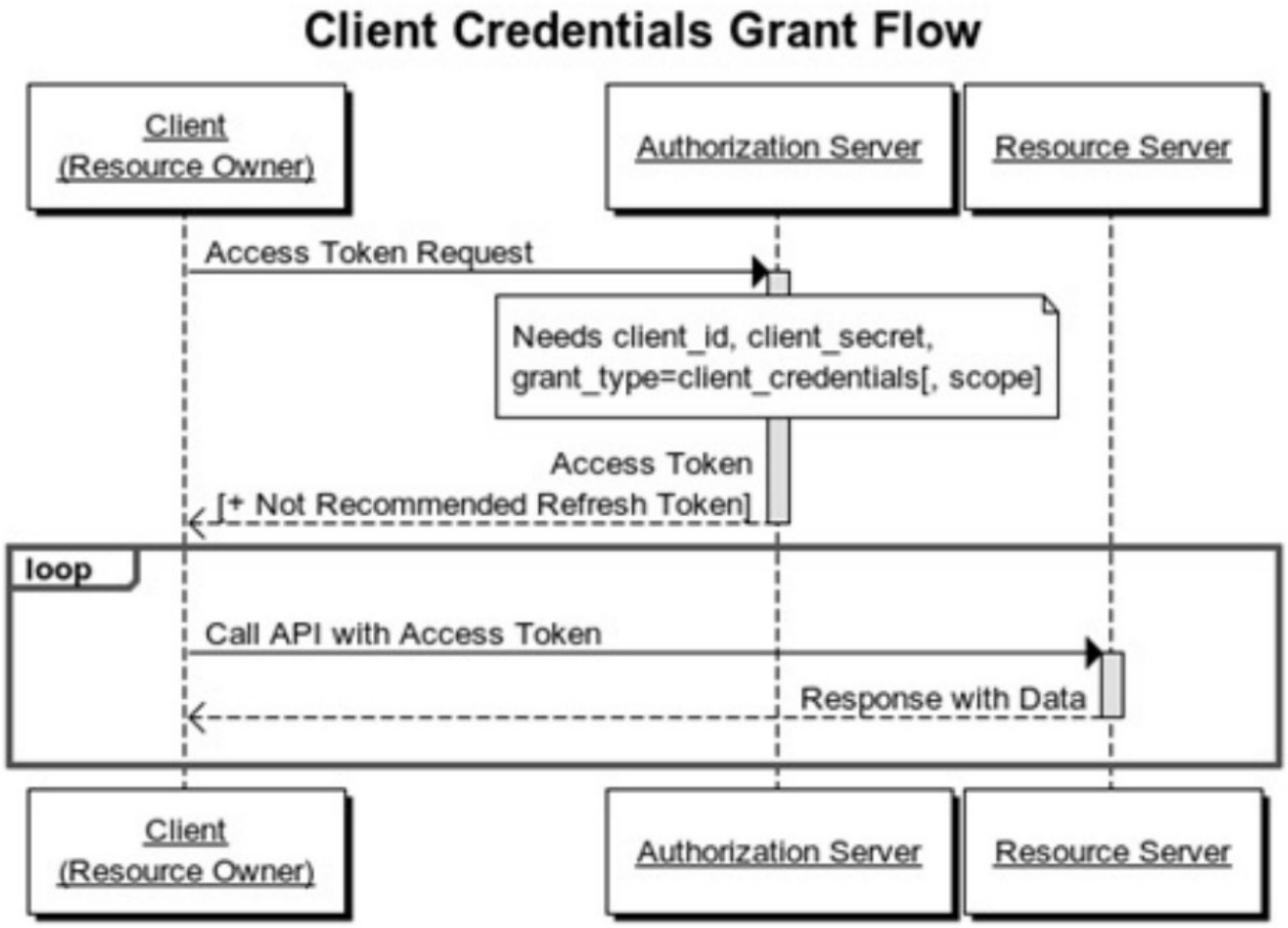


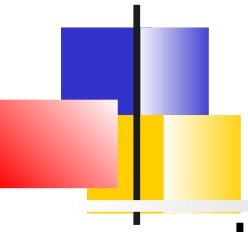


OpenAPI

- 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





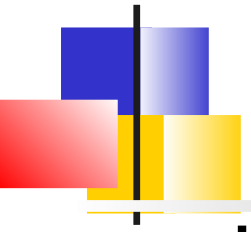
Tokens

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



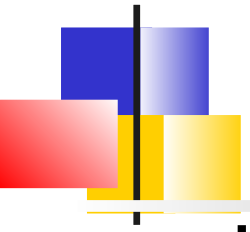
Usage du jeton

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

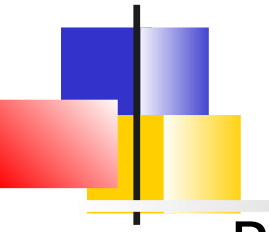
JSON Web Token (JWT) est un standard ouvert défini dans la RFC 75191.

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***



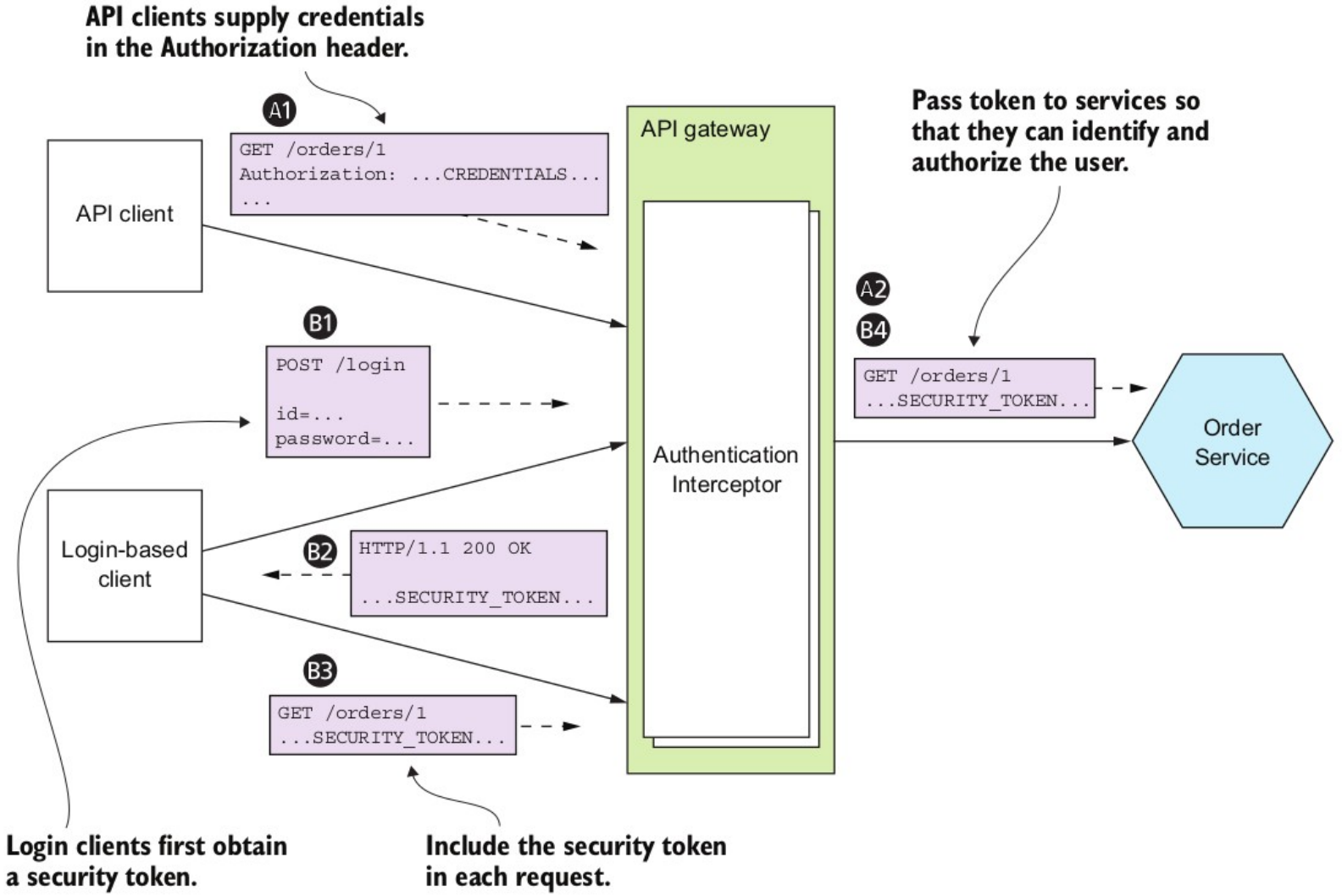
Sécurité micro-services

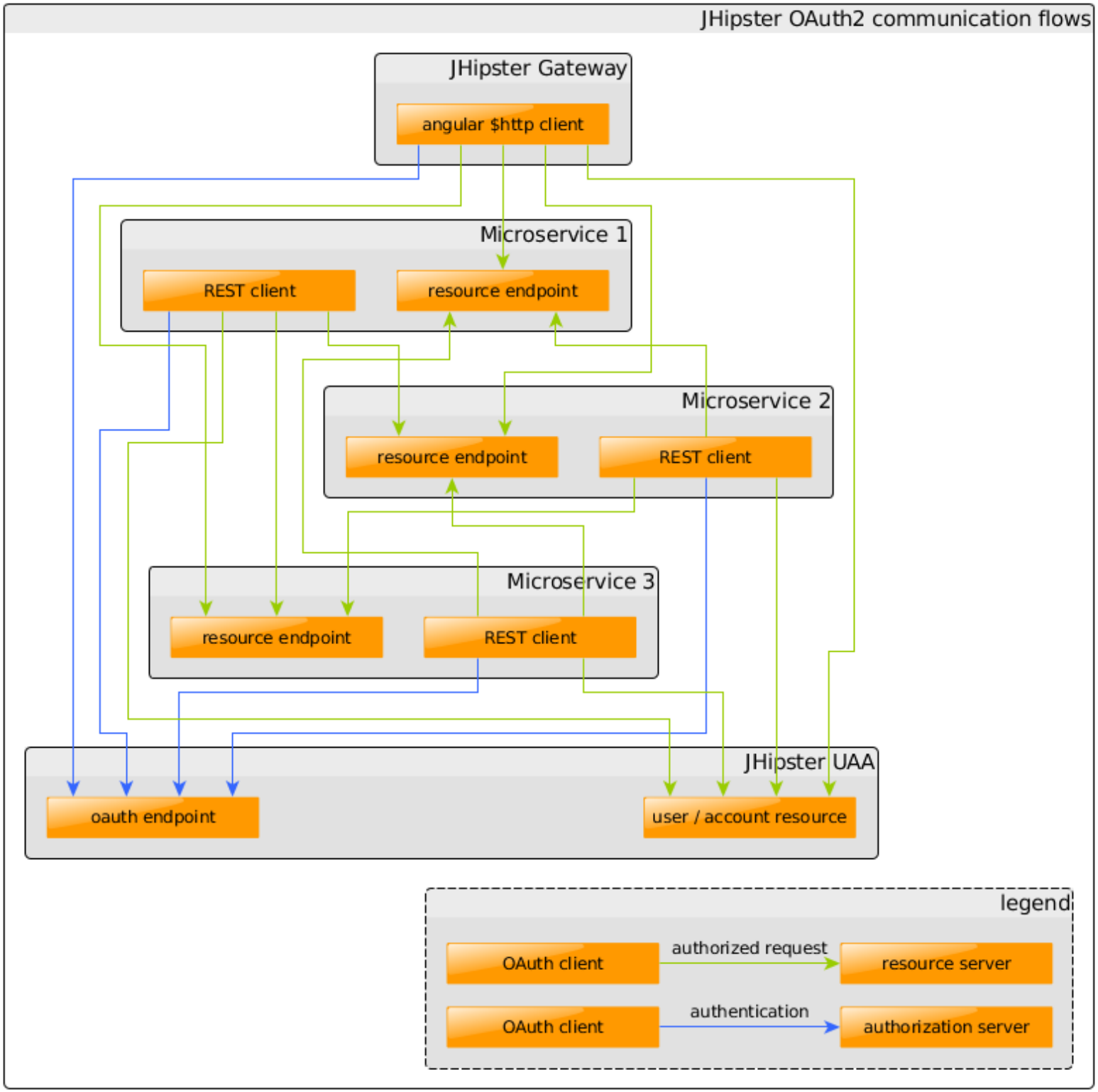
Plusieurs approches pour sécuriser une architecture micro-services :

- 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




JHipster OAuth2 communication flows
JHipster Gateway
angular $http client
Microservice 1
REST client
resource endpoint
Microservice 2
resource endpoint
REST client
Microservice 3
resource endpoint
REST client
JHipster UAA
oauth endpoint
user / account resource
legend
OAuth client
authorized request
resource server
OAuth client
authentication
authorization server