



Spring / Kafka

David THIBAU – 2023

david.thibau@gmail.com



Agenda

Spring coeur et Spring Boot

- Rappels SpringCoeur et principales annotations
- L'accélérateur SpringBoot
- Principaux starters
- Développer avec SpringBoot

Interactions entre services Spring

- Styles d'interaction et API
- RestFul avec Spring
- Messaging avec Spring

Présentation Kafka

Le projet Kafka
Concepts cœur de Kafka, Architecture
Les différents cas d'usage de Kafka

Cluster Kafka

- Nœuds du cluster
- Kraft
- Distribution / installation
- Utilitaires Kafka
- Outils graphiques d'administration

Apache Kafka et ses APIs

- Les APIs Kafka
- Producer API
- Consumer API
- Schema Registry et Avro
- Autres APIS, KafkaAdmin et KafkaStream

Configuration cluster et topics

Stockage réplification des partitions
Garanties de livraison
Configuration pour la latence et le débit
Configuration de la rétention



Agenda (2)

Spring Kafka

- Introduction
- Production de messages
- Consommation de messages
- Transaction et sémantique Exactly Once
- Sérialisation / Désérialisation
- Traitement des Exceptions

Sécurité Kafka

- Configuration des listeners
- Mise en place SSL/TLS
- Authentification via SASL
- ACLs



Spring coeur et Spring Boot

Rappels Spring Coeur et annotations

L'accélérateur SpringBoot

Principaux starters

Développer avec SpringBoot



IoC et framework

- ❖ Spring s'appuie sur le pattern ***IoC (Inversion Of Control)*** :

Ce n'est plus le code du développeur qui a le contrôle de l'exécution mais Spring

- ❖ Le framework est alors responsable d'instancier les objets, d'appeler les méthodes, de libérer les objets, etc..

=> Le code du développeur est réduit au code métier. Les services techniques transverses sont pris en charge par le framework.



Injection de dépendances

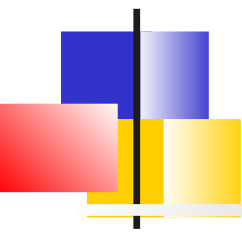
- ❖ L'**injection de dépendance** est une spécialisation du pattern *IoC*
 - Lors de la construction des objets, le framework doit initialiser leurs attributs et en particulier les objets collaborateurs, i.e les dépendances
- ❖ En définissant les dépendances comme des interfaces, on peut définir différentes configurations injectant différentes implémentations
 - Le code devient très évolutif, très testable, adaptable à un environnement particulier (Test, Dév, Prod, ...)



Illustration

```
public interface MovieService {  
    List<Movie> findAll();  
}  
  
---  
@Controller  
public class MovieController {  
    private MovieService movieService ;  
    // Injection de dépendance par constructeur  
    public MovieController(MovieService movieService) {  
        this.movieService = movieService  
    }  
    public List<Movie> moviesDirectedBy(String director) {  
        List<Movie> allMovies = movieService.findAll();  
        return allMovies.stream()  
            .filter(m -> !m.getDirector().equals(director))  
            .collect(Collectors.toList()) ;  
    }  
}
```

- ❖ La classe contrôleur ne dépend que de l'interface *MovieService*.
- ❖ Une configuration particulière Spring indiquera l'implémentation à utiliser pour *MovieService*



Configuration du framework

- ❖ Différents moyens pour configurer le framework :
 - Fichiers de configuration XML
Legacy, toujours supporté peut être intéressant dans certains cas
 - Classes de configuration et annotations Java
Depuis Java5, Technique plus souple pour le développeur
 - Appliquer automatiquement des configurations par défaut
C'est le cas de SpringBoot



ApplicationContext

Quelque soit le moyen utilisé pour configurer Spring, à la fin de l'initialisation, Spring crée une classe ***ApplicationContext*** qui regroupe toutes les configurations des beans et qui permet de les instancier.

ApplicationContext expose des méthodes intéressantes (pour le debug généralement)

- *getBeanDefinitionNames()* : Récupérer tous les noms des bean de l'application
- *getBean(Class<T> requiredType)* : Récupérer un bean à partir d'un type
- *getBean(String name)* : Récupérer un bean à partir de son nom
- ...



Classes de configuration

@Configuration annote les classes pouvant instancier des beans

- Les méthodes annotées par **@Bean** sont exécutées lors de l'initialisation du framework
- Technique utilisée pour les beans *techniques*

@Configuration

```
public class JdbcConfig {  
    @Bean  
    DataSource datasource() {  
        DriverManagerDataSource ds = new DriverManagerDataSource();  
        ...  
        return ds;  
    }  
}
```



Attributs de *@Bean*

@Bean définit 3 attributs :

name : les alias du bean

init-method : Méthode appelée après l'initialisation du bean par Spring

destroy-method : Méthode appelée avant la destruction du bean par Spring

@Configuration

```
public class AppConfig {  
    @Bean(name={"foo", "super-foo"}, initMethod = "init")  
    public Foo foo() {  
        return new Foo();  
    }  
    @Bean(destroyMethod = "cleanup")  
    public Bar bar() {  
        return new Bar();  
    }  
}
```



Annotations *@Enable*

Les classes *@Configuration* sont utilisées pour configurer des ressources externes au métier (une base de données, les beans techniques du framework, ...)

Pour faciliter la configuration, Spring fournit des annotations ***@Enable*** qui appliquent une configuration par défaut

- ***@EnableWebMvc*** : Configuration Spring MVC dans une application
- ***@EnableCaching*** : Permet d'utiliser les annotations *@Cachable*, ...
- ***@EnableScheduling*** : Permet d'utiliser les annotations *@Scheduled*
- ***@EnableJpaRepositories*** : Permet de scanner les classe Repository
- ...



@Component et stereotypes

L'annotation **@Component**, placée sur la classe permet de définir des beans métier.

D'autres annotations appelées stéréotypes servent à classer les beans métier et éventuellement à leur ajouter des comportements génériques.

- **@Repository**,
- **@Service**
- **@Controller** et **@RestController** pour la couche web



Exemple

@Controller

```
public class MovieController {  
    private MovieService movieService;  
  
    public MovieController(MovieService movieService) {  
        this.movieService = movieService;  
    }  
}  
---
```

@Service

```
public class JpaMovieService implements MovieService {  
    ...  
}
```



Cycles de vie

Les beans peuvent avoir 3 cycles de vie (ou scope) définis par l'annotation **@Scope**:

- **Singleton (Défaut)**: Une seule instance de l'objet (qui est donc partagé).
Idéal pour des beans « stateless ».
=> C'est l'écrasante majorité des cas.
- **Prototype** : Une nouvelle instance à chaque utilisation.
=> Quasiment Jamais
- **Custom object "scopes"** : Cycle de vie synchronisé avec d'autres objets, comme une requête HTTP, une session http, une transaction BD
=> Certains beans fournis par Spring. Ex : *EntityManager*

@Scope("prototype")

@Repository

```
public class MovieFinderImpl implements MovieFinder {  
    // ...  
}
```



Méthodes de call-back

Spring supporte les annotations de call-back **@PostConstruct** et **@PreDestroy**

@Component

```
public class CachingMovieLister {  
    @PostConstruct  
    public void populateMovieCache() {  
        // Initialisation après construction...  
    }  
    @PreDestroy  
    public void clearMovieCache() {  
        // Nettoyage avant destruction...  
    }  
}
```




@ComponentScan

La recherche des annotations est généralement déclenchée via **@ComponentScan** placée sur une classe *@Configuration*

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig {
    ...
}
...
public static void main(String[] args) {
    ApplicationContext ctx =
        new AnnotationConfigApplicationContext(AppConfig.class);
    ...
}
```



Injection de dépendances

Pour initialiser les dépendances, Spring peut s'appuyer sur :

- Le **type** de l'attribut (généralement une interface)
- Le **nom** du bean.
Chaque bean instancié ayant 1 ou plusieurs noms (les alias)



@Autowired

L'injection par type peut s'effectuer via ***@Autowired*** qui peut se placer à de nombreux endroits :

- Déclaration d'attributs, arguments de constructeur, méthodes arbitraires...
- *@Autowired* a un attribut supplémentaire *required*, (*true* par défaut)



Examples

```
@Controller
public class MovieController {
    @Autowired
    private MovieService movieService;
    ...
}

...

public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(MovieCatalog mCatalog, CustomerPreferenceDao cPD) {
        this.movieCatalog = mCatalog;
        this.customerPreferenceDao = cPD;
    }
    // ...
}
```



Injection implicite

Dans les dernières versions de Spring, l'annotation *@Autowired* peut disparaître :

- Si la dépendance est présente comme argument de constructeur
- En général, l'attribut est déclaré comme ***final***

=> C'est ce qu'on appelle l'injection implicite¹, c'est une implémentation par type

1. On retrouve la même idée dans le framework Angular




Injection implicite

```
@Controller
public class MovieController {
    private final MovieService movieService ;

    public MovieLister(MovieService movieService) {
        this.movieService = movieService ;
    }

    public List<Movie> moviesDirectedBy(String arg) {
        List<Movie> allMovies = movieService.findAll();
        List<Movie> ret = new ArrayList<Movie>() ;
        for (Movie movie : allMovies ) {
            if (!movie.getDirector().equals(arg))
                ret.add(movie);
        }
        return ret;
    }
}
```



Exceptions dues à l'injection par type

L'injection par type peut provoquer des exceptions au démarrage de Spring.

- Cas 1 : Spring n'arrive pas à trouver de définitions de Beans correspondant au type :

UnsatisfiedDependencyException,
No qualifying bean of type '' available:
expected at least 1 bean which qualifies as autowire candidate.

- Cas 2 : Spring trouve plusieurs Beans du type demandé

UnsatisfiedDependencyException,
No qualifying bean of type '' available:
expected single matching bean but found 2.



@Resource, @Qualifier

@Resource et **@Qualifier** permettent d'injecter un bean par son nom.

- L'annotation prend l'attribut ***name*** qui doit indiquer le nom du bean
- Si l'attribut *name* n'est pas précisé, le nom du bean à injecter correspond au nom de la propriété



Exemples

```
public class MovieRecommender {  
  
    @Resource(name="myPreferenceDao")  
    private CustomerPreferenceDao cpDao;  
  
    // Le nom du bean recherché est "context"  
    @Resource  
    private ApplicationContext context;  
  
    // Plusieurs beans implémentent Formatter  
    @Autowired  
    @Qualifier("fooFormatter")  
    private Formatter formatter;  
  
    public MovieRecommender() {  
    }  
    // ...  
}
```



Propriétés de configuration

Spring permet également d'injecter de simples valeurs (String, entiers, etc.) dans les attributs des beans via des annotations :

- **@PropertySource** permet d'indiquer un fichier *.properties* permettant de charger des valeurs de configuration (clés/valeurs)
- **@Value** permet d'initialiser les propriétés des beans avec une expression **SpEl** référençant une clé de configuration



Exemple

@Configuration

@PropertySource("classpath:/com/myco/app.properties")

public class AppConfig {

@Value("\${my.property:0}") // Le fichier app.properties définit la valeur de la clé "my.property"

Integer myIntProperty ;

@Autowired

Environment env;

@Bean

public static **PropertySourcesPlaceholderConfigurer** properties() {
 return new PropertySourcesPlaceholderConfigurer();

}

@Bean

public TestBean testBean() {

TestBean testBean = new TestBean();

testBean.setIntProperty(myIntProperty) ;

testBean.setName(**env.getProperty("testbean.name")**); // app.properties définit "testbean.name"

return testBean;

}

}



Profils

Les **profils** permettent d'appliquer des configurations différentes dans le même projet.

Ils ont une influence :

- Sur les beans instanciés.
Certains beans ne sont instanciés que si leur profil est activé
- Sur les valeurs des propriétés.
Différentes valeurs peuvent être définies en fonction des profils



Annotations utilisant les profils

Tout *@Component* ou *@Configuration* peut être marqué avec **@Profile** pour limiter son chargement

```
@Configuration
@Profile("production")
@PropertySource("classpath:/com/myco/app-prod.properties")
public class ProductionConfiguration {

    // ...

}

@Service
@Profile("kafka")
public class MyKafkaServiceImpl implements MyService {

    // ...

}
```



Environment

L'interface ***Environment*** est une abstraction modélisant 2 aspects :

- Les **propriétés** : Ce sont les propriétés résolues des beans.
Ils proviennent des fichier *.properties*, d'argument de commande en ligne ou autre ...
- Les **profils activés** : Une liste de String influençant les beans activés



Activation d'un profil

Programmatically :

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();  
ctx.getEnvironment().setActiveProfiles("development");  
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);  
ctx.refresh();
```

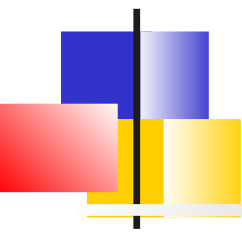
Ligne de commande :

Propriété Java

```
java -jar myJar -Dspring.profiles.active="profile1,profile2"
```

Argument SpringBoot

```
java -jar myJar --spring.profiles.active="profile1"
```



Spring coeur et Spring Boot

Rappels Spring Coeur et annotations

L'accélérateur SpringBoot

Principaux starters

Développer avec SpringBoot



Introduction

Spring Boot a été conçu pour **simplifier le démarrage** et le développement de nouvelles applications Spring

- Ne nécessite aucune configuration préalable :

Dés la première ligne de code, on a une application fonctionnelle

=> Offrir une expérience de développement simplifiant à l'extrême l'utilisation des technologies existantes



Auto-configuration

Le concept principal de *SpringBoot* est l'**auto-configuration**

SpringBoot est capable de détecter la nature de l'application et de configurer les beans Spring nécessaires

- Permet de démarrer rapidement avec une configuration par défaut puis de graduellement surcharger la configuration pour ses besoins spécifiques

Les mécanismes sont différents en fonction du langage : Groovy, Java ou Kotlin



Auto-configuration (Java)

En fonction des librairies présentes au moment de l'exécution, Spring Boot crée tous les beans techniques nécessaires avec une configuration par défaut.

- Par exemple, si il s'aperçoit que des librairies Web sont présentes, il démarre un serveur Tomcat embarqué sur le port 8080 et applique la configuration par défaut de Spring MVC ou Spring Webflux
- Si il s'aperçoit que le driver H2 est dans le classpath, il crée automatiquement un pool de connexions vers la base
- Etc...



Personnalisation de la configuration

La configuration par défaut peut être surchargée par différents moyens

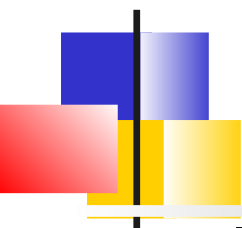
- Les propriétés qui modifient les valeurs par défaut des beans techniques via :
 - Des **variables d'environnement**
 - Des **arguments de la ligne de commande**
 - Des **fichiers de configuration externe** (*.properties* ou *.yml*). Différents fichiers peuvent être activés en fonction de profils
- Avec des classes de configuration redéfinissant les beans par défaut
- En utilisant des **classes spécifiques du framework** (exemple classes **Configurer*)



Autres apports de SpringBoot

En dehors de l'auto-configuration, SpringBoot offre d'autres bénéfices pour les développeurs :

- Simplification de la gestion des dépendances vers les librairies OpenSource via les **starters**
- Assistant de création de projet avec **SpringInitializer**
- Point central de la configuration des propriétés avec ***application.properties/yml***
- Plugins pour les **IDEs** (SpringTools suite ou autre)
- Plugins pour les **outils de build** (Gradle et Maven)



Gestion des dépendances

La gestion des dépendances est simplifiée grâce aux **starters**.

Ce sont des groupes de dépendances permettant d'intégrer une technologie ou apportant une fonctionnalité qui sont déclarés dans le fichier de build

=> Pour les développeurs, ce mécanisme simplifie à l'extrême la gestion des versions des dépendances.

Plus qu'un seul numéro de version à gérer : Celui de SpringBoot

https://start.spring.io/



Project

☒ Gradle - Groovy ☐ Gradle - Kotlin ☒ Java ☐ Kotlin ☐ Groovy
☐ Maven

Spring Boot

☐ 3.2.0 (SNAPSHOT) ☐ 3.2.0 (M1) ☐ 3.1.3 (SNAPSHOT) ☒ 3.1.2
☐ 3.0.10 (SNAPSHOT) ☐ 3.0.9 ☐ 2.7.15 (SNAPSHOT) ☐ 2.7.14

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 20 ☒ 17 ☐ 11 ☐ 8

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Reactive Web

WEB

Build reactive web applications with Spring WebFlux and Netty.

Spring for Apache Kafka

MESSAGING

Publish, subscribe, store, and process streams of records.



Fichiers générés par l'assistant

- *.gitignore, Help.md*
- Scripts de build (*mvnw* ou *gradlew*)
- Classe de démarrage de SpringBoot (*src/main/java*)
- Classe de test de la configuration (*src/test/java*)
- Fichier de configuration des propriétés (*src/main/resources*)



Exemple Gradle

```
plugins {  
    id 'java'  
    id 'org.springframework.boot' version '3.1.2'  
    id 'io.spring.dependency-management' version '1.1.2'  
}  
  
group = 'org.formation'  
version = '0.0.1-SNAPSHOT'  
  
java {  
    sourceCompatibility = '17'  
}  
  
repositories { mavenCentral() }  
  
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-webflux'  
    implementation 'org.springframework.kafka:spring-kafka'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
    testImplementation 'io.projectreactor:reactor-test'  
    testImplementation 'org.springframework.kafka:spring-kafka-test'  
}  
  
tasks.named('test') { useJUnitPlatform() }
```



Plug-in Maven/Gradle de SpringBoot

L'initialiser crée des scripts (***mvnw*** ou ***gradlew***) pour les environnements Linux et Windows.

- Ce sont des wrappers de l'outil de build garantissant que tous les développeurs utilisent la même version de l'outil de build.

La commande la + importante dans un contexte Maven :

./mvnw clean package

=> Génère un *fat-jar*

L'application peut alors être démarrée en ligne de commande par :

java -jar target/myAppli.jar

Avec gradle :

`gradle build`

`java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar`



Autres tâches de build

Packager une image OCI :

spring-boot:build-image

Exécuter l'application

spring-boot:run

Exécuter avec le classpath de tes

spring-boot:test-run

Construire une image native

package -Pnative

Exécuter des tests d'intégration

spring-boot:start

spring-boot:stop

Renseigner le endpoint */actuator/info* avec les informations de build

spring-boot:build-info



Classe de démarrage

La classe de démarrage est annotée via **@SpringBootApplication**, annotation qui englobe :

- **@Configuration** : Permet de définir des méthodes *@Bean*
- **@ComponentScan** : Scan des annotations dans les sous-packages
- **@EnableAutoConfiguration** : Activation du mécanisme d'auto-configuration de SpringBoot

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        // Création du contexte Spring
        SpringApplication.run(DemoApplication.class, args);
    }

}
```



Classe de test

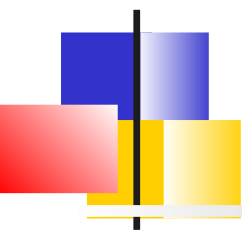
La classe de test est annotée via
@SpringBootTest :

- Permet de créer le contexte Spring avant l'exécution du test junit5

```
@SpringBootTest
class DemoApplicationTests {

    @Test
    void contextLoads() {
        // Si le test passe, la configuration SpringBoot est OK
    }

}
```



Spring coeur et Spring Boot

Rappels Spring Coeur et annotations

L'accélérateur SpringBoot

Principaux starters

Développer avec SpringBoot



Starters les + importants

Web

*-**web** : Application web ou API REST. Framework *Spring MVC*

*-**reactive-web** : Application web ou API REST en mode réactif. Framework *Spring WebFlux*

Cœurs :

*-**logging** : Utilisation de logback (Tjs présent)

*-**test** : Test avec Junit, Hamcrest et Mockito (Tjs présent)



Starters développement

- *-**devtools** : Fonctionnalités pour le développement
- *-**lombok** : Simplification du code Java
- *-**configuration-processor** : Complétion des propriétés de configuration applicatives disponibles dans l'IDE
- *-**docker-compose**¹ : Support pour démarrer les services de support via docker-compose (BD, Kafka, etc..)
- *-**graalvm-native**¹ : Support pour construire des images natives
- *-**modulith**¹ : Support pour construire des applications monolithes modulaires



Sécurité

*-**security** : *Spring Security*, sécurisation des URLs et des services métier

*-**oauth2-client** : Pour obtenir un jeton *oAuth* d'un serveur d'autorisation

*-**oauth2-resource-server** : Sécurisation des URLs via *oAuth*

*-**ldap** : Intégration LDAP

*-**okta** : Intégration avec le serveur d'autorisation Okta



Starters SQL

jdbc : JDBC avec pool de connexions Tomcat

Spring Data JPA: Spring Data avec Hibernate et JPA

Spring Data JDBC : Spring Data avec jdbc

Spring Data R2DBC : Spring Data avec *jdbc reactif*

MyBatis : Framework MyBatis

LiquiBase Migration : Migration de schéma avec Liquibase

Flyway Migration : Migration de schéma avec Flyway

JOOQ Access Layer : API fluent pour construire des requêtes SQL

*-***<drivers>*** : Accès aux driver JDBC (MySQL, Postgres, H2, HyperSonic, DB2)



Starters NOSQL

- *-**data-cassandra**, *-**data-reactive-cassandra**: Base distribuée Cassandra
- *-**data-neo4j** : Base de données orienté graphe de Neo4j
- *-**data-couchbase** *-**data-reactive-couchbase** : Base NoSQL CouchBase
- *-**data-redis** *-**data-reactive-redis** : Base NoSQL Redis
- *-**data-geode** : Stockage de données via Geode
- *-**data-elasticsearch** : Base documentaire indexée ElasticSearch
- *-**data-solr**: Base indexée SolR
- *-**data-mongodb** *-**data-reactive-mongodb** : Base NoSQL MongoDB



Messaging

- *-**integration**: Spring Integration (Abstraction de + haut niveau pour implémenter des patterns d'intégration de façon déclaratif)
- *-**kafka**: Intégration avec Apache Kafka
- *-**kafka-stream**: Intégration avec l'API KafkaStream
- *-**rabbitmq**: Intégration avec Rabbit MQ
- *-**activemq5** : ActiveMQ avec JMS
- *-**artemis** : ApacheMQ avec Artemis
- *-**pulsar**, *-**reactive-pulsar** : Messagerie PULSAR
- *-**websocket** : Servlet avec STOMP et SockJS
- *-**rsocket** : SpringMessaging et Netty
- *-**camel** : Intégration avec Apache Camel
- *-**solacePubSub** : Intégration avec Solace



Autres Starters Web

Moteur de templates HTML

- *-**thymeleaf** : *Spring MVC avec des vues Thymeleaf*
- *-**mustache** : *Spring MVC avec Mustache*
- *-**groovy-templates** : *Spring MVC avec gabarits Groovy*
- *-**freemarker** : *Spring MVC avec freemarker*

Autres

- *-**graphql** : *API GraphQL*
- *-**rest-repository, restrepository-explorer, *-hateoas** :
Génération API Rest à partir des repositories de Spring Data
- *-**jersey** : *API Restful avec JAX-RS et Jersey*
- *-**webservices** : *Services SOAP*
- *-**vaadin, *-hila** : *Framework pour applis web*



Autres Starters

I/O

- *-**batch** : Gestion de batchs
- *-**mail** : Envois de mails
- *-**cache** : Support pour un cache
- *-**quartz** : Intégration avec Scheduler
- *-**shell** : Support pour commandes en ligne

Ops

- *-**actuator** : Points de surveillance REST ou JMX
- *-**spring-boot-admin (client et serveur)** : UI au dessus d'actuator
- *-**sentry** : Intégration sentry (monitoring performance)



Spring Cloud

Services cloud : Facilité de déploiement

Amazon, Google Cloud, Azure, Cloud Foundry, Alibaba

Architecture Micro-services

Services de discovery, de configuration externalisée,
de répartition de charge, de gateway, de circuit
breaker

Spring Cloud Contract : Génération de tests et mock
servers

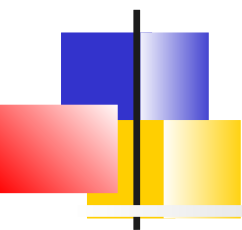
Service de monitoring, de tracing, etc ...



Observabilité

Depuis la version 3.x, SpringBoot s'appuie fortement sur ***Micrometer***.

- Des starters permettent de publier les métriques *micrometer* vers des système de visualisation :
DataDog, Dynatrace, Influx, Graphite, New Relic, Prometheus, Wavefront
- D'autres starters permettent la tracabilité des requêtes dans les architecture micro-services :
Brave et Zipkin



Spring coeur et Spring Boot

Rappels Spring Coeur et annotations

L'accélérateur SpringBoot

Principaux starters

Développer avec SpringBoot

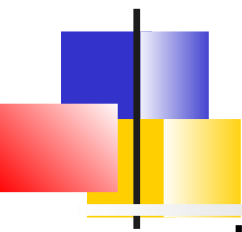


IDEs

Pivotal propose ***SpringToolsSuite***
disponible pour

- VSCode
- Eclipse
- Theia

Pour les autres IDEs, des tierces parties
proposent des plugins qui offrent un
support équivalent



Apport de SpringTools Suite

Les principaux apports sont :

- Assistant de création de projet qui se connecte sur *Spring Initializr*
- Édition du *pom.xml* pour l'ajout de starter
- Assistant d'upgrade de version ou de refactoring
- *Boot Dashboard* : Petite fenêtre permettant de facilement démarrer les applications
- Éditeur de fichier de propriétés (.properties ou yaml) avec auto-complétion
- Éditeur des configuration de démarrage permettant d'activer un profil, le debug ou positionner des propriétés de configuration



Structure typique projet

com

+ - example

+ - myproject

+ - Application.java

+ - OneConfig.java

|

+ - domain

| + - Customer.java

| + - CustomerRepository.java

|

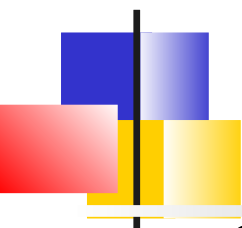
+ - service

| + - CustomerService.java

|

+ - web

+ - CustomerController.java



Propriétés de configuration

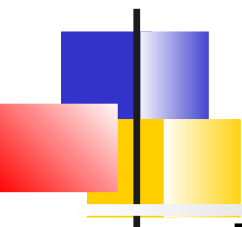
Spring Boot permet d'externaliser la configuration des beans :

- Ex : Externaliser l'adresse de la BD, la configuration d'un client, ...

On peut utiliser des fichiers ***properties*** ou ***YAML***, des variables d'environnement ou des arguments de commande en ligne.

Les valeurs des propriétés sont ensuite injectées dans les beans :

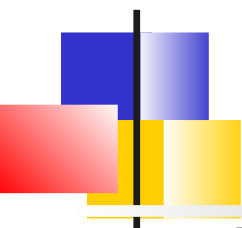
- Directement via l'annotation ***@Value***
- Ou associer à un objet structuré via l'annotation ***@ConfigurationProperties***



Priorités

De nombreux niveaux de propriétés différents mais en résumé l'ordre des propriétés est :

1. *spring-boot-devtools.properties* si *devtools* est activé (SpringBoot)
2. Les propriétés de test
3. **La ligne de commande. Ex : `--server.port=9000`**
4. Environnement REST, Servlet, JNDI, JVM
5. **Variables d'environnement de l'OS**
6. Propriétés avec des valeurs aléatoires
7. **Propriétés spécifiques à un profil**
8. ***application.properties* , *yaml***
9. Annotation `@PropertySource` dans la configuration
10. Les propriétés par défaut spécifié par *SpringApplication.setDefaultProperties*



application.properties (.yml)

Les fichiers de propriétés (***application.properties/.yml***) sont généralement placés dans les emplacements suivants :

- Un sous-répertoire *config*
- Le répertoire courant
- Un package *config* dans le classpath
- A la racine du classpath

En respectant ces emplacements standards, SpringBoot les trouve tout seul



Valeur filtrée

Le fichiers supportent les valeurs filtrées.

```
app.name=MyApp  
app.description=${app.name} is a Boot app.
```

Les valeurs aléatoires :

```
my.secret=${random.value}  
my.number=${random.int}  
my.bignumber=${random.long}  
my.uuid=${random.uuid}
```




Injection de propriété : *@Value*

La première façon de lire une valeur configurée est d'utiliser l'annotation **@Value**.

```
@Value("${my.property}")  
private String myProperty ;
```

```
@Value("${my.property:defaultValue}")  
private String myProperty ;
```

Dans ce cas, aucune validation n'est effectuée sur la valeur de la propriété



Validation des propriétés

Il est possible de forcer la validation des propriétés de configuration à l'initialisation du conteneur.

- Utiliser une classe annotée par **@ConfigurationProperties** et **@Validated**
- Positionner des contraintes de *javax.validation* sur les attributs de la classes



Exemple

@Component

@ConfigurationProperties("app")

@Validated

```
public class MyAppProperties {
```

```
    @Pattern(regexp = "\\d{3}-\\d{3}-\\d{4}")
```

```
    private String adminContactNumber;
```

```
    @Min(1)
```

```
    private int refreshRate;
```

```
        .....
```

```
}
```

Fichier de properties correct :

app.admin-contact-number : 666-777-8888

app.refresh-rate : 5



Propriétés spécifiques à un profil

Les propriétés spécifiques à un profil (ex : intégration, production) sont spécifiées différemment en fonction du format properties ou .yaml.

- Si l'on utilise le format *.properties*, on peut fournir des fichiers complémentaires :
application-{profile}.properties
- Si l'on utilise le format *.yaml* tout peut se faire dans le même fichier



Exemple fichier *.yml*

```
server:
```

```
  address: 192.168.1.100
```

```
---
```

```
spring:
```

```
  config:
```

```
    activate:
```

```
      on-profile:
```

```
        -prod
```

```
server:
```

```
  address: 192.168.1.120
```



Activation des profils

Les profils sont activés généralement par la propriété ***spring.profiles.active*** qui peut être positionnée :

- Dans un fichier de configuration
- En commande en ligne via :
--spring.profiles.active=dev,hsqldb
- Programmatically, via :
SpringApplication.setAdditionalProfiles(...)

Plusieurs profils peuvent être activés simultanément



DevTools

Le module ***spring-boot-devtools*** est recommandé dans un environnement de développement

Cela apporte notamment :

- Ajout automatique de propriétés de configuration propre au développement.

Ex :

spring.thymeleaf.cache=false

- Rechargement du contexte automatique lors d'une modification des fichiers sources.
- ...



Gestion des traces

Spring utilise *Common Logging* en interne mais permet de choisir son implémentation

Des configurations sont fournies pour :

- Java Util Logging
- Log4j2
- Logback (défaut)



Format des traces

Une ligne contient les informations suivantes :

- Timestamp à la ms
- Niveau de trace : ERROR, WARN, INFO, DEBUG or TRACE.
- Process ID
- Un séparateur --- .
- Nom de la thread entouré de [].
- Le nom du Logger <=> Nom de la classe .
- Un message
- Une note entouré par des []



Configurer les traces via Spring

Par défaut, Spring affiche les messages de niveau ERROR, WARN et INFO sur la console

- *java -jar myapp.jar -debug* : Active les messages de debug
- Changer les niveaux de logs au niveau des logger via *application.properties/yml*
logging.level.root=WARN
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
- Propriétés **logging.file** et **logging.path** pour générer un fichier de log



Extensions Logback

Spring propose des extensions à Logback permettant des configurations avancées.

Il faut dans ce cas utiliser le fichier ***logback-spring.xml***

- Multi-configuration en fonction de profil
- Utilisation de propriétés dans la configuration



Rapport d'auto-configuration

Lors de l'activation du mode DEBUG, SpringBoot affiche sur la console un rapport d'auto-configuration divisé en 2 parties

- **Positive Matches** : Liste les configurations par défaut qui ont été appliquées et pourquoi elles ont été appliquées
- **Negative Matches** : Liste les configurations par défaut qui n'ont pas été appliquées et pourquoi elles n'ont pas été appliquées



Interactions entre services Spring

Styles d'interaction et APIs

RestFul avec Spring

Messaging avec Spring



Introduction

Différentes technologies pour la communication entre services :

- Modèles requête/réponse *RESTful*, *gRPC*, *GraphQL* : Synchrones ou Réactifs
- Communications via message : *Point2Point* ou *PubSub*

Différents formats :

- Texte : JSON, XML
- Binaires : Avro, Protocol Buffer



Rôles de l'API

Les APIs/interfaces sont au cœur du développement logiciel.

- Une interface/API bien conçue expose des fonctionnalités utiles tout en masquant la mise en œuvre.
- L'interface/API est un contrat entre le service et ses clients

Avec un langage typé, si l'interface change, le projet ne compile plus et l'on règle le problème.
=> Il faut avoir le même mécanisme dans le cadre des micro-services



Design By Contract

Il est important de définir précisément l'API d'un service en utilisant une sorte de langage de définition d'interface (IDL).

La définition de l'API dépend du style d'interaction :

- *OpenAPI* pour Rest et ProtoBuf, Schéma *GraphQL*
- Nom des canaux de messages, format et type du message mais pas de standard disponible



Offre Spring

Spring Cloud Contract est un projet qui permet d'adopter une approche *Consumer Driven Contract*

Il supporte les différents modes d'interaction qui sont décrit en YAML, Groovy ou Java

A partir d'une spécification d'interaction entre 2 services, cela permet

- De générer des tests d'acceptation côté producteur ou serveur
- De créer des mocks serveur pour les tests côté consommateur ou client



Evolution de l'API

Avec les micro-services, changer l'API d'un service est beaucoup plus difficile.

Les clients étant développés par d'autres équipes.

- => Il est impossible de forcer tous les clients à se mettre à niveau en même temps.
- => Les anciennes et les nouvelles versions d'un service devront s'exécuter simultanément.



Compatibilité d'API

Différents types de compatibilité d'API existent :

- **Descendante/backward/rétrocompatible** : interaction possible avec un système plus vieux
- **Ascendante** : interaction possible avec des systèmes plus récents

Des outils comme *Schema Registry* dans l'écosystème Kafka permettent de déterminer les compatibilités d'API lors d'évolution¹

1. <https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html>



Versionning

Semvers¹ définit le numéro de version composé de trois parties : **MAJOR.MINOR.PATCH** :

- **MAJEUR** : Une modification incompatible à l'API
- **MINEUR** : Améliorations rétrocompatibles à l'API
- **PATCH** : Correction de bogue rétrocompatible

Les changements de version doivent être contrôlés.

=> Le n° de version est alors présent dans l'interaction (URL, Nom du canal, Entête de la requête ou du message)

1. <https://semver.org/lang/fr/>



Gestion des versions majeures

Bonnes pratiques pour la gestion des versions majeures des API :

- Seules 2 versions majeures d'une API peuvent être simultanément en production (versions X = nominale et X-1 = dépréciée)
 - Les évolutions sont réalisées sur la version X
 - Les correctifs peuvent être réalisés sur la version X-1 (et reportés sur la version X)
- La montée de version majeure d'une API ne se justifie que dans le cas d'évolutions non rétrocompatibles
- La publication d'une version X entraîne la dépréciation de la version X-1 si elle existe et le dé-commissionnement de la version X-2 si elle existe



Gestion des versions mineures et correctifs

Dans l'environnement de production :

- Une version majeure X est dans une seule version mineure X.Y
- Une version mineure X.(Y+1) entraîne la suppression de l'ancienne version mineure

Dans l'environnement de qualification :

- Une version majeure X peut être dans 2 versions mineures
- Une version X.Y identique à la version de Production
- Une version X.(Y+1) en cours de développement ou de qualification



Interactions entre services Spring

Styles d'interaction et APIs

RestFul avec Spring

Messaging avec Spring



Introduction

Lors de l'utilisation d'un mécanisme RPC (*Remote Procedure Call*),

- un client envoie une demande à un service,
- le service traite la demande et renvoie une réponse.

L'interaction peut être :

- bloquante : le client attend la réponse
- Réactive ou non bloquante : le client continue son exécution et est prévenu lorsque la réponse arrive

Mais contrairement à l'utilisation d'une messagerie, le client suppose que la réponse arrivera en un temps court.

L'interaction nécessite que le client ET le serveur soit UP en même temps



Avantages de REST

C'est simple et familier.

Un format de description standard existe : *OpenAPI*

Facile à tester, navigateur avec Postman, curl, Swagger

HTTP est compatible avec les pare-feu.

Ne nécessite pas de broker intermédiaire, ce qui simplifie l'architecture.



Inconvénients

Un seul type de communication

Il ne prend en charge que le style demande/réponse.

Les clients doivent connaître les emplacements (URL) des serveurs

Service de discovery nécessaire.

Disponibilité réduite.

Le client et le service doivent tous les deux fonctionner pendant toute la durée de l'échange.

Granularité fine

La récupération de plusieurs ressources en une seule requête est un défi.

Peu de verbes HTTP

Il est parfois difficile de mapper plusieurs opérations de mise à jour sur des verbes HTTP.



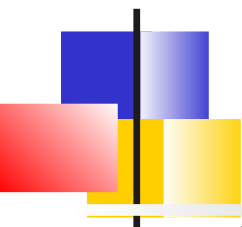
Starters Spring

2 starters sont possibles pour implémenter une API REST :

- **Spring MVC** : *spring-boot-starter-web*
- **Spring WebFlux** : *spring-boot-starter-webflux*

Ils permettent de déclarer des beans de type

- **@RestController**
- Dont les méthodes peuvent être associées à des requêtes HTTP via **@RequestMapping**



Spring WebFlux et Reactor

Spring Webflux et tous les projets réactifs de Spring s'appuient sur une implémentation de *ReactiveStream* : **Reactor**

Reactor apporte principalement :

- Les types **Mono** et **Flux** représentant des flux de 1 ou un nombre infini d'événement
- Un ensemble **d'opérateurs** réactifs permettant la manipulation des flux (transformation, filtrage, agrégation, jointure, etc..)



Exemple SpringMVC

```
@RestController
@RequestMapping(value="/users")
public class UsersController {

    @GetMapping(value="/{id}")
    public User getUser(@PathVariable Long id) {
        // ...
    }

    @GetMapping(value="/{user}/customers")
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @DeleteMapping(value="/{user}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long user) {
        // ...
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }

    @PostMapping
    public ResponseEntity<User> register(@Valid @RequestBody User user) {

        user = userRepository.save(user);

        return new ResponseEntity<>(user, HttpStatus.CREATED);
    }
}
```



Exemple SpringWebFlux

```
@RestController
@RequestMapping(value="/users")
public class UsersController {

    @GetMapping(value="/{id}")
    public Mono<User> getUser(@PathVariable Long id) {
        // ...
    }

    @GetMapping(value="/{user}/customers")
    Flux<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @DeleteMapping(value="/{user}")
    public ResponseEntity<Mono<Void>> deleteUser(@PathVariable Long user) {
        // ...
        return new ResponseEntity<Mono<Void>>(HttpStatus.NO_CONTENT);
    }

    @PostMapping
    public ResponseEntity<Mono<User>> register(@Valid @RequestBody User user) {

        return new ResponseEntity<Mono<User>>(userRepository.save(user), HttpStatus.CREATED);
    }
}
```



Sérialisation JSON

Par défaut, les entrées/sorties des *RestController* utilisent le format JSON.

La librairie responsable des conversion JSON ↔ Objet Java est **Jackson**

Plusieurs alternatives existent pour adapter la sérialisation par défaut de Jackson à ses besoins :

- Créer des classes DTO spécifiques.
- Utiliser les annotations de Jackson sur les classes du domaine
- Permettent différentes sérialisation du même objet avec l'annotation *@JsonView*
- Implémenter ses propres Sérialiseur/Désérialiseur.
Spring propose l'annotation *@JsonComponent*



Gestion des erreurs

Lorsqu'une exception est levée durant le traitement de la requête, il faut générer une réponse REST adéquate (Code réponse 400, 404, ou autre)

Par défaut, Spring MVC/Webflux gère les erreurs de validation d'entrée en renvoyant un code 400, il renvoie un code 500 au format HTML pour les autres exceptions

Pour remplacer le comportement par défaut Spring offre plusieurs alternatives :

- **@ResponseStatus** sur l'exception métier permet d'associer un code d'erreur HTTP à l'exception
- Catcher l'exception et relancer une **ResponseStatusException**
- Les classes **@RestController** et **@RestControllerAdvice** peuvent avoir des méthodes annotées par **@ExceptionHandler**
Ces méthodes sont responsable de générer la réponse en cas de déclenchement de l'exception.



Exemple

@ResponseStatus(value = HttpStatus.NOT_FOUND)

```
public class MyResourceNotFoundException extends RuntimeException {  
    public MyResourceNotFoundException() {  
        super();  
    }  
    public MyResourceNotFoundException(String message, Throwable cause) {  
        super(message, cause);  
    }  
    public MyResourceNotFoundException(String message) {  
        super(message);  
    }  
    public MyResourceNotFoundException(Throwable cause) {  
        super(cause);  
    }  
}
```



ResponseStatusException

```
@GetMapping(value =("/{id}")
public Foo findById(@PathVariable("id") Long id, HttpServletResponse response)
{
    try {
        Foo resourceById = RestPreconditions.checkFound(service.findOne(id));

        eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this,
response));
        return resourceById;
    }
    catch (MyResourceNotFoundException exc) {
        throw new ResponseStatusException(
            HttpStatus.NOT_FOUND, "Foo Not Found", exc);
    }
}
```



Exemple *@ControllerAdvice*

@RestControllerAdvice

```
public class NotFoundAdvice extends ResponseEntityExceptionHandler {
```

```
    @ExceptionHandler(value = {MemberNotFound.class, DocumentNotFound.class})
```

```
    ResponseEntity<Object> handleNotFoundException(HttpServletRequest request,
        Throwable ex) {
        return new ResponseEntity<Object>(
            "Entity was not found", new HttpHeaders(), HttpStatus.NOT_FOUND);
    }
```

@Override

```
    protected ResponseEntity<Object>
        handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<Object>(
            ex.getMessage(), new HttpHeaders(), HttpStatus.BAD_REQUEST);
    }
}
```



OpenAPI

SpringDoc permet de générer la documentation d'une API REST au format OpenAPI et s'intègre avec Swagger-UI

Il suffit de placer la bonne dépendance dans le fichier de build

Par exemple, pour SpringBoot 3.x et Spring MVC

```
org.springdoc : springdoc-openapi-starter-webmvc-ui :2.1.0
```

Par défaut,

- La description OpenAPI est disponible à :
<http://localhost:8080/v3/api-docs/>
- L'interface Swagger à :
<http://localhost:8080/swagger-ui.html>



Clients REST

Le support pour les clients REST est différent en fonction de Spring MVC ou Webflux

- Spring MVC : ***RestTemplate***
- Spring WebFlux : ***WebClient*** qui supporte les interactions bloquantes et non bloquantes



Exemple RestTemplate

```
@Service
public class MyBean {
    private final RestTemplate restTemplate;

    public MyBean(RestTemplateBuilder restTemplateBuilder) {
        this.restTemplate = restTemplateBuilder.rootUri("
            http://localhost:8080/api")
            .basicAuthentication("user", "password")
            .build();
    }

    public Details someRestCall(String name) {
        return this.restTemplate.getForObject("/{name}/details",
            Details.class,
            name);
    }
}
```



Exemple WebClient

// Spécification de la rootUri, des entêtes, etc.

```
WebClient client = WebClient.builder()
    .baseUrl("http://localhost:8080")
    .defaultHeader(HttpHeaders.CONTENT_TYPE,
        MediaType.APPLICATION_JSON_VALUE)
    .build();
```

// Echange avec le back-end

```
Mono<MyDto> myDto = client.post()
    .uri("/resource")
    .bodyValue(jsonData)
    .retrieve()
    .bodyToMono(DTO.class);
```



Interactions entre services Spring

Styles d'interaction et APIs
RestFul avec Spring
Messaging avec Spring



Introduction

Les communications asynchrones entre services apportent plusieurs avantages :

- Découplage du producteur et consommateur de message
- Scaling et montée en charge en scalant les consommateurs
- Implémentation de patterns micro-services comme Saga¹, ou Event-sourcing Pattern²

Des difficultés :

- Gestion de l'asynchronisme
- Mise en place et exploitation d'un message broker

1. <https://microservices.io/patterns/data/saga.html>

2. <http://microservices.io/patterns/data/event-sourcing.html>



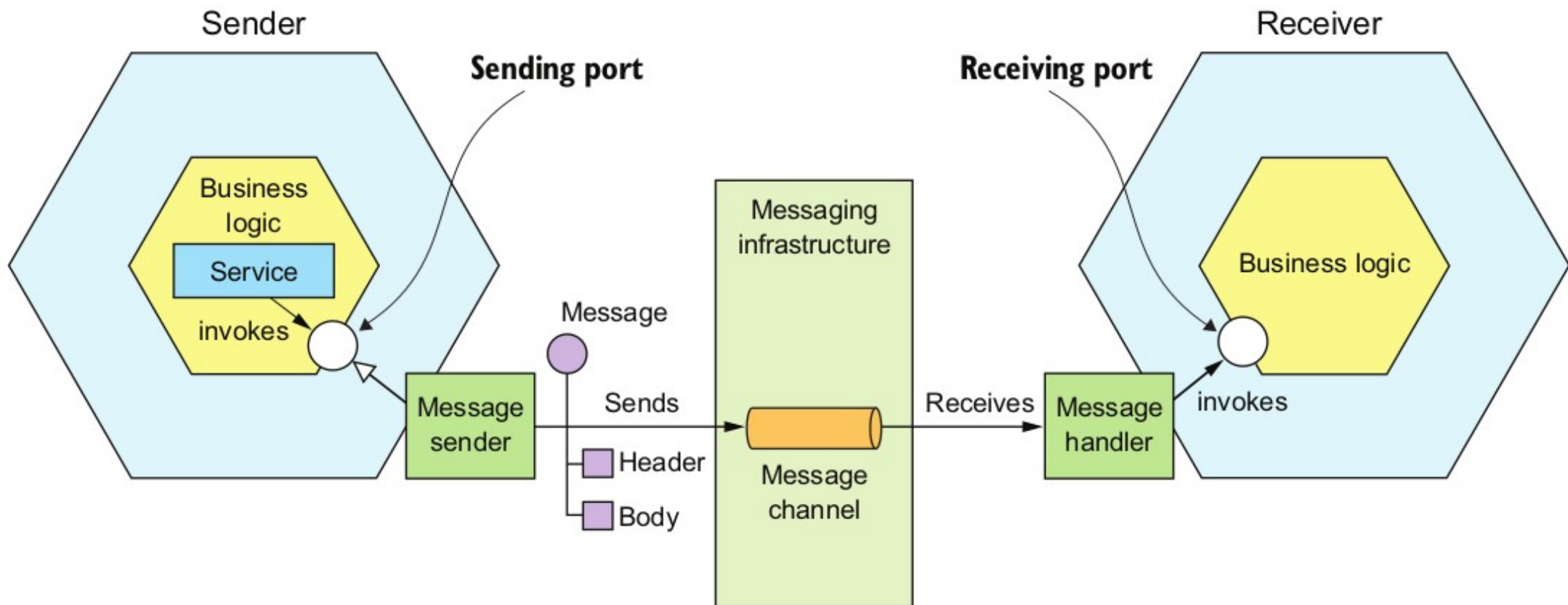
Messaging Pattern

Messaging Pattern¹ : Un client invoque un service en utilisant une messagerie asynchrone

- Le pattern fait intervenir un message broker
- Un client effectue une requête en postant un message asynchrone
- Optionnellement, il s'attend à recevoir une réponse

1. <http://microservices.io/patterns/communication-style/messaging.html>

Architecture

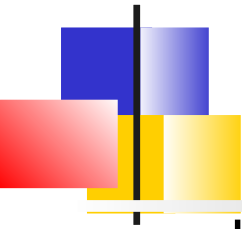




Services d'un broker

Le fait de disposer d'un broker permet en général de nombreux autres patterns distribués car un broker permet un large éventail d'interactions :

- Sémantique du message : Document, Requête ou événement
- Canaux : Point 2 point ou PubSub
- Styles d'interactions : Requête/réponse synchrone ou asynchrone, *Fire and Forget*, Publication vers abonnés avec ou sans acquittement
- Services : Bufferisation, Durabilité, Garantie de livraison et ordre des messages, Scalabilité, Tolérance aux fautes etc.



Architecture event-driven

L'utilisation de message broker influe sur les architectures.

Dans les architecture **event-driven**, très évolutive et très scalable, le message broker a une importance primordiale.

- Les services publient systématiquement leurs changement d'état vers les topics d'un broker.
- Ils s'abonnent également aux topics qui les intéressent afin de réagir ou traiter les événement des autres services.

Les séquences d'événements peuvent être vus comme des flux sur lesquels des pipelines de transformation doivent être appliquées.

- Les librairies réactives sont adaptées à cette problématique.
- Les contraintes BigData, temps-réel nécessitent des architecture en cluster pour les brokers



Messaging transactionnel

L'envoi d'un message requête fait souvent partie d'une transaction mettant à jour une base locale.

L'envoi du message doit faire partie de la transaction

- La solution legacy des transactions distribuées¹ ne sont pas adaptées dans ce cas
- Une solution est d'appliquer le ***Transactional Outbox Pattern***².

1. Exemple de JTA par exemple

2. <https://microservices.io/patterns/data/transactional-outbox.html>



Offre Spring

Starter messaging pur :

- *RabbitMQ, ActiveMQ, Kafka, ActiveMQ Artemis, Solace PubSub*

Pipeline de traitement d'évènements :

- Kafka Stream

Architecture micro-services *event-driven*

- Spring Cloud Stream
- Spring Data Flow



Exemple *spring-kafka*

Envoi de message

```
@Value("${app.my-channel}")
String PAYMENT_REQUEST_CHANNEL;

@Autowired
KafkaTemplate<Long, DomainEvent> kafkaOrderTemplate;

public Order doService(Domain model) {
    ...
    DomainEvent event = new DomainEvent(model);
    kafkaOrderTemplate.send(ORDER_STATUS_CHANNEL, event);
    ...
}
```

Réception de message :

```
@KafkaListener(topics = "#{ '${app.my-channel}' }", id = "oneHandler")
public void handleEvent(DomainEvent domainEvent) {
    ...
}
```




Présentation Kafka

Le projet Kafka

Cas d'usage

Concepts cœur de Kafka



Origine

Initié par *LinkedIn*, mis en OpenSource en 2011

Écrit en *Scala* et *Java*

Au départ, un message broker gérant une file de messages

A évolué pour devenir une plate-forme de streaming d'événements temps-réel

Organisé en cluster, taillé pour le BigData, il est basé sur l'abstraction « d'un journal de commit distribué »

Maintenu par *Confluent* depuis 2014



Objectifs

- Découpler producteurs et consommateurs de messages
- Persister les messages afin qu'ils puissent être consommés par de nombreux consommateurs, (éventuellement à posteriori)
- Atteindre de très haut débit et une latence faible
- Scaling horizontal flexible
- Offrir des garanties de fiabilité de la livraison de messages, malgré des défaillances !



Fonctionnalités

Kafka a trois capacités clés:

- Publier et s'abonner à des flux de messages¹ avec certaines garanties de fiabilité.
- Stocker les flux de messages de manière durable et tolérante aux pannes.
- Traiter, transformer les flux de messages au fur et à mesure qu'ils se produisent.

1. Dans la suite des slides on utilise de façon non-différenciés les termes *message*, *événement*, *enregistrement*



Points forts

Très bonne scalabilité et flexibilité

- Gestion des abonnements multiples
- Facilité d'extension du cluster

Très bonne performance

Disponibilité et tolérance aux fautes

Rétention sur disque

Traitement distribué d'évènements

Intégration avec les autres systèmes



Confluent

Créé en 2014 par *Jay Kreps, Neha Narkhede*, et *Jun Rao*

Mainteneur principal d'Apache Kafka

Plate-forme *Confluent* :

- Une distribution de Kafka
- Fonctionnalités commerciales additionnelles
- Services Cloud

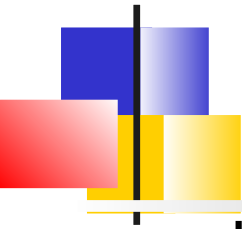


Présentation Kafka

Le projet Kafka

Cas d'usage

Concepts cœur de Kafka



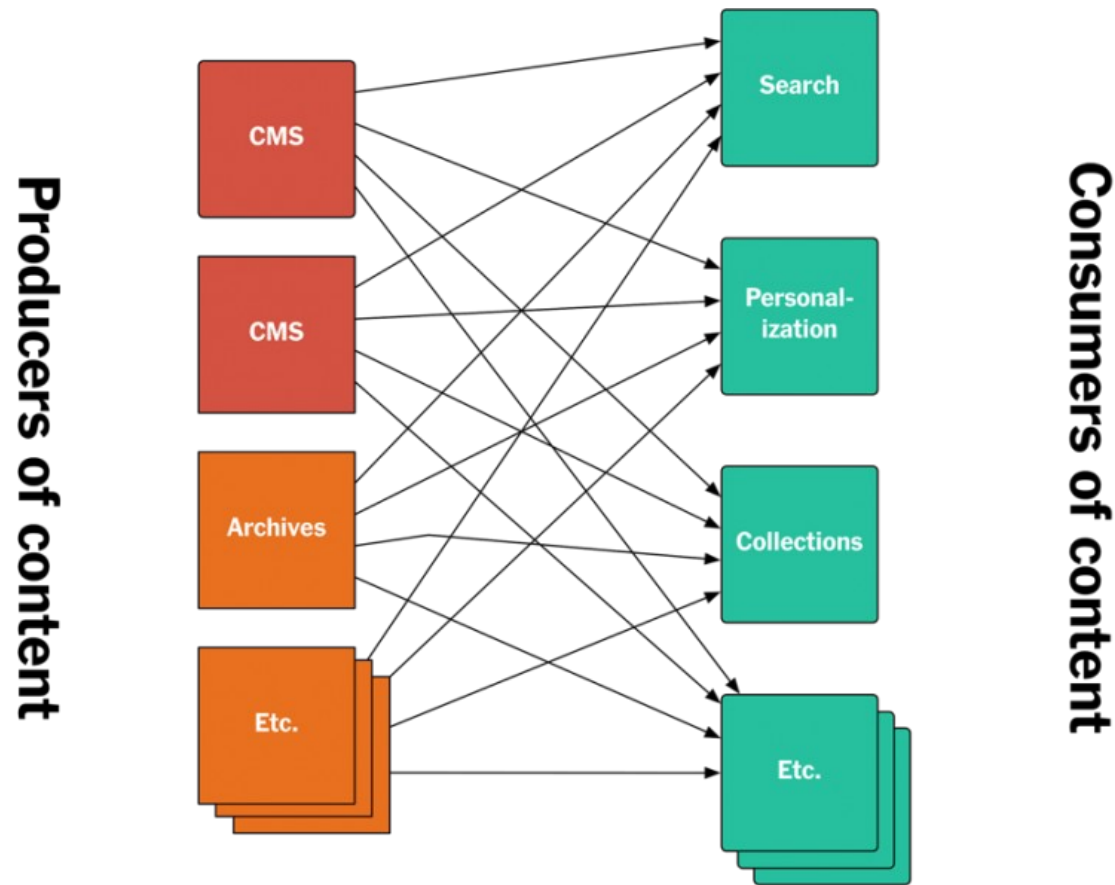
Kafka vs Message broker traditionnel

Kafka peut être utilisé comme **message broker** permettant de découpler un service producteur de services consommateurs

- Kafka n'offre que le modèle **PubSub**.
- Grâce au concept de **groupe de consommateurs**, ce modèle est scalable
- Kafka offre une **garantie plus forte** sur la livraison des messages malgré les défaillances
- Kafka ne supprime pas les messages après consommation. Ils peuvent être **consommés à posteriori**

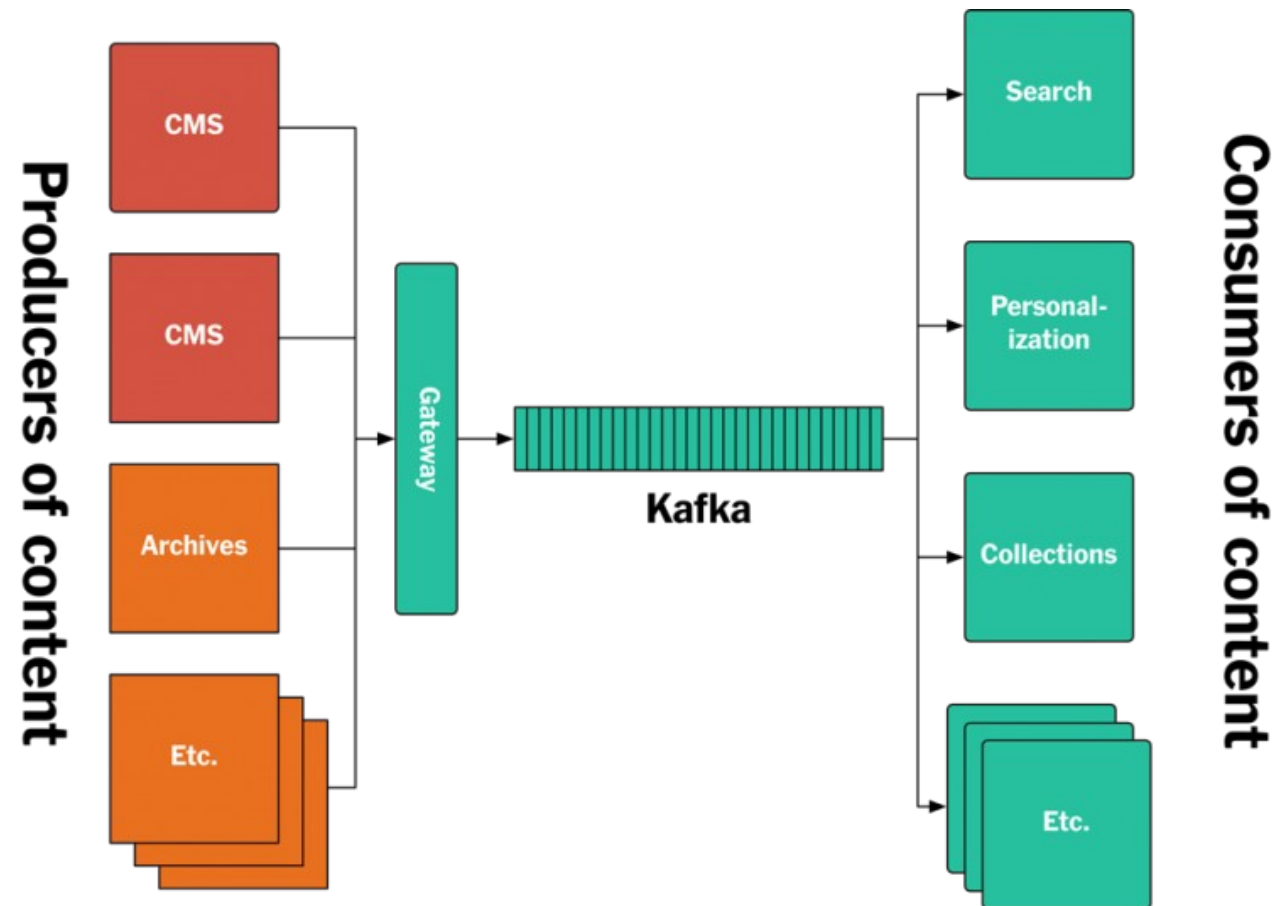
=> Idéal comme plate-forme d'intégration entre services : Architecture micro-services, ESB, EAI

Exemple ESB (New York Times) *Avant*

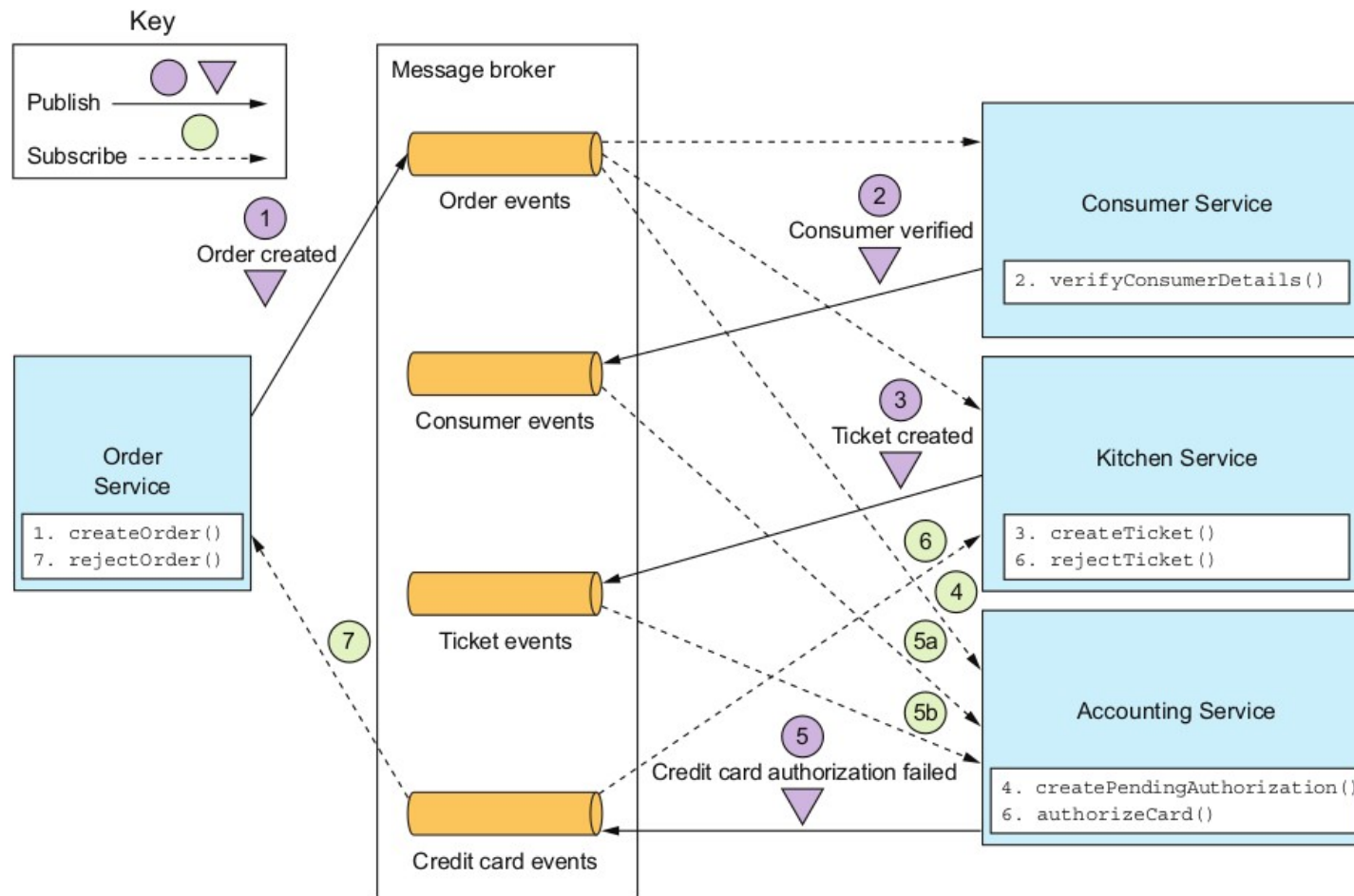


Exemple ESB (New York Times)

Après



Exemple Micro-services SAGA pattern



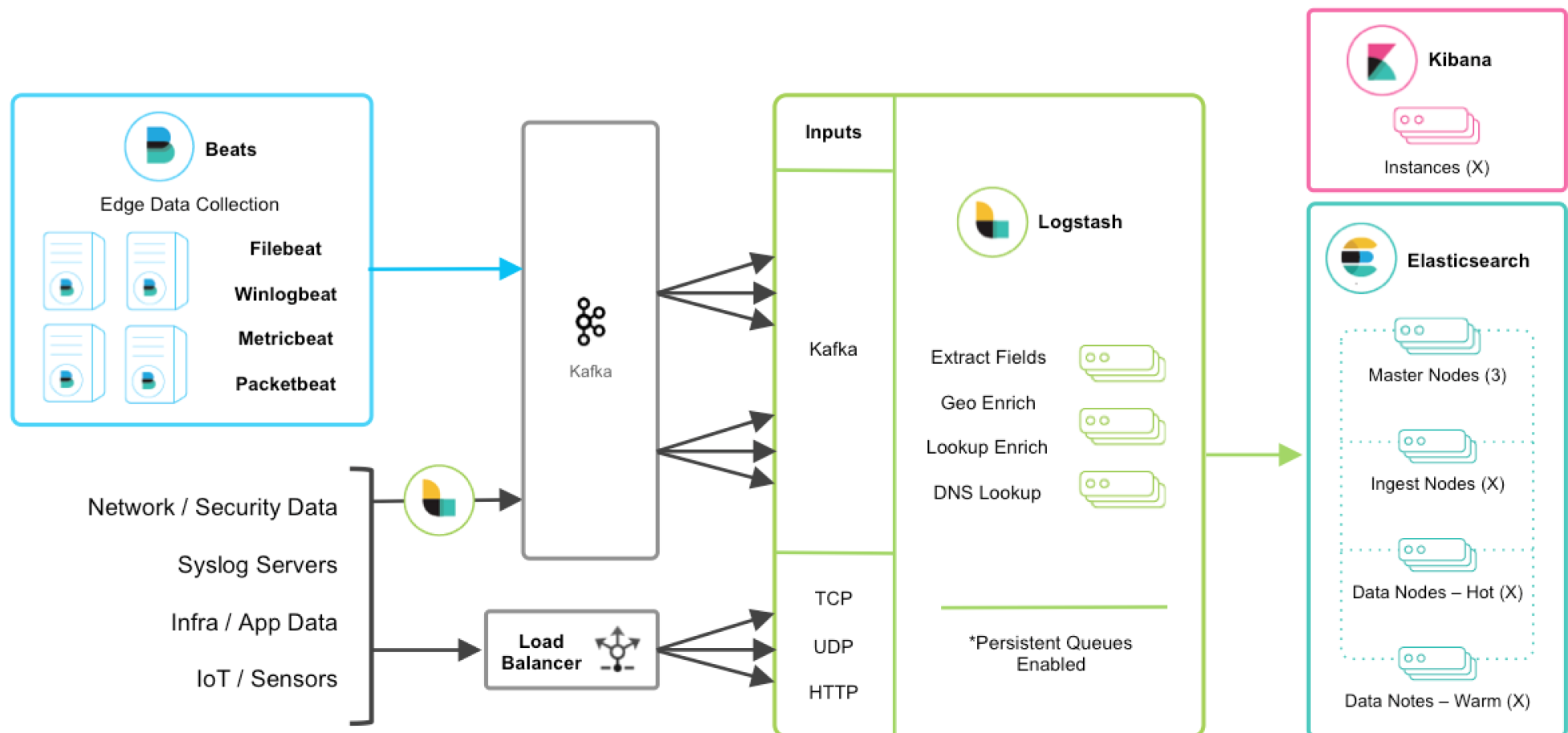


Kafka pour l'ingestion massive de données

Les *topics* Kafka peuvent être utilisés pour bufferiser les événements provenant des producteurs afin que les consommateurs puissent supporter des débits de production élevé

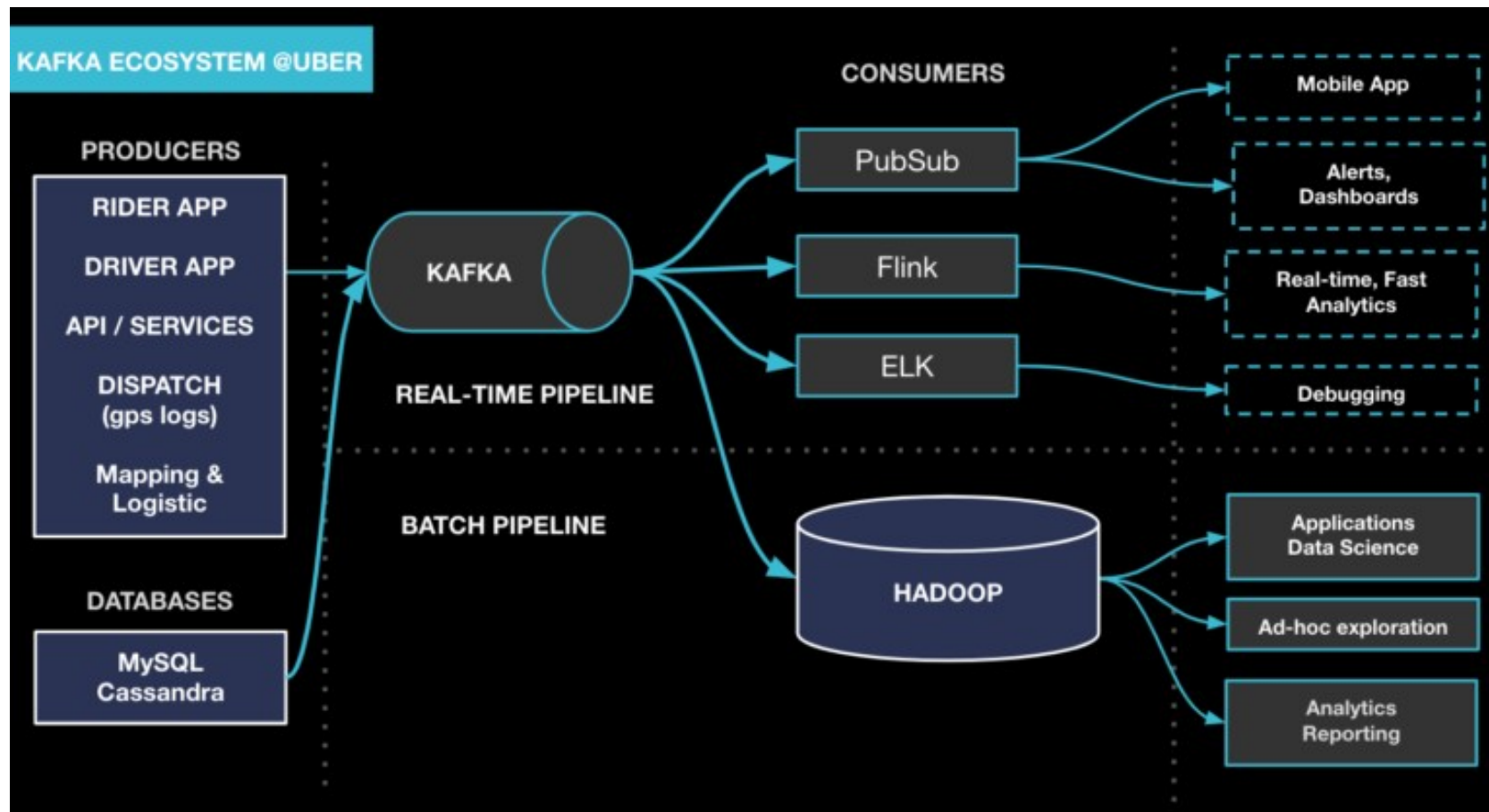
Typique des architectures BigData, d'analyse ou du monitoring temps réel.

Exemple Architecture ELK Bufferisation des événements



Ingestion massive de données

Exemple Uber





Architecture Event-driven

Les architectures *event-driven* sont une alternative intéressante dans un contexte micro-services.

- Cela produit généralement des architectures plus souples et plus réactives.

Chaque micro-service consomme en continu des événements :

- Lit un ou plusieurs topics Kafka en entrée
- Effectue un traitement
- Écrit vers un ou plusieurs topics de sortie

Kafka Stream, Spring Cloud Stream ou Spring Cloud Data Flow facilitent ce type d'architecture



Data Stream

Exemple Spring Cloud Data Flow

Streams

Create a stream using text based input or the visual editor.

Definitions Create Stream

CREATE STREAM

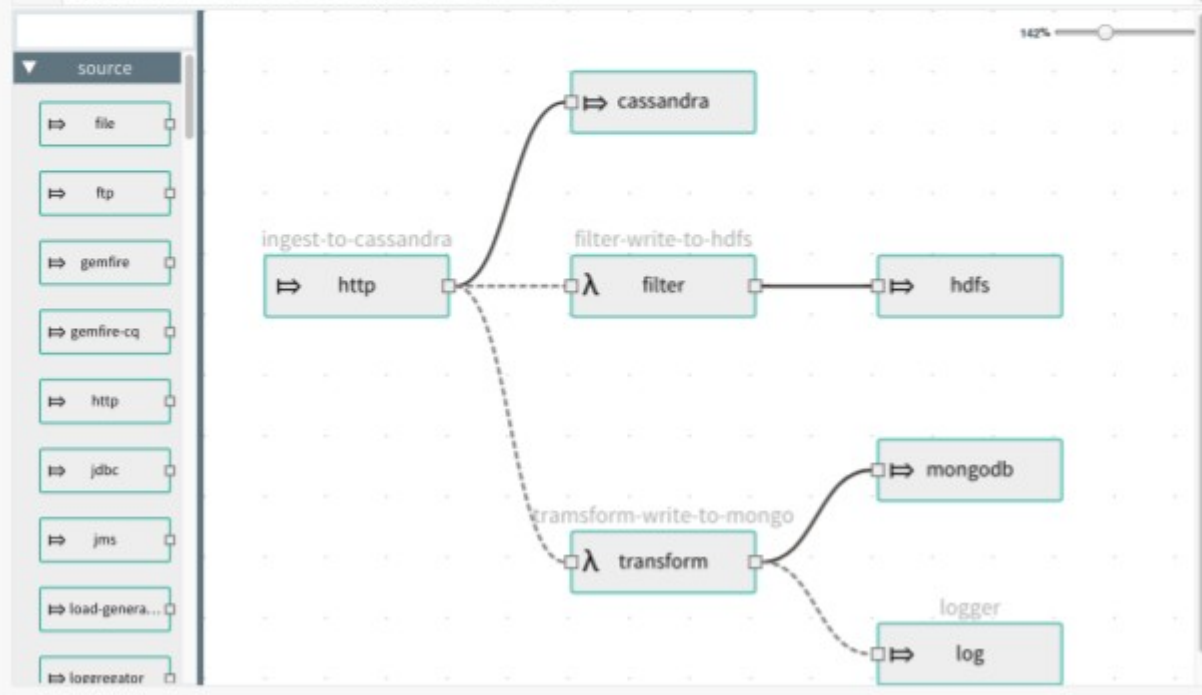
CLEAR

LAYOUT

AUTO LINK

GRID

```
1 ingest-to-cassandra=http | cassandra
2 filter-write-to-hdfs=:ingest-to-cassandra.http > filter | hdfs
3 transform-write-to-mongo=:ingest-to-cassandra.http > transform | mongodb
4 logger=:transform-write-to-mongo.transform > log
```





Kafka comme système de stockage

Les enregistrements sont écrits et répliqués sur le disque.

La structure de stockage est très scalable.

Kafka fonctionne de la même manière avec 50 Ko ou 50 To de données.

=> Kafka peut être considéré comme un système de stockage.

A la différence d'une BD, il stocke l'intégralité de l'historique des données plutôt qu'un simple instantané

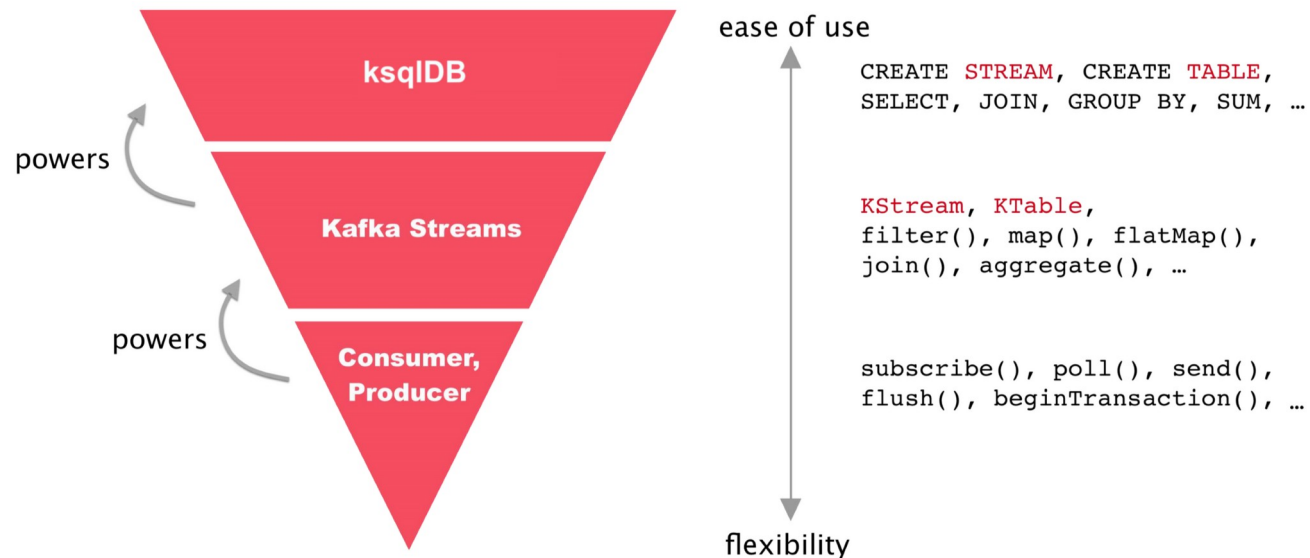
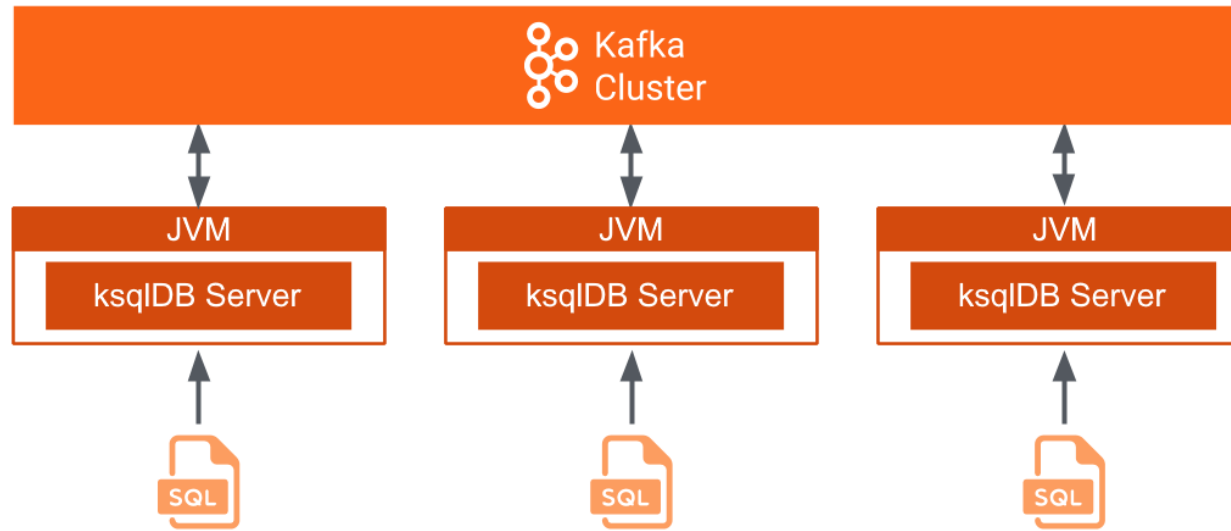
Kafka peut alors être utilisé comme *EventStore* et permet la mise en place du pattern *Event Sourcing*¹ utilisé dans les micro-services

Des abstractions sont proposées pour faciliter la manipulation de l'*EventStore* : Projet **ksqlDB**²

1. <https://microservices.io/patterns/data/event-sourcing.html>

2. <https://ksqldb.io/overview.html>

ksqlDB Standalone Application (Headless Mode)





Présentation Kafka

Le projet Kafka

Cas d'usage

Concepts cœur de Kafka



Concepts de base

Kafka s'exécute en **cluster** sur un ou plusieurs serveurs (**brokers**) pouvant être déployés dans différents data-center.

Le cluster Kafka
stocke des flux d'enregistrements : les **records**
dans des rubriques : les **topics** .

Chaque enregistrement se compose d'une clé éventuelle, d'une valeur et d'un horodatage.



Protocole Client/Serveur

Dans Kafka, la communication entre les clients et les serveurs s'effectue via un protocole TCP simple, performant et indépendant du langage.

- Ce protocole est versionné et maintient une compatibilité ascendante avec les versions plus anciennes.

Apache fournit un client Java, mais les clients sont disponibles dans de nombreuses langages.¹

1. <https://cwiki.apache.org/confluence/display/KAFKA/Clients>

Topic

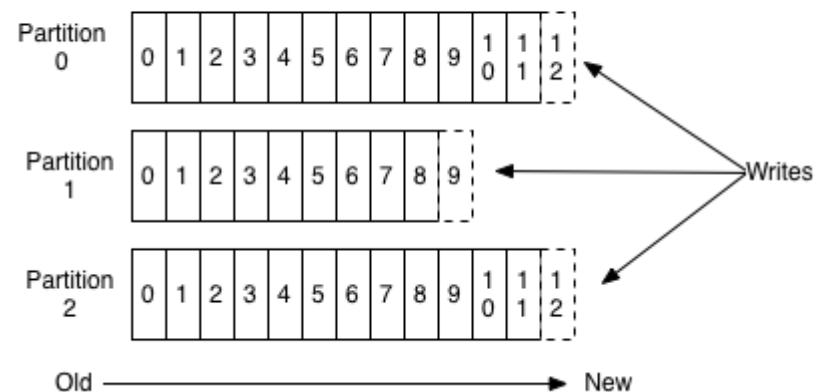
Les *records* sont publiés vers des **topics**.

Les *topics* de Kafka peuvent avoir Zéro, Un ou de multiples abonnés

Les topics peuvent être **partitionnés**.

Le cluster Kafka conserve donc un journal partitionné

Anatomy of a Topic



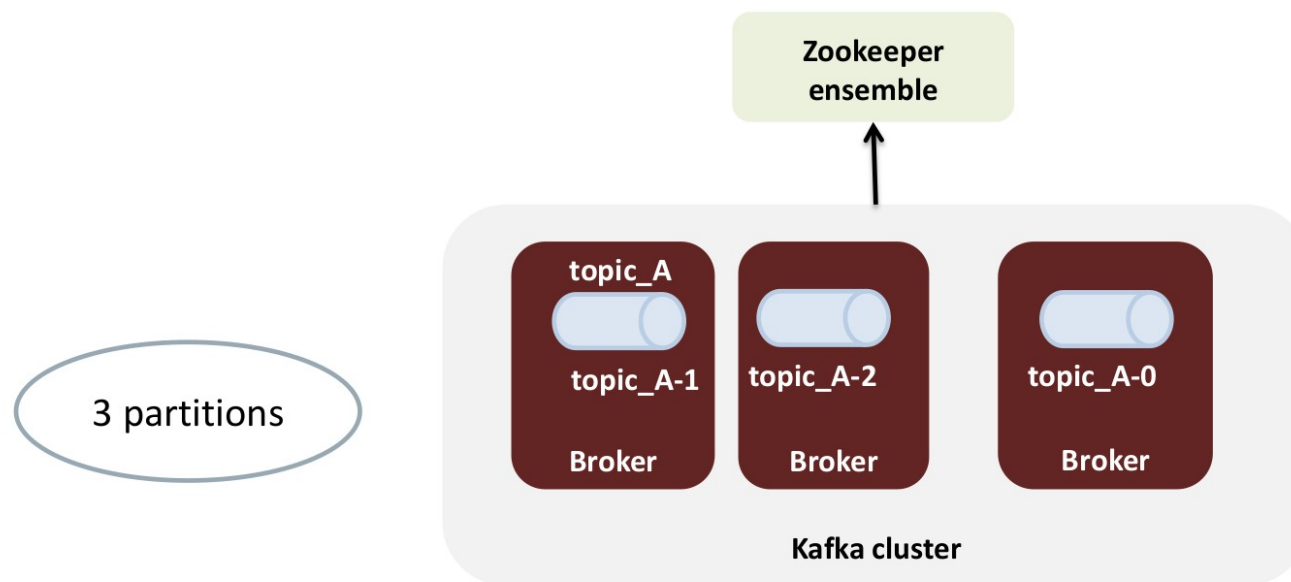


Apport des partitions

Les partitions autorisent le **parallélisme** et augmentent la **capacité de stockage** en utilisant les capacités disque de plusieurs brokers.

L'ordre des messages n'est garanti qu'à l'intérieur d'une partition

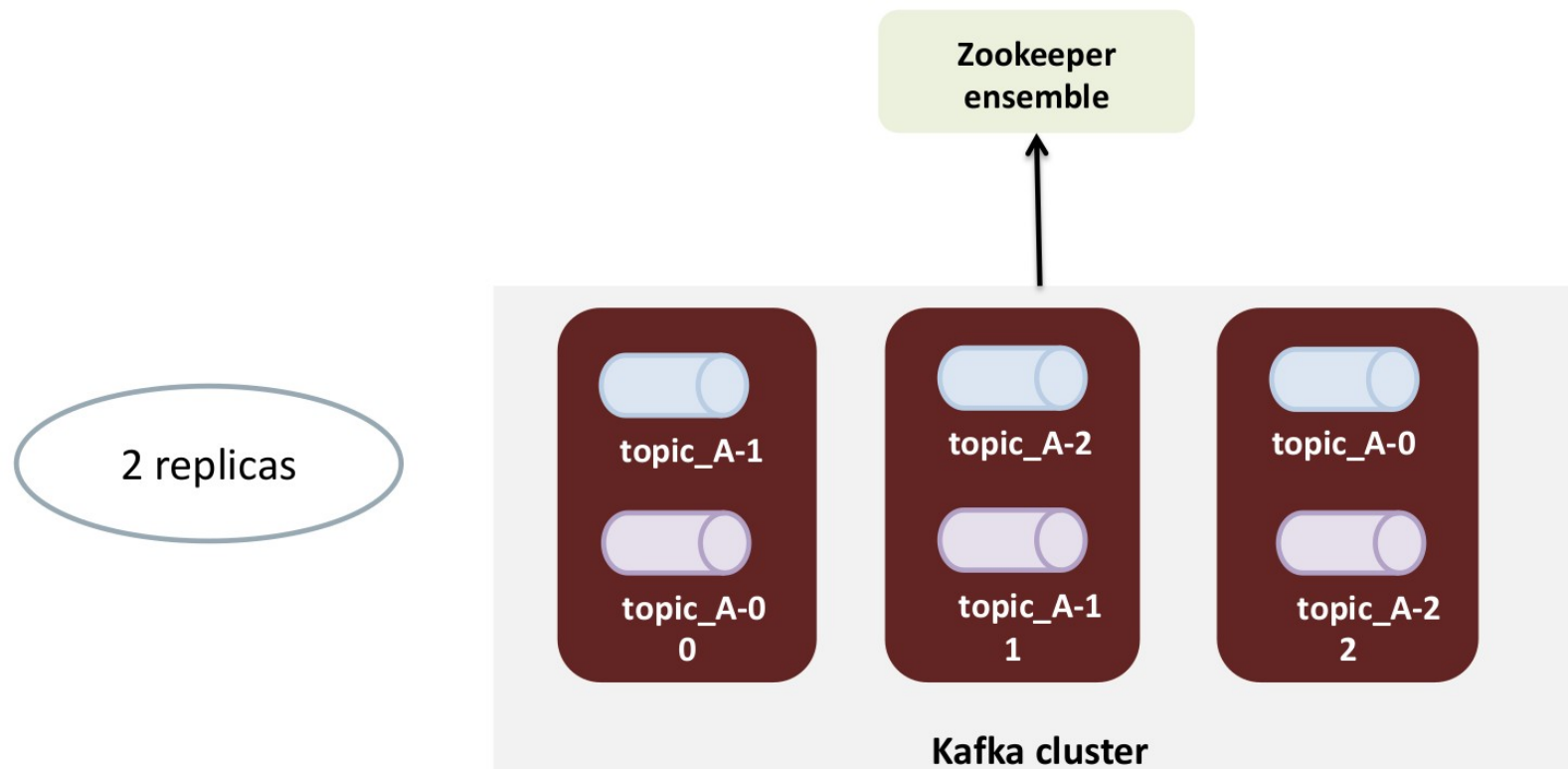
Le nombre de partition est fixé à la création du *topic*

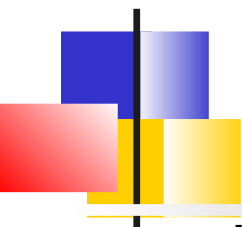


Réplication

Les partitions peuvent être **répliquées**

- La réplication permet la tolérance aux pannes et la durabilité des données





Distribution des partitions

Les partitions sont réparties sur les instances du cluster.

Les répliques sont distribuées sur des instances différentes

Pour chaque partition répliquée, une des instances agit comme **maître (leader)**. Les autres comme **suiveurs (follower)**

- Le maître coordonne les lectures et les écritures sur la partition
- Les suiveurs répliquent passivement le maître
- Si le maître défaille, un processus d'élection choisit un autre maître parmi les répliques



Partition et offset

Chaque partition est une séquence **ordonnée et immuable** d'enregistrements.

Un numéro d'identification séquentiel nommé **offset** est attribué à chaque enregistrement.

Le cluster Kafka conserve durablement tous les enregistrements publiés, qu'ils aient ou non été consommés, en utilisant une **période de rétention** configurable.

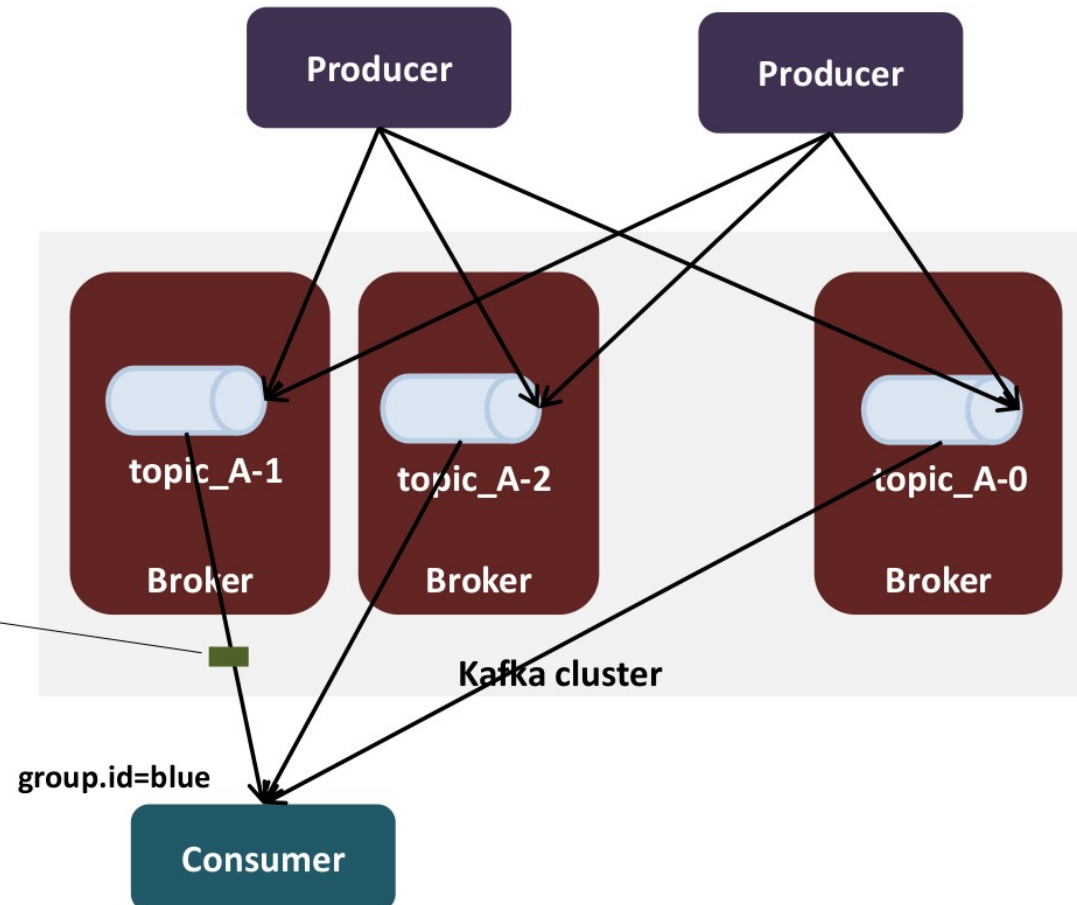
Clients du cluster

Les producteurs et consommateurs sont connectés à tous les brokers détenant le *topic*

Ils découvrent les nœuds grâce à des adresses de *bootstrap*

message (record, event)

- key-value pair
- string, binary, json, avro
- serialized by the producer
- stored in broker as byte arrays
- deserialized by the consumer





Routing des messages

Les producteurs sont responsables du choix de la partition en fonction de l'enregistrement

Cela peut être fait

- Via une stratégie Round-Robin assurant un équilibrage de charge
- En fonction des données de l'enregistrement. Typiquement, la clé

Ce routage est en général délégué au client Kafka mais peut également être pris en charge par l'application.



Groupe de consommateurs

Les consommateurs sont taggés avec un nom de **groupe**

- Plusieurs instances de processus ou plusieurs threads peuvent avoir le même nom de groupe
=> scalabilité de la consommation
- Chaque enregistrement d'un topic est remis à une instance au sein de chaque groupe de consommateurs.



Offset consommateur

Kafka conserve pour un groupe de consommateurs son **offset** de lecture.

Kafka met à jour l'offset lorsqu'il reçoit des ordres de **commit** de la part du consommateur:

- normalement, le consommateur avance son offset au fur et à mesure de sa lecture des enregistrements,
- mais, il peut consommer dans l'ordre qu'il souhaite. Par exemple, retraiter les données les plus anciennes ou repartir d'un offset particulier.



Consommateur vs Partition

Rééquilibrage dynamique

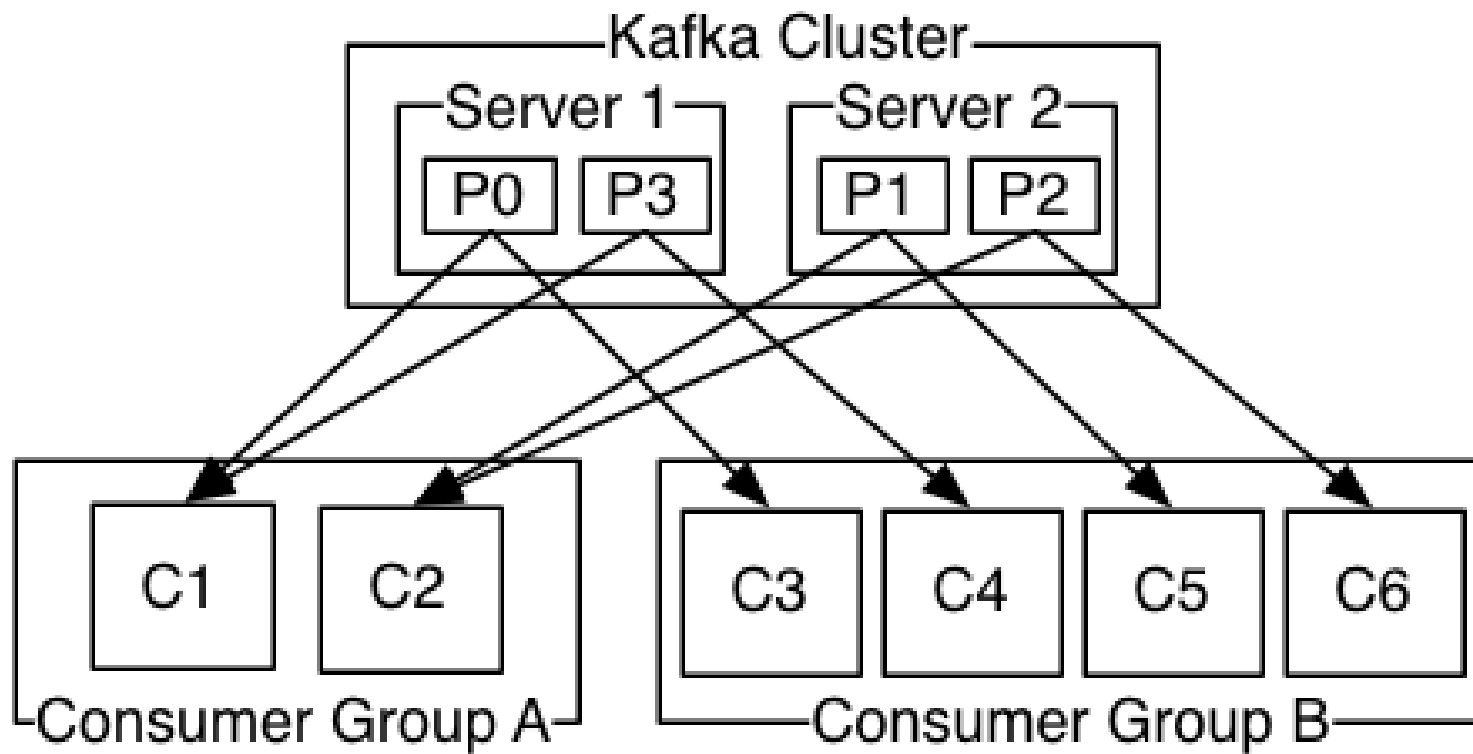
Kafka assigne les partitions à des instances de consommateur d'un même groupe.

- A tout moment, un consommateur est exclusivement dédié à une partition

Kafka gère des rééquilibrages.

- Si de nouvelles instances rejoignent le groupe, elles reprendront certaines partitions des autres membres du groupe;
- Si une instance meurt, ses partitions seront distribuées aux instances restantes.

Example





Ordre des enregistrements

Kafka garantit un ordre total sur les enregistrements d'une partition, mais pas sur les différentes partitions d'un topic.

- L'ordre sur les partitions, combiné à la possibilité de partitionner les données par une clé est suffisant pour la plupart des applications.
- Si une application nécessite un ordre strict sur tous les enregistrements. Il faut que le topic n'est qu'une seule partition



Cluster Kafka

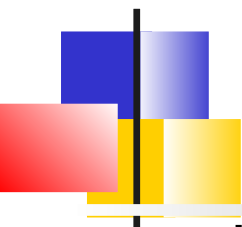
Cluster

KRaft

Distributions / Installation

Utilitaires *Kafka*

Outils graphiques



Cluster

Kafka est exécuté comme un cluster d'un ou plusieurs **serveurs** pouvant s'étendre sur plusieurs centres de données.

- Certains de ces serveurs appelés les **brokers** forment la couche de stockage.
- Un serveur est désigné **contrôleur**.
Son rôle est de prendre des décisions concernant le cluster comme l'affectation de partitions

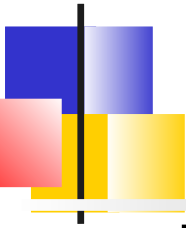
Avant la version 3.x, un cluster Kafka nécessitait également un ensemble Zookeeper faisant office de contrôleur



Nombre de brokers

Pour déterminer le nombre de brokers :

- Premier facteur :
Le niveau de tolérance aux pannes requis
- Second facteur :
La capacité de disque requise pour conserver les messages et la quantité de stockage disponible sur chaque *broker*.
- 3ème facteur :
La capacité du cluster à traiter le débit de requêtes en profitant du parallélisme.



Configuration et démarrage

Kafka fournit un script de démarrage :

kafka-server-start.sh

Chaque serveur est démarré via ce script qui lit sa configuration :

- Dans un fichier ***server.properties***
- Certaines propriétés peuvent être surchargées par la commande en ligne
Option ***--override***



Principales configuration

Les configurations principales sont :

- ***cluster.id***¹ : Identique pour chaque broker.
- ***broker.id/node.id*** : Différent pour chaque broker
- ***log.dirs*** : Ensemble de répertoires de stockage des enregistrements
- ***listeners*** : Ports ouverts pour communication avec les clients et inter-broker

1. Généré automatiquement via Zookeeper, doit être précisé en mode Kraft



Cluster Kafka

Cluster
KRaft

Distributions / Installation

Utilitaires *Kafka*

Outils graphiques



Kraft mode

Apache Kafka Raft (KRaft) basé sur le protocole de consensus *Raft* simplifie grandement l'architecture de Kafka en se séparant du processus séparé *ZooKeeper*.

En mode *KRaft*, chaque serveur Kafka est configuré en tant que contrôleur, broker ou les deux à la fois via la propriété ***process.roles***¹

Si *process.roles* n'est pas définie, il est supposé être en mode *ZooKeeper*.

1. Les valeurs possibles sont donc : ***broker*** ou ***controller*** ou ***broker,controller***



Contrôleurs

Certains processus Kafka sont donc des contrôleurs, ils participent aux quorums sur les méta-données.

- Une majorité des contrôleurs doivent être vivants pour maintenir la disponibilité du cluster
- Il sont donc typiquement en nombre impair. (3 pour tolérer 1 défaillance, 5 pour 2)

Tous les serveurs découvrent les votants via la propriété ***controller.quorum.voters*** qui les liste en utilisant l'*id*, le *host* et le *port*

controller.quorum.voters=id1@host1:port1,id2@host2:port2,id3@host3:port3



Cluster ID

Chaque cluster a un **ID** qui doit être présent dans les configurations de chaque serveur.

Avec Kraft mode, il faut définir soit même cet ID.

L'outil *kafka-storage.sh* permet de générer un ID aléatoire :

bin/kafka-storage.sh random-uuid



Formatting des répertoires de log

Avec le mode Kraft, il est nécessaire de formater les répertoires des logs.

- Le formatting consiste à créer 2 fichiers de méta-données :
 - ***meta.properties***
 - ***bootstrap.checkpoint***
- Ces fichiers sont dépendants des propriétés *cluster.id* et *node.id*

Pour créer les fichiers :

***bin/kafka-storage.sh format -t <cluster_id>
-c server.properties***



Cluster Kafka

Cluster
KRaft

Distributions / Installation

Utilitaires *Kafka*
Outils graphiques



Introduction

Kafka peut être déployé

- sur des serveurs physiques, des machines virtuelles ou des conteneurs
- sur site ou dans le cloud

Différentes distributions peuvent être récupérées :

- Binaire chez Apache.
OS recommandé Linux + pré-installation de Java
(Support de Java 17 pour 3.1.0)
- Images docker ou packages Helm (Bitnami par exemple)
- Confluent Platform (Téléchargement ou cloud)



Hardware

Afin de sélectionner le matériel :

- Débit de disque : Influence sur les producteurs de messages. SSD si beaucoup de clients
- Capacité de disque : Estimer le volume de message * période de rétention
- Mémoire : Faire en sorte que le système ait assez de mémoire disponible pour utiliser le cache de page. Pour la JVM, 5Go permet de traiter beaucoup de message
- Réseau : Potentiellement beaucoup de trafic. Favoriser les cartes réseau de 10gb
- CPU : Facteur moins important



Installation à partir de l'archive

Téléchargement, puis

```
# tar -zxf kafka_<version>.tgz
# mv kafka_<version> /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk17
# cd /usr/local/kafka/bin
# ./kafka-server-start.sh -daemon ../config/server.properties
#
```

Les propriétés de configuration définies dans ***server.properties*** peuvent être surchargées en ligne de commande par l'option ***--override***



Images bitnami

Images basées sur *minideb* (minimaliste Debian)

```
docker run -d --name kafka-server \  
  --network app-tier \  
  -e ALLOW_PLAINTEXT_LISTENER=yes \  
  bitnami/kafka:latest
```

Prises en compte de variables d'environnement pour configurer Kafka

- Variables spécifiques bitnami.
Ex : BITNAMI_DEBUG, ALLOW_PLAINTEXT_LISTENER,
...
- Mapping des variables d'environnement préfixée par KAFKA_CFG_
Ex : KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE



Packages Helm

Bitnami propose des packages Helm pour déployer vers Kubernetes ou des clouds

```
helm install my-release  
oci://registry-1.docker.io/bitnamicharts/kafka
```

De nombreux paramètres sont disponibles :

- *replicaCount* : Nombre de répliques
- *config* : Fichier de configuration
- *existingConfigmap* : Pour utiliser les ConfigMap
- ...



Configuration broker

node.id : Chaque broker doit avoir un identifiant entier unique à l'intérieur du cluster

listeners : Par défaut 9092 et 9093 pour la communication inter-contrôleurs

process.roles : Le rôle du serveur Par défaut *broker,controller*

controller.quorum.voters : Listes des contrôleurs

log.dirs : Liste de chemins locaux où kafka stocke les messages (Doivent être préformattés)



Configuration cluster

3 contraintes pour la configuration d'un cluster :

- Tous les brokers doivent avoir le même paramètre ***cluster.id***
- Chaque broker doit avoir une valeur unique pour ***node.id***
- Au moins 1 broker doit avoir le rôle contrôleur



Configuration par défaut des topics

num.partitions : Nombre de partitions, défaut 1

default.replication.factor : Facteur de réplication, défaut 1

min.insync.replicas : Joue sur les garanties de message.
Défaut 1

log.retention.ms : Durée de rétention de messages. 7J par défaut

log.retention.bytes : Taille maximale des logs. Pas défini par défaut

log.segment.bytes : Taille d'un segment. 1Go par défaut

log.roll.ms : Durée maximale d'un segment avant rotation. Pas défini par défaut

message.max.bytes : Taille max d'un message. 1Go par défaut



Vérifications de l'installation

Création de topic :

```
bin/kafka-topics.sh --create --bootstrap-server  
localhost:9092 --replication-factor 1 --partitions 1 --  
topic test
```

Envois de messages :

```
bin/kafka-console-producer.sh --bootstrap-server  
localhost:9092 --topic test  
This is a message  
This is another message  
^D
```

Consommation de messages :

```
bin/kafka-console-consumer.sh --bootstrap-server  
localhost:9092 --topic test --from-beginning  
This is a message  
This is another message
```



Script d'arrêt

La distribution propose également un script d'arrêt permettant d'arrêter les process Kafka en cours d'exécution.

`bin/kafka-server-stop.sh`



Fichiers de trace

La configuration des fichiers de trace est définie dans ***conf/log4j.properties***

Par défaut sont générés :

- ***server.log*** : Traces générales
- ***state-change.log*** : Traces des changements d'état (topics, partition, brokers, répliques)
- ***kafka-request.log*** : Traces des requêtes clients et inter-broker
- ***log-cleaner.log*** : Suppression des messages expirés
- ***controller.log*** : Logs du contrôleur
- ***kafka-authorizer.log*** : Traces sur les ACLs



Cluster Kafka

Cluster
KRaft

Distributions / Installation

Utilitaires Kafka

Outils graphiques



Introduction

Le répertoire ***\$KAFKA_HOME/bin*** contient de nombreux scripts utilitaires dédiés à l'exploitation du cluster.

Les scripts utilisent l'API Java de Kafka

Une aide est disponible pour chaque script décrivant les paramètres requis ou optionnels



Gestion des topics

Le script ***bin/kafka-topics.sh*** permet de créer, supprimer, modifier visualiser un topic

Les valeurs par défaut des topics sont définis dans *server.properties* mais peuvent être surchargées pour chaque topic

Exemple création de topic :

```
bin/kafka-topics.sh --create \  
  --bootstrap-server localhost:9092 \  
  --replication-factor 1 --partitions 13 \  
  --topic my-topic
```

Exemple listing des topics :

```
bin/kafka-topics.sh --list \  
  --bootstrap-server localhost:9092
```



Choix du nombre de partitions

Une fois un *topic* créé, on ne peut pas diminuer son nombre de partitions. Une augmentation est possible mais peut générer des problèmes.

Lors du choix, il faut tenir compte de :

- Du débit des écritures
- Le débit maximum de consommation
- L'espace disque et la bande passante réseau sur chaque broker
- En général en augmentant le nombre de partitions, on augmente le débit global de l'application en s'autorisant plus de consommateurs
- Mais évitez de surestimer, car chaque partition utilise de la mémoire et le nombre de partitions augmente le temps du processus d'élection du maître



Envoi de message

Le script ***bin/kafka-console-producer.sh*** permet de tester l'envoi des messages sur un topic

Exemple :

```
bin/kafka-console-producer.sh \  
    --bootstrap-server localhost:9092 \  
    --topic my-topic
```

...Puis saisir les messages sur l'entrée standard



Lecture de message

Le script ***bin/kafka-console-consumer.sh*** permet de consommer les messages d'un topic

Exemple lecture depuis l'origine :

```
bin/kafka-console-consumer.sh \  
  --bootstrap-server localhost:9092 \  
  --topic my-topic \  
  --from-beginning
```



Autre utilitaires

kafka-consumer-groups.sh permet de gérer les groupes de consommateurs : les lister et manipuler leurs offsets

kafka-reassign-partitions.sh permet de gérer les partitions, déplacement sur de nouveau brokers, de nouveaux répertoire, extension de cluster, ...

kafka-replica-verification.sh : Vérifie la cohérence des répliques

kafka-log-dirs.sh : Obtient les informations de configuration des log.dirs du cluster

kafka-dump-log.sh permet de parser un fichier de log et d'afficher les informations utiles pour debugger

kafka-delete-records.sh Permet de supprimer des messages jusqu'à un offset particulier



Autre utilitaires (2)

kafka-configs.sh permet des mises à jour de configuration dynamique des brokers

kafka-cluster.sh obtenir l'ID du cluster et sortir un broker du cluster

kafka-metadata-quorum.sh permet d'afficher les informations sur les contrôleurs

kafka-acls.sh permet de gérer les permissions sur les topics

kafka-verifiable-consumer.sh, kafka-verifiable-producer.sh: Utilitaires permettant de produire ou consommer des messages et de les afficher sur la console au format JSON



Cluster Kafka

Cluster
KRaft

Distributions / Installation

Utilitaires Kafka

Outils graphiques



Outils graphiques

Il est recommandé de s'équiper d'outils graphiques aidant à l'exploitation du cluster.

Comme produit gratuit, semi-commerciaux citons :

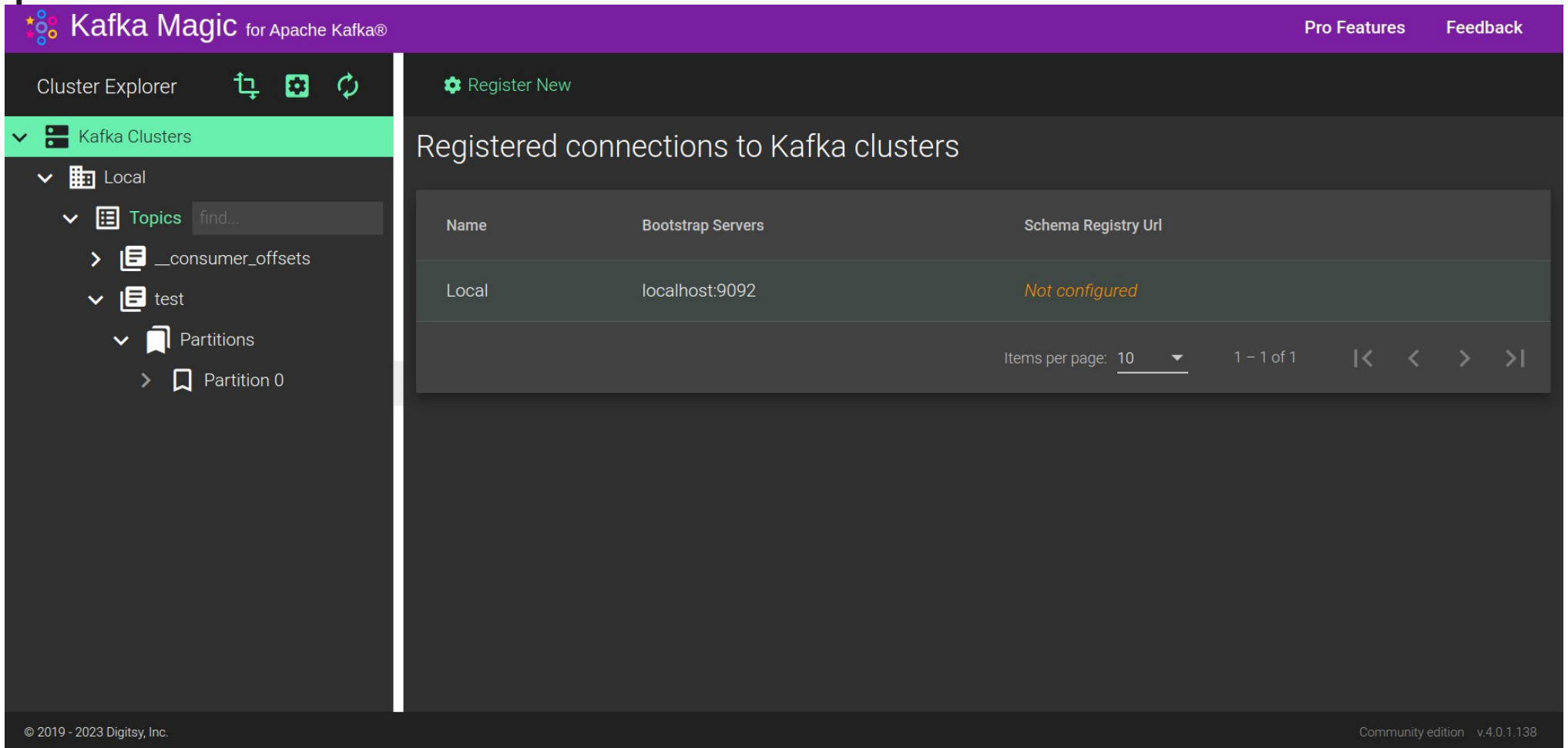
- **Akhq** (<https://akhq.io/>)
- **Kafka Magic** (<https://www.kafkamagic.com/>)
- **Redpanda Console** (<https://docs.redpanda.com/docs/manage/console/>)

Ils proposent une interface graphique permettant :

- De rechercher et afficher des messages,
- Transformer et déplacer des messages entre des topics
- Gérer les topics
- Automatiser des tâches.

La distribution commerciale de Confluent a naturellement un outil graphique d'exploitation

Kafka Magic



The image shows the Kafka Magic web interface. The top navigation bar is purple with the 'Kafka Magic for Apache Kafka®' logo on the left, and 'Pro Features' and 'Feedback' links on the right. Below the navigation bar, the left sidebar is dark grey and contains a 'Cluster Explorer' section with icons for a cluster, settings, and refresh. The 'Kafka Clusters' section is expanded, showing a tree view with 'Local' selected. Under 'Local', the 'Topics' section is expanded, showing a search bar and a list of topics: '__consumer_offsets', 'test', and 'Partitions'. The 'Partitions' section is also expanded, showing 'Partition 0'. The main content area is dark grey and displays 'Registered connections to Kafka clusters'. It features a table with three columns: 'Name', 'Bootstrap Servers', and 'Schema Registry Url'. The table contains one row for the 'Local' cluster, with 'localhost:9092' in the 'Bootstrap Servers' column and 'Not configured' in the 'Schema Registry Url' column. At the bottom of the table, there is a pagination control showing 'Items per page: 10' and '1 - 1 of 1'.

Kafka Magic for Apache Kafka®

Pro Features Feedback

Cluster Explorer

Kafka Clusters

Local

Topics find...

__consumer_offsets

test

Partitions

Partition 0

Register New

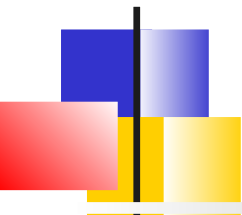
Registered connections to Kafka clusters


Name	Bootstrap Servers	Schema Registry Url
Local	localhost:9092	Not configured

Items per page: 10 1 - 1 of 1


© 2019 - 2023 Digitsy, Inc. Community edition v.4.0.1.138


akhq





akhq.io


0.24.0


my-cluster


Nodes


Topics

Live Tail

Consumer Groups



ACLs

Schema Registry

Settings

Nodes

[default](#)

Id	Host	Controller	Partitions (% of total)	Rack	
1	localhost:9092	False	1 (100.00%)		Q
2	localhost:9192	True			Q
3	localhost:9292	False			Q

Redpanda Console



Redpanda

Overview

Topics

Schema Registry

Consumer Groups

Security

Quotas

Connectors

Reassign Partitions

Cluster > Topics



1

Total Topics

1

Total Partitions

Create Topic

☐ Show internal topics

Name



Partitions

Replicas

CleanupPolicy



Size



test

1

1

delete

145 B

Total 1 items < 1 > 50 / page



Redpanda Console (Platform Version v23.1) (built May 03, 2023) FC2BF3B



Apache Kafka et ses APIs

Les APIs Kafka

Producer API

Consumer API

Sérialisation et Schema Registry

Autres APIs

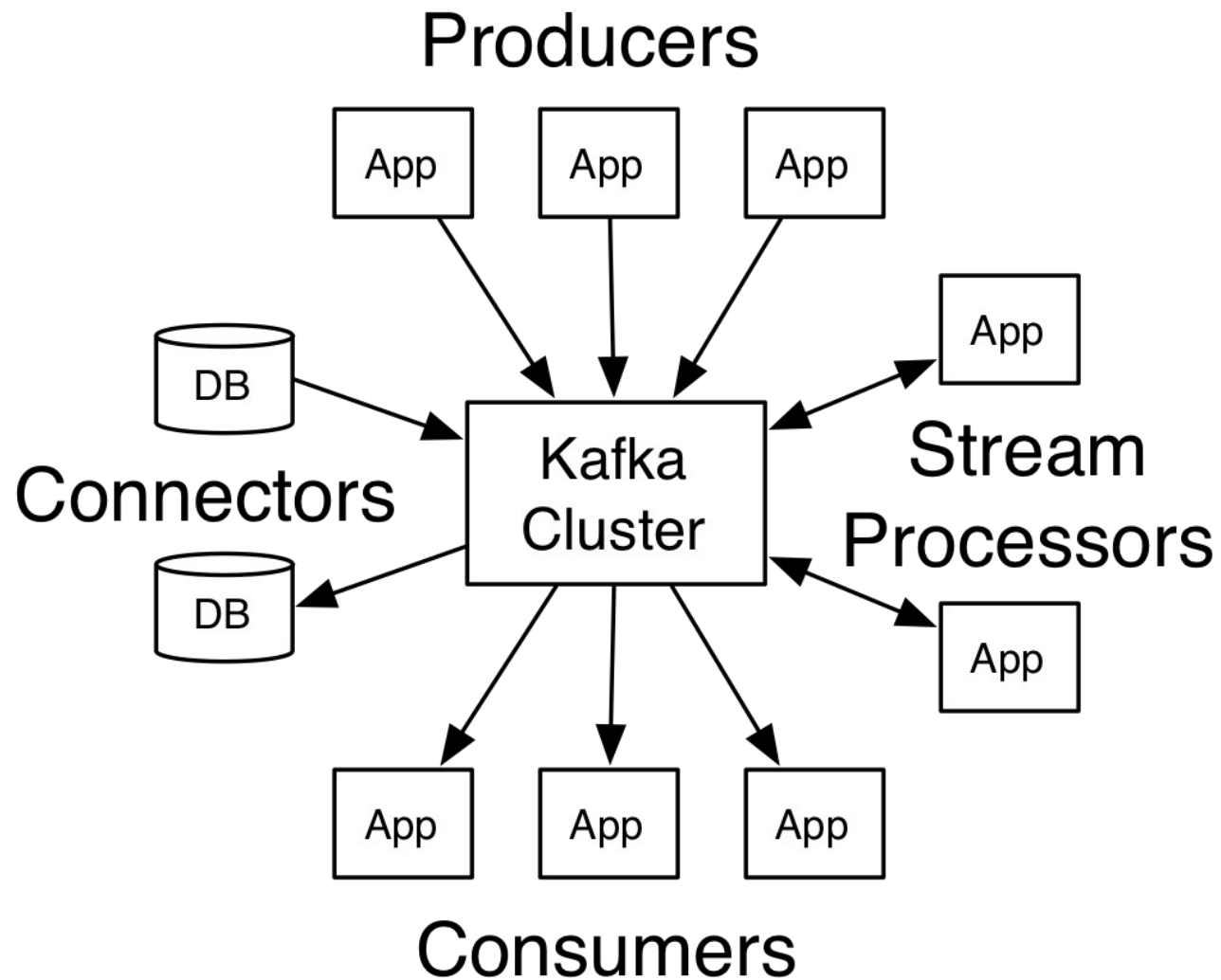


APIs

Kafka propose 5 principales APIs :

- L'API **Producer** permet à une application de publier un flux sur un ou plusieurs topics Kafka.
- L'API **Consumer** permet à une application de s'abonner à un ou plusieurs topics et de traiter le flux d'enregistrements associé.
- L'API **Streams** permet à une application d'agir comme un processeur de flux, consommant un ou plusieurs topic d'entrée et produisant un flux de sortie vers un ou plusieurs topics.
- L'API **Connector** permet de créer et d'exécuter des producteurs ou des consommateurs à partir de système tierces (BD, fichiers, STDOUT, ...)
- L'API **Admin** permet de gérer les topics et le cluster

APIs





Dépendances

Java

```
<dependency>  
    <groupId>org.apache.kafka</groupId>  
    <artifactId>kafka-clients</artifactId>  
    <version>${kafka-version}</version>  
</dependency>
```




Apache Kafka et ses APIs

Les APIs Kafka

Producer API

Consumer API

Sérialisation Json, Avro

KafkaAdmin et KafkaStream



Introduction

L'API est simple mais les contraintes applicatives influencent la façon de l'utiliser ainsi que la configuration des *topics*

Les questions devant être posées :

- Chaque message est-il critique, ou peut-on tolérer des pertes de messages?
- La duplication accidentelle de messages est elle autorisée ?
- Y-a-t-il des exigences strictes de latence ou de débit ?

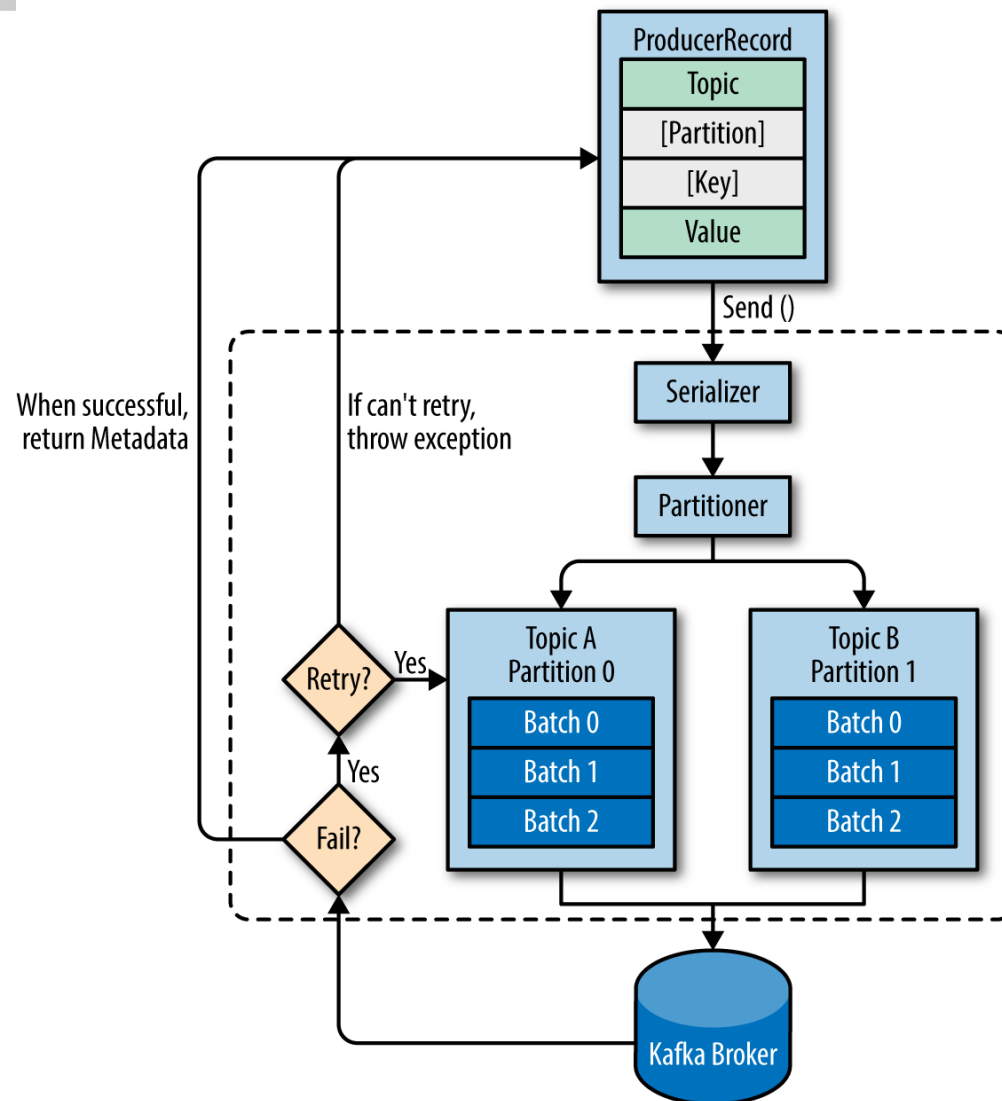


Étapes lors de l'envoi d'un message

L'envoi de message est constitué de plusieurs étapes :

- Création d'un objet **ProductRecord** encapsulant le topic et optionnellement une clé et une partition
- L'objet est **sérialisé** pour préparer sa transmission sur le réseau
- Les données sont ensuite fournies à un **partitionneur** qui détermine la partition de destination, (à partir de la partition indiquée, de la clé du message ou en Round-robin)
- Une fois la partition sélectionnée, le message est ajouté à un **lot de messages** destiné à la même partition.
Une thread séparée envoie le lot de messages.
- Lorsque le broker reçoit le message, il renvoie une réponse sous la forme d'un objet **RecordMetadata** encapsulant le *topic*, la partition, la clé et l'offset
- Si le broker n'arrive pas à écrire le message, il renvoie une erreur et le producteur peut réessayer un certain nombre de fois si l'erreur est **Retriable**

Envoi de message





Construire un Producteur

La première étape consiste à instancier un ***KafkaProducer*** en lui passant des propriétés de configuration

3 propriétés de configurations sont obligatoires :

- ***bootstrap.servers*** : Liste de brokers que le producteur contacte pour découvrir le cluster
- ***key.serializer*** : La classe utilisée pour la sérialisation de la clé
- ***value.serializer*** : La classe utilisée pour la sérialisation du message ...

1 optionnelle est généralement positionnée :

- ***client.id*** : Permet de tracer le producteur de message



Exemple Java

```
private Properties kafkaProps = new Properties();  
kafkaProps.put("bootstrap.servers",  
    "broker1:9092,broker2:9092");  
kafkaProps.put("key.serializer",  
    "org.apache.kafka.common.serialization.StringSerializer");  
kafkaProps.put("value.serializer",  
    "org.apache.kafka.common.serialization.StringSerializer");  
  
producer = new KafkaProducer<String, String>(kafkaProps);
```



ProducerRecord

ProducerRecord représente l'enregistrement à envoyer .
Il contient le nom du *topic*, une valeur et éventuellement une clé, une partition, un timestamp

Constructeurs disponibles :

Sans clé

`ProducerRecord(String topic, V value)`

Avec clé

`ProducerRecord(String topic, K key, V value)`

Avec clé et partition

`ProducerRecord(String topic, Integer partition, K key, V value)`

Avec clé, partition et timestamp

`ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)`



Méthodes d'envoi des messages

Il y a 3 façons d'envoyer des messages :

- ***Fire-and-forget*** : On n'attend pas d'acquittement,
- ***Envoi synchrone*** : La méthode d'envoi retourne un *Future*, l'appel à *get()* est bloquant et contient la réponse du broker. On traite éventuellement les cas d'erreurs
- ***Envoi asynchrone*** : Lors de l'envoi, on passe en argument une fonction de call-back. La méthode est appelée lorsque la réponse est retournée



Méthodes d'envoi

// Fire and forget

```
ProducerRecord<String, String> record =  
new ProducerRecord<>("CustomerCountry", "Precision", "France");  
try {  
    producer.send(record);  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

// Envoi synchrone

```
try {  
    producer.send(record).get();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```



Envoi asynchrone avec callback (Java)

```
private class DemoProducerCallback implements Callback {  
    @Override  
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {  
        if (e != null) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");  
producer.send(record, new DemoProducerCallback());
```

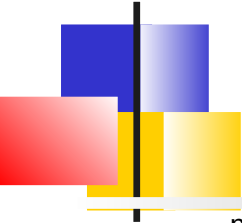


Sérialiseurs

Kafka inclut des sérialiseurs pour les types basiques (***ByteArraySerializer, StringSerializer, LongSerializer, etc.***).

Pour des objets du domaine, il faut implémenter ses propres sérialiseurs/désérialiseurs en s'appuyant sur des librairies comme *Avro, Thrift, Protobuf* ou *Jackson*

La version commerciale de Confluent privilégie le sérialiseur Avro et permet une gestion fine des formats de sérialisation via les ***Schema Registry***



Exemple sérialiseur s'appuyant sur Jackson

```
public class JsonPOJOSerializer<T> implements Serializer<T> {
    private final ObjectMapper objectMapper = new ObjectMapper();
    /**
     * Default constructor requis par Kafka
     */
    public JsonPOJOSerializer() { }

    @Override
    public void configure(Map<String, ?> props, boolean isKey) { }

    @Override
    public byte[] serialize(String topic, T data) {
        if (data == null)
            return null;

        try {
            return objectMapper.writeValueAsBytes(data);
        } catch (Exception e) {
            throw new SerializationException("Error serializing JSON message", e);
        }
    }

    @Override
    public void close() { }
}
```



Exemple désérialiseur basé sur Jackson

```
public class JsonPOJODeserializer<T> implements Deserializer<T> {
    private ObjectMapper objectMapper = new ObjectMapper();
    private Class<T> tClass;
    // Default constructor requis par Kafka
    public JsonPOJODeserializer() { }

    @Override
    public void configure(Map<String, ?> props, boolean isKey) {
        tClass = (Class<T>) props.get("JsonPOJOClass");
    }

    @Override
    public T deserialize(String topic, byte[] bytes) {
        if (bytes == null)
            return null;
        T data;
        try {
            data = objectMapper.readValue(bytes, tClass);
        } catch (Exception e) {
            throw new SerializationException(e);
        }
        return data;
    }
    @Override
    public void close() { }
}
```



Envoi de message

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("value.serializer", "org.myappli.JsonPOJOSerializer");

String topic = "customerContacts";

Producer<String, Customer> producer =
    new KafkaProducer<String, Customer>(props);

Customer customer = CustomerGenerator.getNext();

ProducerRecord<String, Customer> record =
    new ProducerRecord<>(topic, customer.getId(), customer);
producer.send(record);
```



Écriture sur une partition

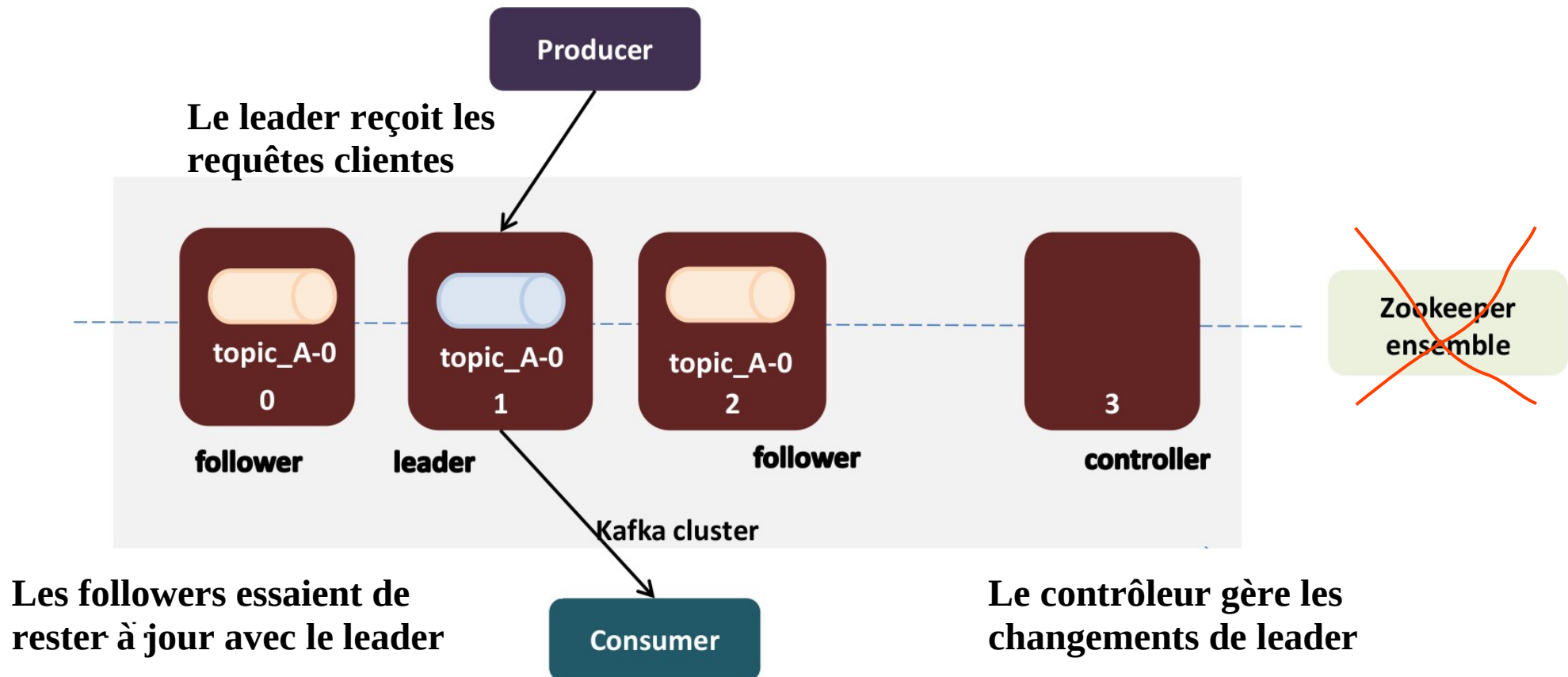
Différents brokers participent à la gestion distribuée et répliquée d'une partition.

- **1 broker leader** : Détient la partition leader responsable de la validation des écritures
- **N followers** : Suivent les écritures effectuées par le leader.
Un décalage est permis mais si il n'arrive plus à suivre la cadence d'écriture, ils sont éliminés

Au niveau du cluster :

- Un **contrôleur** est responsable des élections de leader

Rôle des brokers gérant un topic





Garanties Kafka

Garanties offertes par Kafka grâce à la réplication :

- Garantie de l'ordre à l'intérieur d'une partition.
=> Le consommateur d'une partition lit dans l'ordre d'écriture des messages
- Les messages produits sont considérés **validés** lorsqu'ils sont écrits sur la partition leader et qu'ils ont atteint le minimum de réplication, i.e. minimum de répliques synchronisées
- Les messages validés sont disponibles tant qu'au moins une réplique reste en vie.
- Les consommateurs ne peuvent lire que les messages validés.



ISR

Contrôlé par la propriété :

replica.lag.time.max.ms (défaut 30 000)

Si pendant ce délai, le follower

- N'envoie pas de requêtes fetch
- N'atteint pas l'offset de fin du leader

Alors, le follower est considéré comme désynchronisé

- Il est supprimé de la liste des **ISR (In Sync Replica)**
- Il peut ensuite rattraper son retard et être réintégré aux ISRs

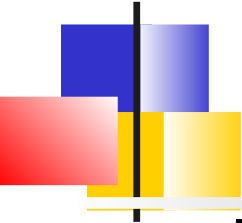
kafka-topics.sh --describe : permet de voir les ISR pour chaque partition d'un topic



min.insync.replica

La propriété ***min.insync.replica***, spécifiée au niveau cluster ou topic, indique le minimum de répliques de l'ISR (incluant le leader) qui doivent avoir écrit un message afin que celui-ci soit considéré comme validé

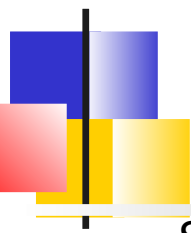
- A la réception d'un message, le leader vérifie si il y a assez d'ISR pour écrire le message, sinon il envoie une *NotEnoughReplicasException*
- Lorsque le message est répliqué par *min.insync.replica* répliques, le leader envoie un acquittement au client.



Conséquences

Un réplique synchronisée légèrement en retard peut ralentir l'acquittement du message ; ce qui peut ralentir le débit.

Une réplique désynchronisée n'a plus d'impact sur les performances mais augmente le risque d'un temps d'arrêt ou d'une perte de données.



Rejet de demande d'émission

Si le **nombre de ISR** < ***min.insync.replicas*** :

- Kafka empêche l'acquittement du message.
En fonction de la configuration du producteur, celui-ci peut être bloqué.

Si le **nombre de répliques disponible** < ***min.insync.replicas***

- Mauvaise configuration, Kafka bloque les émissions de message

En conséquences :

n répliques

=> tolère $n-1$ failures pour que la partition soit disponible à la consommation

n répliques, $min.insync.replicas = m$

=> Tolère $n-m$ failures pour accepter les envois



Configuration des producteurs *fiabilité*

Certains paramètres ont un impact significatif sur l'utilisation de la mémoire, les performances et la fiabilité des producteurs.

Fiabilité :

- **acks** : contrôle le nombre de réplicas qui doivent recevoir l'enregistrement avant que le producteur puisse considérer l'écriture comme réussie
- **retries** : Si l'erreur renvoyée est de type Retriable, le nombre de tentative de renvoi. Si > 0 possibilité d'envoi en doublon
- **max.in.flight.requests.per.connection** : Maximum de message en cours de transmission (sans réponse obtenu)
- **enable.idempotence** : Livraison unique de message
- **transactional.id** : Mode transactionnel



Configuration des producteurs *performance*

- ***batch.size*** : La taille du batch en mémoire pour envoyer les messages.
Défaut 16ko
- ***linger.ms*** : la durée d'attente de messages supplémentaires avant d'envoyer le batch courant. Défaut 0ms
- ***buffer.memory*** : Taille buffer pour stocker les messages que l'on ne peut pas envoyé. Défaut 32Mo
- ***compression.type*** : Par défaut, les messages ne sont pas compressés. Valeurs possibles : *snappy* , *gzip* , ou *lz4*
- ***request.timeout.ms***, ***metadata.fetch.timeout.ms*** et ***timeout.ms***:
Timeouts pour la réception d'une réponse à un message, pour obtenir des méta-données (leader, etc..) pour obtenir le ack des répliques.
- ***max.block.ms*** : Temps maximum d'attente pour la méthode *send()*.
Dans le cas ou le buffer est rempli
- ***max.request.size*** : Taille max d'un message
- ***receive.buffer.bytes*** et ***send.buffer.bytes***: Taille des buffers TCP



Garantie sur l'ordre

En absence de failure, *Kafka* préserve l'ordre des messages au sein d'une partition.

- Si des messages ont été envoyés par le producteur dans un ordre spécifique, le broker les écrit sur une partition dans cet ordre et tous les consommateurs les liront dans cet ordre ...

En cas de failure

- Si *retries* > 0 et *max.in.flights.requests.per.session* > 1 .
Il se peut que lors d'un renvoi l'ordre initial soit inversé.
- Pour avoir une garantie sur l'ordre avec tolérance aux fautes, on configure
retries > 0 et *max.in.flights.requests.per.session* = 1
(au détriment du débit global)



Apache Kafka et ses APIs

Les APIs Kafka

Producer API

Consumer API

Sérialisation et Schema Registry

Autres APIs



Introduction

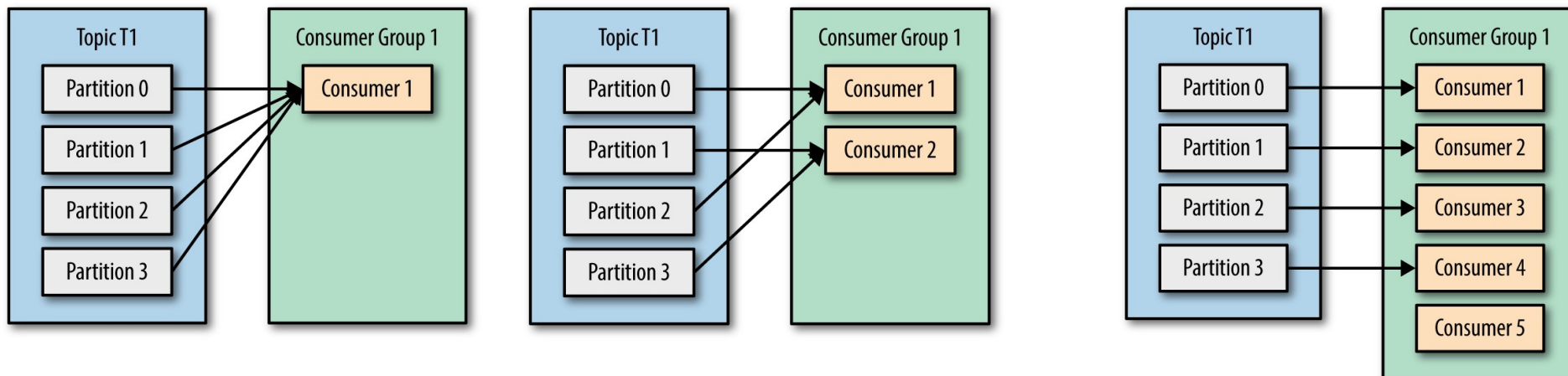
Les applications qui ont besoin de lire les données de Kafka utilisent un ***KafkaConsumer*** pour s'abonner aux topics

Pour bien comprendre l'API, il faut comprendre la notion de groupe de consommateurs et leurs relations avec les partitions

Groupes de consommateurs

Les consommateurs font généralement partie d'un **groupe de consommateurs**.

- Chaque consommateur d'un groupe reçoit les messages d'un sous-ensemble différent des partitions du topic.





Rééquilibrage dynamique des consommateurs

Lors de l'ajout d'un nouveau consommateur, celui-ci peut se faire affecter une partition consommée précédemment par un autre consommateur du groupe.

Lors de l'arrêt d'un consommateur, les partitions qui lui était assignées sont réaffectées à un autre consommateur

Cette répartition dynamique offre la scalabilité et la tolérance aux défaillances mais n'est pas spécialement désirable

- Durant le rééquilibrage les messages ne sont pas consommés
- les consommateurs si ils utilisent des caches sont obligés de les rafraîchir



Création de *KafkaConsumer*

L'instanciation d'un ***KafkaConsumer*** est similaire à celle d'une *KafkaProducer*

Plusieurs propriétés doivent être spécifiées dans une classe *Properties* :

- *bootstrap.servers*
- *key.deserializer* , et *value.deserializer*
- *group.id* qui spécifie le groupe de consommateur



Exemple

```
private Properties kafkaProps = new Properties();  
kafkaProps.put("bootstrap.servers",  
    "broker1:9092,broker2:9092");  
kafkaProps.put("key.deserializer",  
    "org.apache.kafka.common.serialization.StringDeserializer");  
kafkaProps.put("value.deserializer",  
    "org.apache.kafka.common.serialization.StringDeserializer");  
kafkaProps.put("group.id", "myGroup");  
  
consumer = new KafkaConsumer<String, String>(kafkaProps);
```



Abonnement à un *topic*

Un *KafkaConsumer* peut s'abonner à un ou plusieurs *topic(s)*

La méthode ***subscribe()*** prend une liste de *topics* ou une expression régulière.

Ex :

```
consumer.subscribe(Collections.singletonList("myTopic"));
```

```
consumer.subscribe("test.*");
```



Boucle de Polling

Typiquement, les consommateurs *poll* continuellement les *topics* auxquels ils sont abonnés.

Les objets retournés par *poll* sont une collection de ***ConsumerRecord*** qui encapsule :

- le message
- La partition
- L'offset
- Le timestamp



Exemple

```
try {
while (true) {
    // poll bloque pdt au maximum 100ms pour récupérer les messages
    // retourne immédiatement si des messages sont disponibles
    ConsumerRecords<String, String> records = consumer.poll(100);

    for (ConsumerRecord<String, String> record : records) {
        log.debug("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n", record.topic(), record.partition(),
            record.offset(), record.key(), record.value());

        // Traitement du record
        _handleRecord(record) ;
    }
} finally {
    consumer.close();
}
```



Thread et consommateur

Il n'est pas possible d'avoir plusieurs consommateurs du même groupe dans la même thread et il n'est pas possible d'utiliser le même consommateur dans plusieurs threads
=> 1 consommateur = 1 thread

Pour exécuter plusieurs consommateurs d'un même groupe dans une application, il est utile d'utiliser les classes *ExecutorService* de Java ou de démarrer plusieurs processus.



Offsets et Commits

Les consommateurs synchronisent l'état d'avancement de leur consommation, en périodiquement indiquant à Kafka le dernier offset traité.

Kafka appelle la mise à jour d'un offset : un ***commit***

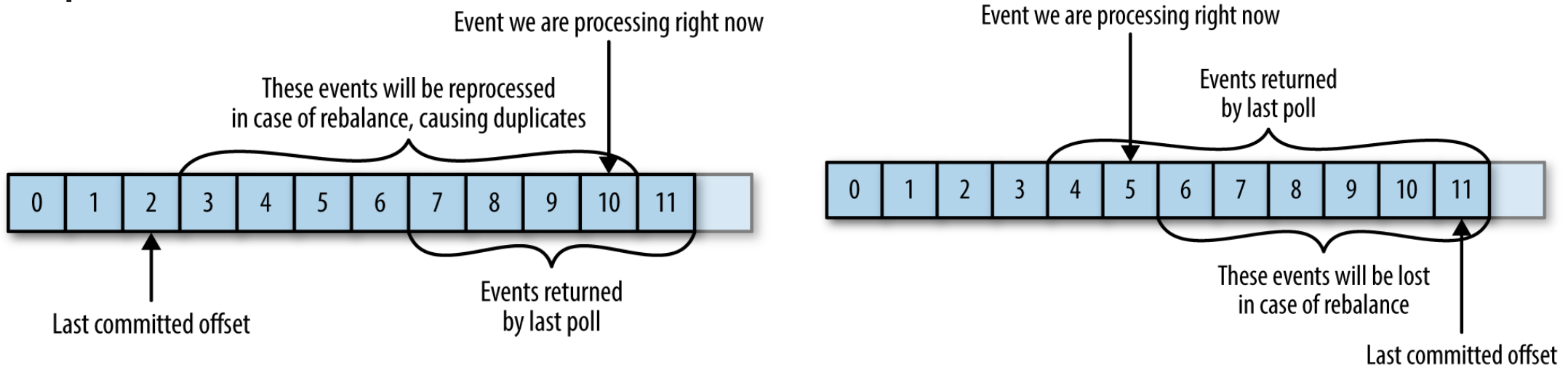
Pour committer, un consommateur envoie un message vers un *topic* particulier de Kafka : ***__consumer_offsets***

- Ce *topic* contient les offsets de chaque partition.

Lors d'une réaffectation de partitions, 2 risques doivent être pris en compte :

- Traiter 2 fois le même message
- Louper des messages

Risques lors d'une réaffectation



Kafka propose plusieurs façons de gérer les commits



Gestion des commits

Différentes alternatives sont possible pour gérer les commits :

- Laisser l'API Kafka, committer automatiquement (défaut)
- Committer manuellement les commits
- Gérer les commits en dehors de kafka et positionner soi-même, l'offset à lire lors de l'appel à poll



Commit automatique

Si ***enable.auto.commit=true*** ,
le consommateur valide automatiquement tout
les ***auto.commit.interval.ms*** (par défaut
5000), les plus grands offset reçus par *poll()*

=> Cette approche (simple) ne protège pas
contre les duplications en cas de ré-affectation

Si le traitement des messages est asynchrone, on
peut également perdre des messages



Commit contrôlé

L'API Consumer permet de contrôler le moment du commit plutôt que de se baser sur un timer.

Si ***auto.commit.offset=false*** ,
l'application doit explicitement committer les offsets

- Soit de façon bloquante avec ***commitSync()***
- Soit en asynchrone avec ***commitAsync()***



Commit Synchrone

La méthode ***commitSync()*** valide les derniers offsets reçus par *poll()*

- La méthode est bloquante et retourne lorsque les offsets sont commités
- Elle lance une exception si un commit échoue

=> En cas de réaffectation, il y a toujours un risque pour que le messages soit traités plusieurs fois



Exemple Java

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        // Si réaffectation, possibilité de duplication
        log.info("topic = %s, partition = %s,
            offset =%d, customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value());
    }
    try {
        consumer.commitSync();
    } catch (CommitFailedException e) {
        log.error("commit failed", e)
    }
}
```



Commit asynchrone

commitAsync() est non bloquante et il est possible de fournir un callback en argument

```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
    for (ConsumerRecord<String, String> record : records) {  
        _handleRecord(record)  
    }  
    consumer.commitAsync(new OffsetCommitCallback() {  
        public void onComplete(Map<TopicPartition,  
            OffsetAndMetadata> offsets, Exception exception) {  
            if (e != null)  
                log.error("Commit failed for offsets {}", offsets, e);  
        }  
    });  
}
```



Committer un offset spécifique

L'API permet également de fournir en argument une *Map* contenant les offsets que l'on veut valider pour chaque partition

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new
    HashMap<>();
int count = 0;
....
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        _handleRecord(record) ;
        currentOffsets.put(new TopicPartition(record.topic(),record.partition()),
new OffsetAndMetadata(record.offset()+1, "no metadata"));
        if (count % 1000 == 0)
            consumer.commitAsync(currentOffsets, null);
        count++;
    }
}
```



Stocker les offsets hors de Kafka

Il n'y a pas d'obligation à stocker les offsets dans Kafka, un consommateur peut stocker lui-même les offsets dans son propre data store.

- Si les offsets et le résultat du traitement sont stockés dans la même BD, on peut alors profiter d'une écriture transactionnelle.

Ce type de scénario permet d'obtenir facilement des garanties de livraison « *Exactly Once* » même en cas de défaillance



Réagir aux réaffectations

Lors de réaffectations de partitions, les consommateurs peuvent être prévenus afin de prendre les mesures adéquates (commit, fermeture de ressources, ...)

Lors du *subscribe()*, il faut fournir une classe de type ***ConsumerRebalanceListener*** qui implémente 2 méthodes :

- `public void onPartitionsRevoked(Collection<TopicPartition> partitions)`
- `public void onPartitionsAssigned(Collection<TopicPartition> partitions)`



Example

```
private class HandleRebalance implements
    ConsumerRebalanceListener {

    public void onPartitionsAssigned(
        Collection<TopicPartition> partitions) { }

    public void onPartitionsRevoked(
        Collection<TopicPartition> partitions) {
        log.info("Lost partitions in rebalance.
            Committing current offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets);
    }
}
```



Consommation de messages avec des offsets spécifiques

L'API de consommation permet d'indiquer un offset spécifique :

- ***seekToBeginning(TopicPartition tp)*** :
Revenir au début de la partition
- ***seekToEnd(TopicPartition tp)*** :
Se placer à la fin
- ***seek(TopicPartition, long)*** :
Se placer à un offset particulier
Cela permet de stocker les offsets en dehors de Kafka et en cas d'erreur ou redémarrage repartir à partir des offsets sauvegardés



Sortie de boucle

Pour sortir de la boucle de *poll*, il faut qu'une autre thread appelle ***consumer.wakeup()*** qui a pour effet de lancer une *WakeupException* lors de l'appel à *poll*.

Le consommateur doit alors faire un appel explicite à *close()*

On peut utiliser
Runtime.addShutdownHook(Thread hook)



Example

```
Runtime.getRuntime().addShutdownHook(new Thread() {  
    public void run() {  
        System.out.println("Starting exit...");  
        consumer.wakeup();  
        try {  
            mainThread.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
});
```



Example (2)

```
try {
// looping until ctrl-c, the shutdown hook will cleanup on exit
while (true) {
    ConsumerRecords<String, String> records = movingAvg.consumer.poll(1000);
    log.info(System.currentTimeMillis() + "-- waiting for data...");
    for (ConsumerRecord<String, String> record : records) {
        log.info("offset = %d, key = %s,value = %s\n",
            record.offset(), record.key(),record.value());
    }
    for (TopicPartition tp: consumer.assignment())
        log.info("Committing offset atposition:"+consumer.position(tp));
    movingAvg.consumer.commitSync();
}
} catch (WakeupException e) {
} finally {
    consumer.close();
    log.info("Closed consumer and we are done");
}
```



Affectation statique des partitions

Si l'application ne comporte que des consommateurs dédiés statiquement à des partitions, alors la fonctionnalité de groupe n'est pas nécessaire.

Dans ce cas, l'application assigne explicitement les partitions à ses consommateurs.

L'API ***assign()*** est alors utilisée à la place de *subscribe()*



Example

```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic");

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(), partition.partition()));

    consumer.assign(partitions);
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(1000);
        for (ConsumerRecord<String, String> record: records) {
            log.info("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```



auto.offset.reset

Si le consommateur n'a pas d'offset défini dans Kafka, le comportement est piloté par la propriété

***auto.offset.reset* :**

- *latest* (défaut), l'offset est initialisé au dernier offset
- *earliest* : L'offset est initialisé au premier offset disponible



Autres propriétés

D'autres propriétés

- ***fetch.min.bytes*** : Volume minimum de données à recevoir. Permet de réduire la charge sur le broker et le consommateur
- ***fetch.max.wait.ms*** : Attente maximale avant de récupérer les données
- ***max.partition.fetch.bytes*** : Maximum de données par partition ramenées lors d'un poll. Par défaut 1Mo
- ***max.poll.records*** : Maximum de record via un *poll()*
- ***session.timeout.ms*** : Le temps faisant expirer la session et déclarer le consommateur comme down. Par défaut 10s
- ***heartbeat.interval.ms*** : L'intervalle d'envoi des heartbeat. Par défaut 3s
- ***partition.assignment.strategy*** : Stratégie d'affectation des partitions
Range (défaut), *RoundRobin* ou *Custom*
- ***client.id*** : Une chaîne de caractère utilisé pour les métriques.
- ***receive.buffer.bytes*** et ***send.buffer.bytes*** : Taille des buffers TCP



Apache Kafka et ses APIs

Les APIs Kafka

Producer API

Consumer API

Sérialisation et Schema Registry

Autres APIs



Introduction

Lors d'évolution des applications, le format des messages est susceptible de changer.

=> Afin de s'assurer que ses évolutions ne génèrent pas de problème chez les consommateurs, il est nécessaire d'utiliser un gestionnaire de schéma capable de détecter les problèmes de compatibilité.

C'est le rôle de ***Schema Confluent Registry*** qui supporte les formats de sérialisation JSON, Avro et ProtoBuf



Apache Avro

Apache Avro est un système de sérialisation de données.

- Il utilise une structure JSON pour définir le **schéma**, permettant la sérialisation entre les octets et les données structurées.
- Les outils associés à Avro sont capables de générer les classes Java¹ correspondantes au schéma.
- Il offrent également un format de sérialisation binaire plus compact

1. Avro supporte C, C++, C#, Java, PHP, Python, et Ruby



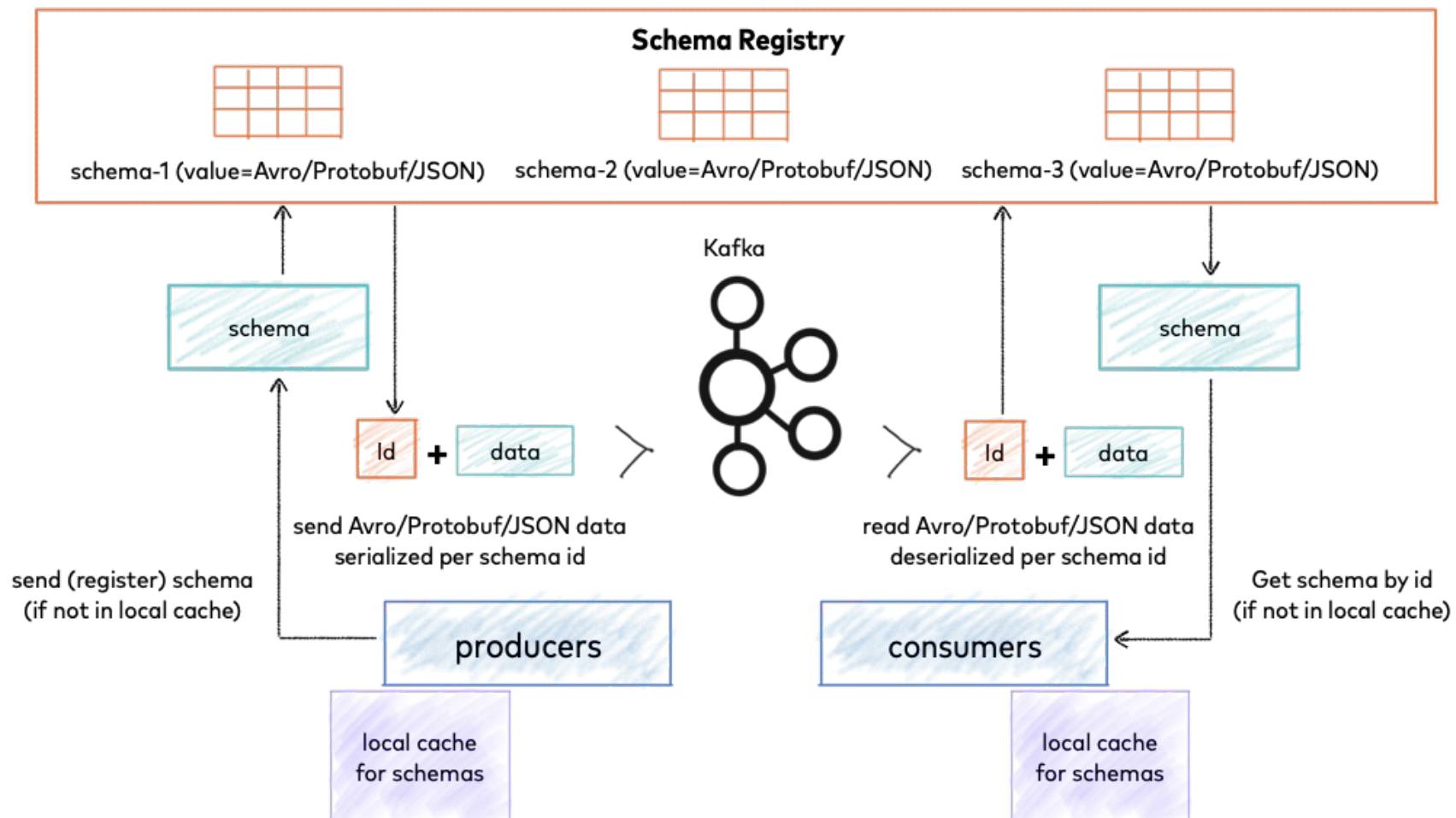
Utilisation du schéma

Confluent Schema Registry offre une API Rest

- permettant de stocker des Schema
- De détecter les compatibilité entre schémas
Si le schéma est incompatible, le producteur est empêché de produire vers le topic
Il faudra publier une autre version vers un autre topic

Les sérialiseurs inclut l'id du schéma dans les messages

Schema Registry





Enregistrement de schéma

L'enregistrement du schéma peut s'effectuer via l'outil de build :

```
mvn schema-registry:register
```

Ou en utilisant la librairie cliente Java de *Schema Registry* :

```
// Lecture du schéma
Schema avroSchema = new Schema.Parser().parse(inputStream);
// Instanciation du client
CachedSchemaRegistryClient client = new CachedSchemaRegistryClient(REGISTRY_URL,
    20);
// Enregistrement du sujet, le nom correspond au nom du topic + suffixe
client.register(topicName + "-value", new AvroSchema(avroSchema));
```



Producteur

Les propriétés du producteur Kafka doivent contenir :

- ***schema.registry.url*** :
L'adresse du serveur de registry
- Le sérialiseur de valeur :
io.confluent.kafka.serializers.KafkaAvroSerializer



Consommateur

KafkaConsumer doit également préciser l'URL et le désérialiseur à ***io.confluent.kafka.serializers.KafkaAvroDeserializer***

Il peut récupérer les messages sous la forme de ***GenericRecord*** plutôt que des classes spécialisés.

```
ConsumerRecords<String, GenericRecord> records =  
    consumer.poll(Duration.ofMillis(sleep));  
for (ConsumerRecord<String, GenericRecord> record : records) {  
    System.out.println("Value is " + record.value());  
}
```



Apache Kafka et ses APIs

Les APIs Kafka

Producer API

Consumer API

Sérialisation et Schema Registry

Autres APIs



Introduction

Kafka propose 2 autres APIs :

- ***Admin API*** : Client d'administration permettant de gérer et inspecter les topics, brokers, configurations et ACLs
- ***Streams API*** : Librairie cliente pour faciliter le traitement du flux dont les entrées/sorties sont des *topics* Kafka



Exemple Admin : Lister les configurations

```
public class ListingConfigs {  
  
    public static void main(String[] args) throws ExecutionException,  
        InterruptedException {  
        Properties config = new Properties();  
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");  
        AdminClient admin = AdminClient.create(config);  
        for (Node node : admin.describeCluster().nodes().get()) {  
            System.out.println("-- node: " + node.id() + " --");  
            ConfigResource cr = new ConfigResource(ConfigResource.Type.BROKER, "0");  
            DescribeConfigsResult dcr = admin.describeConfigs(Collections.singleton(cr));  
            dcr.all().get().forEach((k, c) -> {  
                c.entries()  
                    .forEach(configEntry -> {  
System.out.println(configEntry.name() + "= " + configEntry.value());  
                });  
            });  
        }  
    }  
}
```



Exemple Admin

Créer un topic

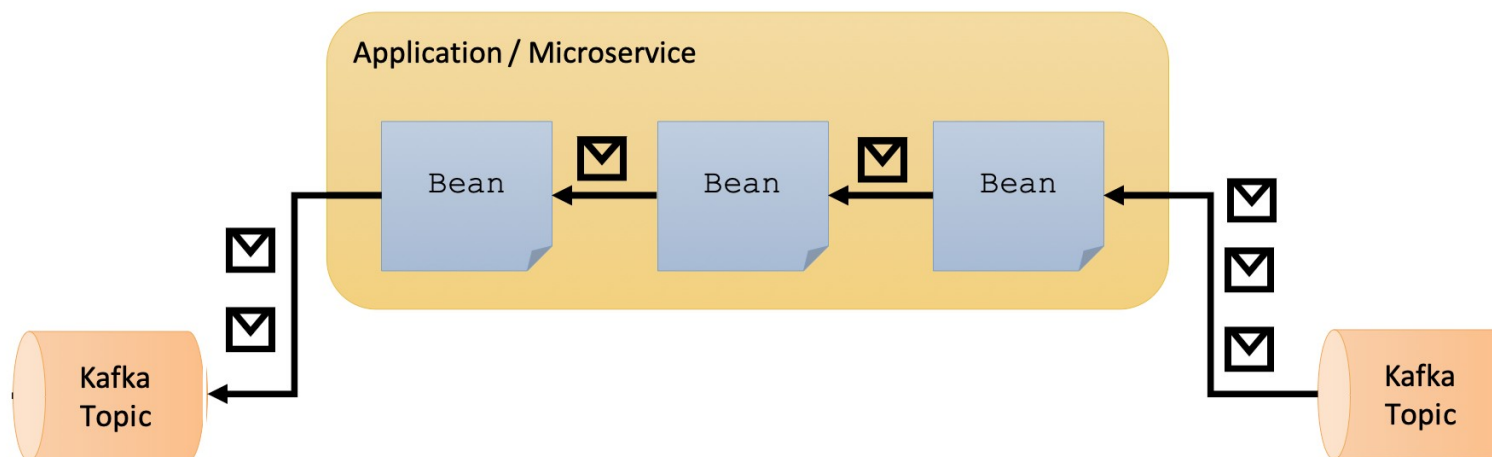
```
public class CreateTopic {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        Properties config = new Properties();
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        AdminClient admin = AdminClient.create(config);
        //Créer un nouveau topics
        System.out.println("-- creating --");
        NewTopic newTopic = new NewTopic("my-new-topic", 1, (short) 1);
        admin.createTopics(Collections.singleton(newTopic));

        //lister
        System.out.println("-- listing --");
        admin.listTopics().names().get().forEach(System.out::println);
    }
}
```



Kafka Streams

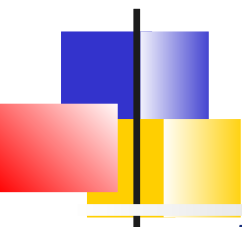
Kafka Streams API est une bibliothèque cliente pour la création d'applications et de micro-services, où les données d'entrée et de sortie sont stockées dans des topics Kafka





Apports de *KafkaStream*

- Librairie simple et légère, facilement intégrable, seule dépendance celle d'Apache Kafka.
- Abstractions ***KStream*** et ***KTable*** permettant la manipulation de flux d'évènements, transformation, filtrage, jointures et agrégations
- Peut garantir que chaque évènement soit traité une et une seule fois, même en cas de défaillance.
- Temps de latence des traitements en millisecondes,
- Supporte des opérations de fenêtrage temporel avec l'arrivée des événements dans le désordre.
- Définition d'une topologie de processeurs d'évènements sous forme de DSL ou programmatiquement.



KStream

KStream est une abstraction représentant un flux de données illimité dont:

- Les événements sont ordonnés
- Les événements sont immuables
- On peut rejouer un flux d'événements

KStream offre un grand nombre d'opérateurs de manipulation d'événements qui produise des déclinaisons de *KStream*

Un *KStream* peut

- être construit à partir d'un topic Kafka
- Être la source d'un topic



KStream et KTable

L'abstraction ***KTable*** représente un ensemble de fait qui évoluent, chaque enregistrement est associé à une clé.

- Une *KTable* peut être construit à partir d'un *Kstream*.
Seule la dernière valeur ou une agrégation, pour une clé donnée, est conservée
- *KTable* peut être transformé en Kstream.
Le flux d'événement correspond aux flux des modifications des entités



Application KafkaStream

Une application KafkaStream définit sa logique de traitement à travers une ou plusieurs **topologies de processeurs**, i.e. un graphe de processeurs de flux connectés

Un processeur représente une étape de traitement qui prend en entrée un événement, le transforme puis produit un ou plusieurs événements

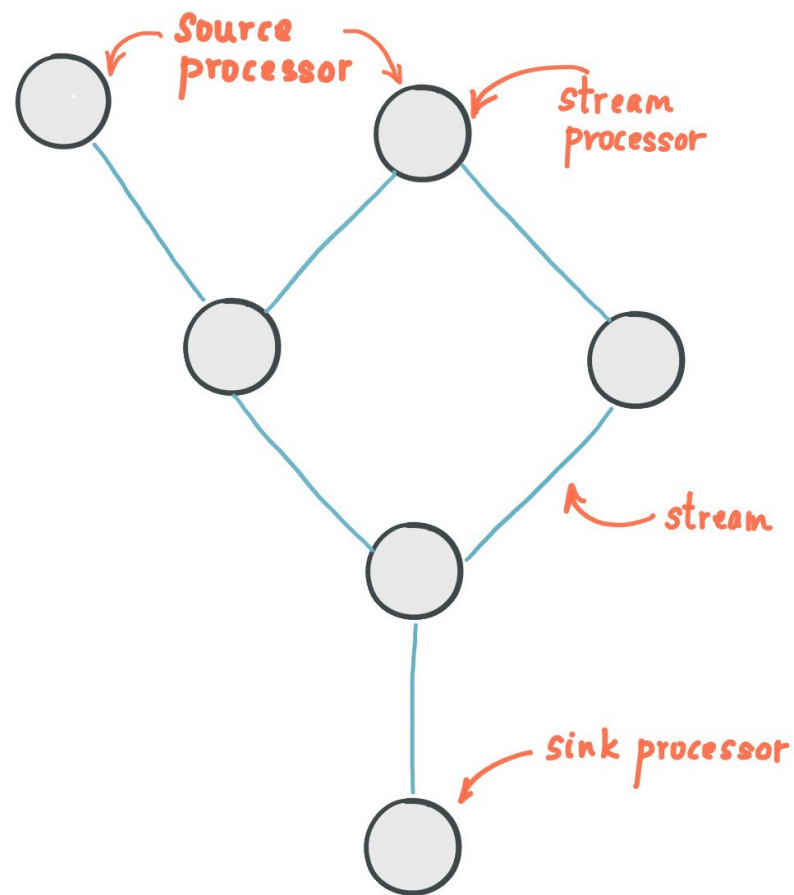
Certains processeurs :

- n'ont pas de connexions entrantes : **Source**
- d'autres n'ont pas de connexions sortantes : **Sink**
- Les autres ont des connexions entrantes et sortantes

La topologie peut être spécifiée programmatiquement ou par un DSL

Certains processeurs sont stateful et stockent un état dans un **StateStore** (Topic Kafka intermédiaire)

Topologie processeurs



PROCESSOR TOPOLOGY



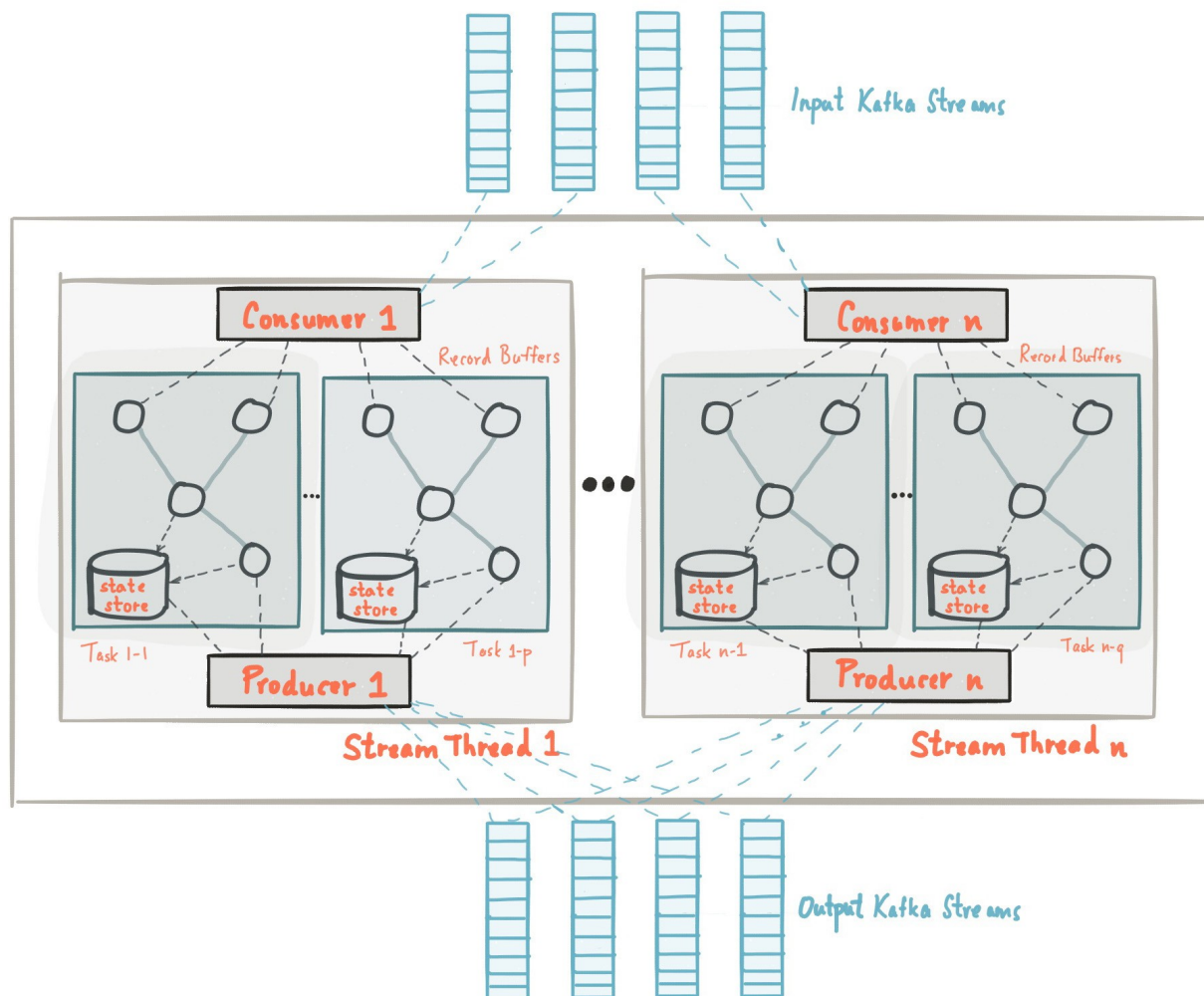
Scalabilité

KafkaStream offre plusieurs moyens pour la scalabilité

- Démarrage de n-processus
- Définition dans un processus de Task (correspondant à des pools de threads)

Bien sûr, le degré de parallélisme est fixé par le nombre de partitions du topic entrant

Architecture et scalabilité





Exemple

// Propriétés : ID, BOOTSTRAP, Serialiseur/Désérialiseur

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
```

// Création d'une topologie de processeurs

```
final StreamsBuilder builder = new StreamsBuilder();
builder.<String, String>stream("streams-plaintext-input")
    .flatMapValues(value -> Arrays.asList(value.split("\\W+")))
    .to("streams-linesplit-output");
```

```
final Topology topology = builder.build();
```

// Instanciation du Stream à partir d'une topologie et des propriétés

```
final KafkaStreams streams = new KafkaStreams(topology, props);
```



Exemple (2)

```
final CountDownLatch latch = new CountDownLatch(1);

// attach shutdown handler to catch control-c
Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
    @Override
    public void run() {
        streams.close();
        latch.countDown();
    }
});

// Démarrage du stream
try {
    streams.start();
    latch.await();
} catch (Throwable e) {
    System.exit(1);
}
System.exit(0);
```



Configuration cluster et topics

Mécanismes de réplication

Garanties de livraison

Débit, latence, durabilité

Rétention des messages



Introduction

Chaque partition peut être répliquée.

Différents brokers participent à la gestion distribuée des répliques.

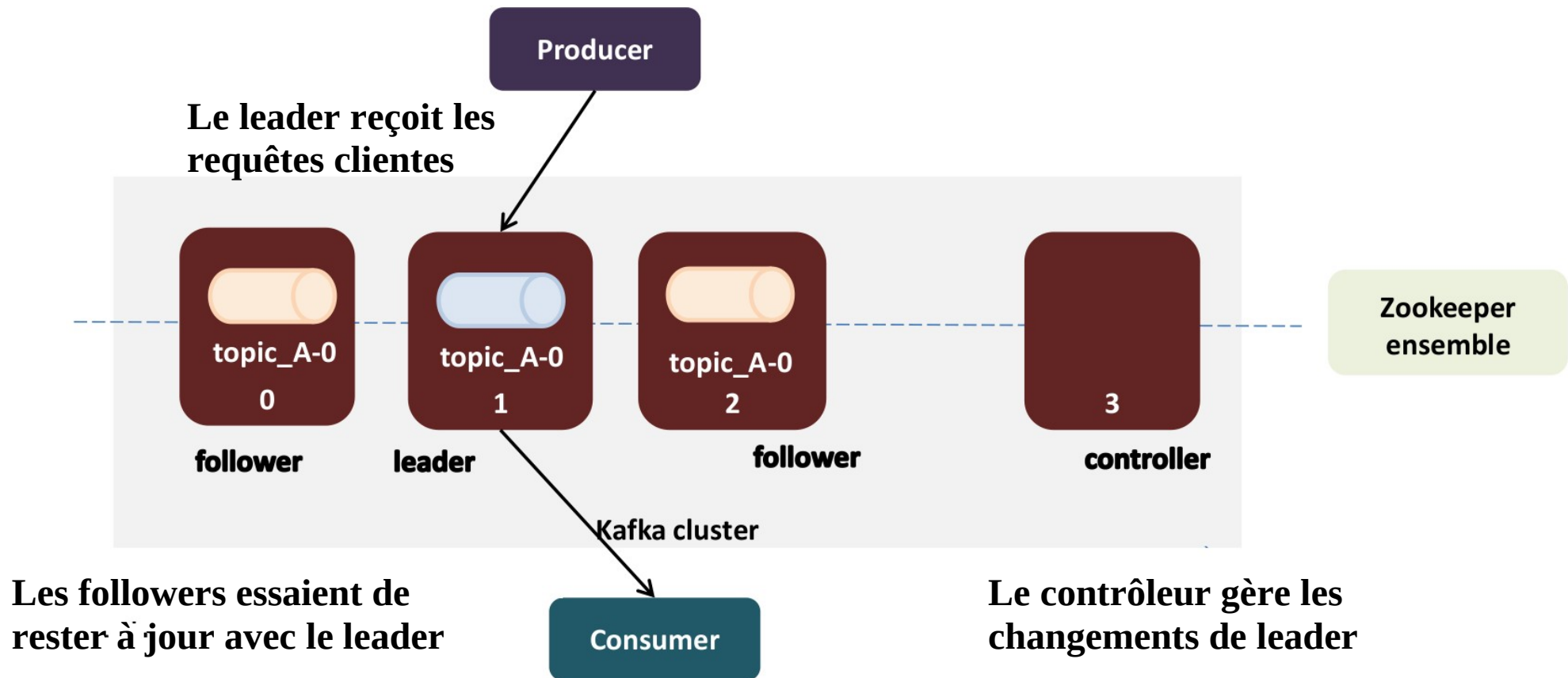
Pour chaque partition d'un topic :

- **1 broker leader** : Détient la partition leader responsable de la validation des écritures
- **N followers** : Suivent les écritures effectuées par le leader.
Un décalage est permis mais si il n'arrive plus à suivre la cadence d'écriture, ils sont éliminés

Au niveau du cluster :

- Un **contrôleur** est responsable des élections de leader

Rôle des brokers gérant un topic





Garanties Kafka

Garanties offertes par Kafka grâce à la réplication :

- Garantie de l'ordre à l'intérieur d'une partition.
=> Le consommateur d'une partition lit dans l'ordre d'écriture des messages
- Les messages produits sont considérés **validés (committed)** lorsqu'ils sont écrits sur la partition leader et qu'ils ont atteint le minimum de réplication, i.e. minimum de répliques synchronisées
- Les messages validés restent disponibles tant qu'au moins une réplique reste en vie.
- Les consommateurs ne peuvent lire que les messages validés.



Synchronisation des répliques

Contrôlé par la propriété :

replica.lag.time.max.ms (défaut 30 000)

Si pendant ce délai, le follower

- N'envoie pas de requêtes fetch
- N'atteint pas l'offset de fin du leader

Alors, le follower est considéré comme désynchronisé

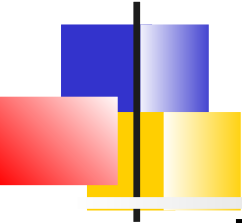
- Il est supprimé de la liste des **ISR (In Sync Replica)**
- Il peut ensuite rattraper son retard et être réintégré aux ISRs



min.insync.replica

La propriété ***min.insync.replica***, spécifiée au niveau cluster ou topic, indique le minimum de répliques de l'ISR (incluant le leader) qui doivent avoir écrit un message afin que celui-ci soit considéré comme *committed/validé*

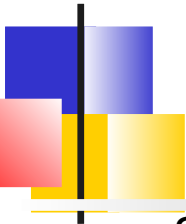
- A la réception d'un message, le leader vérifie si il y a assez d'ISR pour écrire le message, sinon il envoie une *NotEnoughReplicasException*
- Lorsque le message est répliqué par *min.insync.replica* répliques, le leader envoie un acquittement au client.



Conséquences

Un réplique synchronisée légèrement en retard peut ralentir l'acquittement du message ; ce qui peut ralentir le débit.

Une réplique désynchronisée n'a plus d'impact sur les performances mais augmente le risque d'un temps d'arrêt ou d'une perte de données.



Rejet de demande d'émission

Si le **nombre de ISR** < ***min.insync.replicas*** :

- Kafka empêche l'acquittement complet du message.
En fonction de la configuration du producteur, celui-ci peut être bloqué.

Si le **nombre de répliques disponible** < ***min.insync.replicas***

- Mauvaise configuration, Kafka bloque les émissions de message

En conséquences :

n répliques

=> tolère $n-1$ failures pour que la partition soit disponible

n répliques et $\text{min.insync.replicas} = m$

=> Tolère $n-m$ failures pour accepter les envois



Configuration

2 principales configurations affectent la fiabilité et les compromis liés :

- ***default.replication.factor (au niveau cluster)*** et ***replication.factor (au niveau topic)***

Compromis entre disponibilité et matériel requis

Valeur classique pour la fiabilité : 3

- ***min.insync.replicas*** (défaut 1, au niveau cluster ou topic)

Le minimum de répliques qui doivent acquitter pour valider la production d'un message

Compromis entre fiabilité et ralentissement

Valeur classique : 2/3



Configuration cluster et topics

Mécanismes de réplication

Garanties de livraison

Débit, latence, durabilité

Rétention des messages



Introduction

En fonction des configurations du topic, des producteurs et des consommateurs, Kafka peut garantir différents niveaux de livraison malgré un certain nombre de défaillances :

- **At Most Once** : Un message est livré au plus une fois.
On tolère les pertes de message
- **At Least Once** : Le message est livré au moins une fois.
On tolère les doublons
- **Exactly once** : Le message est livré 1 et 1 seule fois.



Côté producteur

Du côté producteur, 2 facteurs influencent la fiabilité de l'émission

- La configuration des **acks** en fonction de *min.insync.replica* du topic
3 valeurs possibles : *0,1,all*
- Les gestion des **erreurs** dans la configuration et dans le code



acks=0

acks=0 : Le producteur considère le message écrit au moment où il l'a envoyé sans attendre l'acquittement du broker
=> Perte de message potentielle (*At Most Once*)





acks=1

acks=1 : Le producteur considère le message écrit lorsque le leader a acquitté l'écriture

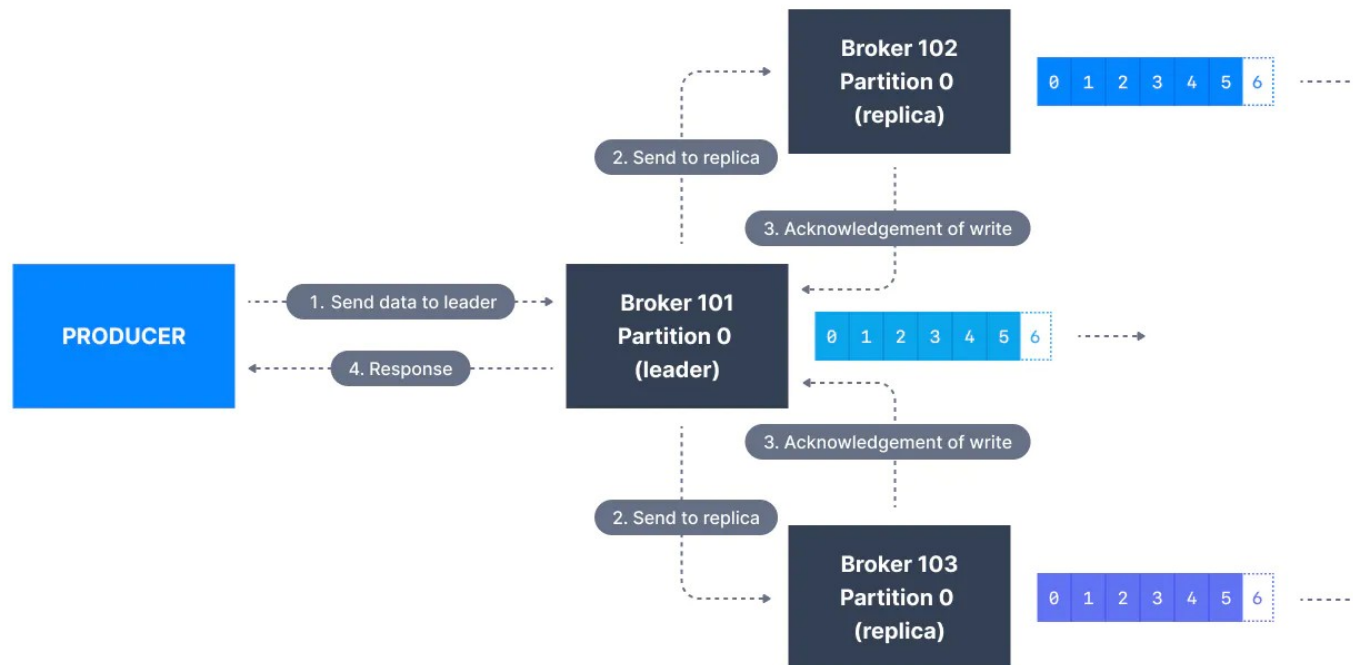
=> Si le leader s'arrête et que les répliques n'ont pas eu le temps d'écrire le message, perte de données

Garantie : *At Most Once*



acks=all

acks=all : Le producteur considère le message comme écrit lorsque il a été répliqué par *min.insync.replica* des ISR.
=> Assure le maximum de durabilité
(Nécessaire pour *At Least Once* et *Exactly Once*)





Gestion des erreurs

2 types d'erreur pour le producteur:

- les erreurs que les producteurs gèrent automatiquement .
Ce sont les erreurs ré-essayable
(ex : LEADER_NOT_AVAILABLE,
NOT_ENOUGH_REPLICA)
Nombre d'essai configurable via **retries**.
=> Attention, peut générer des doublons
- les erreurs qui doivent être traitées par le code.
(ex : INVALID_CONFIG,
SERIALIZATION_EXCEPTION)



Côté consommateur

Du point de vue de la fiabilité, les consommateurs doivent s'assurer qu'ils gardent une trace des offsets qu'ils ont traités en cas de rééquilibrage des consommateurs.

Pour cela, ils committent leur offset auprès du cluster Kafka qui stocke les informations dans le topic **_consumer_offsets**

=> La perte de messages peut se produire si l'on a committé avant le traitement du message

=> Les traitements en doublon peuvent se produire si l'on a committé après le traitement du message



Configuration

Les propriétés de configuration importantes pour les garanties de livraison du consommateur consommateur :

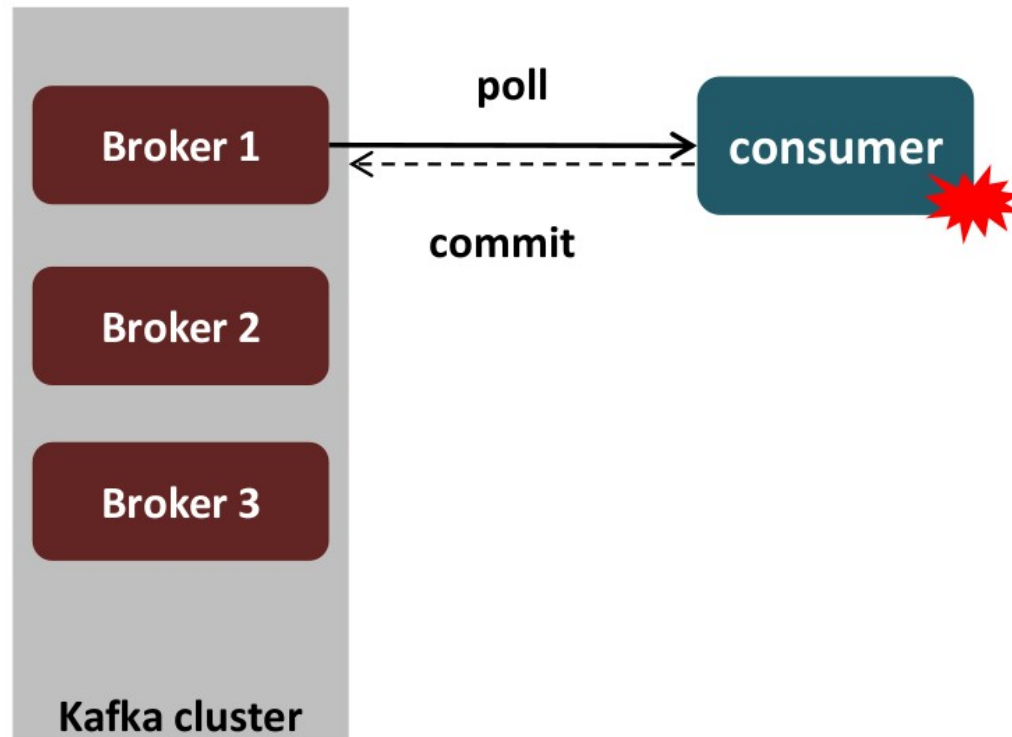
enable.auto.commit : Commit manuel ou non

- Si true : ***auto.commit.interval.ms*** : Intervalle des commits

auto.offset.reset : Contrôle le comportement du consommateur lorsqu'aucun offset est commité ou lorsqu'il demande un offset qui n'existe pas

- ***earliest*** : Le consommateur repart au début, garantie une perte minimale de message mais peut générer beaucoup de traitements en doublon
- ***latest*** : Minimise les doublons mais risque de louper des messages

Scénario At Most Once



- L'offset est commité
- Traitement d'un ratio de message puis plantage



Configuration *At Most Once*

***enable.auto.commit = true* + Traitement asynchrone dans la boucle de poll**

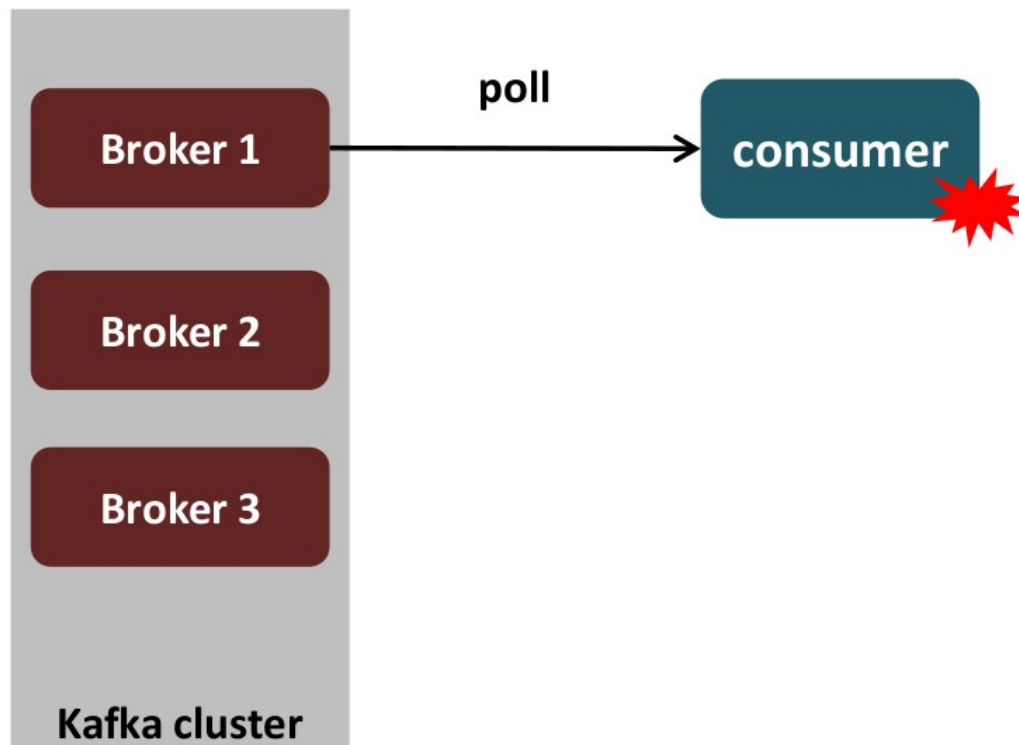
OU

Commit manuel avant le traitement des messages

Exemple : Auto-commit et traitement asynchrone

```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
    long lastOffset = 0;  
    for (ConsumerRecord<String, String> record : records) {  
        System.out.printf("\n\roffset = %d, key = %s, value = %s", record.offset(), record.key(),  
            record.value());  
        lastOffset = record.offset();  
    }  
    System.out.println("lastOffset read: " + lastOffset);  
    // Traitement asynchrone  
    process();  
}
```


Scénario: *At Least Once*



- Traitement d'un ratio de message puis plantage
- L'offset n'est pas commité



Configuration *At Least Once*

Configuration par défaut avec traitement synchrone dans la boucle de poll

Ou

***enable.auto.commit* = false ET Commit explicite après traitement via *consumer.commitSync()*;**

Exemple : Commit manuel après traitement

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    long lastOffset = 0;

    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("\n\roffset = %d, key = %s, value = %s", record.offset(),
            record.key(), record.value());
        lastOffset = record.offset();
    }
    System.out.println("lastOffset read: " + lastOffset);
    process();
    consumer.commitSync();
}
```

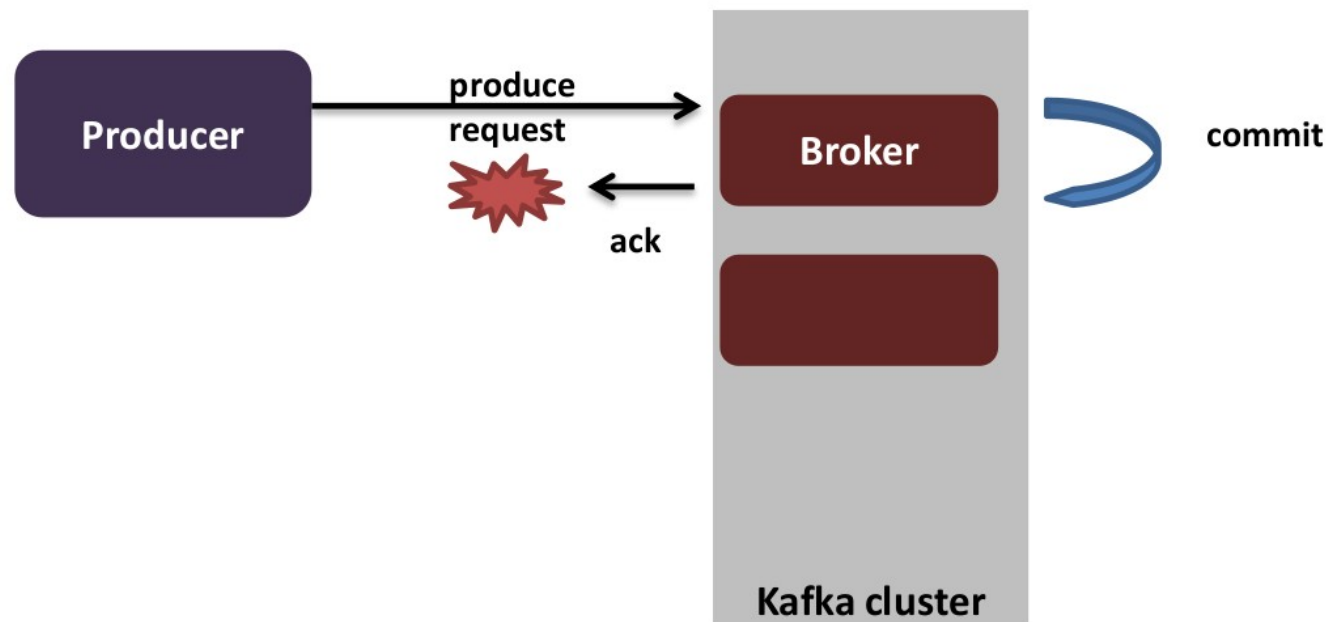


Exactly Once

La sémantique *Exactly Once* est basée sur *At least Once* (défaut) et empêche les messages en double en cours de traitement par les applications clientes

Pris en charge par les APIs *Kafka Producer, Consumer et Streams*

Producteur idempotent



Le producteur ajoute une
nombre séquentiel et un ID
de producteur

Le broker détecte le
doublon
=> Il envoie un ack sans le
commit



Configuration

enable.idempotence = true

L'idempotence nécessite les configurations suivantes :

- *max.in.flight.requests.per.connection* ≤ 5
- *retries* > 0
- *acks = all*

Si des valeurs incompatibles sont configurées, une *ConfigException* est lancée



Consommateur

Du côté du consommateur, traiter une et une seule fois les messages consistent à :

- gérer manuellement les offsets des partitions dans un support de persistance transactionnel, partagé par tous les consommateurs et gérer les rééquilibrages
- Si le traitement consiste à produire un message vers le même cluster Kafka, on peut utiliser les transactions Kafka¹.

1. C'est le cas de KafkaStream



Exemple

enable.auto.commit=false

```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
    for (ConsumerRecord<String, String> record : records) {  
        System.out.printf("offset = %d, key = %s, value = %s\n",  
            record.offset(),  
            record.key(), record.value());  
  
        // Sauvegarder l'offset traité .  
        offsetManager.saveOffsetInExternalStore(record.topic(),  
            record.partition(), record.offset());  
    }  
}
```



Example (2)

```
public class MyConsumerRebalancerListener implements
    org.apache.kafka.clients.consumer.ConsumerRebalanceListener {

    private OffsetManager offsetManager = new OffsetManager("storage2");
    private Consumer<String, String> consumer;

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        for (TopicPartition partition : partitions) {
            offsetManager.saveOffsetInExternalStore(partition.topic(),
                partition.partition(), consumer.position(partition));
        }
    }

    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        for (TopicPartition partition : partitions) {
            consumer.seek(partition,
                offsetManager.readOffsetFromExternalStore(partition.topic(),
                    partition.partition()));
        }
    }
}
```




Transaction

Les transactions Kafka permettent la production atomique d'un lot de messages

La configuration consiste à la propriété ***transactional.id*** à une chaîne de caractères unique par producteur

- Le mode transactionnel inclut le mode idempotent
- Les topics impliqués doivent être configurés avec *replication.factor* ≥ 3 et *min.insync.replicas* = 2
- L'API *send()* devient bloquante



Exemple

```
// initTransaction démarre le mode transactionnel
producer.initTransactions();
Try {
    // délimitation des envois transactionnels
    producer.beginTransaction();
    for (int i = 0; i < 100; ++i) {
        ProducerRecord record = new ProducerRecord("topic_1", null,
i);
        producer.send(record);
    }
    producer.commitTransaction();
} catch(ProducerFencedException e) { producer.close(); }
    catch(KafkaException e) { producer.abortTransaction(); }
```



Configuration du consommateur

Afin de ne lire que les messages transactionnels validés, les consommateurs doivent modifier la configuration par défaut :

– ***isolation.level=read_committed***
(Défaut: *read_uncommitted*)

- *read_committed*: Messages (transactionnels ou non) validés
- *read_uncommitted*: Tous les messages (même les messages transactionnels non validés)

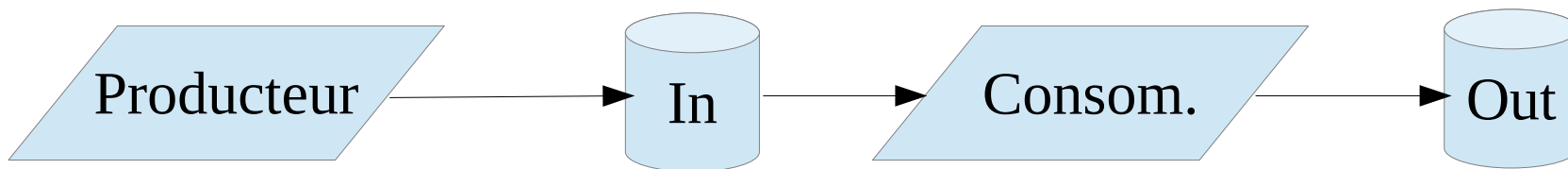


Sémantique Exactly Once

Lorsque le traitement d'un consommateur consiste à produire un résultat vers un autre topic, on peut utiliser les transactions pour garantir une livraison *Exactly Once*

afin d'écrire l'offset vers Kafka dans la même transaction que le topic de sortie

Si la transaction est abandonnée, la position du consommateur reviendra à son ancienne valeur et les données produites sur les topics de sortie ne seront pas visibles pour les autres consommateurs, en fonction de leur "niveau d'isolement".





Producteur

```
producer.initTransactions();
```

```
try {  
    producer.beginTransaction();  
    Stream.of(DATA_MESSAGE_1, DATA_MESSAGE_2)  
        .forEach(s -> producer.send(new ProducerRecord<String, String>("input", null, s)));  
    producer.commitTransaction();  
} catch (KafkaException e) {  
    producer.abortTransaction();  
}
```



Consommateur

```
while (true) {  
  
    ConsumerRecords<String, String> records = consumer.poll(ofSeconds(60));  
  
    // Transform message  
    Map<String, Integer> wordCountMap = ...  
  
    producer.beginTransaction();  
  
    wordCountMap.forEach((key, value) ->  
        producer.send(new ProducerRecord<String, String>(OUTPUT_TOPIC, key, value.toString())));  
  
    Map<TopicPartition, OffsetAndMetadata> offsetsToCommit = new HashMap<>();  
  
    // Retrieve offsets for each partition  
    for (TopicPartition partition : records.partitions()) {  
        List<ConsumerRecord<String, String>> partitionedRecords = records.records(partition);  
        long offset = partitionedRecords.get(partitionedRecords.size() - 1).offset();  
        offsetsToCommit.put(partition, new OffsetAndMetadata(offset + 1));  
    }  
    // Commit Offset for consumer associated with the commit of the transaction  
    producer.sendOffsetsToTransaction(offsetsToCommit, CONSUMER_GROUP_ID);  
    producer.commitTransaction();  
}
```



Configuration cluster et topics

Mécanismes de réplication
Garanties de livraison
Débit, latence, durabilité
Rétention des messages



Configuration pour favoriser le débit

Côté producteur :

Augmenter

- *batch.size*: ex : 100000 (défaut 16384)
- *linger.ms*: ex : 100 (default 0)
- *buffer.memory*: si il y a beaucoup de partitions (défaut 32 Mb)

Puis

- *compression.type*=lz4 (défaut none)
- *acks*=1 (défaut all)

Côté consommateur

Augmenter

- *fetch.min.bytes*: ex : 100000 (défaut 1)



Configuration pour favoriser la latence

Cluster

Si followers pas suffisamment rapide, augmenter leur nombre

- *num.replica.fetchers* : (défaut 1)

Côté producteur

- *linger.ms*: 0
- *compression.type*=none
- *acks*=1

Côté consommateur

- *fetch.min.bytes*: 1



Configuration pour la durabilité

Cluster

- *replication.factor*: 3
- *min.insync.replicas*: 2 (défaut 1)
- *unclean.leader.election.enable* : false (défaut false)
- *broker.rack*: rack du broker (défaut null)

Producteur

- *acks:all* (défaut all)
- *enable.idempotence:true* (défaut false)
- *max.in.flight.requests.per.connection*: ≤ 5

Consommateur

- *isolation.level: read_committed*



Configuration cluster et topics

Mécanismes de réplication
Garanties de livraison
Débit, latence, durabilité
Rétention des messages



Introduction

L'unité de stockage de Kafka est une réplique de partition.

- => Les partitions ne peuvent pas être divisées entre plusieurs brokers ni entre plusieurs disques du même broker

La propriété ***log.dirs*** définit les répertoires de stockage des partitions



Allocation des partitions

A la création des *topics*, Kafka décide comment allouer les partitions sur les brokers

Ses objectifs sont :

- Répartir uniformément les répliques entre les brokers
- S'assurer que chaque réplique d'une partition se trouve sur un broker différent
- Si les brokers ont des informations sur le rack, s'assurer que les répliques sont affectés à des racks différents si possible



Rétention des données

L'administrateur Kafka configure une période de rétention pour chaque topic

- Soit une durée
- Soit un volume

Pour accélérer la purge, Kafka utilise les ***segments***

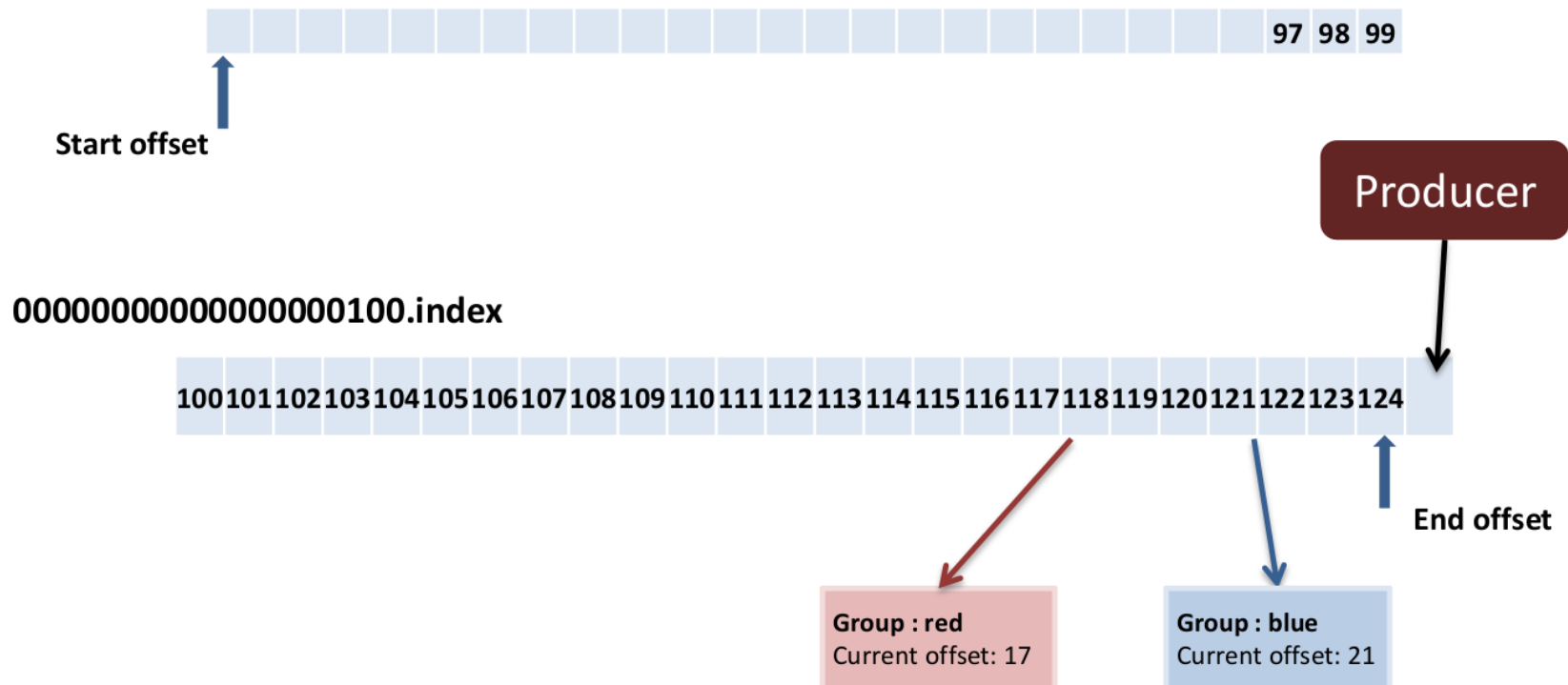
- Les segments sont des fragments de partition au maximum de 1Go et au maximum d'une semaine de données
- Lors de l'écriture d'un segment, lorsque la limite est atteinte, le broker ferme le fichier. Il devient éligible pour la purge
- Il existe donc un seul *segment actif* qui d'ailleurs ne peut pas être purgé

Segments

Log segment

closing segment parameters :

- log.roll.ms (ignored if log.roll.hours is set)
- log.roll.hours (default 168)
- log.segment.bytes (default 1073741824)





Indexation

Kafka permet de récupérer des messages à partir de n'importe quel offset disponible.

Pour accélérer cette opération, Kafka maintient un ***index*** pour chaque partition

- L'index associe les offsets aux segments et aux positions dans le fichier
- Les index sont également divisés en segments
- En cas de suppression, ils peuvent être régénérés automatiquement par Kafka



Principales configurations

log.retention.hours (défaut 168 : 7 jours),

log.retention.minutes (défaut null),

log.retention.ms (défaut null, si -1 infini)

Période de rétention des vieux segment avant de les supprimer

log.retention.bytes (défaut -1)

La taille maximale du log

offsets.retention.minutes (défaut 10080 : 7 jours)

Le nombre de minutes pour conserver l'index d'offset d'un groupe n'ayant plus de consommateurs



Nettoyage des logs

Propriété ***log.cleanup.policy***, 2 stratégies disponibles :

- ***delete*** (défaut) :
 - Suppression des vieux segments en fonction de l'âge et la taille du log (partition)
 - Meilleur contrôle de l'espace disque
- ***compact***
 - Suppression basée sur les clés des messages quelque soit l'âge et la taille des données

Possibilité de combiner les 2 stratégies (*delete* et *compact*)



Stratégie compact

2 propriétés de configuration supplémentaire pour cette stratégie :

- ***cleaner.min.compaction.lag.ms*** : Le temps minimum qu'un message reste non compacté
- ***cleaner.max.compaction.lag.ms*** : Le temps maximum qu'un message reste inéligible pour la compactage

Lors du nettoyage, le segment actif est fermé (un nouveau segment est créé)

Sauvegarde la dernière valeur (dernière mise à jour) pour chaque clé

=> Consomme CPU et RAM



Exemple *compact*

1	2	3	4	5	6	7
K1	K2	K3	K4	K4	K5	K1
6	3	2	2	1	3	2



2	3	5	6	7
K2	K3	K4	K5	K1
3	2	1	3	2



Spring-Kafka

Introduction

Production de message

Consommation

Transaction

Sérialisation / Désérialisation

Traitement des exceptions



Introduction

Le starter *SpringKafka* apporte :

- Les concepts Spring Ioc et DI
- Les auto-configurations SpringBoot
- Un template pour envoyé les messages
- @KafkaListener sur des POJOs pour recevoir les messages
- Une similarités avec les autres intégrations de MessageBroker, en particulier RabbitMQ et JMS



Connexions à Kafka

Spring permet 3 types de client aux clusters Kafka

- ***KafkaAdmin*** : Utilisé principalement pour créer et configurer les topics
- ***ProducerFactory*** : Permettant de configurer la production de messages
- ***ConsumerFactory*** : Permettant de configurer la consommation

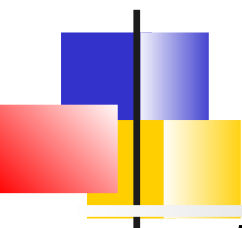
SpringBoot configure automatiquement un instance de ces classes en s'appuyant sur les propriétés de configuration



Configuration des topics

Si un bean *KafkaAdmin* est disponible (automatique avec SpringBoot), il est possible de définir des beans ***NewTopic*** qui permettent la création automatique de topic

```
@Bean
public NewTopic topic2() {
    return TopicBuilder.name("thing2")
        .partitions(10)
        .replicas(3)
        .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
        .build();
}
```

KafkaAdmin et AdminClient

KafkaAdmin fournit les méthodes pour créer et examiner des topics : *createOrModifyTopics()*, *describeTopics()*

On peut également accéder directement à la classe *Kafka AdminClient* pour des fonctionnalités plus avancées

```
@Autowired
private KafkaAdmin admin;

...
AdminClient client =
    AdminClient.create(admin.getConfigurationProperties());
    ...
    client.close()
```



Listeners

Les *factories* pour les producteurs et les consommateurs de messages peuvent être configurés avec des Listeners

```
interface Listener<K, V> {  
    default void producerAdded(String id, Producer<K, V> producer) {}  
    default void producerRemoved(String id, Producer<K, V> producer) {}  
}
```

```
interface Listener<K, V> {  
  
    default void consumerAdded(String id, Consumer<K, V> consumer) {}  
    default void consumerRemoved(String id, Consumer<K, V> consumer) {}  
}
```



Spring-Kafka

Introduction

Production de message

Consommation

Transaction

Sérialisation / Désérialisation

Traitement des exceptions



Production de messages

Différents objets *Template* sont fournis pour l'émission de messages

- ***KafkaTemplate*** encapsulant un *KafkaProducer*
- ***RoutingKafkaTemplate*** permet de sélectionner un *KafkaProducer* en fonction du nom du *Topic*
- ***ReplyingKafkaTemplate*** permettant des interactions Request/Response
- ***DefaultKafkaProducerFactory*** pouvant être utilisé pour gérer plus finement les templates



Méthodes d'envoi

KafkaTemplate propose 2 méthodes d'envoi retournant un ***CompletableFuture<SendResult>*** :

- ***sendDefault*** qui nécessite d'avoir défini un topic par défaut
- ***send*** ou le topic est fourni en argument

Différentes signatures sont possible pour ces 2 méthodes :

- Juste la donnée
- La donnée et la clé
- La donnée, la clé et le timestamp
- *ProducerRecord<K, V>* record de Kafka;
- *Message<?>* message de spring-integration



Configuration sans SpringBoot

@Bean

```
public ProducerFactory<Integer, String> producerFactory() {  
    return new DefaultKafkaProducerFactory<>(producerConfigs());  
}
```

@Bean

```
public Map<String, Object> producerConfigs() {  
    Map<String, Object> props = new HashMap<>();  
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");  
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class);  
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);  
  
    return props;  
}
```

@Bean

```
public KafkaTemplate<Integer, String> kafkaTemplate() {  
    return new KafkaTemplate<Integer, String>(producerFactory());  
}
```



Avec SpringBoot

SpringBoot configure automatiquement un *ProducerFactory* (***DefaultKafkaProducerFactory***) à partir des propriétés trouvées dans *application.yml*

```
spring:
  kafka:
    bootstrap-servers:
      - localhost:9092
    producer:
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
      key-serializer: org.apache.kafka.common.serialization.IntegerSerializer
```

Il suffit alors de s'injecter un *KafkaTemplate* (singleton par défaut partagé par toutes les threads)

@Autowired

```
KafkaTemplate<Integer, String> kafkaTemplate
```



Réponse à l'envoi

La réponse est encapsulée dans un *CompletableFuture* permettant différents types d'interaction

- Fire and Forget
- Synchrone

```
CompletableFuture<SendResult<Integer, String>> future = template.send("myTopic", "something");  
SendResult result = future.get();
```

- Asynchrone avec call-back

```
CompletableFuture<SendResult<Integer, String>> future = template.send("myTopic", "something");  
future.whenComplete((result, ex) -> {  
    ...  
});
```

SendResult encapsule les classes de Kafka

- *ProducerRecord* : L'enregistrement envoyé
- *RecordMetadata* : Partition, offset, timestamp, objets sérialisés



ProducerListener

On peut également associer un ***ProducerListener*** au template via sa méthode *setProducerListener()*

Les méthodes de call-back seront alors appelées pour tous les envois du Template

```
public interface ProducerListener<K, V> {  
    void onSuccess(ProducerRecord<K, V> producerRecord, RecordMetadata  
        recordMetadata);  
  
    void onError(ProducerRecord<K, V> producerRecord, RecordMetadata  
        recordMetadata, Exception exception);  
}
```

Par défaut, *KafkaTemplate* est configuré avec un *LoggingProducerListener*, qui trace les erreurs et ne fait rien lorsque l'envoi réussit.



RoutingKafkaTemplate

```
@Bean
```

```
public RoutingKafkaTemplate routingTemplate(GenericApplicationContext context, ProducerFactory<Object, Object> pf) {
```

```
    // Cloner le ProducerFactory par défaut avec un sérialiseur différent
```

```
    Map<String, Object> configs = new HashMap<>(pf.getConfigurationProperties());
```

```
    configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, ByteArraySerializer.class);
```

```
    DefaultKafkaProducerFactory<Object, Object> bytesPF = new DefaultKafkaProducerFactory<>(configs);
```

```
    context.registerBean(DefaultKafkaProducerFactory.class, "bytesPF", bytesPF);
```

```
    Map<Pattern, ProducerFactory<Object, Object>> map = new LinkedHashMap<>();
```

```
    map.put(Pattern.compile("two"), bytesPF); // Topic « two » utilise le ByteSerializer
```

```
    map.put(Pattern.compile(".*"), pf); // Défaut : StringSerializer
```

```
    return new RoutingKafkaTemplate(map);
```

```
}
```

```
@Bean
```

```
public ApplicationRunner runner(RoutingKafkaTemplate routingTemplate) {
```

```
    return args -> {
```

```
        routingTemplate.send("one", "thing1");
```

```
        routingTemplate.send("two", "thing2".getBytes());
```

```
    };
```

```
}
```

```
}
```



ProducerFactory

Par défaut, *DefaultKafkaProducerFactory* crée un producer singleton

- La propriété *producerPerThread* permet de créer un Producer par thread

Il peut également être utile d'accéder à *ProducerFactory* pour mettre à jour dynamiquement sa configuration via :

- `updateConfigs(Map<String, Object> updates)`
- `void removeConfig(String configKey);`

Utile pour les rotations de clés par exemple



ReplyingKafkaTemplate

Sous-classe de *KafkaTemplate* permettant un mode requête/réponse :

- Un message est envoyé sur un Topic *request*
- Un consommateur répond sur un Topic *Response*

Lors de son instanciation, un *GenericMessageListenerContainer* représentant un consommateur du topic *Response* est fournie.

L'envoi de message peut alors se faire via un des 2 méthodes supplémentaires :

```
RequestReplyFuture<K, V, R> sendAndReceive(ProducerRecord<K, V> record);  
RequestReplyFuture<K, V, R> sendAndReceive(ProducerRecord<K, V> record, Duration  
    replyTimeout);
```

La valeur de retour est un *Completable* renseigné de façon asynchrone qui contient :

- La clé
- La valeur envoyée
- La réponse du consommateur du Topic Request



Exemple (1)

Création des beans

@Bean

```
public ReplyingKafkaTemplate<String, String, String> replyingTemplate(  
    ProducerFactory<String, String> pf,  
    ConcurrentMessageListenerContainer<String, String> repliesContainer) {  
    return new ReplyingKafkaTemplate<>(pf, repliesContainer);  
}
```

@Bean

```
public ConcurrentMessageListenerContainer<String, String> repliesContainer(  
    ConcurrentKafkaListenerContainerFactory<String, String> containerFactory) {  
    ConcurrentMessageListenerContainer<String, String> repliesContainer =  
    containerFactory.createContainer("kReplies");  
    repliesContainer.getContainerProperties().setGroupId("repliesGroup");  
    repliesContainer.setAutoStartup(false);  
    return repliesContainer;  
}
```



Exemple (2)

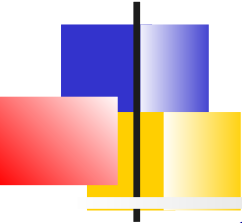
Envoi de message

@Bean

```
public ApplicationRunner runner(ReplyingKafkaTemplate<String, String, String> template) {
    return args -> {
        if (!template.waitForAssignment(Duration.ofSeconds(10))) {
            throw new IllegalStateException("Reply container did not initialize");
        }
        ProducerRecord<String, String> record = new ProducerRecord<>("kRequests", "foo");
        RequestReplyFuture<String, String, String> replyFuture = template.sendAndReceive(record);
        // Vérification de l'envoi de la requête
        SendResult<String, String> sendResult = replyFuture.getSendFuture().get(10,
                                                                                     TimeUnit.SECONDS);

        System.out.println("Résultat de l'envoi ok: " + sendResult.getRecordMetadata());
        // Lecture de la valeur de la réponse
        ConsumerRecord<String, String> consumerRecord = replyFuture.get(10, TimeUnit.SECONDS);
        System.out.println("Valeur de la réponse : " + consumerRecord.value());
    };
}
```

Exemple (3)



```
@KafkaListener(id="server", topics = "kRequests")
@SendTo
public String listen(String in) {
    System.out.println("Server received: " + in);
    return in.toUpperCase();
}
```

Le topic de réponse est indiqué dans l'entête
KafkaHeaders.REPLY_TOPIC.

L'annotation *@SendTo* a pour effet de renvoyer sur le topic de réponse la valeur retournée par la méthode



Spring-Kafka

Introduction
Production de message
Consommation
Transaction
Sérialisation / Désérialisation
Traitement des exceptions



Introduction

Pour consommer des messages, il est nécessaire de définir :

- ***MessageListenerContainer*** : Définit le modèle de threads des listeners
- ***MessageListener*** : Mode de réception des messages

L'annotation ***@KafkaListener*** sur une méthode permet d'associer une méthode d'un POJO à un *MessageListener*



MessageListener

MessageListener est l'interface permettant de consommer les *ConsumerRecords* de Kafka retournés par la boucle poll

Il offre différentes alternatives :

- Traitement individuel ou par lot
- Commit géré par le container ou manuel via la classe ***Acknowledgment***
- Accès au *KafkaConsumer* sous-jacent



Interfaces *MessageListener*

// Traitement individuel des ConsumerRecord retournée par poll() de Kafka : commit gérées par le container

MessageListener<K, V> : onMessage(ConsumerRecord<K, V>);

ConsumerAwareMessageListener<K, V> : onMessage(ConsumerRecord<K, V>, Consumer<?, ?>);
}

// Traitement indivisuel avec commit manuel (Acknowledgment)

AcknowledgingMessageListener<K, V> : onMessage(ConsumerRecord<K, V>, Acknowledgment);

AcknowledgingConsumerAwareMessageListener<K, V> : onMessage(ConsumerRecord<K, V>, Acknowledgment, Consumer<?, ?>);

//Traitement par lot avec commit géré par container

BatchMessageListener<K, V> : onMessage(List<ConsumerRecord<K, V>>);

BatchConsumerAwareMessageListener<K, V> : onMessage(List<ConsumerRecord<K, V>>, Consumer<?, ?>);

//Traitement par lot avec commit manuel

BatchAcknowledgingMessageListener<K, V> : onMessage(List<ConsumerRecord<K, V>>, Acknowledgment);

BatchAcknowledgingConsumerAwareMessageListener<K, V> :
onMessage(List<ConsumerRecord<K, V>>, Acknowledgment, Consumer<?, ?>);}



MessageListenerContainer

MessageListenerContainer contient la configuration de la connexion à Kafka, et les topics ou partitions à consommer.
Il effectue la boucle de poll()

2 implémentations sont fournies :

- ***KafkaMessageListenerContainer*** :
Implémentation mono-thread
- ***ConcurrentMessageListenerContainer*** :
Implémentation multi-thread permettant d'ajuster le nombre de threads au nombre de partitions du Topic



Instanciación

Le constructeur des 2 implémentations de *MessageListenerContainer* nécessite 2 arguments :

- ***ConsumerFactory<K, V>*** : Encapsule les propriétés Kafka
- ***ContainerProperties*** : Encapsule les informations sur les topics ou partition à consommer

Constructeur *ContainerProperties* :

```
public ContainerProperties(TopicPartitionOffset... topicPartitions)
public ContainerProperties(String... topics)
public ContainerProperties(Pattern topicPattern)
```

ContainerProperties propose également la méthode *setMessageListener()* permettant d'associer le listener au Container



ConcurrentMessageListener

ConcurrentMessageListenerContainer a une propriété **concurrency** supplémentaire déterminant le degré de multi-threading :

- A chaque thread est associé une propriété ***client-id = <préfixe>-<n>***
- A chaque thread est associé 1 ou plusieurs partitions avec la distribution par défaut de Kafka
 - Peut être influencé par la propriété :
spring.kafka.consumer.properties.partition.assignment.strategy



Commits

Différentes options pour la gestion des commits :

- Si ***enable.auto.commit = true***. On s'appuie sur les commits automatiques de Kafka
- Sinon (défaut), on s'appuie sur les commits Spring via l'énumération ***AckMode***
Il propose de nombreuses options qui influent sur le nombre de commits et donc sur des compromis entre performance et risque



AckMode

RECORD : Commit au retour du listener => à chaque enregistrement

BATCH (Défaut) : Commit lorsque tous les enregistrements renvoyés par poll() ont été traités.

TIME : Commit l'offset du dernier poll si ackTime a été dépassé depuis la dernière validation.

COUNT : Commit l'offset du dernier poll si ackCount messages ont été reçus depuis la dernière validation.

COUNT_TIME : similaire à TIME et COUNT, le commit est effectué si l'une ou l'autre des conditions est vraie.

MANUAL : Le messageListener est responsable d'effectuer le commit via la méthode *Acknowledgement.acknowledge()*. Le commit vers Kafka est envoyé avec BATCH

MANUAL_IMMEDIATE : Commit effectué lorsque la méthode *Acknowledgement.acknowledge()* est appelée par le *messageListener*.



Acknowledgment

L'interface **Acknowledgment** propose donc 3 méthodes :

- ***void acknowledge()*** : Utilisé pour le commit manuel en mode individuel ou BATCH
- ***default void nack(int index, Duration sleep)*** : Acknowledge négatif d'un index d'un batch
=> Commite les offsets avant l'index puis repositionne à index+1 après le sleep
- ***default void nack(Duration sleep)*** : Acknowledge négatif du record courant
=> Se repositionne après l'enregistrement



Configuration sans SpringBoot

```
@Bean
KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Integer, String>>
    kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setConcurrency(3);
    factory.getContainerProperties().setPollTimeout(3000);
    return factory;
}

@Bean
public ConsumerFactory<Integer, String> consumerFactory() {
    return new DefaultKafkaConsumerFactory<>(consumerConfigs());
}

@Bean
public Map<String, Object> consumerConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, embeddedKafka.getBrokersAsString());
    ...
    return props;
}
```



Configuration SpringBoot

Les valeurs de configuration relatives à *DefaultConsumerFactory* sont fixées par la propriété ***spring.kafka.consumer*** :

- *bootstrap-servers, group-id, Deserialisers, enable-auto-commit, auto-offser-reset, ...*

Les valeurs relatives influant le container et l'implémentation de *MessageListener* sont fixées par ***spring.kafka.listener*** :

- *type : batch | single*
- *concurrency, client-id, log-container-config*
- *ack-mode, ack-count, ack-time*
- *poll-timeout, idle-between-poll*
- *monitor-poll*



Exemple

```
spring:
  kafka:
    consumer:
      group-id: consumer
      bootstrap-servers:
        - localhost:9092
        - localhost:9192
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.formation.model.JsonDeserializer
      enable-auto-commit: true

    listener:
      concurrency: 3 # Container concurrent
      log-container-config: true
```



@KafkaListener

L'annotation **@KafkaListener** permet de désigner une méthode comme *MessageListener*.

Ses attributs¹ surchargeant ceux de *application.yml* dans le cas de SpringBoot permettent de spécifier :

- Le(s) topics à écouter ou même plus précisément les partitions et les offsets
- L'identité du groupe de consommateur
- Le préfixe des clientId
- La propriété *concurrency*
- N'importe quelle propriété

La méthode supporte différents arguments :

- Données ou *ConsumerRecord*
- *Acknowledgment* : Commits manuels
- Entêtes du message, l'argument doit être annoté

1. Supporte les expressions SpEL "#{someBean.someProperty}" ou les propriétés "\${some.property}"



Examples

```
@KafkaListener(id = "myListener", topics = "myTopic", concurrency = "${listen.concurrency:3}")
public void listen(String data) {
```

```
@KafkaListener(id = "thing2", topicPartitions =
    { @TopicPartition(topic = "topic1", partitions = { "0", "1" }),
      @TopicPartition(topic = "topic2", partitions = "0",
        partitionOffsets = @PartitionOffset(partition = "1", initialOffset = "100"))  })
public void listen(ConsumerRecord<?, ?> record) {
```

```
@KafkaListener(id = "cat", topics = "myTopic", containerFactory =
    "kafkaManualAckListenerContainerFactory")
public void listen(String data, Acknowledgment ack) {
    ...
    ack.acknowledge();
}
```

```
@KafkaListener(id = "qux", topicPattern = "myTopic1")
public void listen(@Payload @Valid MyObject foo,
    @Header(name = KafkaHeaders.RECEIVED_KEY, required = false) Integer key,
    @Header(KafkaHeaders.RECEIVED_PARTITION) int partition,
    @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
    @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long ts
) {
```



Exemples en mode batch

```
@KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<String> list) {
```

```
@KafkaListener(id = "listCRs", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<ConsumerRecord<Integer, String>> list) {
```

```
@KafkaListener(id = "listCRsAck", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<ConsumerRecord<Integer, String>> list, Acknowledgment ack) {
```

```
@KafkaListener(id = "list", topics = "myTopic", containerFactory = "batchFactory")
public void listen(List<String> list,
    @Header(KafkaHeaders.RECEIVED_KEY) List<Integer> keys,
    @Header(KafkaHeaders.RECEIVED_PARTITION) List<Integer> partitions,
    @Header(KafkaHeaders.RECEIVED_TOPIC) List<String> topics,
    @Header(KafkaHeaders.OFFSET) List<Long> offsets) {
```

```
@KafkaListener(id = "listMsgAck", topics = "myTopic", containerFactory = "batchFactory")
public void listen15(List<Message<?>> list, Acknowledgment ack) {
```

```
@KafkaListener(id = "listMsgAckConsumer", topics = "myTopic", containerFactory = "batchFactory")
public void listen16(List<Message<?>> list, Acknowledgment ack, Consumer<?, ?> consumer) {
    ...
}
```



Annotation de classe

@KafkaListener peut également être utilisée sur la classe permettant de mutualiser des propriétés du listener.

Ensuite des annotations ***@KafkaHandler*** précisent les méthodes de réception. La méthode est sélectionnée en fonction de la conversion de la charge utile.

```
@KafkaListener(id = "multi", topics = "myTopic")
static class MultiListenerBean {
    @KafkaHandler
    public void listen(String foo) { ... }

    @KafkaHandler
    public void listen(Integer bar) { ... }

    @KafkaHandler(isDefault = true)
    public void listenDefault(Object object) { ... }
}
```

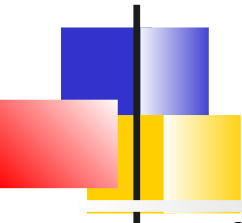



Rééquilibrage des listeners

ContainerProperties a une propriété ***consumerRebalanceListener***, qui prend une implémentation de *ConsumerRebalanceListener* (API Kafka)

- Si pas renseignée, un logger par défaut est associé. Il trace en mode INFO les rééquilibrages
- Sinon Spring fournit une sous-interface ***ConsumerAwareRebalanceListener*** qui peut être utilisée

```
public interface ConsumerAwareRebalanceListener extends ConsumerRebalanceListener {  
  
    void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer, Collection<TopicPartition> partitions);  
  
    void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer, Collection<TopicPartition> partitions);  
  
    void onPartitionsAssigned(Consumer<?, ?> consumer, Collection<TopicPartition> partitions);  
  
    void onPartitionsLost(Consumer<?, ?> consumer, Collection<TopicPartition> partitions);  
  
}
```



Exemple :

Stockage des offsets dans support externe

```
containerProperties.setConsumerRebalanceListener(new ConsumerAwareRebalanceListener() {  
  
    @Override  
    public void onPartitionsRevokedBeforeCommit(Consumer<?, ?> consumer,  
                                                Collection<TopicPartition> partitions) {  
        // acknowledge any pending Acknowledgments (if using manual acks)  
    }  
  
    @Override  
    public void onPartitionsRevokedAfterCommit(Consumer<?, ?> consumer, Collection<TopicPartition>  
    partitions) {  
        // ...  
        store(consumer.position(partition));  
        // ...  
    }  
  
    @Override  
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {  
        // ...  
        consumer.seek(partition, offsetTracker.getOffset() + 1);  
        // ...  
    }  
});
```



Spring-Kafka

Introduction

Production de message

Consommation

Transaction

Sérialisation / Désérialisation

Traitement des exceptions



Support Spring-kafka

Le support Spring pour les transactions :

- ***KafkaTransactionManager***: Implémentation de *PlatformTransactionManager* qui permet le support classique Spring : @Transactional, TransactionTemplate etc)
- ***KafkaMessageListenerContainer* transactionnel**
- Transactions locales avec *KafkaTemplate*, transaction indépendante Kafka
- Synchronisation avec d'autres gestionnaires de transaction (*TransactionManager* BD par exemple)

Les transactions sont activées dès que la propriété ***transactionIdPrefix*** est renseignée



Mécanisme

Lorsque les transactions sont activées, au lieu de gérer un seul producteur partagé, Spring maintient un cache de producteurs transactionnels.

La propriété Kafka *transactional.id* de chaque producteur est ***transactionIdPrefix + n***, où n commence par 0 et est incrémenté pour chaque nouveau producteur.

transactionIdPrefix doit être unique par instance de processus/application.



Envoi transactionnel synchronisé

@Transactional

```
public void process(List<Thing> things) {  
    things.forEach(thing -> this.kafkaTemplate.send("topic", thing));  
    updateDb(things);  
}
```

A l'entrée de méthode, l'intercepteur de *@Transactional* démarre la transaction synchronisée avec le gestionnaire de transaction (ici la BD)

A la sortie de méthode sans exception, la transaction BD est committée, suivie de la transaction Kafka.

Attention : Si le 2nd commit échoue, une exception sera envoyée à l'appelant qui devra compenser les effets de la 1ère transaction.



Transaction locale Kafka

KafkaTemplate propose la méthode
executeInTransaction

```
boolean result = template.executeInTransaction(t -> {  
    t.sendDefault("thing1", "thing2");  
    t.sendDefault("cat", "hat");  
    return true;  
});
```



Sémantique Exactly-Once

L'utilisation de transactions permet la sémantique *Exactly Once* :

- Pour une séquence ***read→process→write***, la séquence est terminée exactement une fois

Cette sémantique est donc valable pour les consommateurs de messages qui écrivent dans un topic Kafka.

- Cela est permis grâce à l'API Kafka `producer.sendOffsetsToTransaction()`.



Spring-Kafka

Introduction
Production de message
Consommation
Transaction
Sérialisation / Désérialisation
Traitement des exceptions



Introduction

Spring Kafka fournit des sérialiseurs/désérialiseurs pouvant être configurés dans les producteurs/consommateurs :

- **Format String** : *StringSerializer/ParseStringSerializer*
- **Format JSON** : *JsonSerializer/JsonDeserializer*
- **Delegating(De)Serializer** : Permettant d'utiliser un (de)serializer différent en fonction d'une clé, d'un type ou d'un topic,
- **RetryingDeSerializer** : Permettant plusieurs essais de désérialisation
- **ErrorHandlingDeserializer** : Désérialiseur permettant de traiter les erreurs de désérialisation

Spring Kafka fournit également une intégration avec *SpringMessaging* permettant de convertir les messages Kafka en classe **Message**



JsonSerializer / JsonDeserializer

Basé sur **ObjectMapper** de Jackson

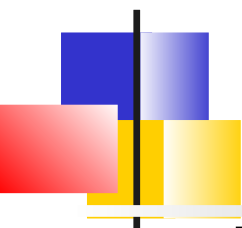
Propriétés de configuration :

– *JsonSerializer*

- **ADD_TYPE_INFO_HEADERS** (défaut true) : Ajoute des informations de type dans les entêtes
- **TYPE_MAPPINGS** : Permet de définir des correspondances de type

– *JsonDeserializer*

- **USE_TYPE_INFO_HEADERS** (défaut true) : Utilise les informations de type pour la désérialisation
- **KEY_DEFAULT_TYPE, VALUE_DEFAULT_TYPE** : Types par défaut si l'entête de type n'est pas présente
- **TRUSTED_PACKAGES** : Pattern de packages autorisés pour la désérialisation. Par défaut : *java.util, java.lang*
- **TYPE_MAPPINGS** : Permet de définir des correspondances de type
- **KEY_TYPE_METHOD, VALUE_TYPE_METHOD** : Définition d'une méthode pour déterminer le type à désérialiser



Comportement par défaut

Par défaut, le nom complet de la classe sérialisé est présent dans une entête du message

Au moment de la désérialisation,

- Spring désérialise dans le type indiqué
- Il vérifie que la classe appartient à un package de confiance

=> Cela implique :

- que la classe sérialisée soit présente dans le même package du côté consommateur et producteur
- Que la propriété ***spring.json.trusted.packages*** spécifie le package de la classe



Mapping de types

Le mapping de type permet aux consommateurs/producteurs de ne pas partager la même structure de package :

- Côté producteur, le nom de classe est mappé sur un jeton.
- Côté consommateur, le jeton dans l'en-tête de type est mappé sur une classe.

```
senderProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
senderProps.put(JsonSerializer.TYPE_MAPPINGS, "cat:com.mycat.Cat, hat:com.myhat.hat");
...
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, JsonDeserializer.class);
consumerProps.put(JsonDeSerializer.TYPE_MAPPINGS, "cat:com.yourcat.Cat, hat:com.yourhat.hat");
```

SpringBoot :

```
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties.spring.json.type.mapping=cat:com.mycat.Cat,hat:com.myhat.Hat
```



Méthodes pour déterminer les types

Il est possible de spécifier une méthode permettant de déterminer le type cible

```
spring.kafka.consumer.properties.spring.json.type.value.type.method=  
org.formation.TypeResover.myMethod
```

La méthode doit être *public static*, retourner un *Jackson JavaType*. 3 signatures possibles :

- (String topic, byte[] data, Headers headers),
- (byte[] data, Headers headers)
- (byte[] data)

```
JavaType thing1Type = TypeFactory.defaultInstance().constructType(Thing1.class);  
JavaType thing2Type = TypeFactory.defaultInstance().constructType(Thing2.class);
```

```
public static JavaType thingOneOrThingTwo(byte[] data, Headers headers) {  
    return data[21] == '1' ? thing1Type : thing2Type;  
}  
}
```



Délégation

DelegatingSerializer / DelegatingDeserializer permettent de sélectionner différents sérialiseur en fonction :

- Des entêtes
- Du type du message produit
- Le nom du topic



Par entête

Les producteurs envoient des clés dans les entêtes
VALUE_SERIALIZATION_SELECTOR et
KEY_SERIALIZATION_SELECTOR

Les consommateurs utilisent les mêmes entêtes pour
sélectionner le désérialiseur correspondant à la clé

La configuration consiste donc à fournir des Maps des 2
côtés :

```
producerProps.put(DelegatingSerializer.VALUE_SERIALIZATION_SELECTOR_CONFIG,  
    "thing1:com.example.MyThing1Serializer, thing2:com.example.MyThing2Serializer")
```

```
consumerProps.put(DelegatingDeserializer.VALUE_SERIALIZATION_SELECTOR_CONFIG,  
    "thing1:com.example.MyThing1Deserializer,  
    thing2:com.example.MyThing2Deserializer")
```




Autres types de délégation

Par type

@Bean

```
public ProducerFactory<Integer, Object> producerFactory(Map<String, Object> config) {  
    return new DefaultKafkaProducerFactory<>(config,  
        null, new DelegatingByTypeSerializer(Map.of(  
            byte[].class, new ByteArraySerializer(),  
            Bytes.class, new BytesSerializer(),  
            String.class, new StringSerializer())));  
}
```

Par regexp sur le nom du topic

```
producerConfigs.put(DelegatingByTopicSerializer.VALUE_SERIALIZATION_TOPIC_CONFIG,  
    "topic[0-4]:" + ByteArraySerializer.class.getName()  
    + ", topic[5-9]:" + StringSerializer.class.getName());  
...  
ConsumerConfigs.put(DelegatingByTopicDeserializer.VALUE_SERIALIZATION_TOPIC_CONFIG,  
    "topic[0-4]:" + ByteArrayDeserializer.class.getName()  
    + ", topic[5-9]:" + StringDeserializer.class.getName());
```



Configuration programmatische

Programmatically, on peut faire plus de choses qu'avec les propriétés de configuration :

Sérialiseurs customs :

```
@Bean
public ConsumerFactory<String, Thing> kafkaConsumerFactory(JsonDeserializer
    customValueDeserializer) {
    Map<String, Object> properties = new HashMap<>();
    // properties.put(..., ...)
    // ...
    return new DefaultKafkaConsumerFactory<>(properties,
        new StringDeserializer(), customValueDeserializer);
}
```

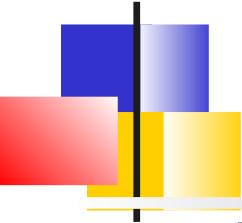
Fournir la classe de désérialisation en ignorant les informations de type dans l'entête :

```
DefaultKafkaConsumerFactory<Integer, Cat> cf = new
    DefaultKafkaConsumerFactory<>(props,
        new IntegerDeserializer(), new JsonDeserializer<>(Cat.class, false));
```



Spring-Kafka

Introduction
Production de message
Consommation
Transaction
Sérialisation / Désérialisation
Traitement des exceptions



Types d'erreur

Lors de la consommation de messages différents types d'erreur peuvent survenir

- Erreur de désérialisation Kafka : Dans ce cas le message n'est pas récupéré par le framework Spring
- Erreur de traitement du message : Une exception est lancée lors du traitement



Configuration du traitement d'erreur

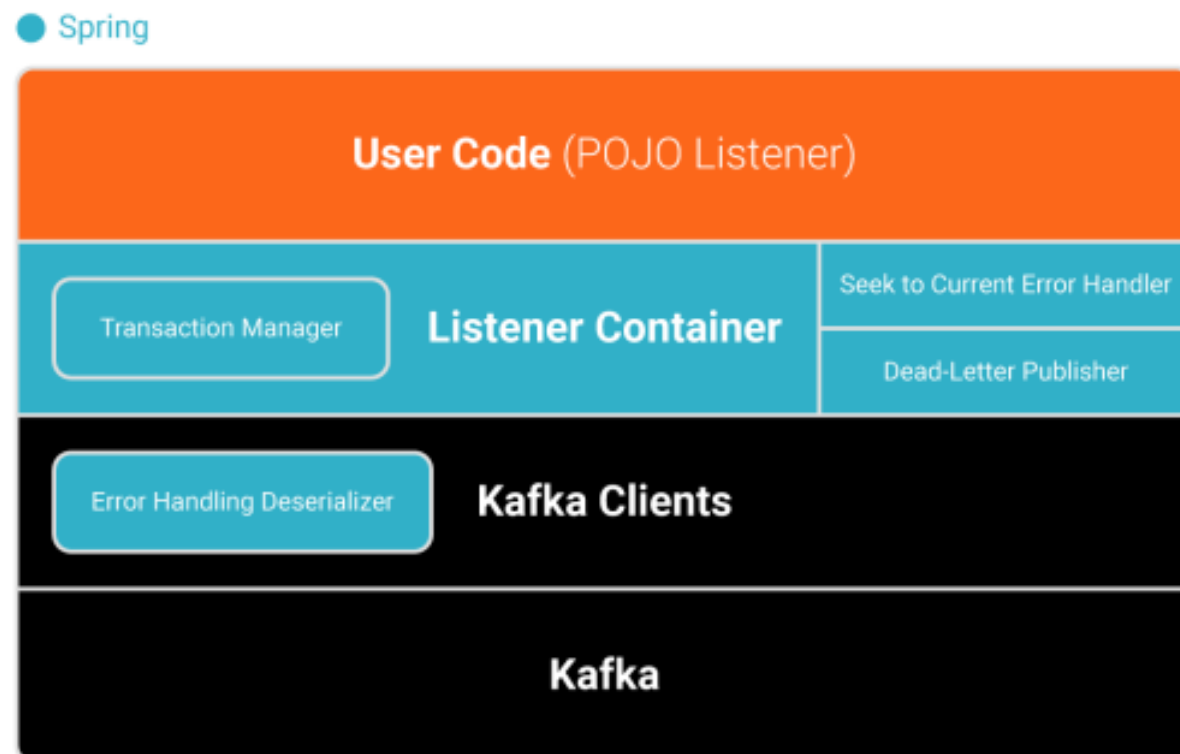
Par défaut, le message fautif est sauté (skip) et la consommation continue

Il est possible de configurer des gestionnaires d'exception au niveau container ou listener

- Ils peuvent alors retenter un certain nombre de fois la consommation du message
 - De façon synchrone : les messages suivants sont bloqués pendant les tentatives
 - De façon asynchrone : les messages suivants sont consommés pendant les tentatives. (L'ordre n'est plus respecté)

Il est possible de configurer un désérialiseur Spring gérant les erreurs de sérialisation

Support Spring





DefaultErrorHandler

DefaultErrorHandler est utilisé pour rejouer un message en erreur. Le traitement est synchrone

- ***N*** tentatives sont essayées, si elles échouent toutes :
 - Soit un simple trace
 - Soit un *DeadLetterPublishingRecoverer* est configuré, permettant d'envoyer le message vers un topic dead Letter (même nom avec le suffixe DLT)

si une réussit : la consommation continue

```
// Configuration automatique dans le container factory grâce à SpringBoot
@Bean
public CommonErrorHandler errorHandler(KafkaTemplate<Object, Object> template) {
    return new DefaultErrorHandler(
        new DeadLetterPublishingRecoverer(template), new FixedBackOff(1000L, 2));
}
```



Non blocking-retry

Spring via **@RetryableTopic** et **RetryTopicConfiguration**, permet un traitement asynchrone des messages en erreur

- Le message fautif est transmis à un autre topic avec un délai d'activation.
- A l'échéance du délai, la consommation du topic est tenté, si elle échoue le message est retransmis à un autre topic
- Etc, jusqu'à finir dans un DeadLetterTopic (dlt)

Par exemple, si le topic principal est *myTopic* et que l'on configure un *RetryTopic* avec un BackOff de 1000 ms avec un multiplicateur de 2 et 4 tentatives max

=> Spring créera les topics *myTopic-retry-1000*, *myTopic-retry-2000*, *myTopic-retry-4000* et *myTopic-dlt*



Erreur de désérialisation

ErrorHandlingDeserializer s'appuie sur un désérialiseur délégué

- En cas de sérialisation, l'exception est catchée, une valeur nulle est une entête d'erreur est positionné dans le message.


Lors de la consommation, l'entête est testée et si une erreur est présente, l'enregistrement est sauté ou le ErrorHandler est appelé

- Il est également possible de configurer une fonction qui génère la valeur de fall-back
- Dans le mode batch, c'est à nous de tester l'entête



Exemple Configuration

```
... // other props
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, ErrorHandlingDeserializer.class);
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, ErrorHandlingDeserializer.class);
props.put(ErrorHandlingDeserializer.KEY_DESERIALIZER_CLASS, JsonSerializer.class);
props.put(JsonDeserializer.KEY_DEFAULT_TYPE, "com.example.MyKey")
props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
    JsonSerializer.class.getName());
props.put(JsonDeserializer.VALUE_DEFAULT_TYPE, "com.example.MyValue")
props.put(JsonDeserializer.TRUSTED_PACKAGES, "com.example")
return new DefaultKafkaConsumerFactory<>(props);
```



Exemple consommation Batch

```
@KafkaListener(id = "test", topics = "test")
void listen(List<Thing> in,
    @Header(KafkaHeaders.BATCH_CONVERTED_HEADERS) List<Map<String, Object>> headers) {
    for (int i = 0; i < in.size(); i++) {
        Thing thing = in.get(i);
        if (thing == null
            && headers.get(i).get(SerializationUtils.VALUE_DESERIALIZER_EXCEPTION_HEADER) != null) {
            DeserializationException deserEx =
                ListenerUtils.byteArrayToDeserializationException(this.logger,
                    (byte[]) headers.get(i).get(SerializationUtils.VALUE_DESERIALIZER_EXCEPTION_HEADER));
            if (deserEx != null) {
                logger.error(deserEx, "Record at index " + i + " could not be deserialized");
            }
            throw new BatchListenerFailedException("Deserialization", deserEx, i);
        }
        process(thing);
    }
}
```



Sécurisation des clusters

Configuration des listeners

SSL / TLS

Authentication via SASL

ACLs



Introduction

Différentes alternatives supportées pour la sécurité :

- Authentification des connections clients vers les brokers via SSL ou SASL
- Authentification des connections entre contrôleurs Kraft OU des brokers vers *Zookeeper*
- Cryptage des données transférées avec les clients/brokers via TLS/SSL
- Autorisation des opérations read/write/create/delete/... par client
- Possibilité d'intégration avec d'autres mécanismes d'authentification et d'autorisation

Naturellement, dégradation des performances avec SSL



Listeners

Chaque serveur doit définir l'ensemble des *listeners* utilisés pour recevoir les requêtes des clients ainsi que des autres serveurs.

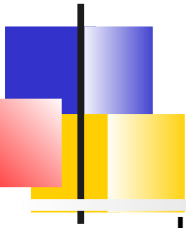
Chaque listener peut être configuré pour authentifier les clients et pour garantir que le trafic entre le serveur et le client est crypté.



Combinaisons des protocoles

Chaque protocole combine une couche de transport (PLAINTEXT ou SSL) avec une couche d'authentification optionnelle (SSL ou SASL) :

- **PLAIN_TEXT** : En clair sans authentification. Ne convient qu'à une utilisation au sein de réseaux privés pour le traitement de données non sensibles
- **SSL** : Couche de transport SSL avec authentification client SSL en option. Convient pour une utilisation dans réseaux non sécurisés car l'authentification client et serveur ainsi que le chiffrement sont prise en charge.
- **SASL_PLAINTEXT** : Couche de transport PLAINTEXT avec authentification client SASL. Ne prend pas en charge le cryptage et convient donc uniquement pour une utilisation dans des réseaux privés.
- **SASL_SSL** : Couche de transport SSL avec authentification SASL. Convient pour une utilisation dans des réseaux non sécurisés.



Configuration des listeners

Les listeners sont déclarés via la propriété ***listeners*** :

`{LISTENER_NAME}://{hostname}:{port}`

Ou LISTENER_NAME est un nom descriptif

Exemple :

```
listeners=CLIENT://localhost:9092,BROKER://localhost:9095
```

Le protocole utilisé pour chaque listener est spécifié dans la propriété ***listener.security.protocol.map***

Exemple : `listener.security.protocol.map=CLIENT:SSL,BROKER:PLAINTEXT`

Les protocoles supportés sont : *PLAINTEXT*, *SSL*, *SASL_PLAINTEXT*, *SASL_SSL*

Il est possible de déclarer le listener utilisé pour la communication inter broker via ***inter.broker.listener.name*** et ***controller.listener.names***



Sécurité

Configuration des listeners

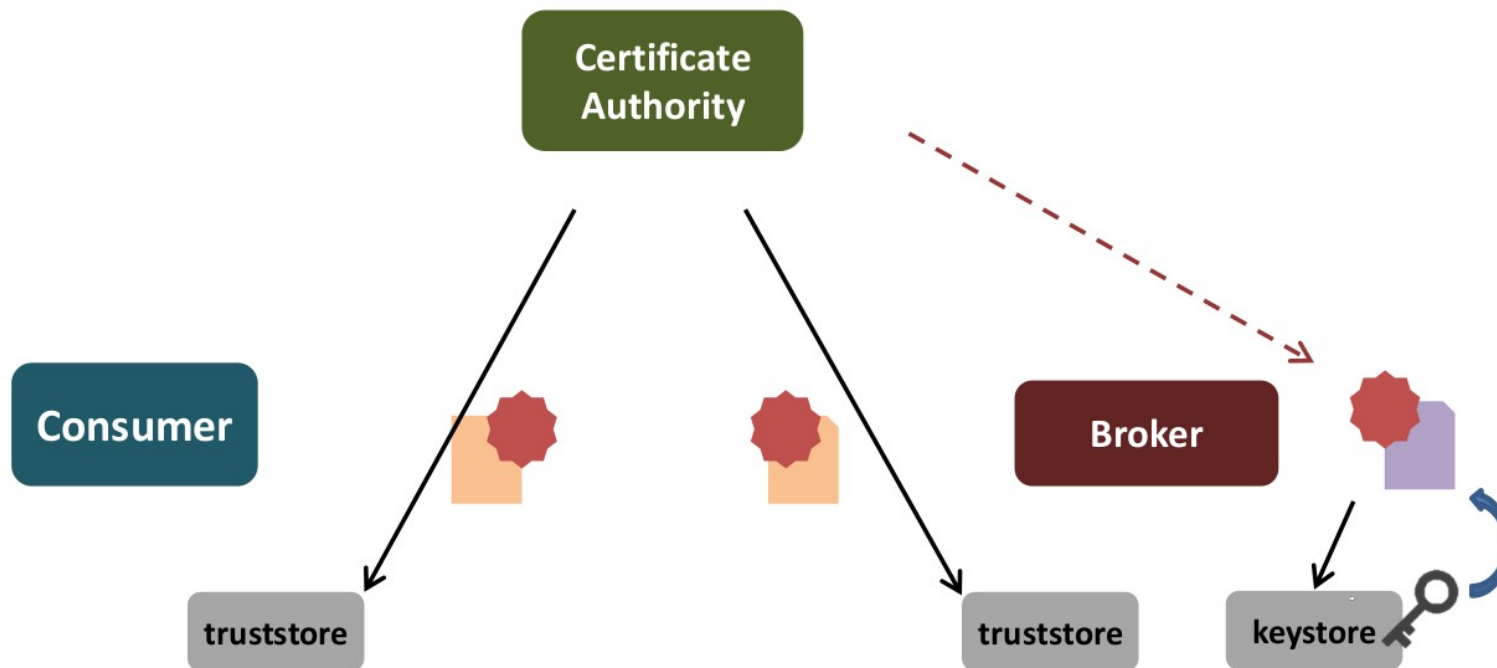
SSL / TLS

Authentication via SASL

ACLs

SSL

SSL pour le cryptage et l'authentification





Configuration TLS et Vérification d'hôte

Les certificats des brokers doivent contenir le nom d'hôte du broker en tant que nom alternatif du sujet (SAN) extension ou comme nom commun (CN) pour permettre aux clients de vérifier l'hôte du serveur.

Les certificats génériques utilisant les wildcards peuvent être utilisés pour simplifier l'administration en utilisant le même keystore pour tous les brokers d'un domaine



Préparation des certificats

Générer une paire clé publique/privé pour chaque serveur

```
keytool -keystore server.keystore.jks -alias localhost -validity 365 -keyalg RSA -genkey
```

Créer son propre CA (Certificate Authority)

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

Importer les CA dans les truststore

```
keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert
```

```
keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert
```

Créer une CSR (Certificate signed request)

```
keytool -keystore server.keystore.jks -alias localhost -certreq -file cert-file
```

Signer la CSR avec le CA

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed -days 365 -  
CAcreateserial -passin pass:servpass
```

Importer le CA et la CSR dans le keystore

```
keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert
```

```
keytool -keystore server.keystore.jks -alias localhost -import -file cert-signed
```



Configuration du broker

server.properties :

```
listeners=PLAINTEXT://host.name:port,SSL://  
host.name:port
```

```
ssl.keystore.location=/home/ubuntu/ssl/server.keystore.jks
```

```
ssl.keystore.password=servpass
```

```
ssl.key.password=servpass
```

```
ssl.truststore.location=/home/ubuntu/ssl/server.truststore.jks
```

```
ssl.truststore.password=servpass
```



Configuration du client

```
security.protocol=SSL
```

```
ssl.truststore.location=/var/private/ssl/  
client.truststore.jks
```

```
ssl.truststore.password=clipass
```

```
--producer.config client-ssl.properties
```

```
--consumer.config client-ssl.properties
```



Authentification des clients via SSL

Si l'authentification du client est requise, un *keystore* doit également être créé et la configuration de *client.properties* doit contenir :

```
ssl.keystore.location=/var/private/client.keystore.jks  
ssl.keystore.password=test1234  
ssl.key.password=test1234
```



Sécurité

Configuration des listeners
SSL/TLS

Authentication via SASL
ACLs



SASL

Simple Authentication and Security Layer pour
l'authentification

Kafka utilise JAAS pour la configuration SASL.

Mécanismes:

- GSSAPI: Kerberos
- SCRAM-SHA-256, SCRAM-SHA-512: hashed passwords
- PLAIN: username/password en clair
- OAUTHBEARER (Depuis Kafka 3.x)



Configuration

L'authentification s'applique :

- Aux brokers (communication inter-broker authentifiée)
- Aux clients (communication client-broker authentifié)
- Dans le mode Kraft, cela concerne également les contrôleurs



Mécanismes

SASL/Kerberos : Nécessite un serveur (Active Directory par exemple), d'y créer les *Principals* représentant les brokers. Tous les hosts Kafka doivent être atteignables via leur FQDNs. SpringKafka offre un support pour Kerberos.

SASL/PLAIN est un mécanisme simple d'authentification par login/mot de passe. Doit être utilisé avec TLS. Kafka fournit une implémentation par défaut qui peut être étendue pour la production.

SASL/SCRAM (256/512) (*Salted Challenge Response Authentication Mechanism*) : Mot de passe haché stocké dans Zookeeper.

SASL/OAUTHBEARER : Basé sur OAuth2 mais pas adapté en l'état à la production.



Configuration JAAS

La configuration JAAS s'effectue soit

- via un **fichier jaas** contenant une section *KafkaServer* spécifiant le LoginModule JAAS associé au mécanisme d'authentification.

```
KafkaServer {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="admin"  
    password="admin-secret";  
};
```

- via la propriété **sasl.jaas.config** préfixée par le mécanisme SASL utilisé

```
listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=  
org.apache.kafka.common.security.scram.ScramLoginModule required \  
username="admin" \  
password="admin-secret";
```



Exemple Kerberos

Utiliser un serveur Kerberos existant ou en installer un¹

Créer des utilisateurs pour le broker et les clients :

S'assurer que tous les brokers soient accessibles via leur FQDNs

Configuration fichier JAAS :

```
KafkaServer {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    keyTab="/etc/security/keytabs/kafka_server.keytab"  
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";  
};
```

1. <https://help.ubuntu.com/community/Kerberos>,
https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Managing_Smart_Cards/installing-kerberos.html



Exemple Kerberos

Configuration server.properties :

```
listeners=SASL_PLAINTEXT://host.name:port
security.inter.broker.protocol=SASL_PLAINTEXT
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.enabled.mechanisms=GSSAPI
# Doit correspondre au nom du principal du broker
sasl.kerberos.service.name=kafka
```



Spring Kafka et *Kerberos*

Le bean

KafkaJaasLoginModuleInitializer offre du support pour la configuration Kerberos.

@Bean

```
public KafkaJaasLoginModuleInitializer jaasConfig() throws IOException {  
    KafkaJaasLoginModuleInitializer jaasConfig = new KafkaJaasLoginModuleInitializer();  
    jaasConfig.setControlFlag("REQUIRED");  
    Map<String, String> options = new HashMap<>();  
    options.put("useKeyTab", "true");  
    options.put("storeKey", "true");  
    options.put("keyTab", "/etc/security/keytabs/kafka_client.keytab");  
    options.put("principal", "kafka-client-1@EXAMPLE.COM");  
    jaasConfig.setOptions(options);  
  
    return jaasConfig;  
}
```



Exemple SASL PLAIN

1. Configurer le fichier JAAS
2. Passer l'emplacement du fichier de configuration JAAS en tant que paramètre JVM à chaque broker

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

3. Configurer SASL dans *server.properties*

```
listeners=SASL_SSL://host.name:port
security.inter.broker.protocol=SASL_SSL
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN
```




Configuration du client

Créer le fichier Jaas

```
KafkaClient {  
    org.apache.kafka.common.security.plain.PlainLoginModule  
    required  
    username="alice"  
    password="alice-secret"  
} ;
```

Ajouter un paramètre JVM

```
-Djava.security.auth.login.config=/home/ubuntu/ssl/client_jaas.conf
```

client.properties

```
security.protocol=SASL_SSL  
sasl.mechanisms=PLAIN
```



SASL / PLAIN en production

Doit être utilisé avec SSL

Possibilité d'éviter de stocker des mots de passe clairs en configurant des *callback handler* qui obtiennent le nom d'utilisateur et le mot de passe d'une source externe via :

sasl.server.callback.handler.class

et

sasl.client.callback.handler.class.

Possibilité d'intégrer des *callback handler* utilisant des serveurs d'authentification externes pour la vérification du mot de passe via

sasl.server.callback.handler.class



Sécurité

Configuration des listeners
SSL/TLS
Authentication via SASL
ACLs



Introduction

Les brokers Kafka gèrent le contrôle d'accès à l'aide d'une API définie par l'interface

org.apache.kafka.server.authorizer.Authorizer

L'implémentation est configurée via la propriété :

authorizer.class.name

Kafka fournit 2 implémentations

- Pour Zookeeper,
kafka.security.authorizer.AclAuthorizer
- En kraft mode
org.apache.kafka.metadata.authorizer.StandardAuthorizer



Autorisation

Les ACLs sont stockées dans les méta-données gérées par les contrôleurs et peuvent être gérées par l'utilitaire ***kafka-acls.sh***

Chaque requête Kafka est autorisée si le *KafkaPrincipal* associé à la connexion a les autorisations pour effectuer l'opération demandée sur les ressources demandées.

Les règles peuvent être exprimées comme suit :

Principal P is [Allowed/Denied] Operation O From Host H on any Resource R matching ResourcePattern RP



Exemple

Principals User:Bob et User:Alice sont autorisés à effectuer les opérations Read et Write sur le Topic Test-Topic à partir des IP 198.51.100.0 et IP 198.51.100.1

```
bin/kafka-acls.sh --bootstrap-servers
localhost:9092 --add --allow-principal
User:Bob --allow-principal User:Alice --
allow-host 198.51.100.0 --allow-host
198.51.100.1 --operation Read --operation
Write --topic Test-topic
```



Ressources et opérations

Chaque ACL consiste en :

- **Type de ressource** : *Cluster | Topic | Group | TransactionalId*
- **Type de pattern** : *Literal | Prefixed*
- **Nom de la ressource** : Possibilité d'utiliser les wildcards
- **Opération** : *Describe | Create | Delete | Alter | Read | Write | DescribeConfigs | AlterConfigs*
- **Type de permission** : *Allow | Deny*
- **Principal** : De la forme *<principalType>:<principalName>*
Exemple : *User:Alice, Group:Sales, User :**
- **Host** : Adresse IP du client, * si tout le monde

Exemple Complet :

*User:Alice has Allow permission for Write to Prefixed
Topic:customer from 192.168.0.1*



Règles

AclAuthorizer autorise une action s'il n'y a pas d'ACL de DENY qui corresponde à l'action et qu'il y a au moins une ACL ALLOW qui correspond à l'action.

- L'autorisation Describe est implicitement accordée si l'autorisation Read, Write, Alter ou Delete est accordée.
- L'autorisation Describe Configs est implicitement accordée si l'autorisation AlterConfigs est accordée.



Permissions clientes

Brokers : ***Cluster:ClusterAction*** pour autoriser les requêtes de contrôleur et les requêtes de fetch des répliques.

Producteurs simples : ***Topic:Write***

- idempotents sans transactions : ***Cluster:IdempotentWrite***.
- Transactionnels : ***TransactionalId:Write*** à la transaction et ***Group:Read*** pour que les groupes de consommateurs valident les offsets.

Consommateurs : ***Topic:Read*** et ***Group:Read*** s'ils utilisent la gestion de groupe ou la gestion des offsets.

Clients admin : ***Create, Delete, Describe, Alter, DescribeConfigs, AlterConfigs*** .



Exceptions

2 options de configuration permettant d'accorder un large accès aux ressources permet de simplifier la mise en place d'ACL à des clusters existants :

- ***super.users*** : Permet de définir les utilisateurs ayant droit à tout
- ***allow.everyone.if.no.acl.found=true*** : Tous les utilisateurs ont accès aux ressources sans ACL.



Exemple

Principals User:Bob et User:Alice sont autorisés à effectuer les opérations Read et Write sur le Topic Test-Topic à partir des IP 198.51.100.0 et IP 198.51.100.1

```
bin/kafka-acls.sh --authorizer-properties
zookeeper.connect=localhost:2181 --add --
allow-principal User:Bob --allow-principal
User:Alice --allow-host 198.51.100.0 --
allow-host 198.51.100.1 --operation Read --
operation Write --topic Test-topic
```



Audit

Les brokers peuvent être configurés pour générer des traces d'audit.

La configuration s'effectue dans *conf/log4j.properties*

- Le fichier par défaut est *kafka-authorizer.log*
- Le niveau INFO trace les entrées pour chaque refus
- Le niveau DEBUG pour chaque requête acceptée