



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

- Modèles d'interaction entre service
- Modèle RestFul
- Modèles asynchrones, le rôle d'un message Broker
- Gérer les évolutions d'API

Apache Kafka et ses APIs

- Le projet Kafka et Cas d'usage
- Concepts coeur, Architecture
- Producer et Consumer API
- Configuration Cluster et Topic
- Autres APIS, KafkaAdmin et KafkaStream

Intégration Spring Kafka

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

Sécurité Kafka

- Configuration des listeners
- SSL/TLS
- Authentification via SASL, oAuth2
- ACL

Spring Security

- Rappels Spring Security
- Support pour oAuth2
- Spring Kafka et Jaas



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, ...)

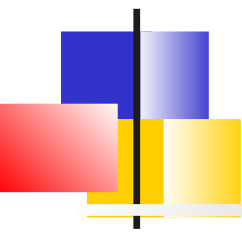


Illustration

```
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 utiliser 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
- ...



Configuration XML

```
<beans>
```

```
  <bean id="movieLister" class="spring.MovieLister">
```

```
    <property name="finder" ref="movieFinder"/>
```

```
  </bean>
```

```
  <bean id="movieFinder" class="spring.ColonMovieFinder">
```

```
    <property name="filename">
```

```
      <value>movies1.txt</value>
```

```
    </property>
```

```
  </bean>
```

```
</beans>
```



Test

```
public void testWithSpring()
    throws Exception {

    ApplicationContext ctx =
        new FileSystemXmlApplicationContext("spring.xml");

    MovieLister lister = (MovieLister)ctx.getBean("movieLister");

    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");

    assertEquals("Once Upon a Time in the West",movies[0].getTitle());
}
```



Classes de configuration

Depuis la 3.0, **@Configuration** permet de définir des classes pouvant instancier des beans

- Toutes les méthodes annotées par **@Bean** sont alors exécutées par le framework lors de son initialisation et permettent de fournir des implémentations
- 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 généralement utilisées pour configurer des ressources externes à l'applcatif (une base de données par exemple, les beans techniques du framework, ...)

Pour faciliter la configuration de ces ressources, Spring fournit des annotations ***@Enable*** qui permettent d'appliquer une configuration par défaut de la ressource



Exemples *@Enable*

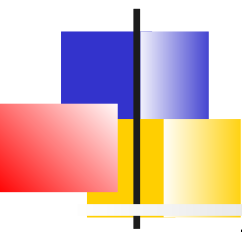
@EnableWebMvc : Configuration Spring MVC dans une application

@EnableCaching : Permet d'utiliser les annotations *@Cacheable*, ...

@EnableScheduling : Permet d'utiliser les annotations *@Scheduled*

@EnableJpaRepositories : Permet de scanner les classe Repository

...



@Component et stereotypes

Des annotations permettent de marquer une classe comme bean Spring.

- L'instanciation du bean est alors fait par le framework.
- Il s'agit en général de beans métier.

L'annotation générique est ***@Component***, placée sur la classe



Stéréotypes

Spring introduit d'autres stéréotypes :

@Repository, **@Service**, **@Controller** et **@RestController** qui sont des spécialisations de *@Component* pour des cas d'usage plus spécifique (persistance, service, et couche de présentation)

Les stéréotypes servent à classer les beans et éventuellement à leur ajouter des comportements génériques.

Par ex : La sérialisation JSON pour les @RestController



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 qu'ils soient instanciés via les classes de Configuration ou les annotations stéréotypées peuvent avoir 3 cycles de vie (ou scope) :

- **Singleton**: Il existe une seule instance de l'objet (qui est donc partagé). Idéal pour des beans « stateless ».
=> C'est l'écrasante majorité des cas.
- **Prototype** : A chaque fois que le bean est utilisé via son nom, une nouvel instance est créé.
=> Quasiment Jamais
- **Custom object "scopes"** : Leur cycle de vie est généralement synchronisé avec d'autre objets, comme une requête HTTP, une session http, une transaction BD
=> Certains beans fournis par Spring. Ex : *EntityManager*



@Scope

L'annotation **@Scope** permet de préciser un des scopes prédéfinis de Spring ou un scope personnalisé

```
@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...  
    }  
}
```



Recherche des annotations

2 alternatives afin que le framework traite les annotations :

- Lui indiquer la localisation des classes de configuration ou component
- Lui indiquer un package à scanner afin qu'il détecte les annotations

Dans la pratique (avec SpringBoot), ce sera la 2ème alternative qui est utilisée.



Configuration du framework via les annotations

Indication d'une classe de *@Configuration*

```
public static void main(String[] args) {  
    ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(AppConfig.class);  
    MovieService movieService = ctx.getBean(MovieService.class);  
    movieService.findAll();  
}
```

Scan de package :

```
public static void main(String[] args) {  
    AnnotationConfigApplicationContext ctx = new  
    AnnotationConfigApplicationContext();  
    ctx.scan("org.formation");  
    ctx.refresh();  
    MyService myService = ctx.getBean(MyService.class);  
}
```



@ComponentScan

Le scan de packages peut également être déclenché via **@ComponentScan** habituellement 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'annotation 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;
    }
    // ...
}
```



Examples (2)

```
public class MovieRecommender {  
    @Autowired  
    private MovieCatalog movieCatalog;  
    private CustomerPreferenceDao customerPreferenceDao;  
    @Autowired  
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
    // ...  
}  
...  
public class MovieRecommender {  
    @Autowired  
    private MovieCatalog[] movieCatalogs;  
    // ...  
}
```



Injection implicite

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

- Si la dépendance est présent 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

@Resource permet 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;  
  
    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 des propriétés de configuration des beans.
Ils proviennent des fichier *.properties*,
d'argument de commande en ligne ou
autre ...
- Les **profils activés** : Groupes nommés de Beans, les beans sont enregistrés seulement si le profil est activé au démarrage



Activation d'un profil

Programmatically :

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

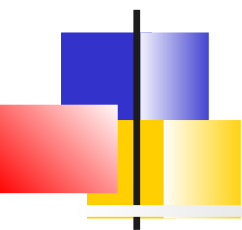
Ligne de commande :

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
- Dès 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 automatiquement la nature de l'application et de configurer les beans Spring nécessaires

- Cela permet de démarrer rapidement avec une configuration par défaut et de graduellement surcharger la configuration par défaut pour les besoins de l'application

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...

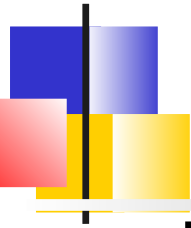
=> Le projet est donc exécutable avec le minimum de configuration préalable



Personnalisation de la configuration

La configuration par défaut peut être surchargée par différents moyens

- Les propriétés de configuration 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
- Du code 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 des dépendances vers les librairies OpenSource avec 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)



Java

Gestion des dépendances

Dans un environnement Java, Spring Boot simplifie la gestion de dépendances et de leurs versions :

- Il organise les fonctionnalités de Spring en modules.
=> Des groupes de dépendances peuvent être ajoutés à un projet en important des **"starter" modules**.
- Un assistant nommé **"Spring Initializr"**, est utilisé pour générer ou mettre à jour des configurations Maven ou Gradle

=> Pour les développeurs : ce mécanisme simplifie à l'extrême la gestion des dépendances et de leurs versions

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 (Maven ou gradle)
- 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**
- **@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

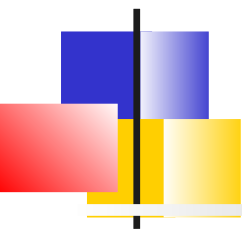
@SpringBootTest :

- Permet de créer le contexte Spring avant le 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
- *-**reactive-web** : Application web ou API REST en mode réactif

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** : Support pour démarrer les services de support via docker-compose (BD, Kafka, etc..)
- *-**graalvm-native** : Support pour construire des images natives
- *-**modulith** : 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** : Spring Mobile
- *-**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

Amazon, Google Cloud, Azure, Cloud Foundry, Alibaba

Micro-services, SpringCloud (Ex : Netflix)

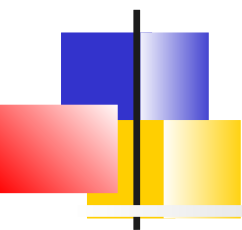
Services de discovery, de configuration externalisée, de répartition de charge, de gateway, de monitoring, de tracing, de messagerie distribuée, de circuit breaker, etc ...



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

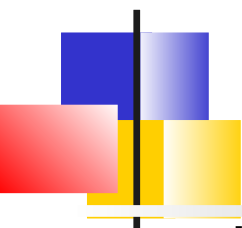


IDEs

Pivotal propose ***SpringToolsSuite***
disponible pour

- VSCode
- Eclipse
- Theia

Pour les autres IDEs, des tierces partie
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 propriété (.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

|

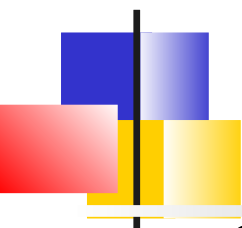
+ - service

| + - 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 :

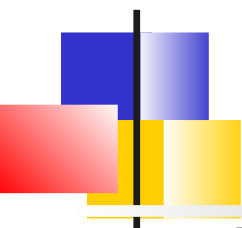
- Directement via l'annotation **@Value**
- Ou associer à un objet structuré via l'annotation **@ConfigurationProperties**



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 *.yaml*

```
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 recommandée 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 découpé 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



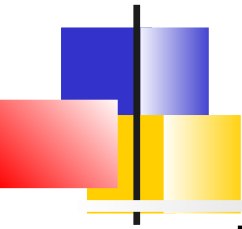
Interactions entre services Spring

Modèles d'interaction

API synchrones : RestFul, GraphQL

API asynchrones et message Broker

Gérer les évolutions d'API



Les modèles d'interaction : synchrone, asynchrone, réactif



Introduction

Différentes technologies pour la communication entre micro-services :

- Synchrones sur le mode requête/réponse :
REST, gRPC, GraphQL
- Asynchrones, Communications via message : AMQP, STOMP

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, *GraphiQL* pour GraphQL
- Canaux de messages, format et type du message pour l'asynchrone



Interactions entre services Spring

Modèles d'interaction

API synchrones : RestFul

API asynchrones et message Broker

Gérer les évolutions d'API



Introduction

Lors de l'utilisation d'un mécanisme IPC (Inter Process Communication) basé sur l'invocation de procédure à distance,

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

Certains clients peuvent bloquer l'attente d'une réponse, et d'autres peuvent avoir une architecture réactive et non bloquante.

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



Pattern

Remote procedure invocation

Pattern¹ : Un client invoque un service utilisant un protocole d'invocation de procédure à distance comme REST

D'autres protocoles existent :

- GraphQL (Facebook)
- Netflix Falcor
- gRPC (Google)

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



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 instances de service.

Nécessité d'utiliser des mécanismes de découverte de service pour localiser les instances de service.

Disponibilité réduite.

Le client et le service communiquent directement sans intermédiaire , ils 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.



Spring MVC

Le starter ***spring-boot-starter-web*** permet de charger le framework Spring MVC

Spring MVC permet de déclarer des beans de type

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

Dans la suite du support, nous nous concentrons sur les RestController



Spring WebFlux

Le starter ***spring-boot-starter-webflux*** charge le framework *SpringWebflux* qui permet une programmation réactive

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..)



Modèles de programmation de *SpringWebflux*

Spring Webflux offre en plus 2 modèles de programmation :

- **Contrôleurs annotés** : Idem à Spring MVC avec les mêmes annotations.
Les méthodes des contrôleurs retournent des types réactifs, des arguments réactifs sont associés à *@RequestBody*.
- **Endpoints fonctionnels** : Programmation fonctionnelle basée sur les lambdas.
Idéal pour de petites libraires permettant de router et traiter des requêtes.
Dans ce cas, l'application est en charge du traitement de la requête du début à la fin.

Dans les versions récentes de *SpringWebflux*, on peut mixer du code réactif et du code impératif.



Spécification des Endpoints

@RestController se positionne sur de simples classes dont les méthodes publiques sont généralement accessible via HTTP

@RequestMapping

- Placée au niveau de la classe, elle permet de spécifier un préfixe d'URL pour toutes les endpoints du contrôleur
- Au niveau d'une méthode, l'annotation précise les attributs suivants :
 - **path** : Valeur fixe ou gabarit d'URI
 - **method** : Pour limiter la méthode à une action HTTP
 - **produce/consume** : Préciser le format des données d'entrée/sortie
Dans le cadre d'une API Rest il n'est pas nécessaire de préciser ces attributs car le format est toujours JSON

Au niveau des méthodes, on utilise généralement les variantes de @RequestMapping : **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**, **@PatchMapping**, ..



Arguments de méthode

Une méthode annotée via *@*Mapping* peut se faire injecter des arguments de type :

- La requête ou réponse HTTP
- La session HTTP
- La locale, la time zone
- La méthode HTTP
- L'utilisateur authentifié par HTTP (Principal)
- ..

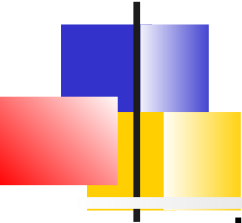
Si l'argument est d'un autre type, il nécessite d'être annoté afin que Spring puisse l'injecter



Annotations d'arguments

Les annotations d'argument permettent de se faire injecter des attributs de la requête

- **@PathVariable** : Une partie de l'URI
- **@RequestParam** : Un paramètre HTTP
- **@RequestBody** : Corps de la requête par défaut au format Json qui sera converti en un objet Java
- **@RequestHeader** : Une entête HTTP
- **@RequestPart** : Une partie d'une requête multi-part



Types des valeurs de retours des méthodes

Les types des valeurs de retour possibles pour un contrôleur REST sont :

- Une classe **Modèle ou DTO** qui par défaut sera converti en JSON via la librairie Jackson.
- Un classe Modèle ou DTO encapsulé dans un type réactif *Mono* ou *Flux*
- Un objet ***ResponseEntity<T>*** permettant de positionner les codes retour et les entêtes HTTP voulues dans le cas de *SpringMVC*
- ***HttpEntity***, ***ResponseEntity*** : La réponse complète dans le cas de *SpringWebFlux*
- ***Flux<ServerSentEvent>*** : Émission d'événements serveur (*text/event-stream*)



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 {

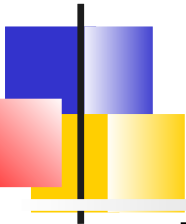
    @GetMapping(value="/{id}")
    public Mono<User> getUser(@PathVariable Long id) {
        // ...
    }

    @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);
    }
}
```



Adaptation de la sérialisation

Pour adapter la sérialisation par défaut de Jackson à ses besoins, plusieurs alternatives :

- Créer des classes DTO spécifiques.
- Utiliser les annotations de Jackson sur les classes du domaine (DTO ou Entité)
- Utiliser l'annotation *@JsonView*
Le même objet Entité ou Dto peut alors être sérialisé différemment en fonction des cas d'usage
- Implémenter ses propres Sérialiseur/Désérialiserur.
Spring propose l'annotation *@JsonComponent*



Gestion des erreurs

*Lorsqu'une exception est levée durant le traitement de la requête, Spring Boot forward la requête sur **/error***

- Un comportement par défaut en REST ou en Web permet de visualiser la cause de l'erreur

Pour remplacer le comportement par défaut dans un modèle RestFul, plusieurs alternatives :

- L'annotation **ResponseStatus** sur la méthode du contrôleur permet de positionner un code d'erreur HTTP dans la réponse
- Utiliser la classe **ResponseStatusException** pour associer un code retour à une Exception
- Les classes **@Controller** et **@ControllerAdvice** 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*

@ControllerAdvice

```
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);
    }
}
```



SpringDoc

SpringDoc est un outil qui simplifie la génération et la maintenance de la documentation des API REST

Il est basé sur la spécification OpenAPI 3 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

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.1.0</version>
</dependency>
```



Fonctionnalités

Par défaut,

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

SpringDoc prend en compte

- les annotations `javax.validation` positionnées sur les DTOs
- Les Exceptions gérées par les `@ControllerAdvice`
- Les annotations de OpenAPI
<https://javadoc.io/doc/io.swagger.core.v3/swagger-annotations/latest/index.html>

SpringDoc peut être désactivé via la propriété :
`springdoc.api-docs.enabled=false`



SpringMVC et RestTemplate

Spring MVC fournit la classe ***RestTemplate*** facilitant les appels aux services REST.

Spring Boot ne fournit pas de bean auto-configuré de type *RestTemplate* mais il auto-configuré un ***RestTemplateBuilder*** permettant de les créer

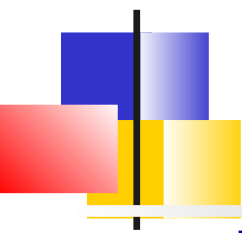


Exemple

```
@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);
    }
}
```

Spring Webflux et WebClient

WebClient est la nouvelle interface permettant d'effectuer les requêtes Web apportée par *Spring Webflux*.

Elle supporte les interactions synchrones et asynchrones et peut donc être utilisées pour les 2 stacks web (servlet et reactive)



Exemple

// 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

Modèles d'interaction

API synchrones : RestFul, GraphQL

API asynchrones et message Broker

Gérer les évolutions d'API



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 de micro-services Saga¹, 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>



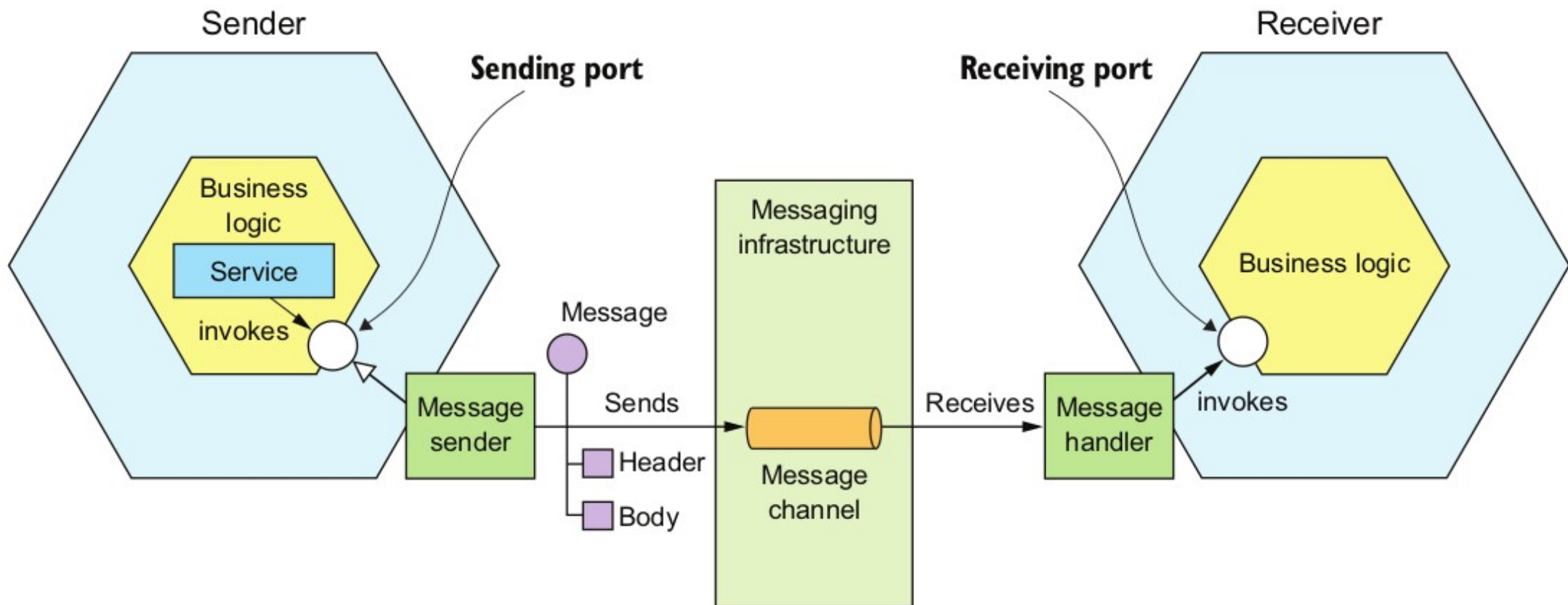
Messaging Pattern

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





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 traditionnelle des transactions distribuées¹ ne sont pas adaptées au clustering
- 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>



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'interaction :

- Sémantique du message : Document, Requête ou événement
- Canaux : Point 2 point ou PubSub
- Styles de communication : Requête synchrone/asynchrone, Fire and Forget, Publication vers abonnés avec ou sans acquittement
- Garanties de service : Bufferisation, Ordre des messages, Durabilité, Garantie de livraison At Most, At Least ou Exactly Once, etc.

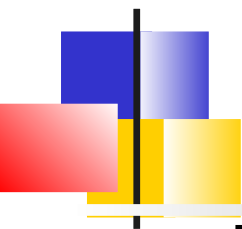


Spécification de l'API

La spécification consiste à définir

- Les noms des canaux
- Les types de messages et leur format.
(JSON, Avro, ProtoBuf)

Par contre à la différence de REST et OpenAPI, il n'y a pas de standard pour décrire les contrats



Architecture event-driven

Dans les architectures event-driven, 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 l'événement.

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

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



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) {
    ...
}
```



Interactions entre services Spring

Modèles d'interaction

API synchrones : RestFul, GraphQL

API asynchrones et message Broker

Gérer les évolutions d'API



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.



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 de l'API Rest, Mime-type ou contenu d'un message asynchrone)

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



Changements mineurs

Les changements rétrocompatibles sont des ajouts :

- Ajout d'attributs optionnels à la requête
- Ajout d'attributs à la réponse
- Ajout de nouvelles opérations

Si les services ont été développés pour fournir des valeurs par défaut des attributs de requête manquant, d'ignorer les attributs de réponse dont on n'a pas besoin, les changements mineurs de posent pas de problème.

(Voir également *Robustness principle*¹⁾)

1.https://en.wikipedia.org/wiki/Robustness_principle



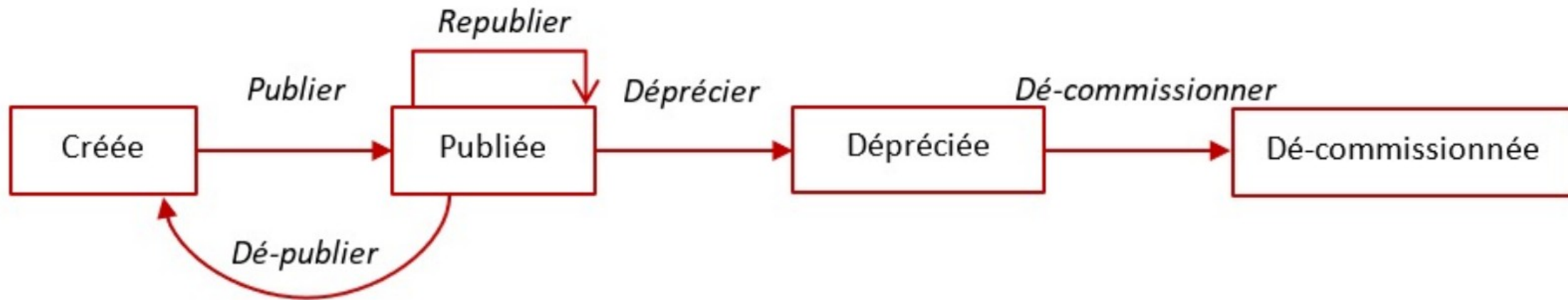
Changements majeurs

En cas de changements majeurs, les services doivent supportés pendant un certain temps plusieurs versions d'API

Les adaptateurs du service qui implémentent les API contiendront une logique qui traduit les requêtes des anciennes versions dans la nouvelle version.

Cycle de vie des APIs

Le cycle de vie d'une API comprend en général 4 états :



Dans chaque état, l'API est visible pour certains acteurs et invisible pour d'autres .

Pour REST, la version de l'API est en général dans l'URL du endpoint



Gestion des versions majeures

Les 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



Apache Kafka et ses APIs

Le projet Kafka

Cas d'usage

Concepts coeur de Kafka, Architecture

Producer API

Consumer APIs

Sérialisation Json, Avro

KafkaAdmin et KafkaStream



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



Apache Kafka et ses APIs

Le projet Kafka

Cas d'usage

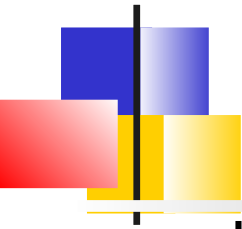
Concepts coeur, Architecture

Producer API

Consumer APIs

Sérialisation Json, Avro

KafkaAdmin et KafkaStream



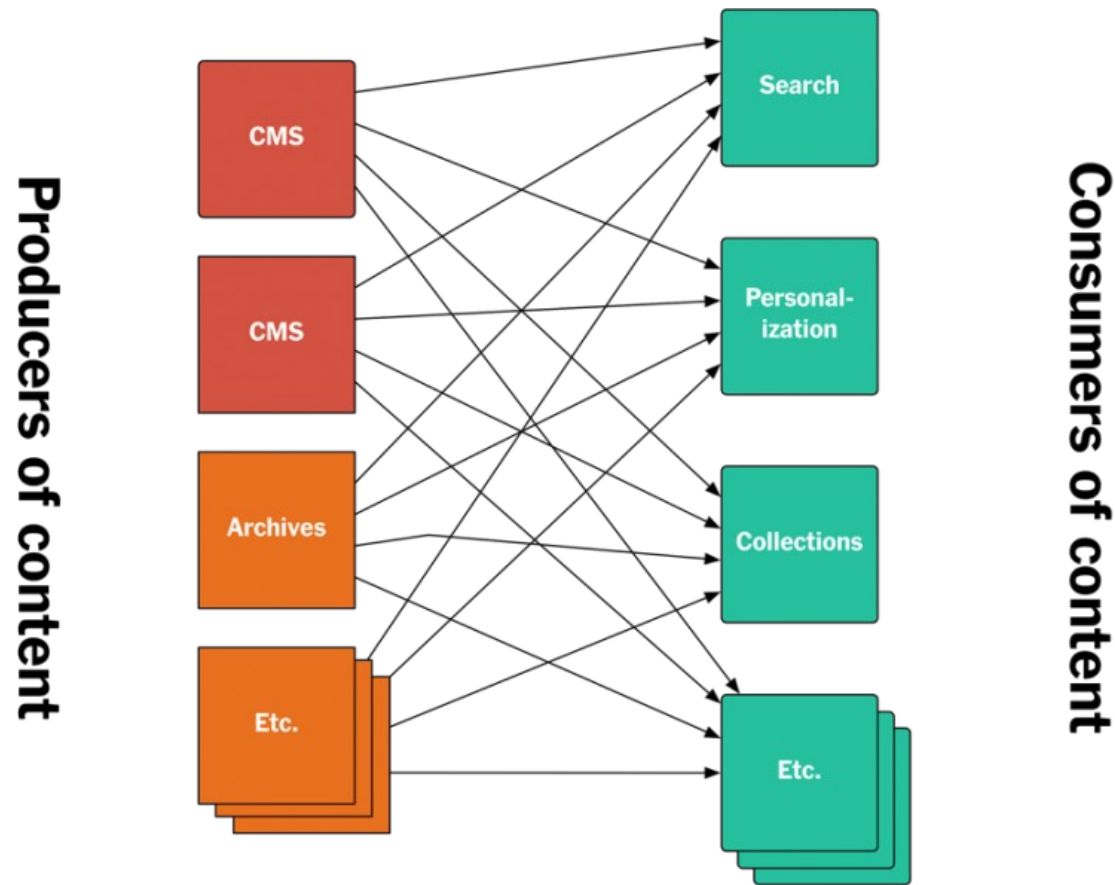
Kafka vs Message broker traditionnel

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 consommateur**, ce modèle est scalable
- Kafka offre une **garantie plus forte** sur l'ordre de livraison des messages
- Kafka ne supprime pas les messages après consommation. Ils peuvent être **consommés à posteriori**

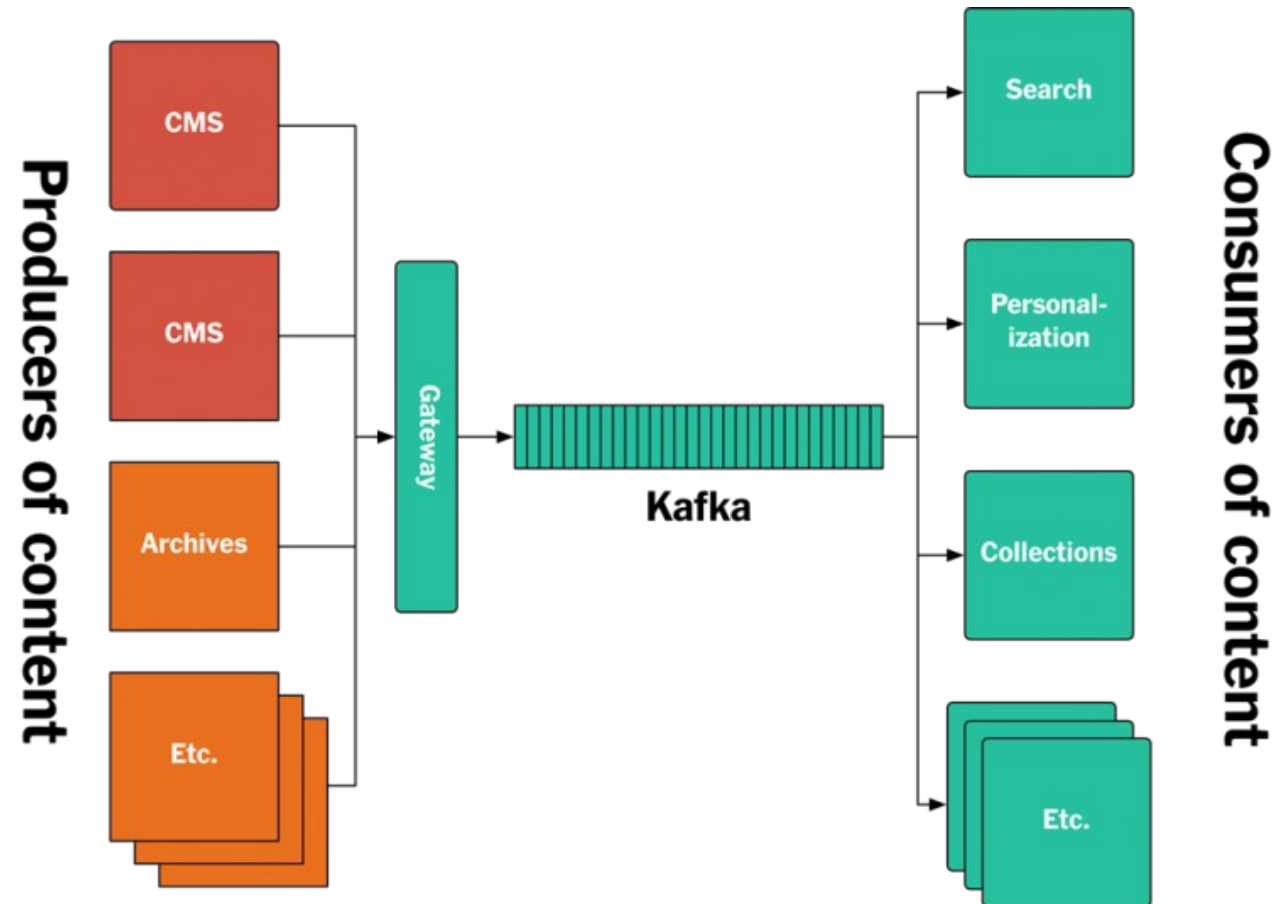
=> Idéal comme plate-forme d'intégration entre services : Architecture micro-services, ESB

Exemple ESB (New York Times) *Avant*



Exemple ESB (New York Times)

Après





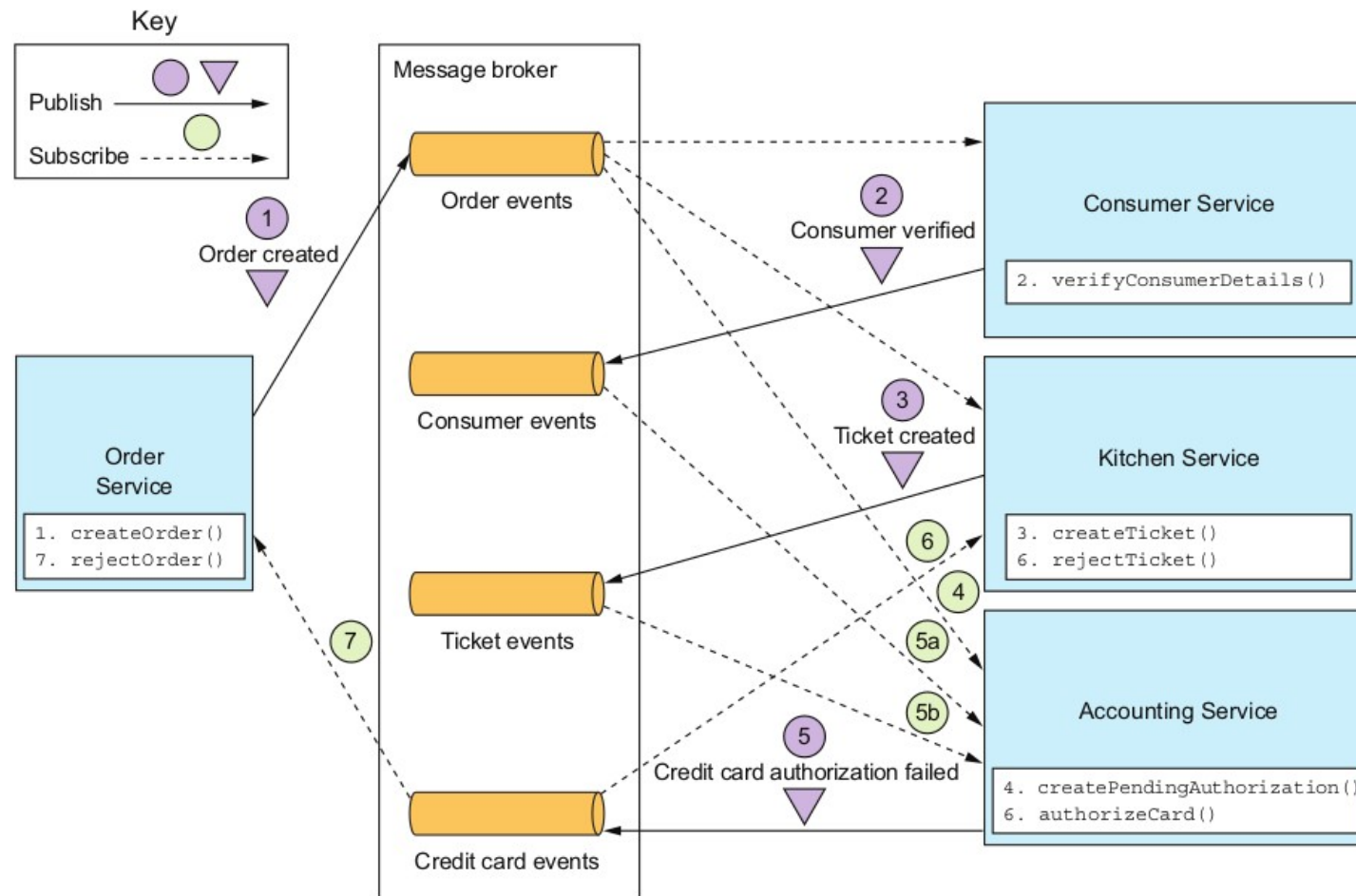
Exemple : micro-services

Kafka est souvent utilisé pour permettre des communications asynchrones entre les services d'une architecture micro-services

- Permet tous les styles d'interaction :
Requête/Réponse synchrone asynchrone, One way notification, Publish and Subscribe, Publish et réponse asynchrones
- Permet l'implémentation de patterns micro-services, par exemple SAGA¹ (~ Transaction distribuée)

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

Exemple Micro-services Message Broker et Design pattern SAGA



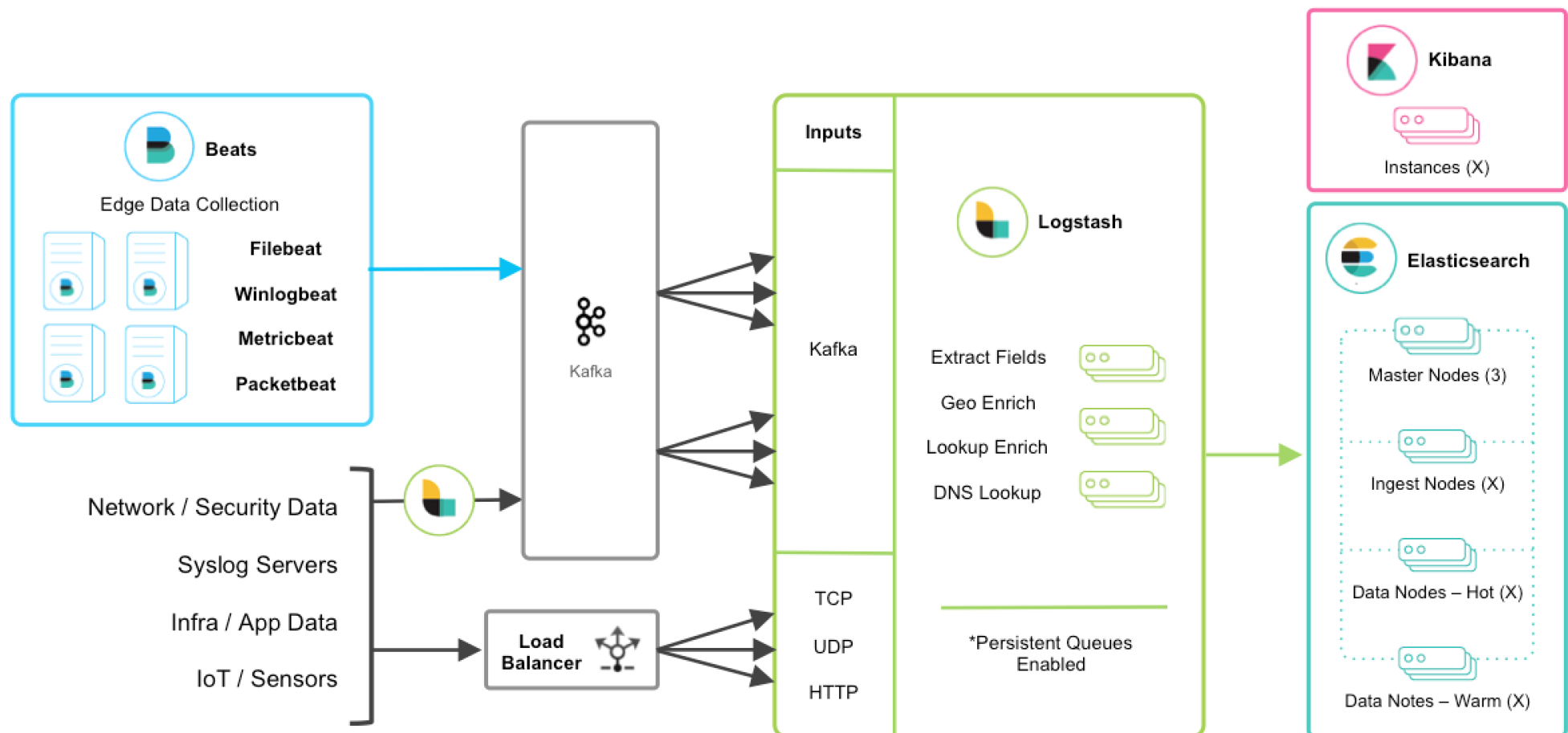


Kafka pour l'ingestion massive de données

Les *topics* Kafka peuvent être utilisés pour bufferiser les événements provenant de multiples sources

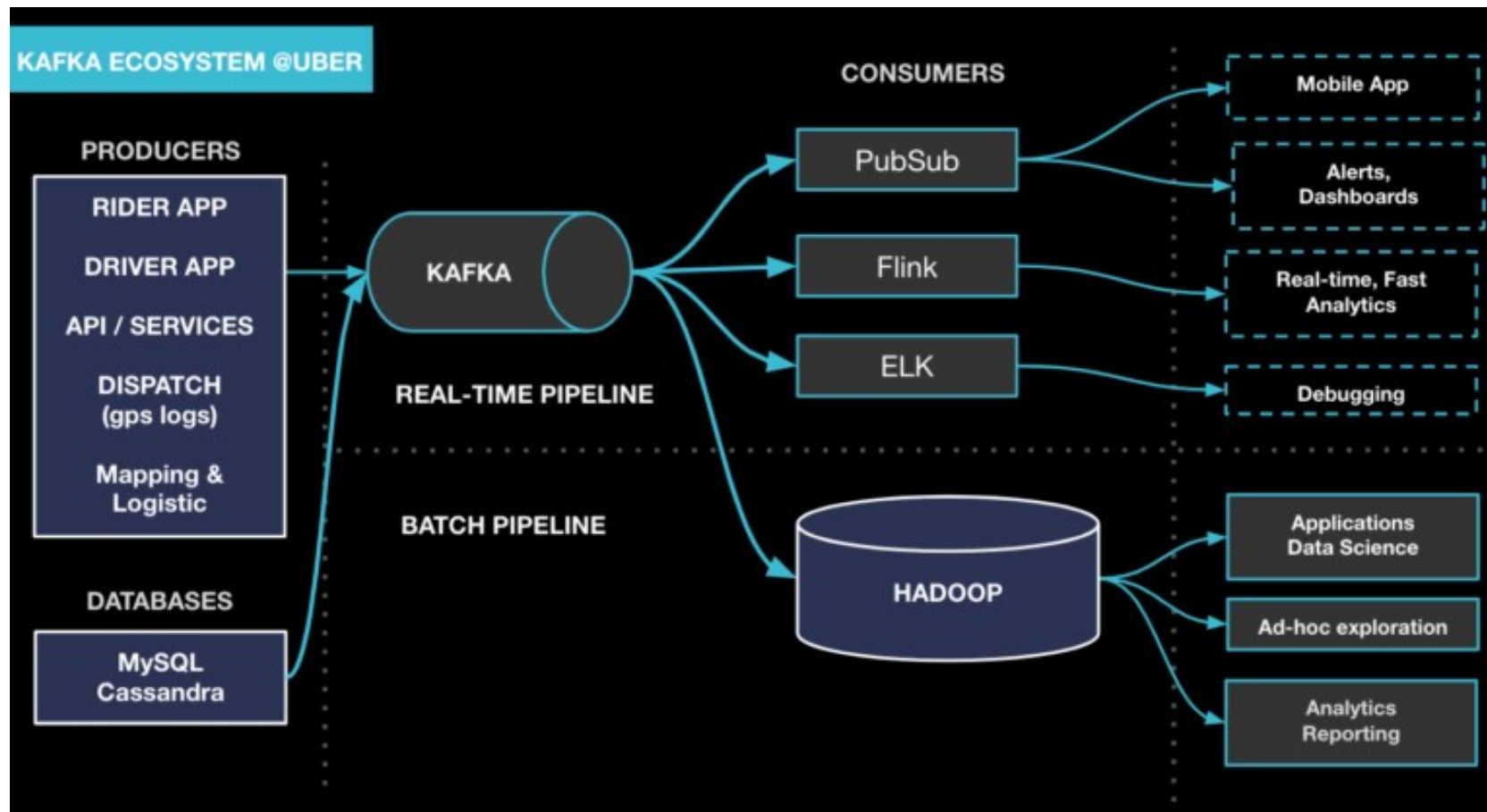
Ces événements peuvent alors être traités par des pipelines généralement destinées au stockage dans des solutions d'analyse temps-réel, d'alerting ou d'archivage souvent relié au machine learning

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
```

source

file

ftp

gemfire

gemfire-cq

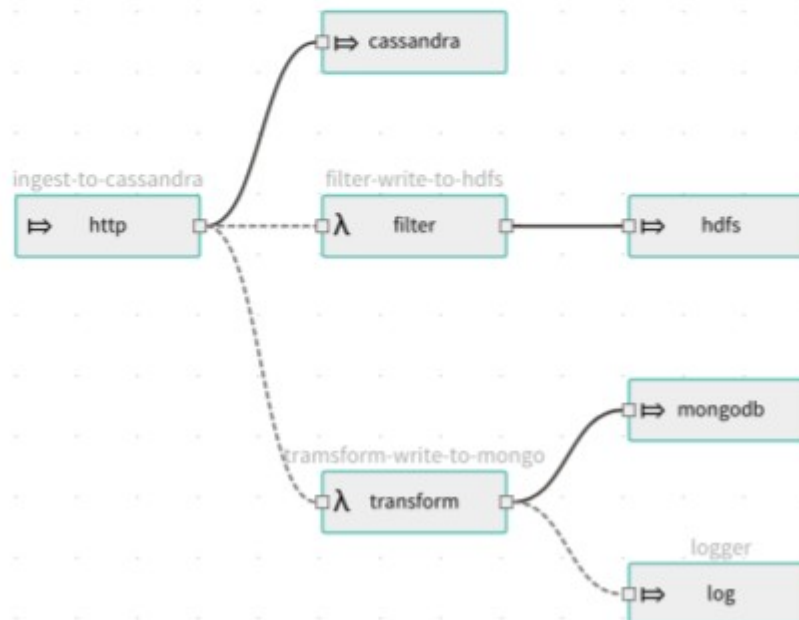
http

jdbc

jms

load-genera...

loggregator





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>



Event sourcing Pattern

Event sourcing Pattern¹ : Persiste un agrégat comme une séquence d'événements du domaine représentant les changements d'état

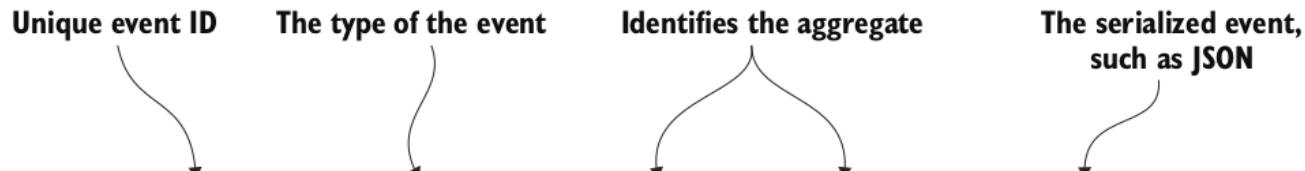
=> Une application recrée l'état courant d'un agrégat en rejouant les événements

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



Event Store

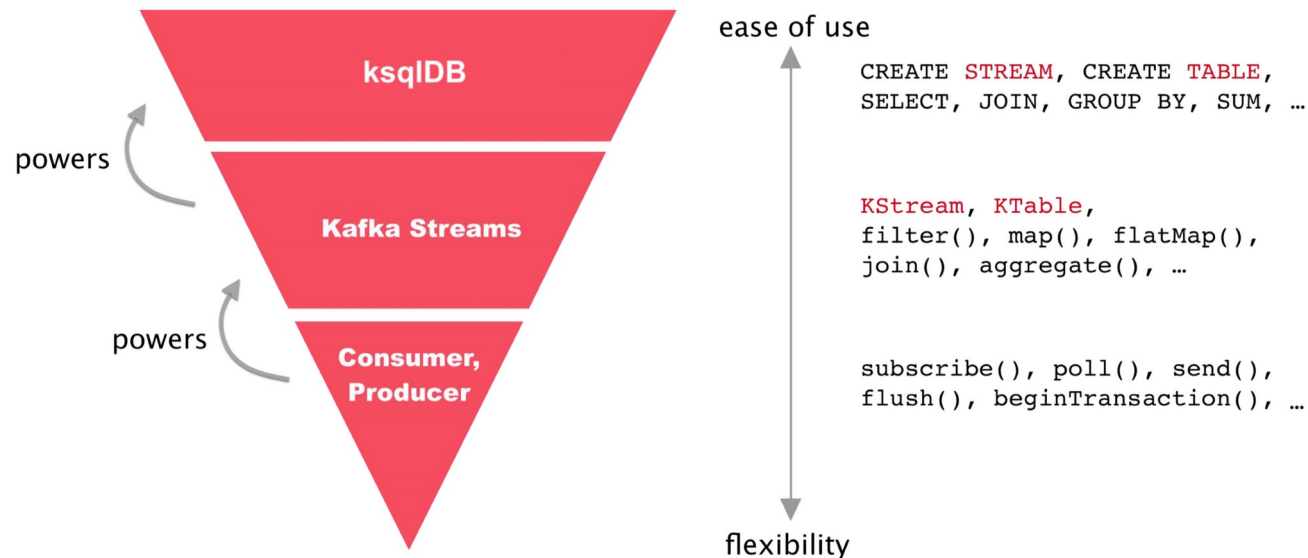
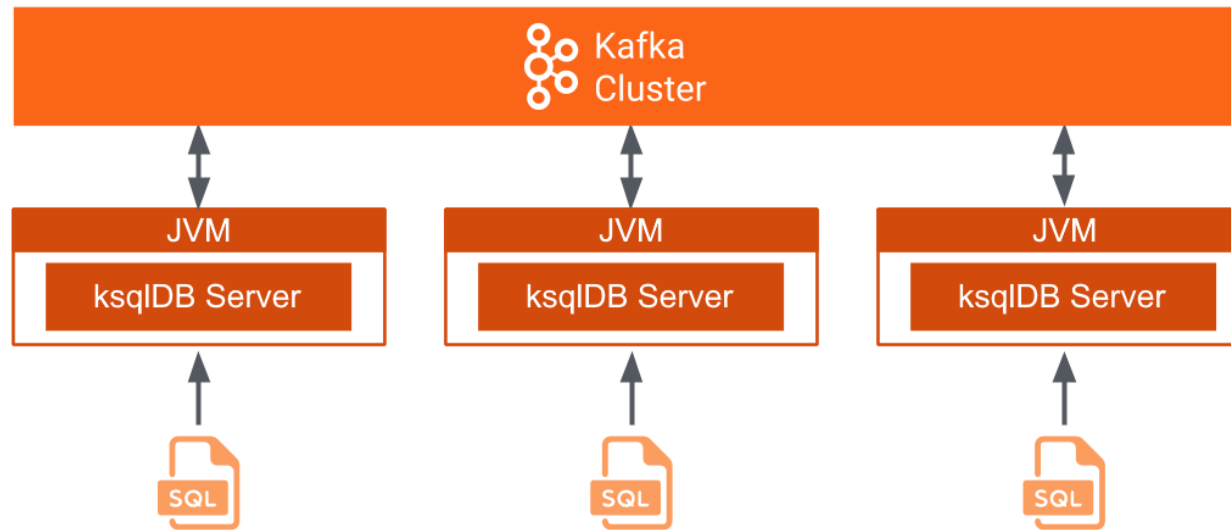
Au lieu de stocker l'agrégat dans un schéma traditionnel classique, l'agrégat est stocké sous forme d'événements dans un ***EventStore***



event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...

EVENTS table

ksqlDB Standalone Application (Headless Mode)





Apache Kafka et ses APIs

Le projet Kafka

Cas d'usage

Concepts coeur, Architecture

Producer API

Consumer APIs

Sérialisation Json, Avro

KafkaAdmin et KafkaStream



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.

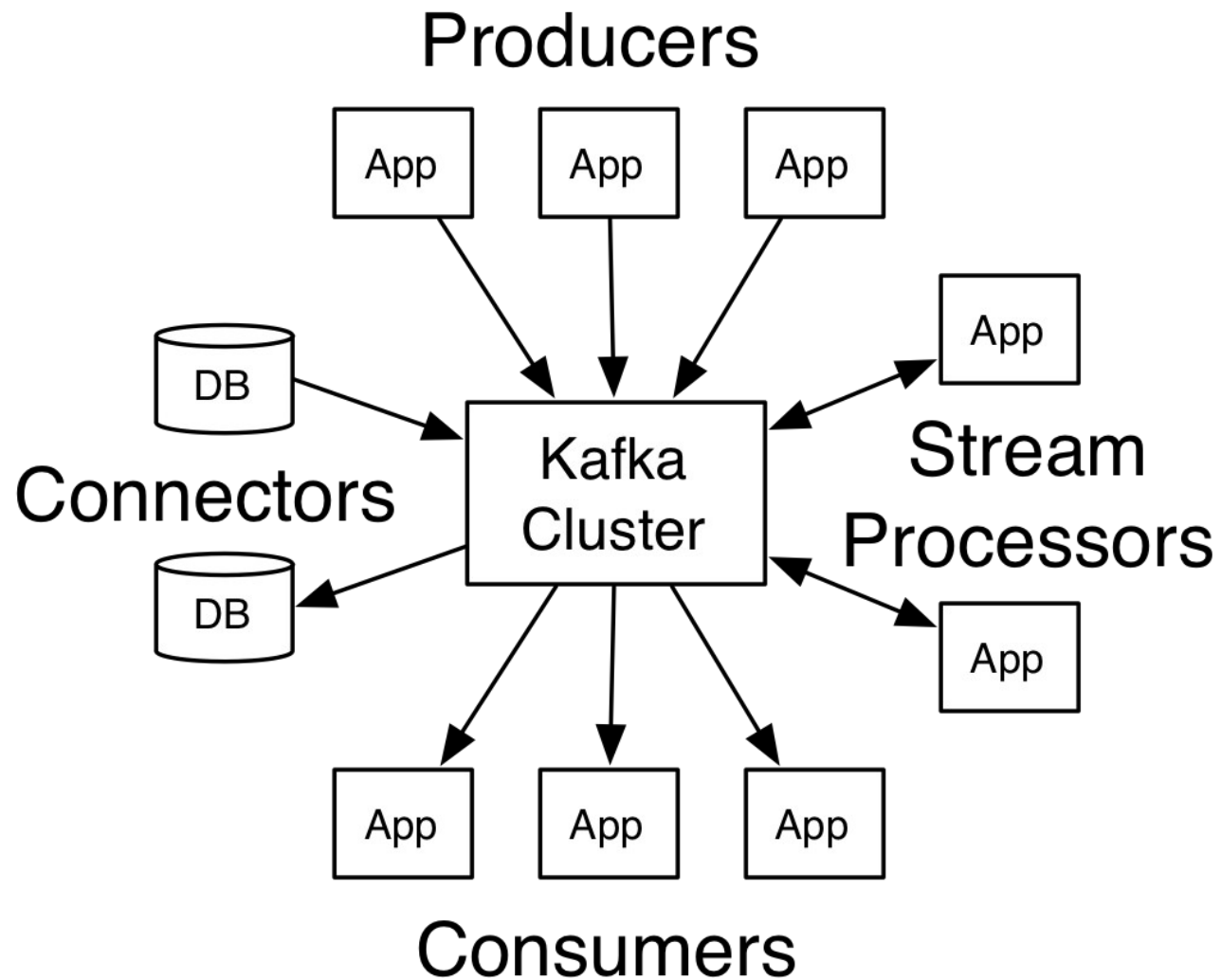


APIs

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





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

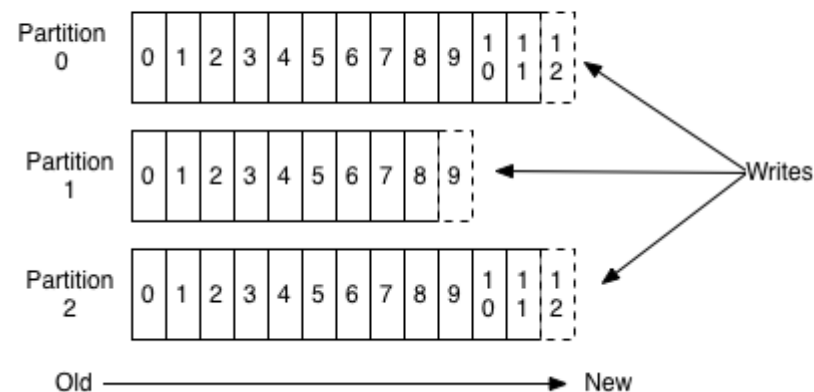
Les *records* sont publiés vers des **topics**.

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



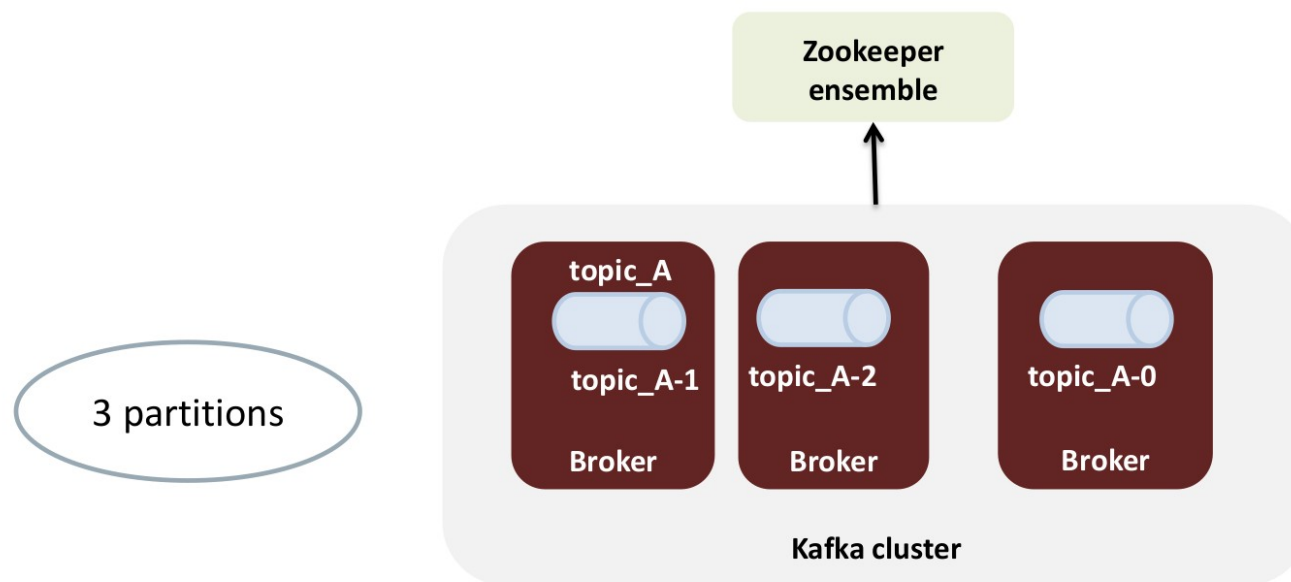


Apport des partitions

Les partitions autorisent le parallélisme et augmentent la capacité de stockage en utilisant les capacités disque de chaque nœud.

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

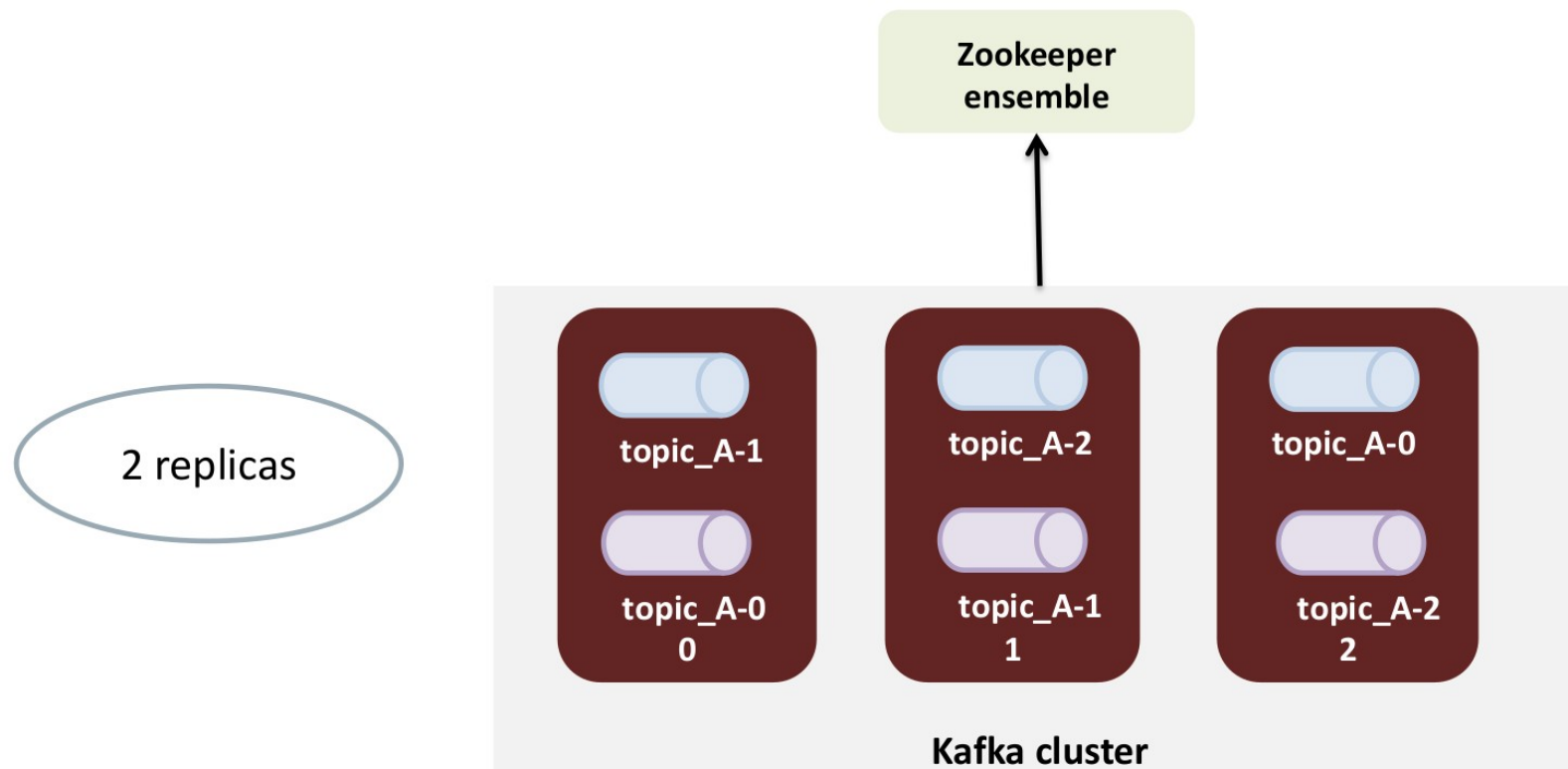
Le nombre de partition est fixé à la création du *topic*

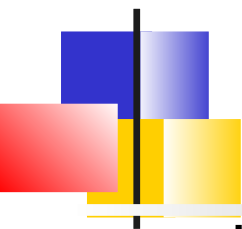


Réplication

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.

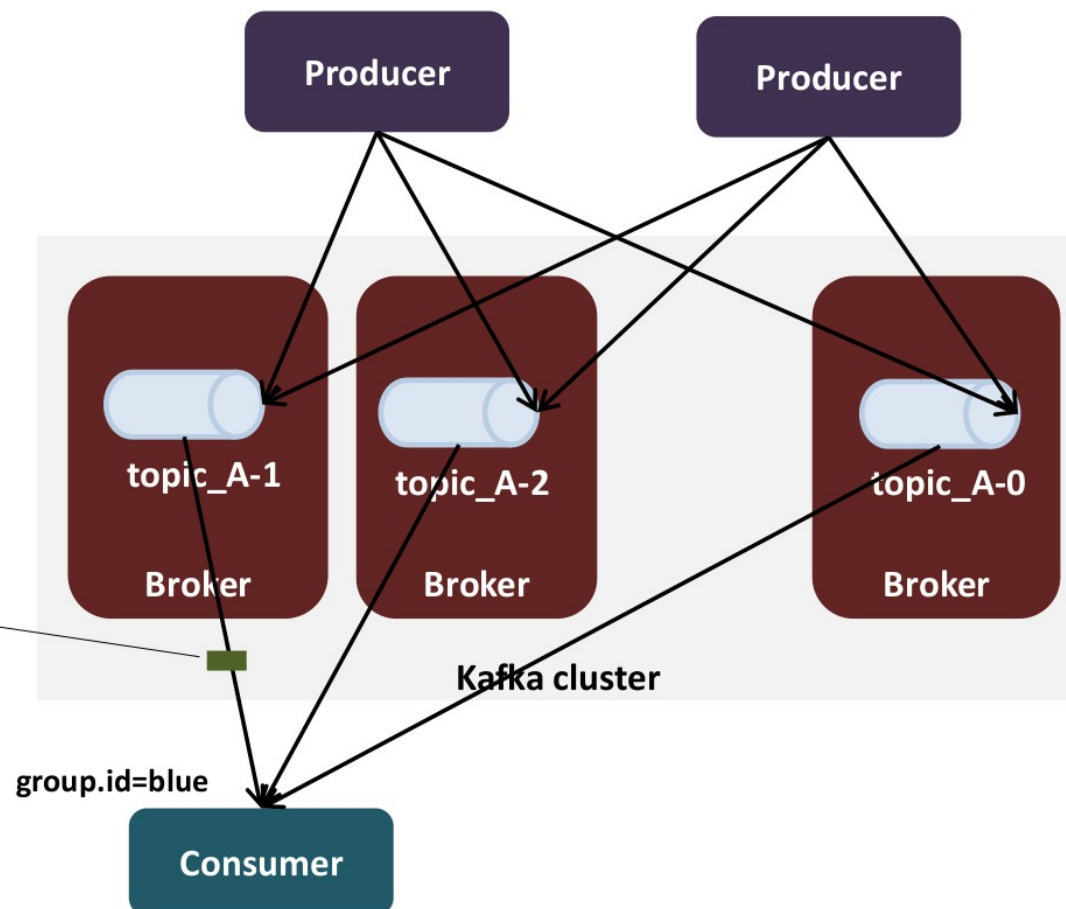
Le cluster Kafka conserve durablement tous les enregistrements publiés, qu'ils aient ou non été consommés, en utilisant une **période de rétention** configurable.

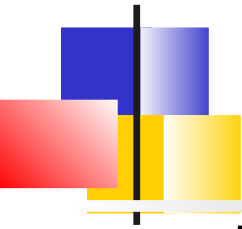
Clients du cluster

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

message (record, event)

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



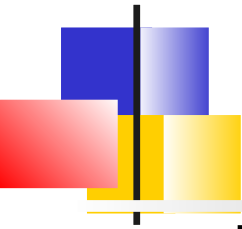


Routing des messages

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



Groupe de consommateurs

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

- Chaque enregistrement d'un topic est remis à une instance de consommateur au sein de chaque groupe.
- Les instances de consommateurs peuvent se trouver dans des threads, processus ou machines distincts.
=> Scalabilité et Tolérance aux fautes



Offset consommateur

La seule métadonnée conservée pour un groupe de consommateurs est son **offset** du journal.

Cet offset est contrôlé par le 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

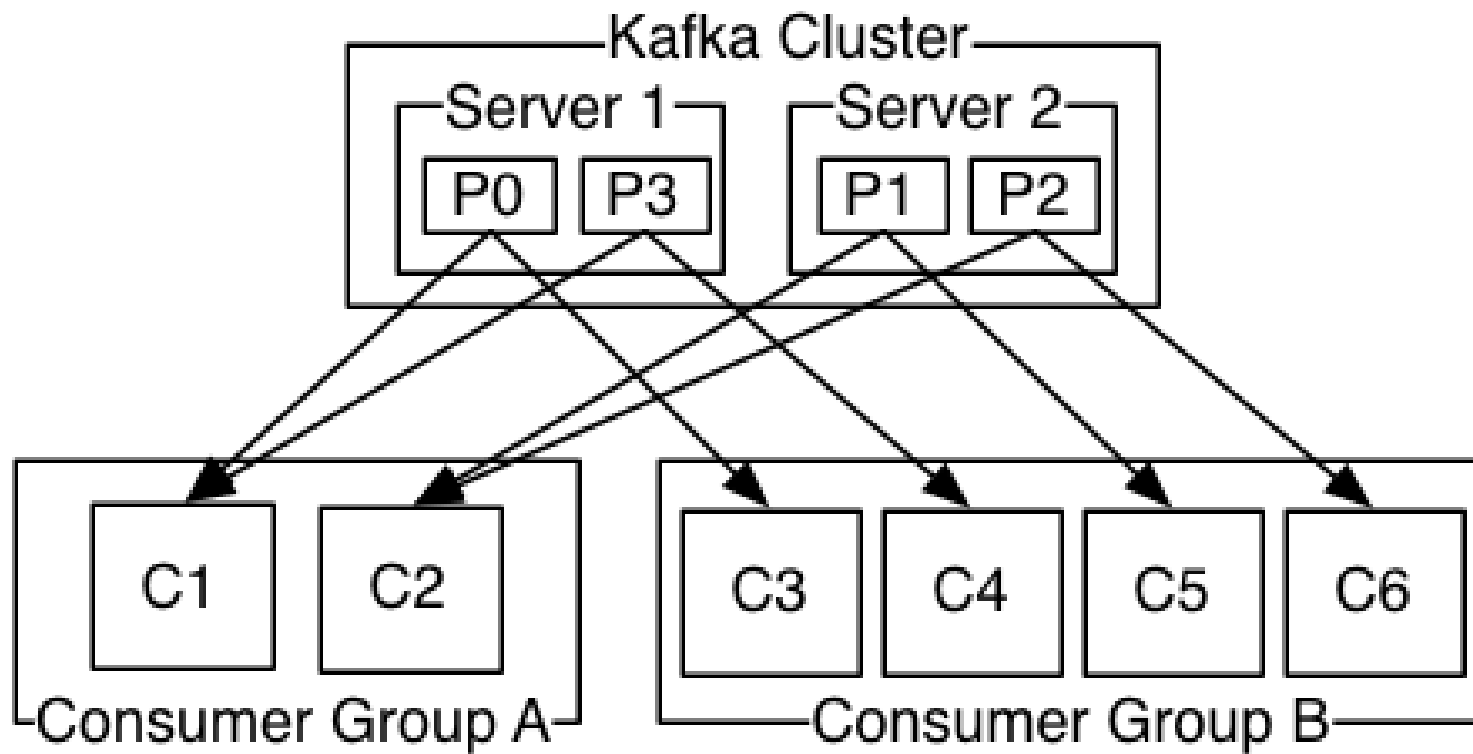
Kafka assigne les partitions à des instances de consommateur d'un même groupe.

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

Ceci est géré dynamiquement par le protocole Kafka.

- 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



Apache Kafka et ses APIs

Le projet Kafka
Cas d'usage
Concepts coeur, Architecture
Producer API
Consumer APIs
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 ?



Dépendances

Java

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

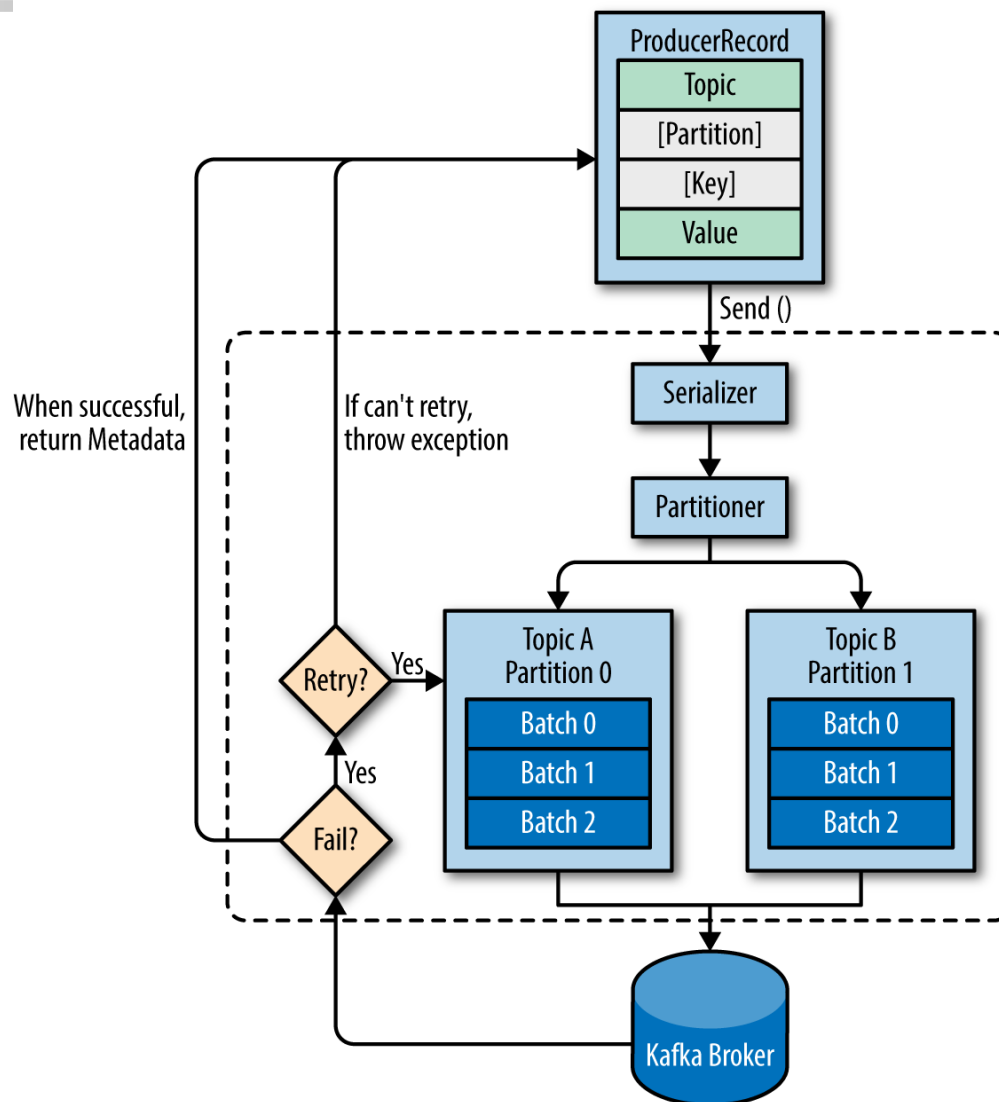


Étapes lors de l'envoi d'un message

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 de 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());
```

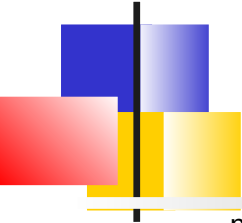


Sérialiseurs

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

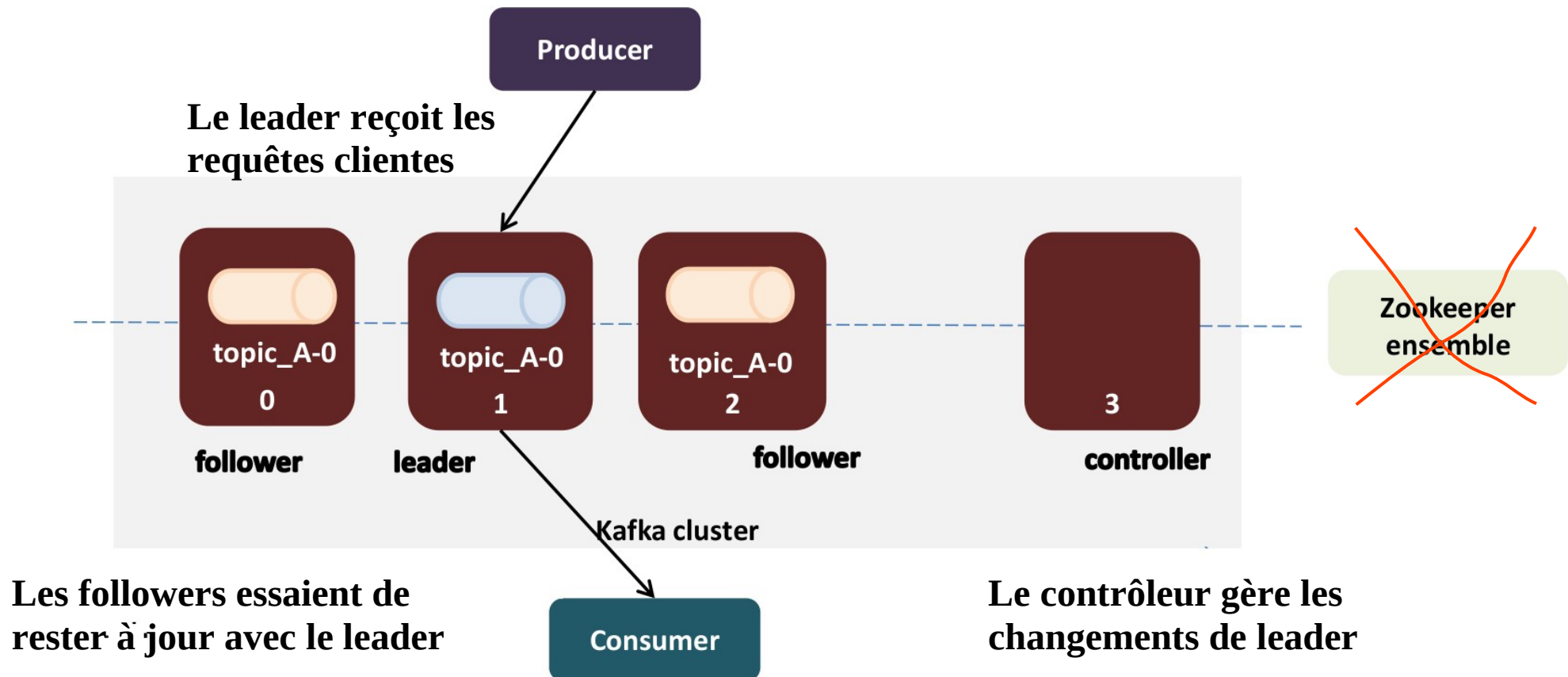
Différents brokers participent à la gestion distribuée et répliquée d'une partition.

- **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

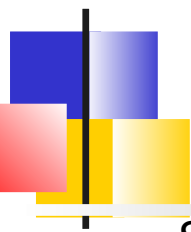
- A la réception d'un message, le leader vérifie si il y a assez d'ISR pour écrire le message, sinon il envoie une *NotEnoughReplicasException*
- Lorsque le message est répliqué par *min.insync.replica* répliques, le leader envoie un acquittement au client.



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

Le projet Kafka

Cas d'usage

Concepts coeur, Architecture

Producer API

Consumer APIs

Sérialisation Json, Avro

KafkaAdmin et KafkaStream



Introduction

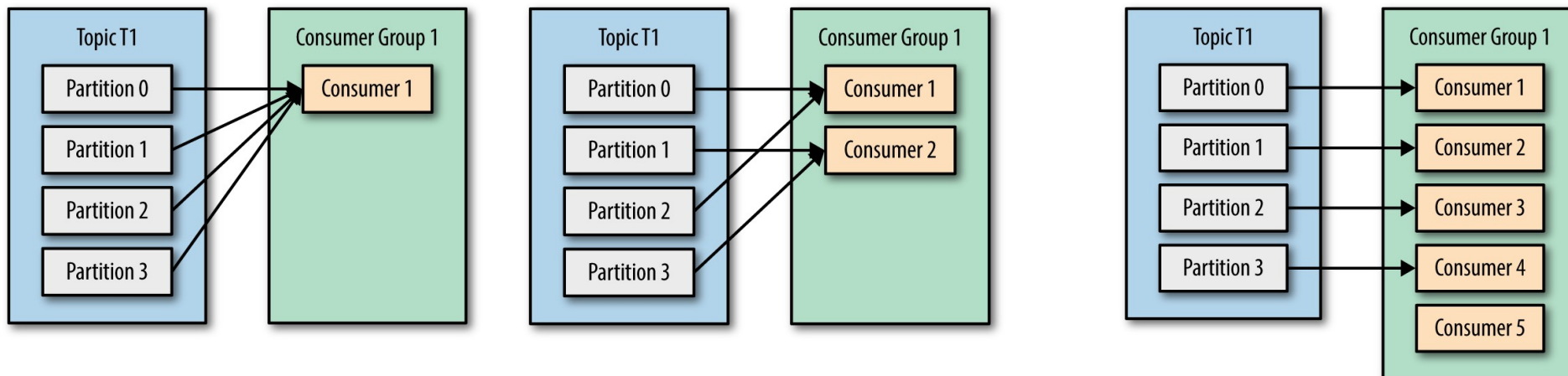
Les applications qui ont besoin de lire les données de Kafka utilisent un ***KafkaConsumer*** pour s'abonner aux topics

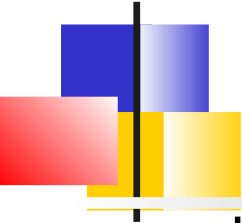
Pour bien comprendre l'API, il faut comprendre la notion de groupe de consommateurs et leurs relations avec les partitions

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.

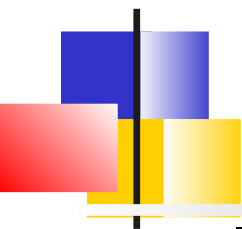
Les objets retournés par *poll* sont une collection de ***ConsumerRecord*** qui encapsule :

- 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***

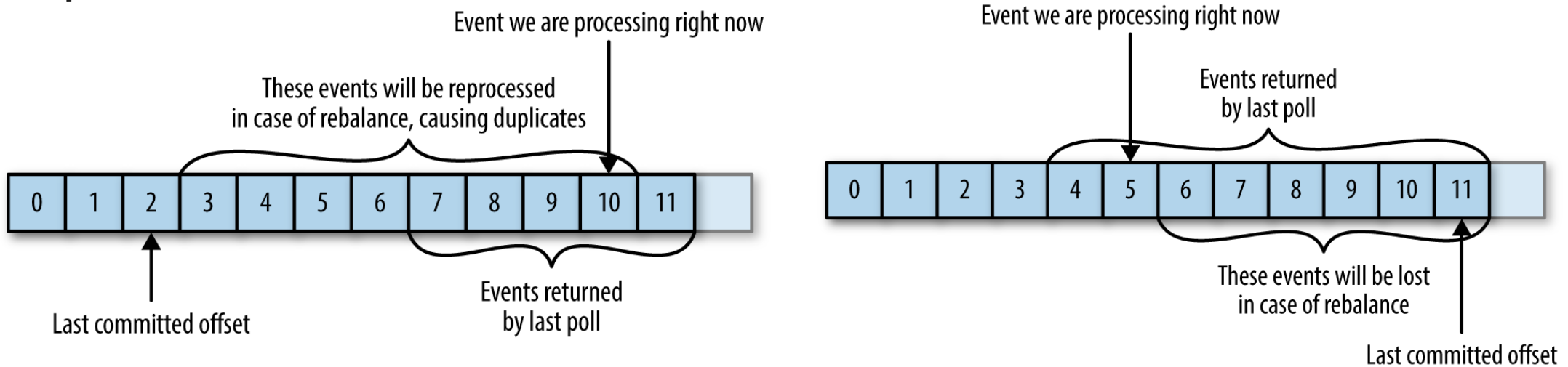
Pour committer, un consommateur envoie un message vers un *topic* particulier de Kafka : ***__consumer_offsets***

- 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 soimême, 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



APIs

Producer API
Consumer API
Frameworks
Connect API

Admin et Stream API



Introduction

Kafka propose 2 autres APIs :

- ***Admin API*¹** : Client d'administration permettant de gérer et inspecter les topics, brokers, configurations et ACLs
- ***Streams API*²** : Librairie cliente pour des micro-services dont les entrées/sorties sont des *topics* Kafka

1. API existante dans les autres langages chez Confluent
2. Équivalent Python : *Faust*



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

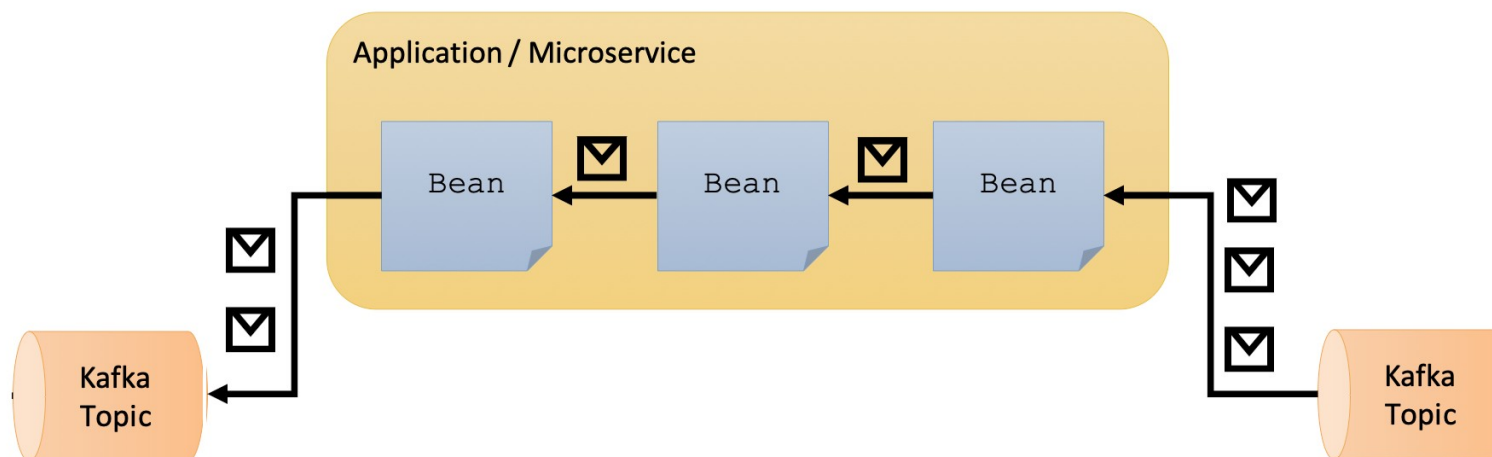
Créer un topic

```
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 transformation de flux d'évènements infinis, des jointures et des 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.
- Offre les primitives de traitement de flux nécessaires sous forme de DSL haut niveau ou d'une API bas niveau.



Définitions

Un ***KStream*** est une abstraction représentant un ensemble de données illimité, c'est à dire infini et sans cesse croissant car de nouveaux enregistrements continuent d'arriver

D'autres attributs caractérisent ce flux d'événements :

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



KStream et KTable

KafkaStream fournit également l'abstraction ***KTable*** qui représente un ensemble de fait qui évoluent.

- Une *KTable* peut être construit à partir d'un *Kstream*. Seule la dernière valeur ou une agrégation d'une clé donnée est conservée
- Un *KStream* peut effectuer des jointures sur une *KTable* et produit alors un *KStream* enrichi
- Une *KTable* peut être transformé en *Kstream* contenant les événements d'update



Agrégation, Jointure, fenêtrage

Une opération **d'agrégation** prend un flux d'entrée ou une table et génère une nouvelle table en combinant plusieurs enregistrements d'entrée en un seul enregistrement de sortie.

– Exemples : Somme, moyenne, ...

Une opération **de jointure** prend plusieurs flux d'entrée et fournit un flux de sortie

Le **fenêtrage** permet de contrôler comment regrouper des enregistrements qui ont une même clé pour des opérations d'agrégations ou des jointures.



State store

Certaines applications (stateful) nécessitent de conserver un état pour grouper, joindre, agréger

Kafka Streams fournit des **State stores** qui permettent aux applications de stocker

On peut y définir des *interactive queries* permettant un accès en lecture à ces données données.

Tout cela dans un contexte de tolérance aux pannes



ksqlDB

ksqlDB fournit donc une couche d'abstraction SQL permettant de manipuler les *KTable* de *KafkaStream*

Via le modèle SQL, il est alors possible de faire tout ce dont on a besoin avec les données en mouvement : acquisition, traitement et interrogation des données.



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

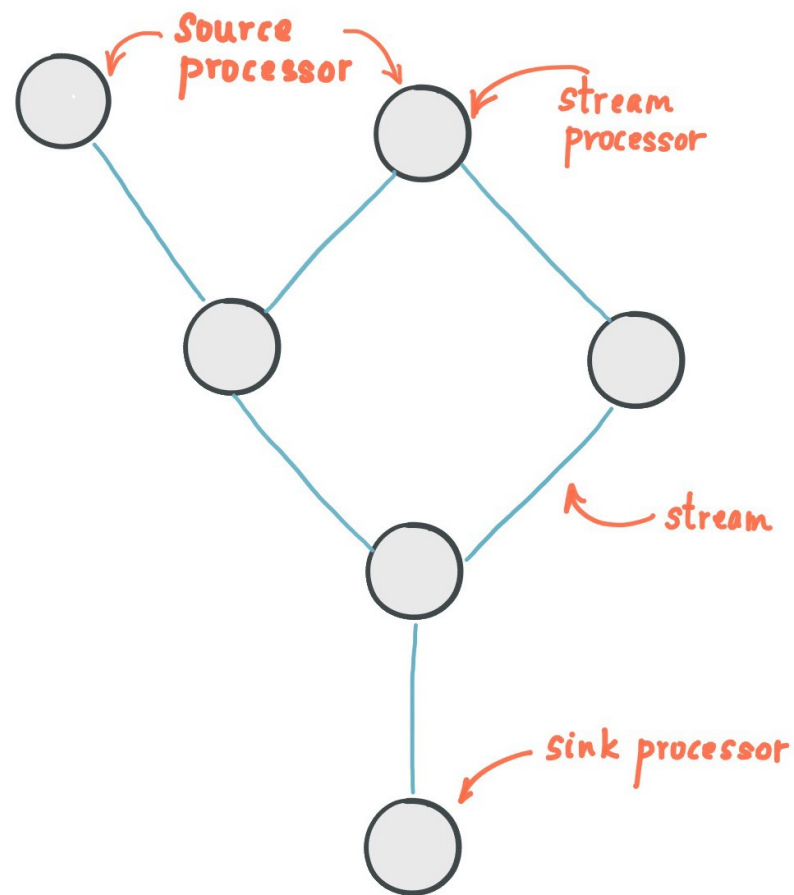
Un processeur représente une étape de traitement qui prend en entrée un événement, le transforme puis produit un ou plusieurs événements

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

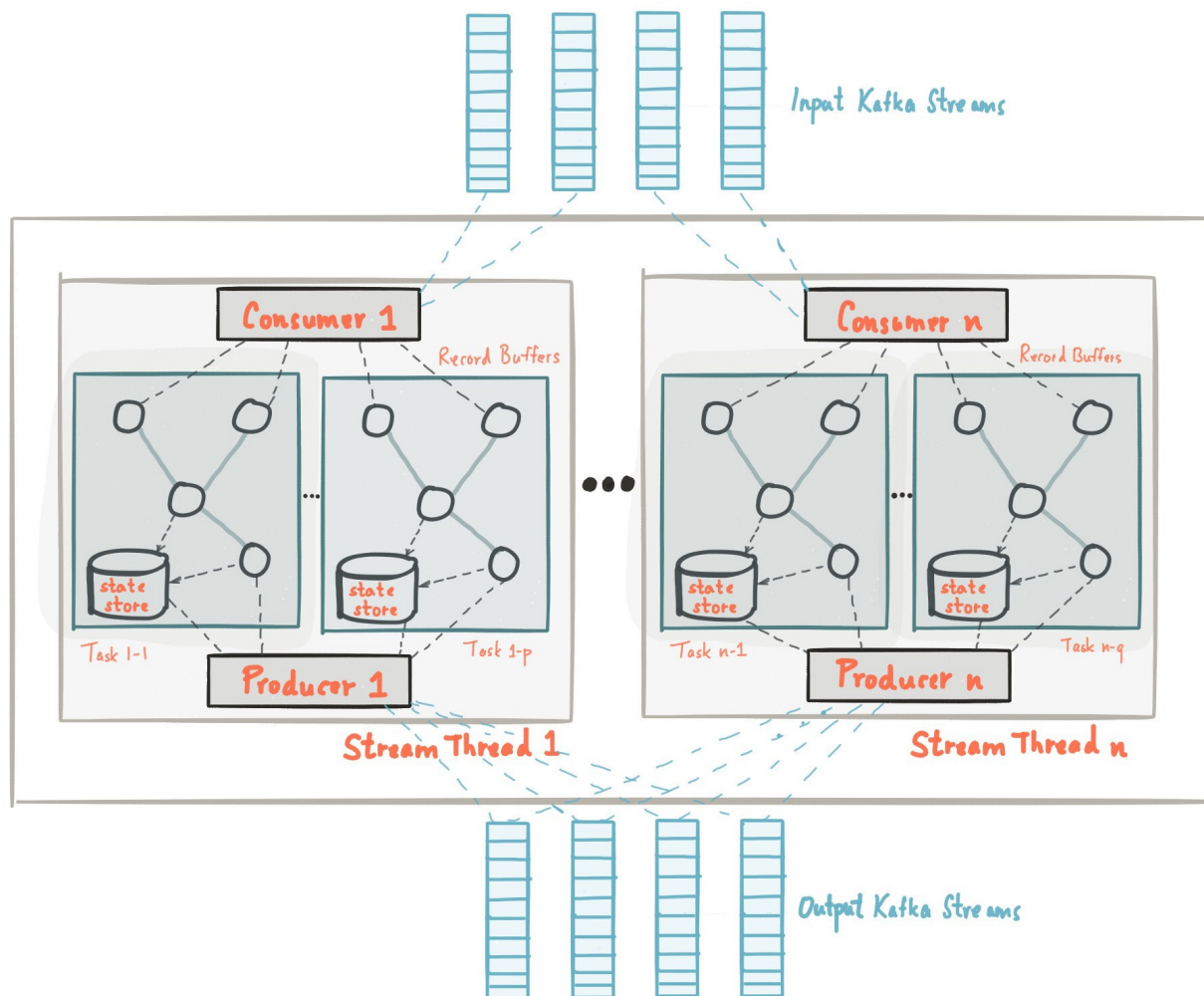
La topologie peut être spécifiée programmatiquement ou par un DSL

Topologie processeurs



PROCESSOR TOPOLOGY

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);
```



Garanties Kafka

Mécanismes de réplication

At Most One, At Least One

Exactly Once

Débit, latence, durabilité



Introduction

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

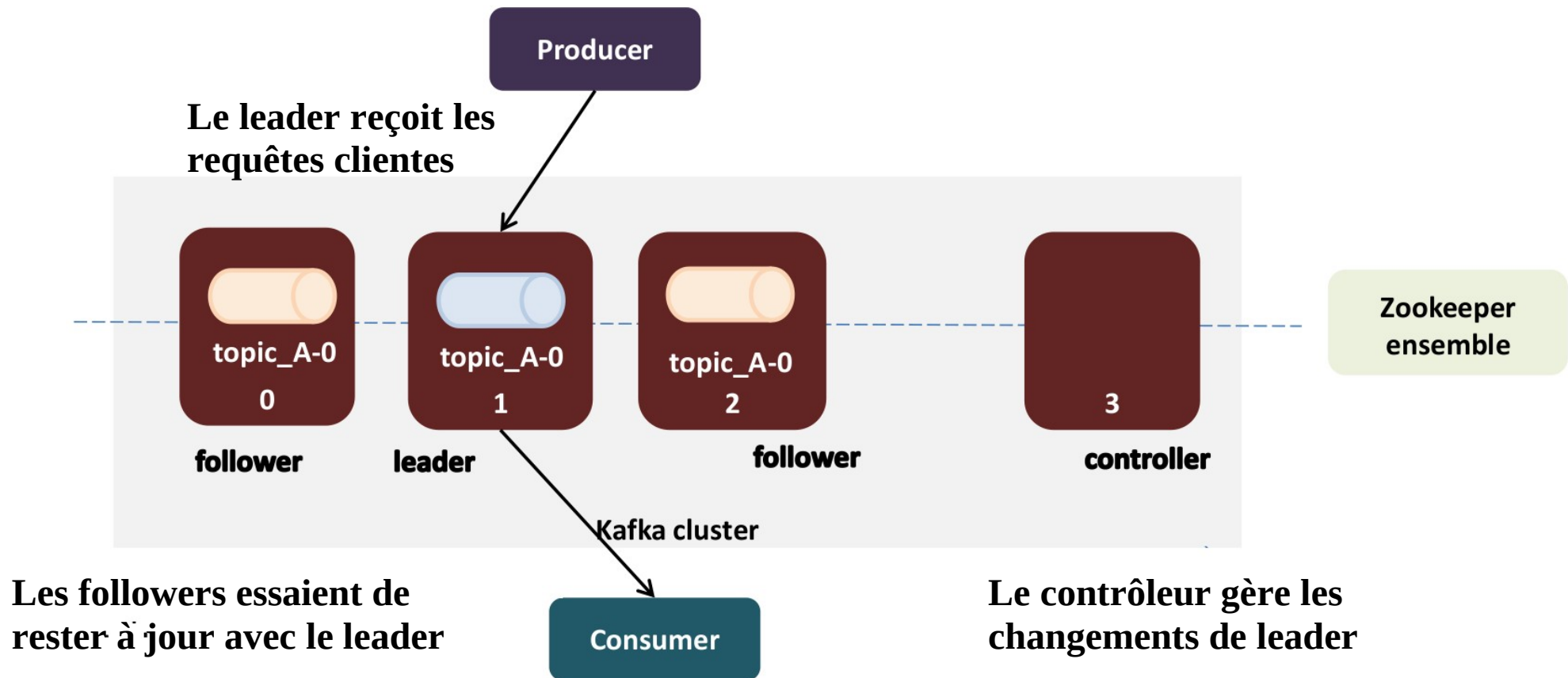
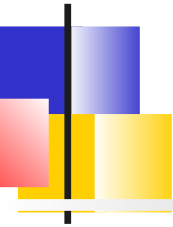
Pour chaque partition d'un topic :

- **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 “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 sont 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



Rôles des brokers vis à vis de l'ISR

Leader

- Une réplique élue pour chaque partition
- Qui reçoit toutes les requêtes des producteurs et consommateurs
- Gère la liste de ses ISR

Followers

- Essaie de rester à jour avec le leader
- Si le leader tombe, un follower devient le nouveau leader

Controller

- Responsable pour élire les leaders de partition et diffuser l'information au leader et aux ISR
- Persiste le nouveau leader et l'ISR vers *Zookeeper* ou les disques des contrôleurs *Kraft*



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*

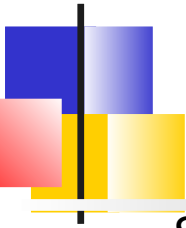
- A la réception d'un message, le leader vérifie si il y a assez d'ISR pour écrire le message, sinon il envoie une *NotEnoughReplicasException*
- Lorsque le message est répliqué par *min.insync.replica* répliques, le leader envoie un acquittement au client.



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** $< \text{min.insync.replicas}$:

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

Si le **nombre de répliques disponible** $< \text{min.insync.replicas}$

- Mauvaise configuration, Kafka bloque les émissions de message

En conséquences :

n répliques

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

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

=> 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 (valeurs hautes) et matériel (valeur basse)
Valeur classique 3
- ***min.insync.replicas*** (défaut 1, au niveau cluster ou topic)
Le minimum de répliques qui doivent acquitter pour valider un message
Compromis entre perte de données (les répliques synchronisées tombent) et ralentissement
Valeur classique 2/3



Garanties Kafka

Mécanismes de réplication

At Most Once, At Least Once

Exactly Once

Débit, latence, durabilité



Côté producteur

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

- La configuration des **acks** en fonction du *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*)





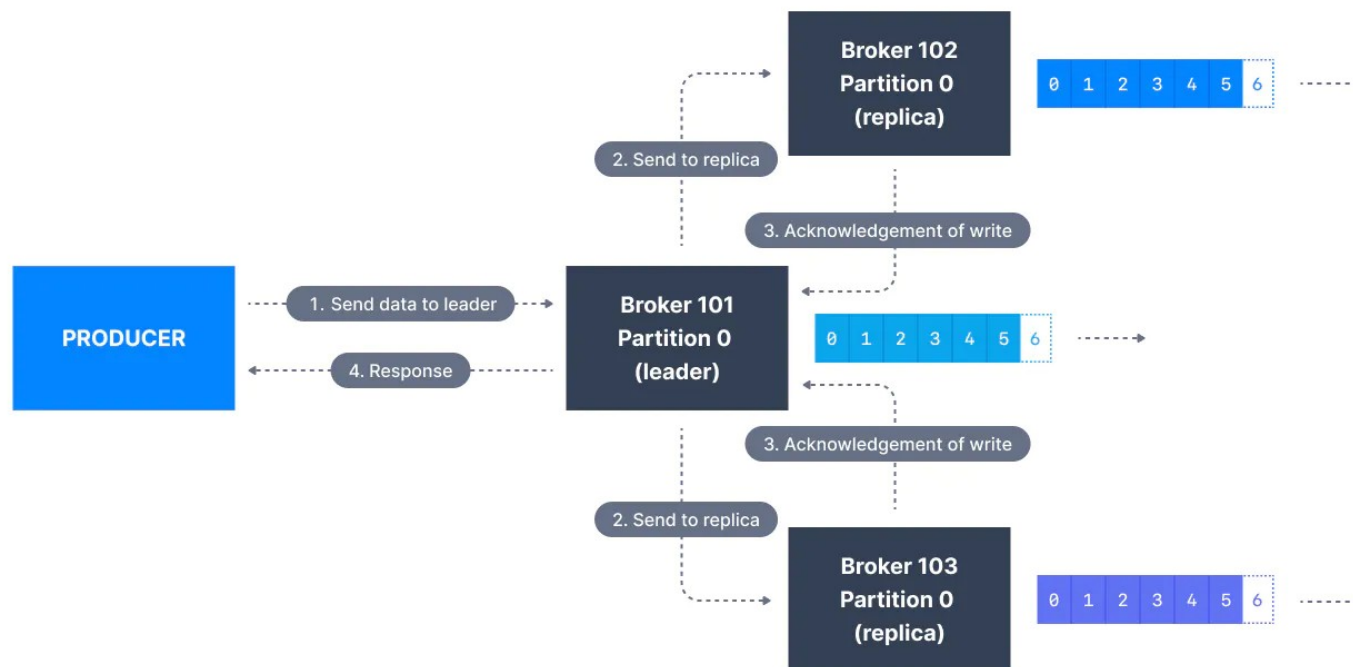
acks=1

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

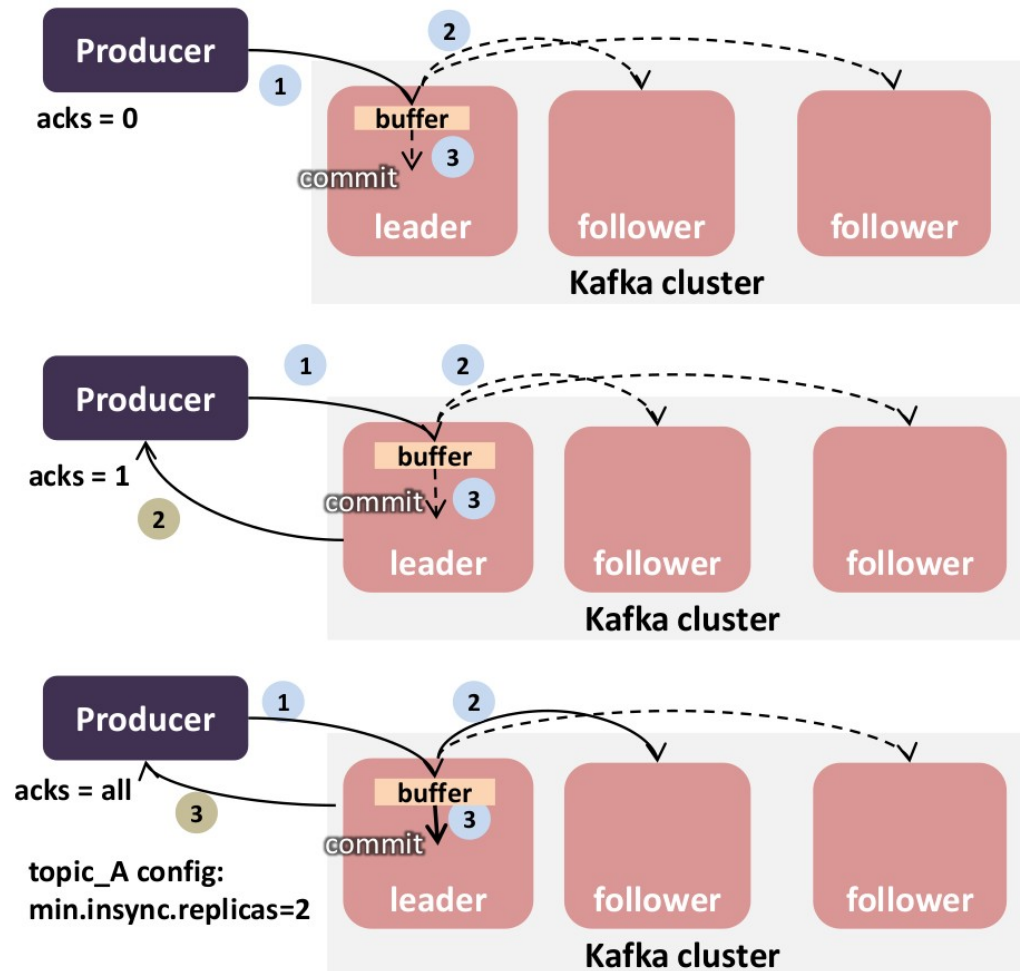


acks=all

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*)



Acquittement et durabilité



Latency

Data loss risk





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)



Rebalancing

En cas de crash ou d'ajout d'un consommateur, Kafka réaffecte les partitions.

Lors d'une réaffectation, un consommateur récupère l'offset de lecture auprès de Kafka

Si l'offset n'est pas synchronisé avec les traitements effectués, cela peut :

- Générer des traitements en doublon
- Perdre des traitements de message

Il est possible de s'abonner aux événements de rebalancing



Côté consommateur

Du point de vue de la fiabilité, la seule chose que les consommateurs ont à faire est de s'assurer qu'ils gardent une trace des offsets qu'ils ont traités en cas de rebalancing.

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

=> La seule façon de perdre des messages est alors de committer des offsets de messages lus mais pas encore traités



Configuration

3 propriétés de configuration sont importantes pour la fiabilité du consommateur :

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

enable.auto.commit : Commit manuel ou non

- Automatique : Si tout le traitement est effectué dans la boucle de poll. Garantie que les offsets commités ont été traités mais pas de contrôle sur le nombre de doublons
Attention si le traitement est fait dans une thread différente de la boucle de poll

auto.commit.interval.ms : Valable en mode automatique

- Diminuer l'intervalle ajoute de la surcharge mais réduit le risque de doublon lors d'un arrêt de consommateur

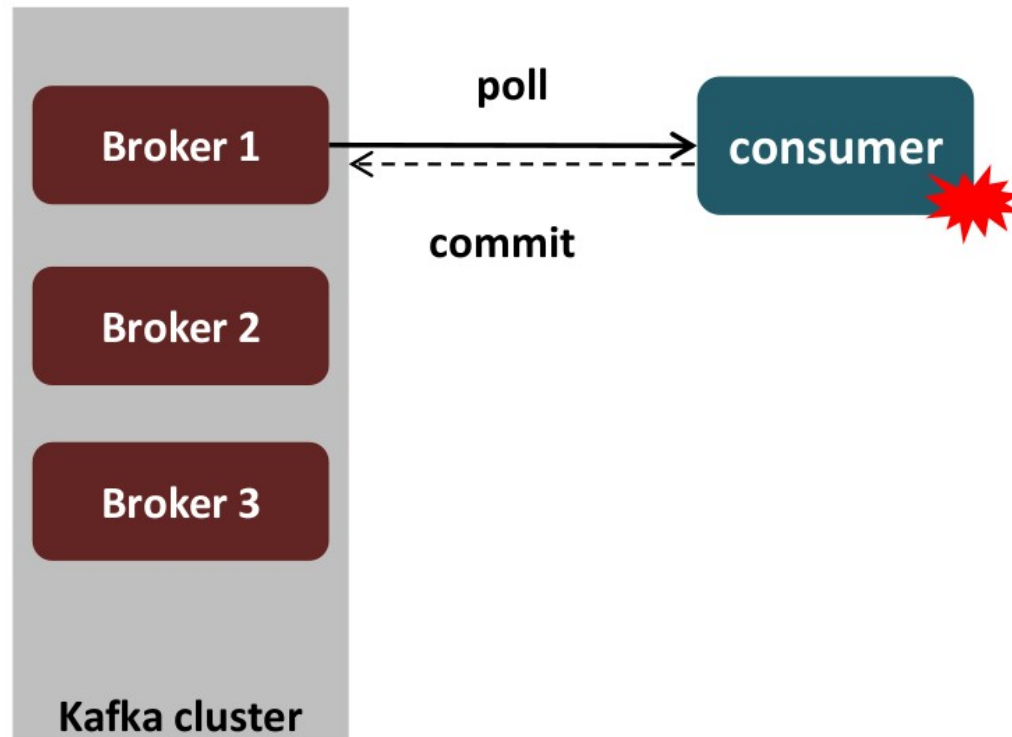


Auto commit

En mode automatique, le commit des offset est effectué lors de l'appel à *poll()*.

Si lors d'un appel à poll, *auto.commit.interval.ms* a été atteint les offsets du dernier poll sont committés

Réception : 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

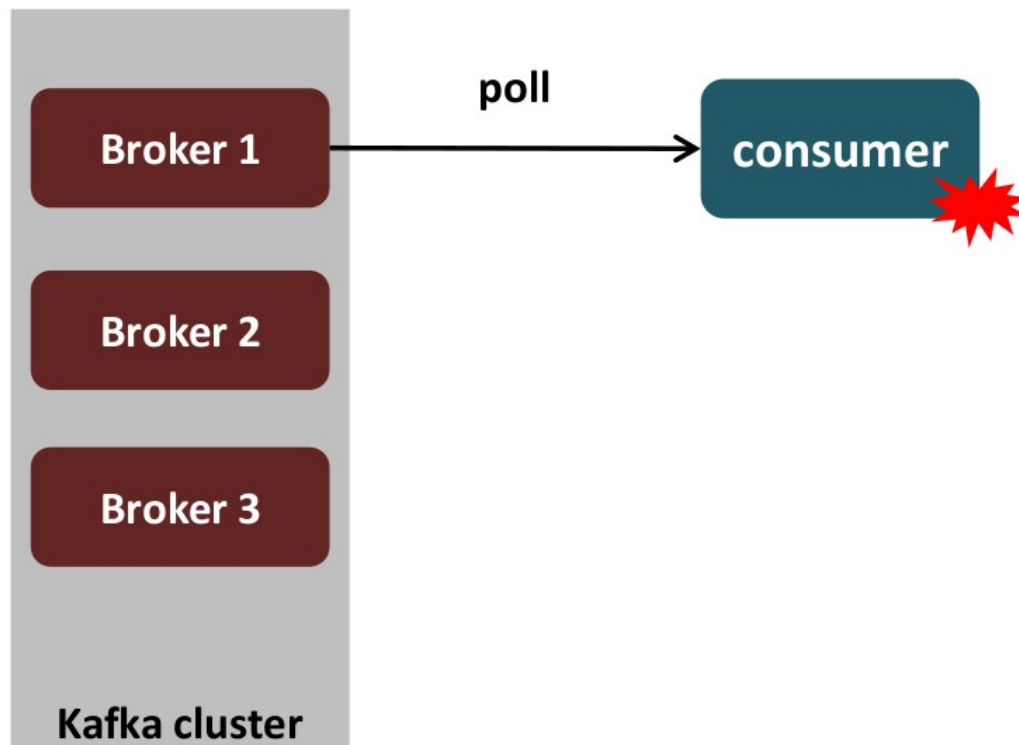
Commit manuel avant le traitement des messages

Exemple : Auto-commit et traitement asynchrone

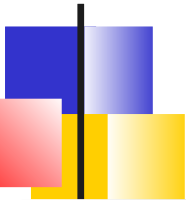
```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
    long lastOffset = 0;  
    for (ConsumerRecord<String, String> record : records) {  
        System.out.printf("\n\roffset = %d, key = %s, value = %s", record.offset(), record.key(),  
            record.value());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




Réception : *At Least Once*



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



Configuration *At Least Once*

Configuration par défaut et traitement synchrone dans la boucle de poll

Ou

enable.auto.commit à 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();  
}
```

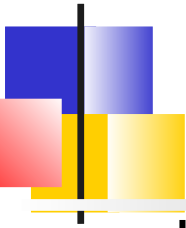


Validation de la fiabilité

Il est nécessaire de valider si sa configuration remplit les garanties voulues.

La validation s'effectue à 3 niveaux :

- Validation de la configuration
- Validation de l'application
- Surveillance



Validation de la configuration

Kafka fournit 2 outils permettant de tester la configuration en isolation de l'application :

- ***kafka-verifiable-producer.sh*** produit une séquence de messages contenant des nombres en séquence. On peut configurer le nombre de ack, de retry et les cadences des messages
- ***kafka-verifiable-consumer.sh*** consomme les messages et les affiche dans l'ordre de consommation. Il affiche également des informations sur les commit et les rééquilibrage

On peut alors exécuter ces commandes pendant différents scénarios de test : Élection de leader, de contrôleur, redémarrage des brokers, ...



Monitoring

Kafka propose des métriques JMX.

Pour la fiabilité du producteur :

- ***error-rate*** et ***retry-rate*** permettent de déceler des anomalies systèmes.
- Également voir les traces du producteur (WARN)

Pour la fiabilité du consommateur :

- ***consumer lag*** : Indique le décalage des consommateurs



Garanties Kafka

Mécanismes de réplication
At Most Once, At Least Once
Exactly Once
Débit, latence, durabilité

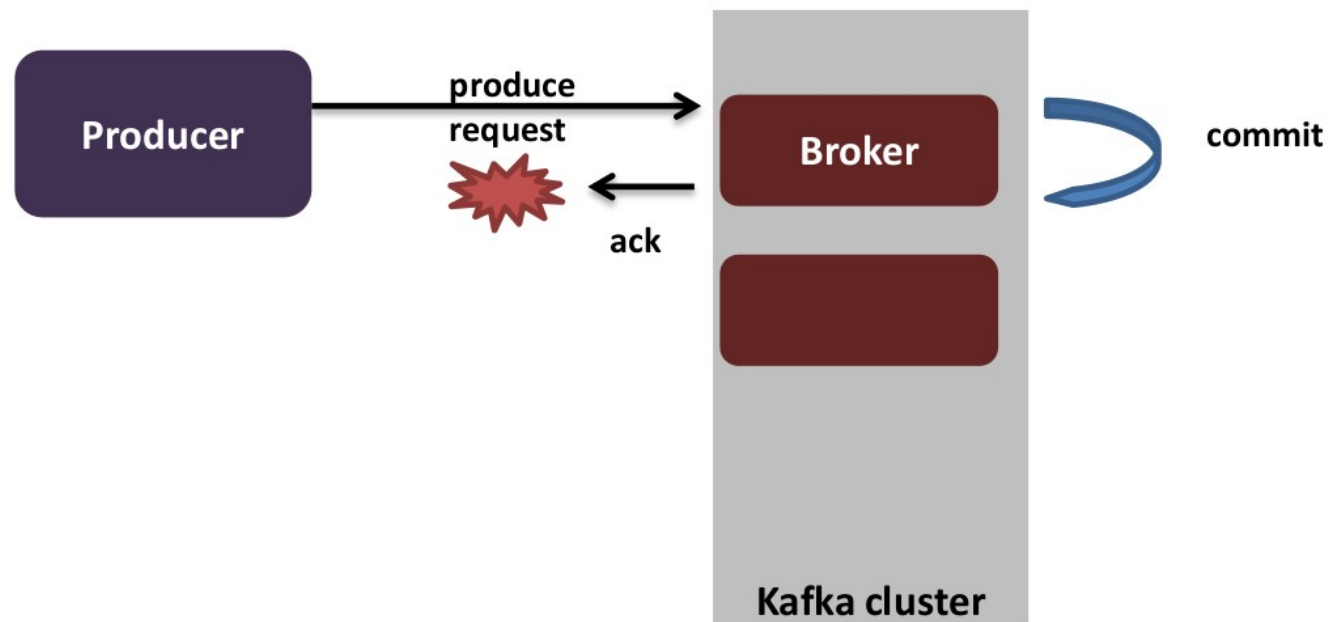


Introduction

La sémantique *Exactly Once* est basée sur *At least Once* 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

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 et 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 transactions¹.

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

Pour utiliser le mode transactionnel et l'API correspondante, il faut positionner 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
- Les consommateurs doivent avoir la propriété *isolation.level* à *read committed*
- L'API *send()* devient bloquante



Transaction

Permet la production atomique de messages sur plusieurs partitions

Les producteurs produisent l'ensemble des messages ou aucun


```
// initTransaction démarre une transaction avec l'id.  
// Si une transaction existe avec le même id, les messages sont roll-backés  
producer.initTransactions();  
try {  
    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 lire les messages transactionnels validés :

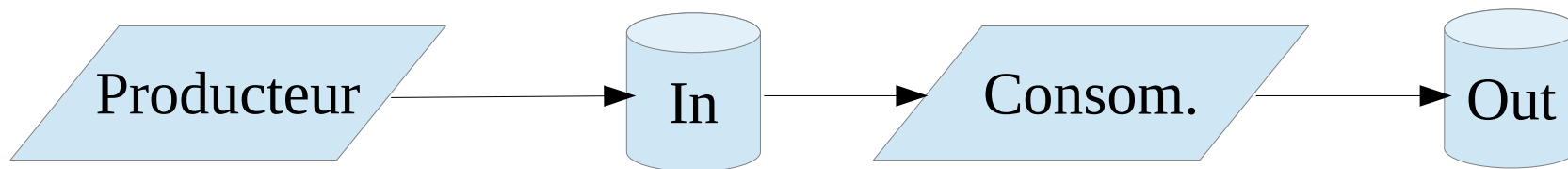
- ***isolation.level=read_committed***
(Défaut: *read_uncommitted*)
 - *read_committed*: Messages (transactionnels ou non) validés
 - *read_uncommitted*: Tous les messages (même les messages transactionnels non validés)



Exactly Once pour le transfert de messages entre topics

Lorsque un consommateur produit vers un autre topic, on peut utiliser les transactions 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();  
}
```



Garanties Kafka

Mécanismes de réplication
At Most Once, At Least Once
Exactly Once
Débit, latence, durabilité



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

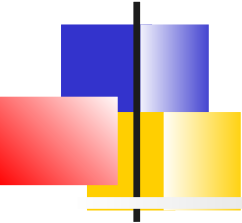
- *replication.factor*: 3
- *min.insync.replicas*: 2 (défaut 1)
- *unclean.leader.election.enable* : false (défaut false)
- *broker.rack*: rack du broker (défaut null)

Producteur

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

Consommateur

- *isolation.level: read_committed*



Spring-Kafka

Introduction
Production de message
Consommation



Connexion à Kafka

Spring permet 3 types de client aux clusters Kafka

- KafkaAdmin : Utilisé principalement pour configurer les topics
- ProducerFactory : Pour la production de messages
- ConsumerFactory : Pour la consommation



ApplicationEvents



Configuration des topics

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

Le builder `TopicBuilder` permet de construire de tel Beans



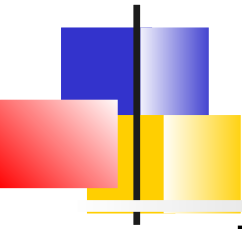
Examples

```
@Bean
public NewTopic topic1() {
    return TopicBuilder.name("thing1")
        .partitions(10)
        .replicas(3)
        .compact()
        .build();
}

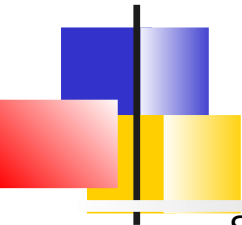
@Bean
public NewTopic topic2() {
    return TopicBuilder.name("thing2")
        .partitions(10)
        .replicas(3)
        .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
        .build();
}

@Bean
public NewTopic topic3() {
    return TopicBuilder.name("thing3")
        .assignReplicas(0, Arrays.asList(0, 1))
        .assignReplicas(1, Arrays.asList(1, 2))
        .assignReplicas(2, Arrays.asList(2, 0))
        .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
        .build();
}

@Bean
public NewTopic topic4() {
    return TopicBuilder.name("defaultBoth")
        .build();
}
```



By default, if the broker is not available, a message is logged, but the context continues to load. You can programmatically invoke the admin's `initialize()` method to try again later. If you wish this condition to be considered fatal, set the admin's `fatalIfBrokerNotAvailable` property to `true`. The context then fails to initialize.



Starting with version 2.7, the `KafkaAdmin` provides methods to create and examine topics at runtime.

`createOrModifyTopics`

`describeTopics`

For more advanced features, you can use the `AdminClient` directly. The following example shows how to do so:

```
@Autowired
```

```
private KafkaAdmin admin;
```

```
...
```

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

```
...
```

```
client.close()
```



Spring-Kafka

Introduction
Production de message
Consommation



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 template



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* 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)

`@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)

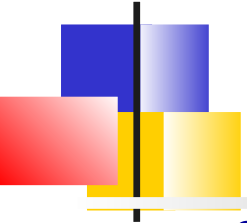
Envoi de message

@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());
    };
}
```

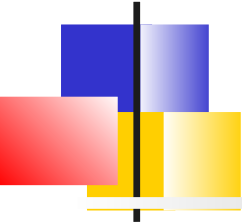
Exemple (3)



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

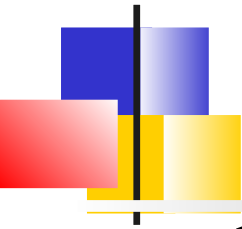
Le topic de réponse est indiqué dans l'entête
KafkaHeaders.REPLY_TOPIC.

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



2 façon de consommer des messages :

- Définir un `MessageListenerContainer` puis un *MessageListener*
- Utiliser l'annotation `@KafkaListener`



MessageListener

MessageListener est l'interface permettant de consommer les ConsumerRecords 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 Consumer Kafka sous-jacent



Interface *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, partitions à écouter. 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

Constructeur :

```
public KafkaMessageListenerContainer(  
    ConsumerFactory<K, V> consumerFactory,  
    ContainerProperties containerProperties  
)  
  
public ConcurrentMessageListenerContainer(  
    ConsumerFactory<K, V> consumerFactory,  
    ContainerProperties containerProperties  
)
```

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 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 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) { ... }
}
```

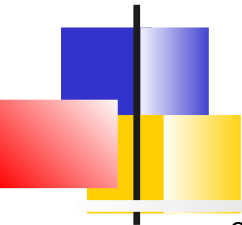


Rééquilibrage des listeners

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 spécifié

```
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 permettant le support classique Spring (@Transactional, TransactionTemplate etc) et la synchronisation avec d'autres TransactionManager (BD par exemple)
- ***KafkaMessageListenerContainer* transactionnel**
- Transactions locales avec KafkaTemplate, transaction indépendante Kafka
- Synchronisation avec d'autres gestionnaires de transaction

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.

- Lorsque l'utilisateur appelle `close()` sur un producteur, il est renvoyé dans le cache pour être réutilisé au lieu d'être réellement fermé.

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 envoyé 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

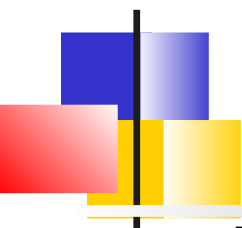
Propriétés de configuration :

– *JsonSerializer*

- **ADD_TYPE_INFO_HEADERS** (défaut true) : Ajouté 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
- **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** : Méthode pour déterminer le type



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 au 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
```




Configuration programmatische

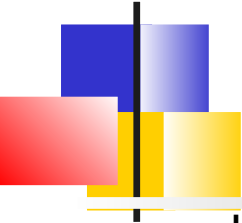
Programmatiquement, on peut faire plus de choses qu'avec les propriétés de configuration :

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));
```



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())));  
}
```

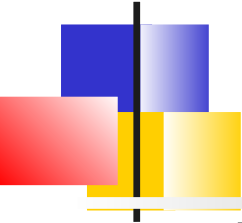
Par regexp sur le nom du topic

```
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());
```



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

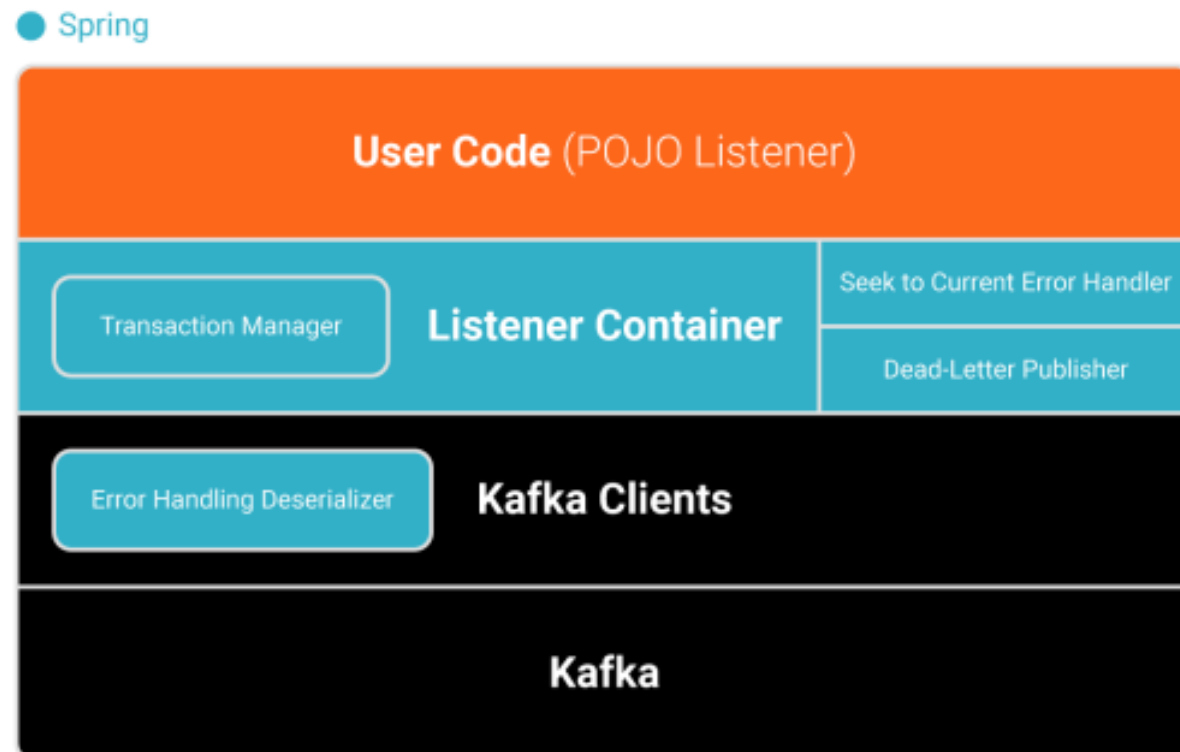
Par défaut, le message fautif est sauté (skip) et la consommation continue

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);
    }
}
```



Spring-Kafka

Introduction
Production de message
Consommation
Transaction
Sérialisation / Désérialisation
Traitement des exceptions
Schema Registry et Avro



Introduction

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

Afin que les consommateurs supportent ces évolutions sans être obligés d'être mis à jour, une simple sérialisation JSON ne suffit pas.

Des bibliothèques de sérialisation comme **Avro** ou **ProtoBuf** adressent ce problème.

Elles offrent également un format de sérialisation binaire plus compact



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.

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



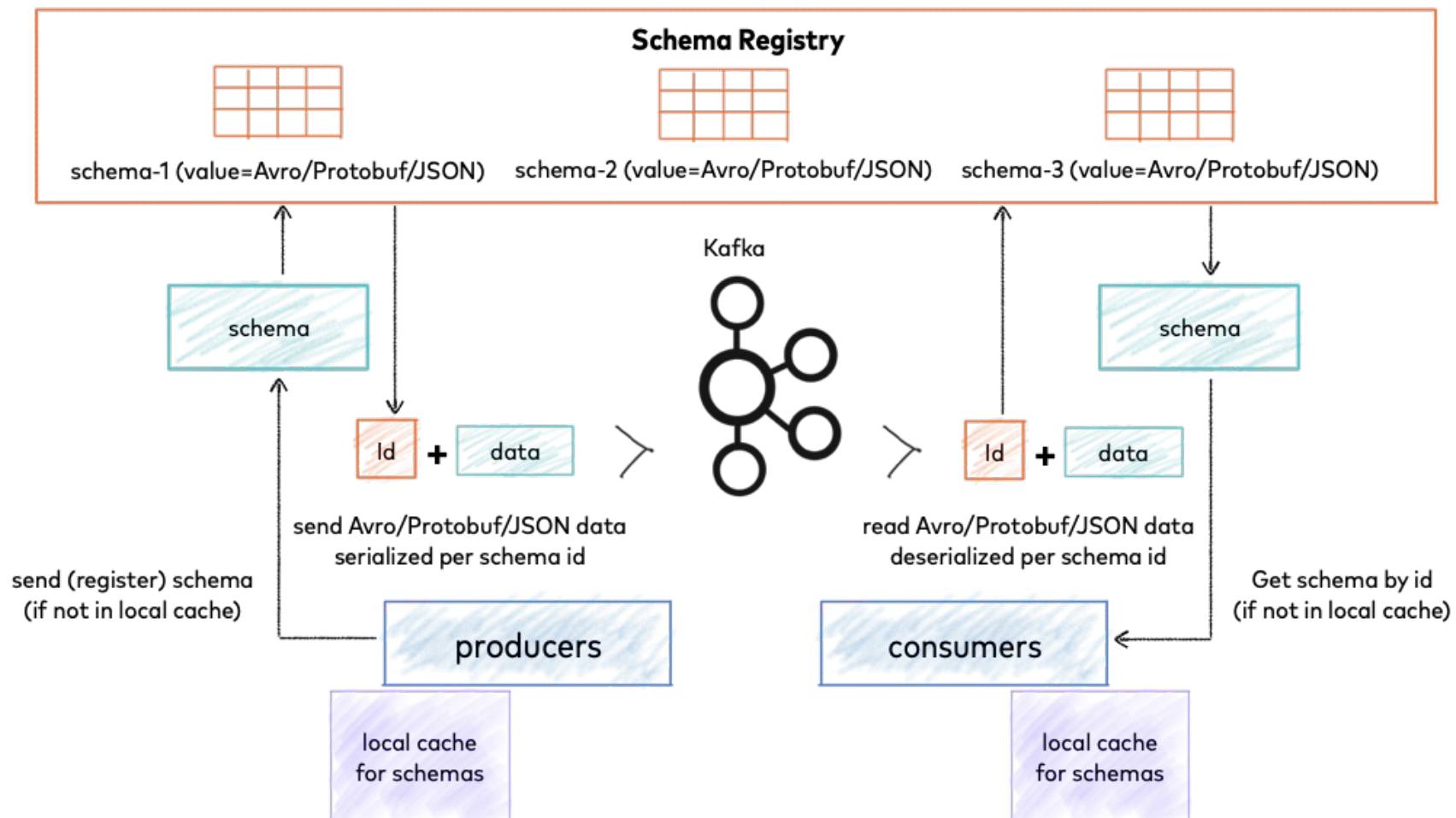
Utilisation du schéma

Avro suppose que le schéma est présent lors de la lecture et l'écriture des fichiers.

- La disponibilité du schéma offre un énorme plus pour l'évolutivité du producteur.
=> Le format du message peut évoluer sans impacter le code des consommateurs

Confluent Platform intègre le composant **Schema Registry** qui offre une API Rest permettant d'accéder aux schémas stockés par les producteurs aux formats Avro, Protobuf et JSON

Schema Registry





Dépendances Confluent

```
<dependencies>
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-schema-registry</artifactId>
    <version>7.2.1</version>
  </dependency>
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>7.2.1</version>
  </dependency>
</dependencies>
<repositories>
  <repository>
    <id>confluent</id>
    <url>https://packages.confluent.io/maven/</url>
  </repository>
</repositories>
```



Schéma Avro

```
{
  "type": "record",
  "name": "Courier",
  "namespace": "org.formation.model",
  "fields": [
    {
      "name": "id",
      "type": "int" },
    {
      "name": "firstName",
      "type": "string" },
    {
      "name": "lastName",
      "type": "string" },
    {
      "name": "position",
      "type": [
        {
          "type": "record",
          "name": "Position",
          "namespace": "org.formation.model",
          "fields": [
            {
              "name": "latitude",
              "type": "double" },
            {
              "name": "longitude",
              "type": "double" }
            ]
          }
        ]
      ]
    }
  ]
}
```



Plug-in Avro

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <version>1.8.2</version>
  <executions>
    <execution>
      <id>schemas</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>schema</goal>
        <goal>protocol</goal>
        <goal>idl-protocol</goal>
      </goals>
      <configuration>
        <sourceDirectory>./src/main/resources/</sourceDirectory>
        <outputDirectory>./src/main/java/</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

=> *mvn compile* génère les classes du modèle



Producteur (1)

Le producteur de message doit contenir du code permettant d'enregistrer le schéma en utilisant la librairie cliente 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 (2)

Les propriétés du *KafkaProducer* 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());  
}
```



Monitoring

Si Micrometer est dans le classpath et qu'un bean MeterRegistry est présent, le ListenerContainer Spring crée et met à jour 2 Timer Micrometer par thread :

- Un pour les succès
- Un pour les échecs



Administration

Gestion des topics

Stockage et rétention des partitions

Gestion du cluster

Sécurité

Dimensionnement

Surveillance



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***

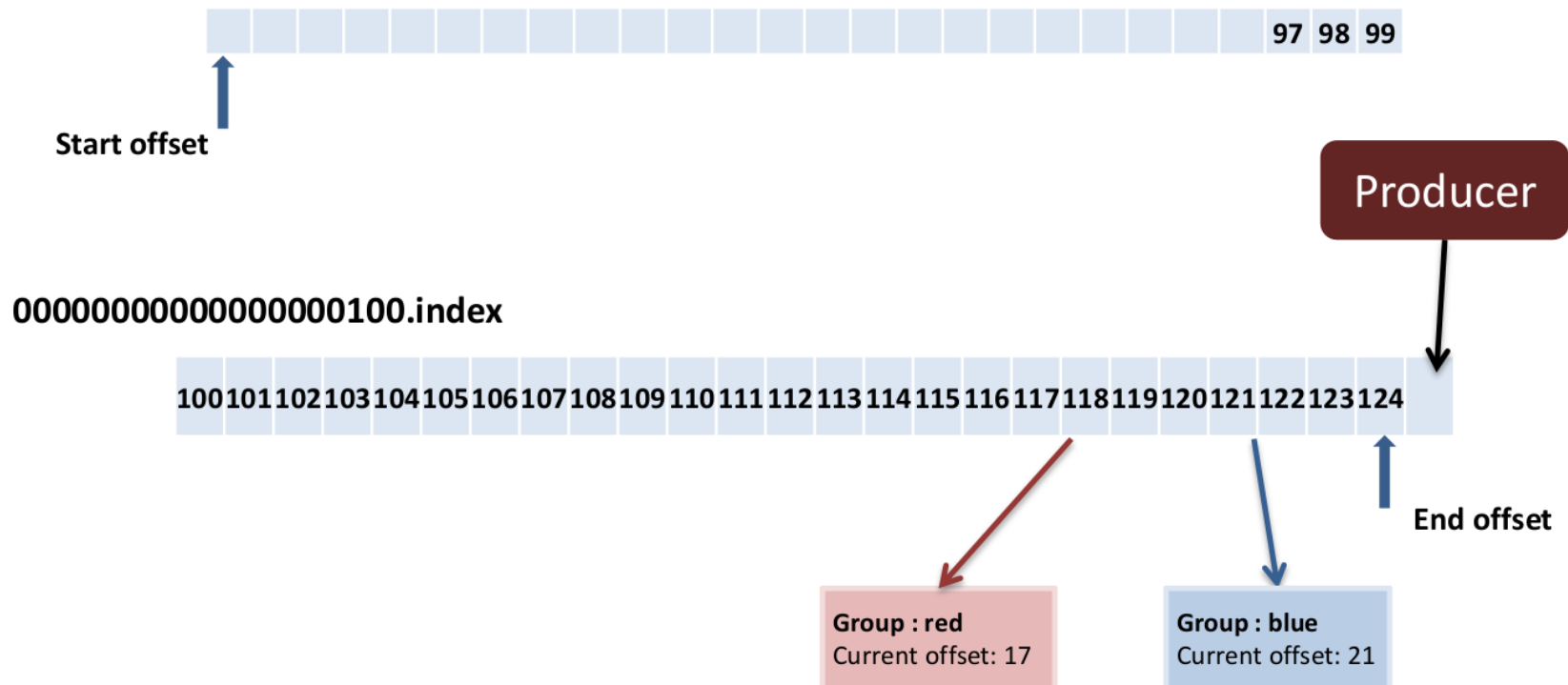
- Les segments sont des fragments de partition au maximum de 1Go et au maximum d'une semaine de données
- Lors de l'écriture d'un segment, lorsque la limite est atteinte, le broker ferme le fichier. Il devient éligible pour la purge
- Il existe donc un seul *segment actif* qui d'ailleurs ne peut pas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



Compactage

Kafka autorise également la stratégie de rétention ***compact***, qui ne stocke que la valeur la plus récente pour chaque clé du *topic*.

- Propriétés ***cleanup.policy=compact*** et ***log.cleaner.enabled=true***
- Les événements doivent alors contenir une clé
- Le compactage est effectué par une thread séparé qui périodiquement purge les messages *dirty*



Nettoyage des logs

Propriété `log.cleanup.policy`, 2 stratégies disponible :

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

Possibilité de combiner les 2 stratégies (delete and compact)



Stratégie *delete*

La stratégie *delete* s'appuie sur :

- ***log.retention.bytes*** (défaut -1 : infinite)
- ***log.retention.ms*** (défaut null)
- ***log.retention.minutes*** (défaut null)
- ***log.retention.hours*** (défaut 168, 1 semaine)

=> Meilleur contrôle de l'usage disque



Stratégie compact

2 propriétés de configuration importante 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

Le nettoyeur (*log cleaner*) est implémenté par un pool de threads.

Le pool est configurable :

- *log.cleaner.enable* : doit être activé si stratégie compact
- *Log.cleaner.threads* : Le nombre de threads
- *log.cleaner.backoff.ms* : Le temps de pause lorsqu'il n'y pas de travail (défaut 15 secondes)
-

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



Administration

Gestion des topics
Stockage et rétention des partitions
Gestion du cluster
Sécurité
Dimensionnement
Surveillance



Redémarrage du cluster

Redémarrage progressif, broker par broker

Attendre que l'état se stabilise
(isr=replicas)

Vérification de l'état des topics

```
bin/kafka-topics.sh --zookeeper zk_host:port --  
describe --topic my_topic_name
```

Exemple d'outillage :

<https://github.com/deviceinsight/kafkactl>



Mise à jour du cluster

Avant la mise à niveau:

- Pour toutes les partitions:
liste de répliques = liste ISR

Garanties:

- Pas de temps d'arrêt pour les clients (producteurs et consommateurs)
- Les nouveaux brokers sont compatibles avec les anciens clients Kafka



Étapes de mise à jour

server.properties: définissez la configuration suivante (redémarrage progressif)

- *inter.broker.protocol.version* : version actuelle
- *log.message.format.version* : version actuelle du format de message

Mettre à niveau les brokers un par un (redémarrage)

- Attendre l'état stable (ISR = réplicas)

Mettre à jour en dernier le contrôleur

Mettre à jour *inter.broker.protocol.version* (redémarrage progressif)

Mettre à niveau les clients

Mettre à jour *log.message.format.version* (redémarrage progressif)



Administration

Gestion des topics
Stockage et rétention des partitions
Gestion du cluster
Sécurité
Dimensionnement
Surveillance



Introduction

Différentes alternatives supportées pour la sécurité :

- Authentification des connections clients vers les brokers via SSL ou SASL
- Authentification des connections des brokers vers *Zookeeper*
- Cryptage des données transférées avec les clients via SSL
- Autorisation des opérations read/write/create/delete/... par client
- Possibilité d'intégration avec d'autres mécanismes d'authentification et d'autorisation

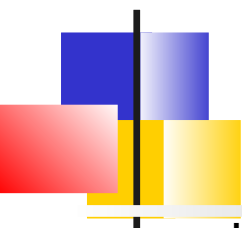
Naturellement, dégradation des performances avec SSL



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.



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***

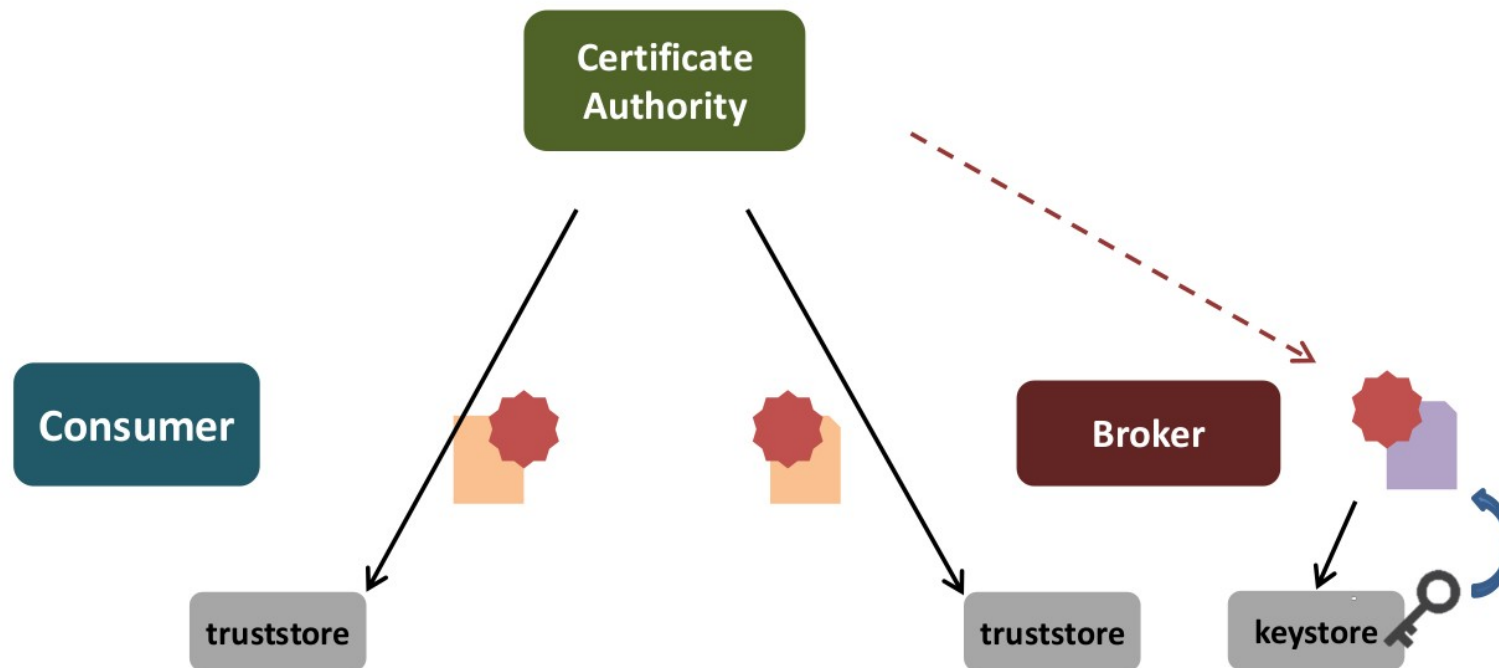
Exemple : `listener.security.protocol.map=CLIENT:SSL,BROKER:PLAINTEXT`

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***

SSL

SSL pour le cryptage et l'authentification





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
```



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 JAAS

La configuration JAAS s'effectue :

- Soit via un **fichier jaas** contenant :
 - Une section *KafkaServer* pour l'authentification auprès d'un broker
 - Une section *Client* pour s'authentifier auprès de Zookeeper

L'exemple suivant définit 2 utilisateurs admin et alice qui pourront accéder au broker et l'identité avec laquelle le broker initiera les requêtes inter-broker

```
KafkaServer {  
    org.apache.kafka.common.security.plain.PlainLoginModule required  
    username="admin"  
    password="admin-secret"  
    user_admin="admin-secret"  
    user_alice="alice-secret";  
};
```

–



Configuration JAAS

La configuration JAAS peut également se faire via la propriété ***sasl.jaas.config*** préfixé par le mécanisme SASL

```
listener.name.sasl_ssl.scram-sha-  
256.sasl.jaas.config=org.apache.kafka.common.security.scram  
.ScramLoginModule required \  
  username="admin" \  
  password="admin-secret";
```



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

SASL/PLAIN est un mécanisme simple d'authentification par login/mot de passe. Il 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 à la production



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



Autorisation

Kafka fournit une implémentation permettant de définir des ACL

Pour l'activer en mode Kraft :

```
authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer
```

Les ACLs sont stockés dans les méta-données gérées par les contrôleurs

Les règles peuvent être exprimées comme suit :

Principal P is [Allowed/Denied] Operation O From Host H on any Resource R matching ResourcePattern RP

L'utilitaire ***kafka-acls.sh*** permet de gérer les ACLs



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
```



Administration

Gestion des topics
Stockage et rétention des partitions
Gestion du cluster
Sécurité
Dimensionnement
Surveillance



ZooKeeper

CPU : Typiquement pas un goulot d'étranglement

- 2 - 4 CPU

Disque : Sensible à la latence I/O, Utilisation d'un disque dédié. De préférence SSD

- Au moins 64 Gb

Mémoire : Pas d'utilisation intensive

- Dépend de l'état du cluster
- 4 Gb - 16 Gb (Pour les très grand cluster : plus de 2000 partitions)

JVM : Pas d'utilisation intensive de la heap

- Au moins 1 Gb pour le cache de page
- 1 Gb - 4 Gb

Réseau : La bande passante ne doit pas être partagée avec d'autres applications



Kafka Broker

CPU : Pas d'utilisation intensive du CPU

Disque :

- RAID 10 est mieux
- SSD n'est pas plus efficace•

Mémoire : Pas d'utilisation intensive (sauf pour le compactage)

- 16 Gb - 64 Gb

JVM :

- 4 Gb - 6 Gb

Réseau:

- 1 Gb - 10 Gb Ethernet (pour les gros cluster)
- La bande passante ne doit pas être partagée avec d'autres applications



Configuration système

Étendre la limite du nombre de fichiers ouverts :

- *ulimit -n 100000*

Configuration Virtual Memory
(*/etc/sysctl.conf pour ubuntu*)

- *vm.swappiness=1*
- *vm.dirty_background_ratio=5*
- *vm.dirty_ratio=60*



JVM

KAFKA_HEAP_OPTS peut être utilisée

Heap size :

– -Xms4G -Xmx4G or -Xms6G -Xmx6G

Options JVM

-XX:MetaspaceSize=96m -XX:+UseG1GC -
XX:MaxGCPauseMillis=20

-XX:InitiatingHeapOccupancyPercent=35 -
XX:G1HeapRegionSize=16M

-XX:MinMetaspaceFreeRatio=50 -
XX:MaxMetaspaceFreeRatio=80



Dimensionnement cluster

Généralement basée sur les capacités de stockage :

$\text{Nb de msg-jour} * \text{Taille moyenne} * \text{Rétention} * \text{Replication} / \text{stockage par Broker}$

Les ressources doivent être surveillées et le cluster doit être étendu plus de 70% est atteint pour :

- CPU
- L'espace de stockage
- La bande passante



Zookeeper

Un nombre impair de serveurs
Zookeeper

- Nécessité d'un Quorum (vote majoritaire)
- 3 nœuds permet une panne
- 5 nœuds permet 2 pannes



Partitionnement des topics

Augmenter le nombre de partitions
permet plus de consommateurs

Mais augmente généralement la taille du
cluster

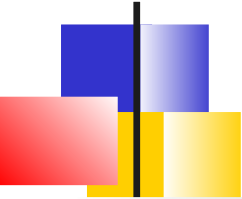
Évaluer l'utilisation d'un autre topic ?

Généralement 24 est suffisant



Administration

Gestion des topics
Stockage et rétention des partitions
Gestion du cluster
Sécurité
Dimensionnement
Surveillance



Principaux métriques brokers accessibles via JMX

Métrique	Description	Alerte
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions	Nombre de partition sous-répliquée	Si > 0
kafka.controller:type=KafkaController,name=OfflinePartitionsCount	Nombre de partitions qui n'ont pas de leader actif	Si > 0
kafka.controller:type=KafkaController,name=ActiveControllerCount	Nombre de contrôleur actif dans le cluster	Si != 1
kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce FetchConsumer FetchFollower}	Nombre de requêtes par seconde, pour produire et récupérer	Si changement significatif
kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica	Retard maximal des messages entre les répliques et le leader	
kafka.server:type=ReplicaManager,name=IsrShrinksPerSec	Cadence de shrink des ISR	
kafka.server:type=ReplicaManager,name=IsrExpandsPerSec	Cadence d'expansion des ISR	



Métriques clients

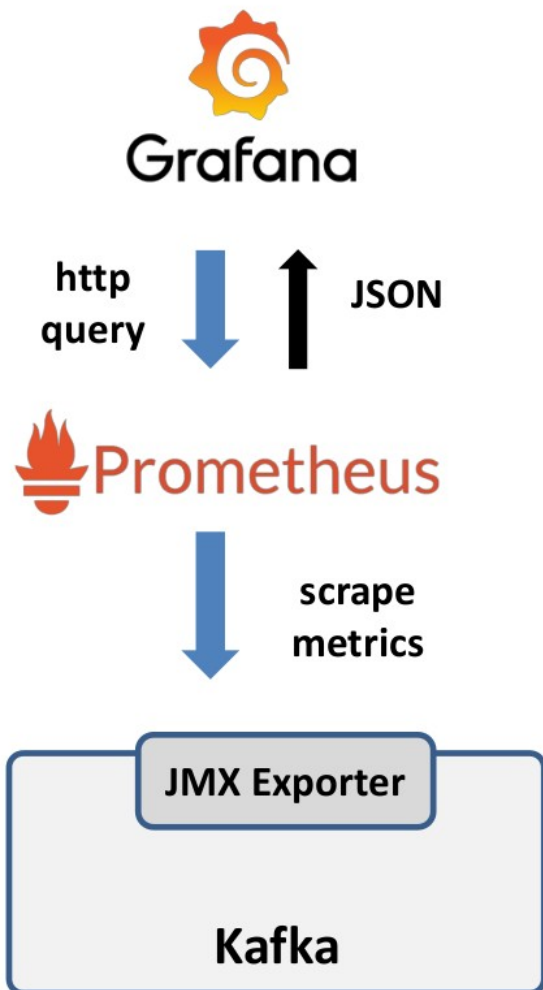
Producteur

Métrique	Description
kafka.producer:type=producermetrics,client-id=([-w]+),name=io-ratio	Fraction de temps de la thread passé dans les I/O
kafka.producer:type=producer-metrics,client-id=([-w]+),name=io-wait-ratio	Fraction de temps de la thread passé en attente

Consommateur

Métrique	Description
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-w]+),records-lag-max	Le décalage maximum en termes de nombre d'enregistrements pour n'importe quelle partition

Outil de visualisation



Tdb dispo :

<https://grafana.com/grafana/dashboards/721>

Exportateur JMX

https://github.com/prometheus/jmx_exporter/blob/master/example_configs/kafka-2_0_0.yml



Autres outils

Confluent Control Center

JConsole

Graphite

CloudWatch

DataDog

SMM of Data Flow (Hortonworks)

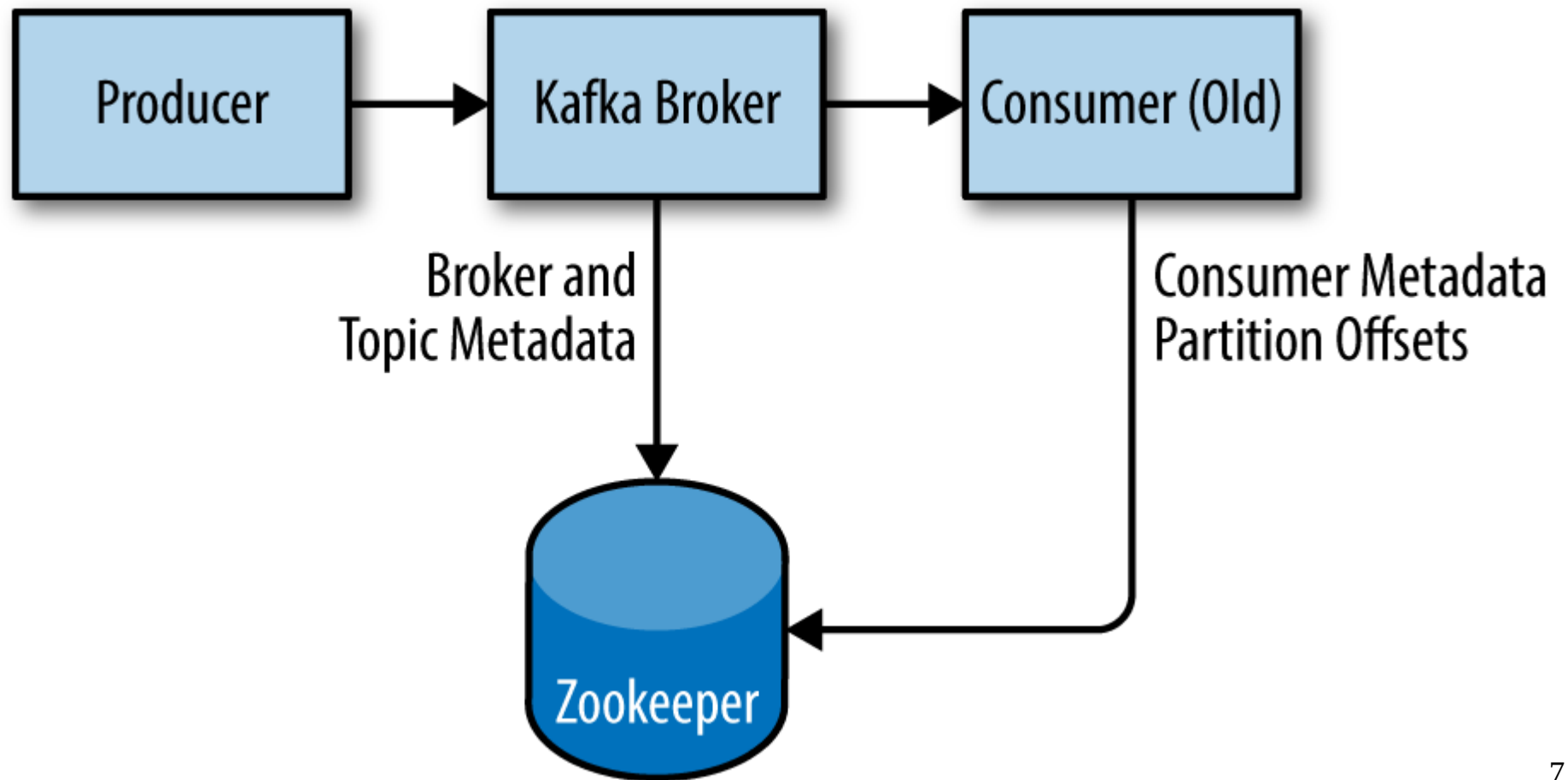
...



Annexes

Apache Zookeeper Contrôleur et Zookeeper SASL SCRAM

Kafka et Zookeeper



Zookeeper

Principes

“High-performance coordination service for distributed applications”

Utilisé par Kafka pour la gestion de configuration et la synchronisation

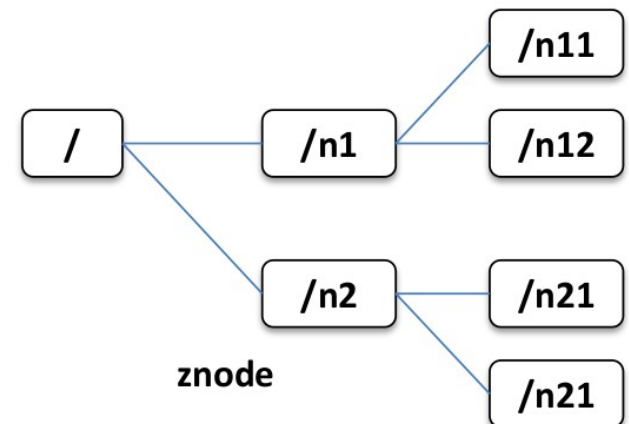
Stocke les méta-données du cluster Kafka

Répliqué sur plusieurs hôtes formant un *ensemble*

Fournit un espace de noms hiérarchiques

Exemples de nœuds pour Kafka :

- */controller*
- */brokers/topics/*
- */config*





Vocabulaire Zookeeper

Nœud (zNode) : Donnée identifiée par un chemin

Client : Utilisateur du service Zookeeper

Session : Établie entre le client et le service Zookeeper

Noeud éphémère : le nœud existe aussi longtemps que la session qui l'a créé est active

Watch :

- Déclenché et supprimé lorsque le nœud change
- Clients peuvent positionné un watch sur un nœud *znode*



Zookeeper Distribution

Kafka contient des scripts permettant de démarrer une instance de Zookeeper mais il est préférable d'installer une version complète à partir de la distribution officielle de *Zookeeper*

<https://zookeeper.apache.org/>



Exemple installation standalone

```
# tar -zxf zookeeper-3.4.6.tar.gz
# mv zookeeper-3.4.6 /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cat > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```



Vérification Zookeeper

```
# telnet localhost 2181
```

```
Trying ::1...
```

```
Connected to localhost.
```

```
Escape character is '^['.
```

```
srvr
```

```
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
```

```
Latency min/avg/max: 0/0/0
```

```
Received: 1
```

```
Sent: 0
```

```
Connections: 1
```

```
Outstanding: 0
```

```
Zxid: 0x0
```

```
Mode: standalone
```

```
Node count: 4
```

```
Connection closed by foreign host.
```

```
#
```




Ensemble Zookeeper

Un cluster Zookeeper est appelé un **ensemble**

Une instance est élue comme ***leader***

L'ensemble contient un nombre impair d'instances
(algorithme de consensus basé sur la notion de quorum).

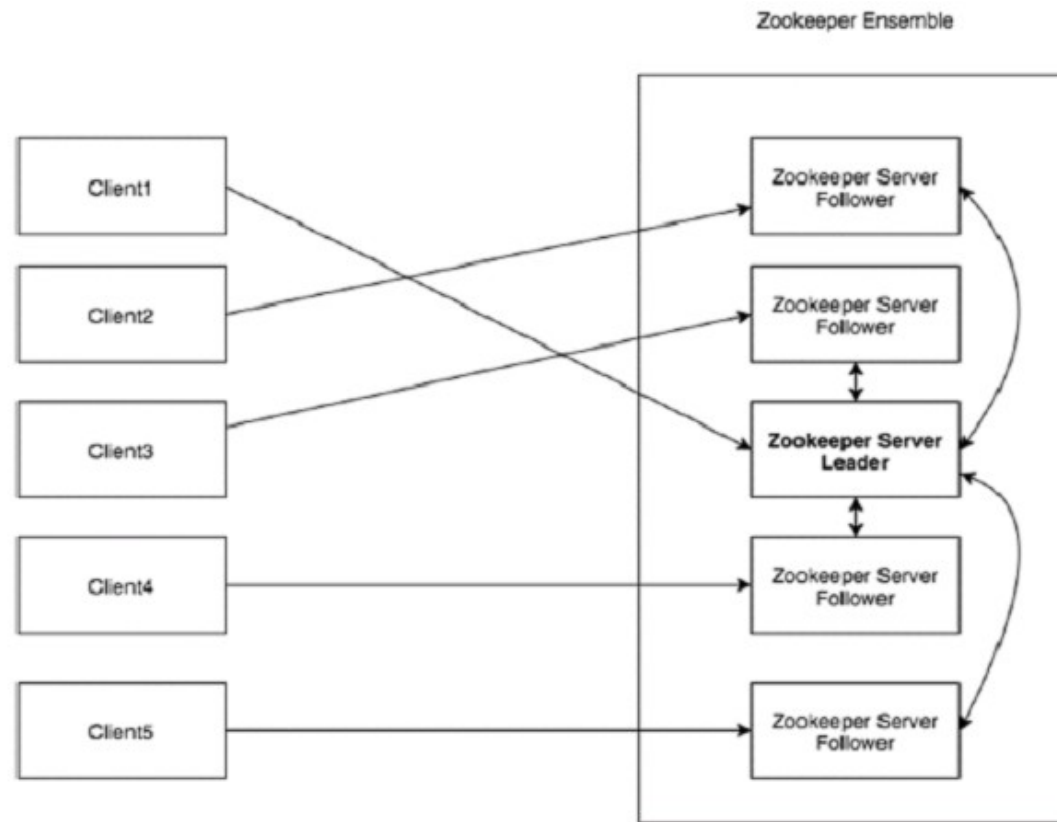
– Le nombre dépend du niveau de tolérance aux pannes
voulu :

- 3 nœuds => 1 défaillance
- 5 nœuds => 2 défaillances

Les clients peuvent s'adresser à n'importe quelle instance
pour lire/écrire les données

Lors d'une écriture, le *leader* coordonne les écritures sur
les *followers*

Architecture *ZooKeeper*





Propriétés de config

tickTime : L'unité de temps en ms

initLimit : nombre de tick autorisé pour la connexion des suiveurs au leader

syncLimit : nombre de tick autorisé pour la synchronisation suiveur/leader

dataDir : Répertoire de stockage des données

clientPort : Port utilisé par les clients

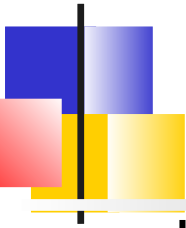
La configuration répertorie également chaque sous la forme :

server.X = nom d'hôte: peerPort: leaderPort

- ***X*** : numéro d'identification du serveur.
- ***nom d'hôte*** : IP
- ***peerPort*** : Port TCP pour communication entre serveurs
- ***leaderPort*** : Port TCP pour l'élection.

Optionnel :

4lw.commands.whitelist : Les commandes d'administration autorisées



Configuration d'un ensemble

Les serveurs doivent partager une configuration commune listant les serveurs et chaque serveur doit contenir un fichier *myid* dans son répertoire de données contenant son identifiant

Exemple de config :

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=zoo1.example.com:2888:3888
server.2=zoo2.example.com:2888:3888
server.3=zoo3.example.com:2888:3888
```



Vérifications Ensemble Zookeeper

Se connecter à une instance :

```
./zkCli.sh -server 127.0.0.1:2181
```

Voir le mode (leader ou suiveur) d'une instance

si la commande *stat* est autorisée

```
echo stat | nc localhost 2181 | grep Mode
```



Quelques commandes

ls [path] : Lister un zNode

create [zNode] : Créer un nœud

delete/deleteall : Suppression (récursive) d'un nœud

get/set [zNode] : Lire/écrire la valeur du nœud

history : Historique des commandes

quit : Quitter zkCli



Annexes

Apache Zookeeper
Contrôleur et Zookeeper
SASL SCRAM



Rôles du contrôleur

Un des brokers Kafka (nœud éphémère dans *Zookeeper*)

Pour le visualiser :

```
./bin/zookeeper-shell.sh [ZK_IP] get /controller
```

- Gère le cluster en plus des fonctionnalités habituelles d'un broker
- Détecte le départ / l'arrivée de broker via *Zookeeper* (*/brokers ids*)
- Gère les changements de Leaders

Si le contrôleur échoue:

- un autre broker est désigné comme nouveau contrôleur
- les états des partitions (liste des leaders et des ISR) sont récupérés à partir de *Zookeeper*



Responsabilités

Lors d'un départ de broker, pour toutes les partitions dont il est le leader, le contrôleur :

- Choisit un nouveau leader et met à jour l'ISR
- Met à jour le nouveau Leader et l'ISR (État des partitions) dans *Zookeeper*
- Envoie le nouveau Leader/ISR à tous les brokers contenant le nouveau leader ou les followers existants

Lors de l'arrivée d'un broker, le contrôleur lui envoie le Leader et l'ISR



Annexes

Apache Zookeeper
Contrôleur et Zookeeper
SASL SCRAM



SASL SCARM

Utilisé avec SSL pour une
authentification sécurisée

- *Zookeeper* est utilisé pour stocker les
crédentiels
- Sécurisé via l'utilisation d'un réseau
privé



Configuration des créden*t*iels SASL SCRAM

Communication Inter-Broker : user “admin”

```
kafka-configs.sh --zookeeper host:port --alter \  
--add-config 'SCRAM-SHA-256=[password=adminpass],SCRAM-  
SHA-512=[password=adminpass]' \  
--entity-type users --entity-name admin \  

```

Communication Client-Broker : user “user”

```
kafka-configs.sh --zookeeper host:port --alter \  
--add-config 'SCRAM-SHA-256=[password=userpass],SCRAM-  
SHA-512=[password=userpass]' \  
--entity-type users --entity-name user\  

```



Configuration du broker

Créer le fichier Jaas

```
KafkaServer {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="admin"  
    password="adminpass"  
} ;
```

Ajouter un paramètre JVM

```
-Djava.security.auth.login.config=/home/ubuntu/ssl/kafka_jaas.conf
```

server.properties

```
listeners=SASL_SSL://host.name:port  
security.inter.broker.protocol=SASL_SSL  
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512  
sasl.enabled.mechanisms=SCRAM-SHA-512
```



Configuration du client

Créer le fichier Jaas

```
KafkaClient {  
    org.apache.kafka.common.security.scram.ScramLoginModule  
    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=SCRAM-SHA-512
```