07/10/2023
Version 2

SPRING CLOUD

JÉRÉMY PERROUAULT

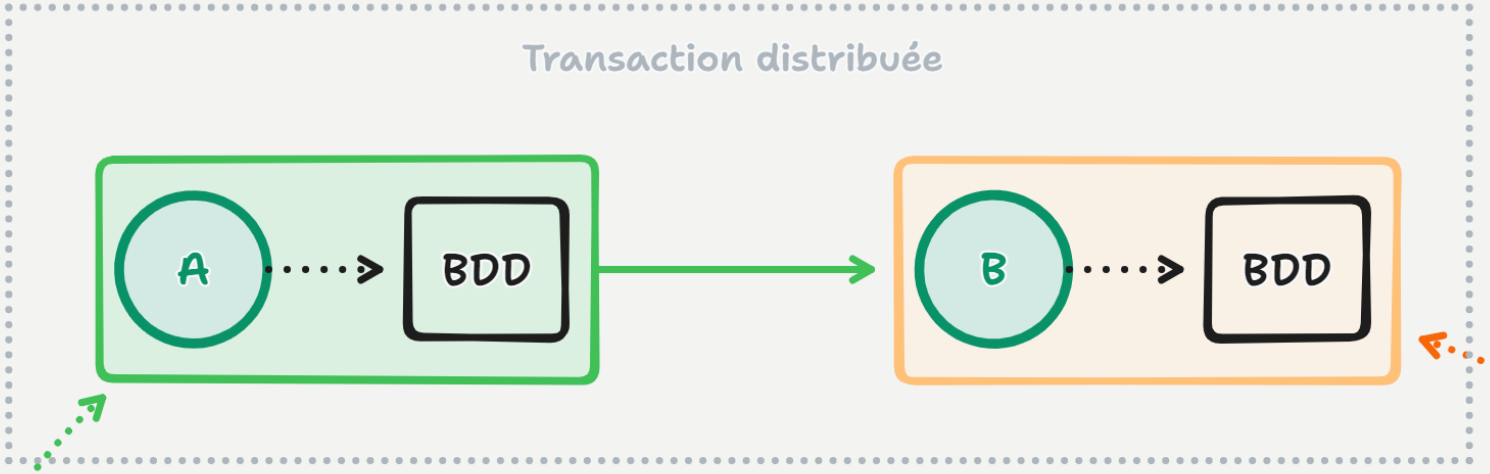
SAGA

GESTION DES TRANSACTIONS

SAGA

- Dans l'architecture implémentée, il y a un problème majeur
  - La (non) gestion d'une transaction n'est pas compensée

SAGA



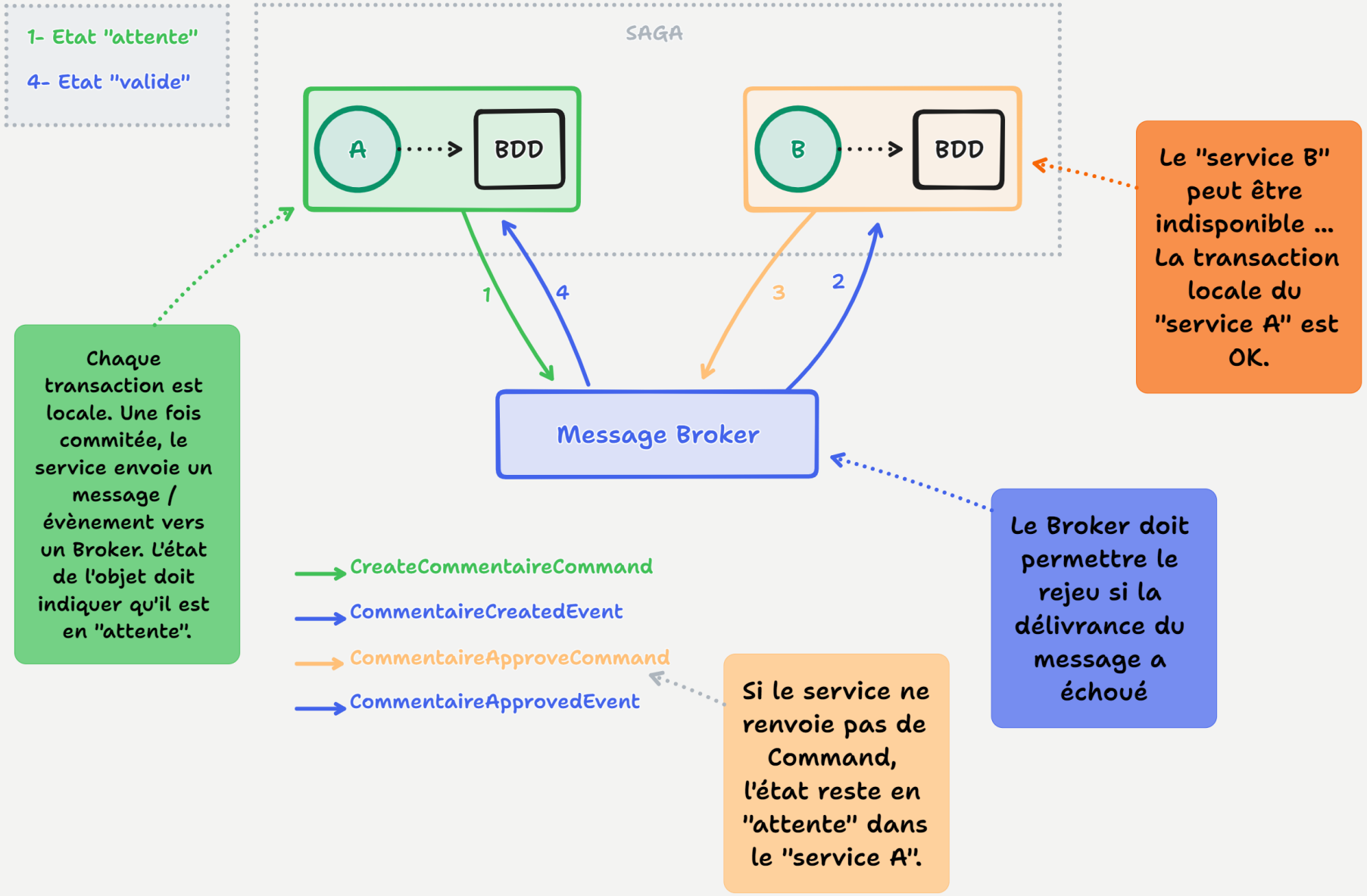
Une transaction démarre. Elle se commit lorsqu'on a reçu une réponse de "service B"

Le "service B" doit être disponible pour valider la transaction

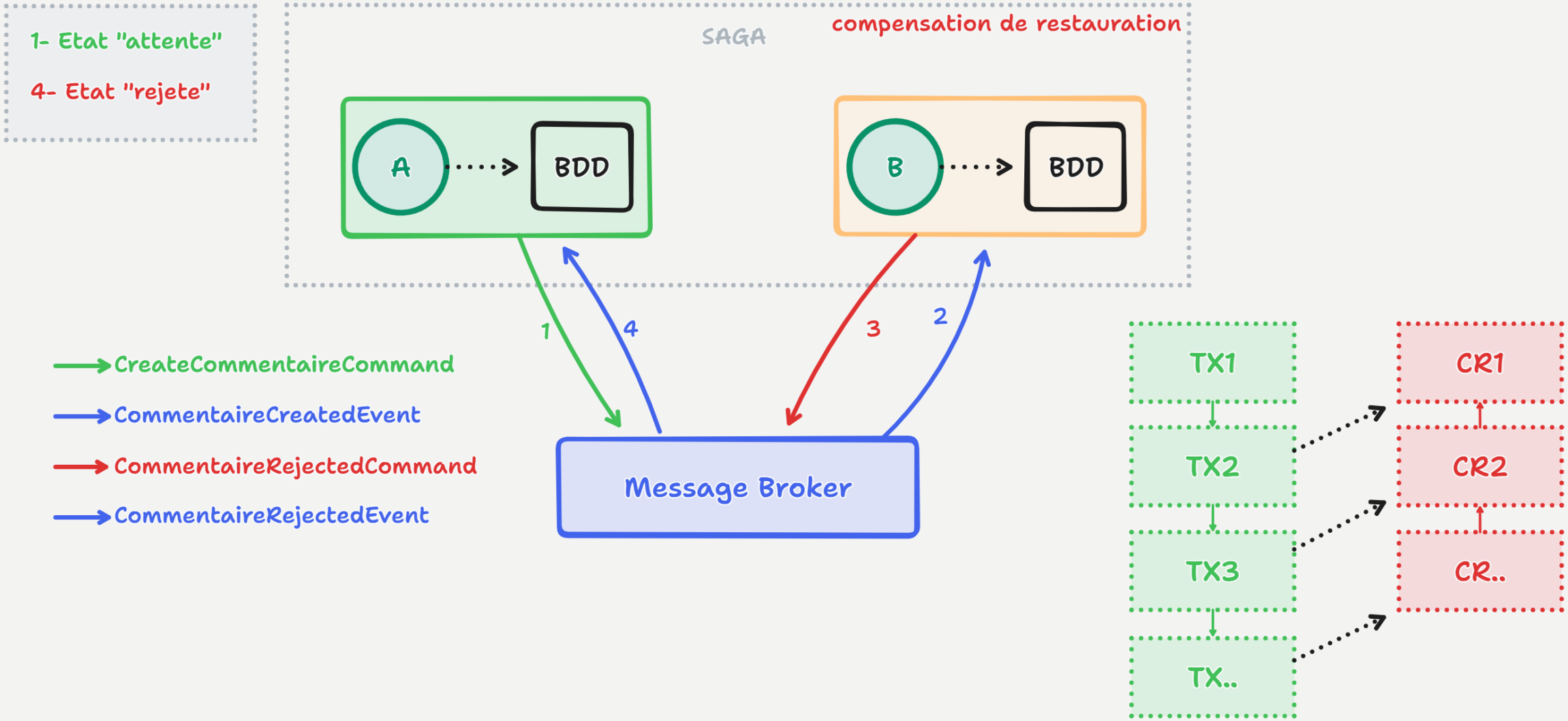
SAGA

- Utilisation du Pattern **Saga**
  - L'idée, c'est d'avoir une séquence de transactions locales
    - Chaque transaction met à jour les changements locaux
    - Puis publie un message ou un évènement pour que les autres services puissent se mettre à jour
    - Et en cas d'échec lors de la transaction locale, un autre message / évènement est publié
      - Chaque service impliqué pourra « revenir en arrière » en déclenchant son mécanisme de compensation de transaction

SAGA


1- Etat "attente"
4- Etat "valide"
SAGA
A
BDD
B
BDD
1
4
3
2
Message Broker
Chaque transaction est locale. Une fois commitée, le service envoie un message / évènement vers un Broker. L'état de l'objet doit indiquer qu'il est en "attente".
Le "service B" peut être indisponible ... La transaction locale du "service A" est OK.
Le Broker doit permettre le rejeu si la délivrance du message a échoué
CreateCommentaireCommand
CommentaireCreatedEvent
CommentaireApproveCommand
CommentaireApprovedEvent
Si le service ne renvoie pas de Command, l'état reste en "attente" dans le "service A".

SAGA


1- Etat "attente"
4- Etat "rejete"
SAGA
compensation de restauration
A
BDD
B
BDD
1
4
3
2
Message Broker
CreateCommentaireCommand
CommentaireCreatedEvent
CommentaireRejectedCommand
CommentaireRejectedEvent
TX1
TX2
TX3
TX..
CR1
CR2
CR..

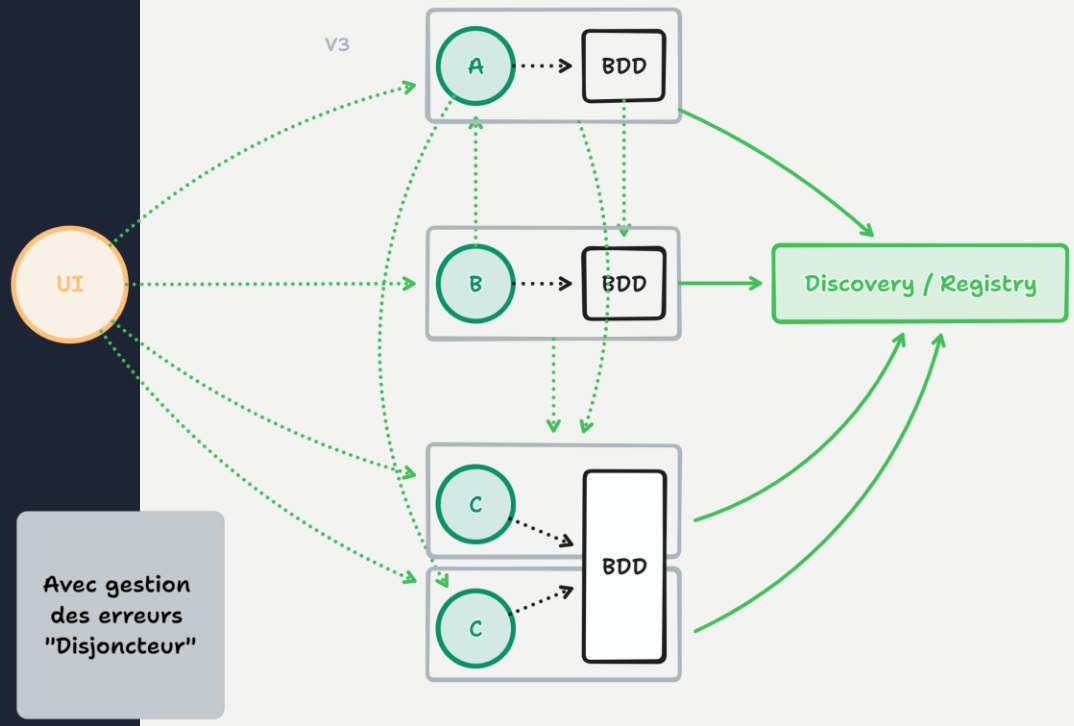
SAGA

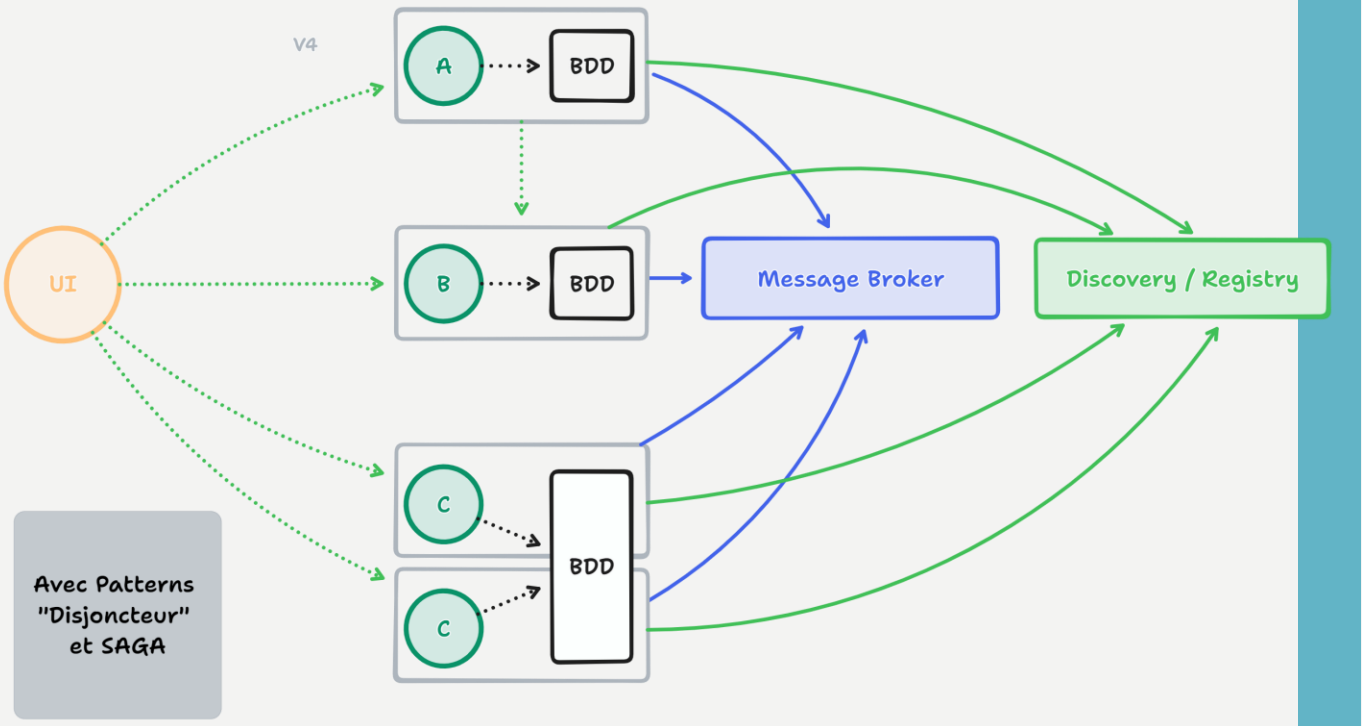
- Utilisation d'une technologie **Broker** existante …
  - **RabbitMQ**
  - **Kafka**
  - **Amazon SQS**
  - **Axon**
  - **EventStoreDb**
  - **Redis Stream**
  - …

SAGA

- Attention, selon ce qu'on veut faire, on utilisera différentes implémentations
  - Messaging infrastructure (**RabbitMQ**)
  - Event Streaming / Storing / Processing (**Kafka**)

SAGA


V3
A
BDD
B
BDD
C
C
BDD
UI
Discovery / Registry
Avec gestion des erreurs "Disjoncteur"


V4
A
BDD
B
BDD
C
C
BDD
UI
Message Broker
Discovery / Registry
Avec Patterns "Disjoncteur" et SAGA

SAGA

- Peu importe le **Message Broker** utilisé, il y aura
  - Un Producer Qui émet un message
  - Un Consumer Qui reçoit un message

SAGA

- Mettre en place **Kafka**
- Configurer les 2 services

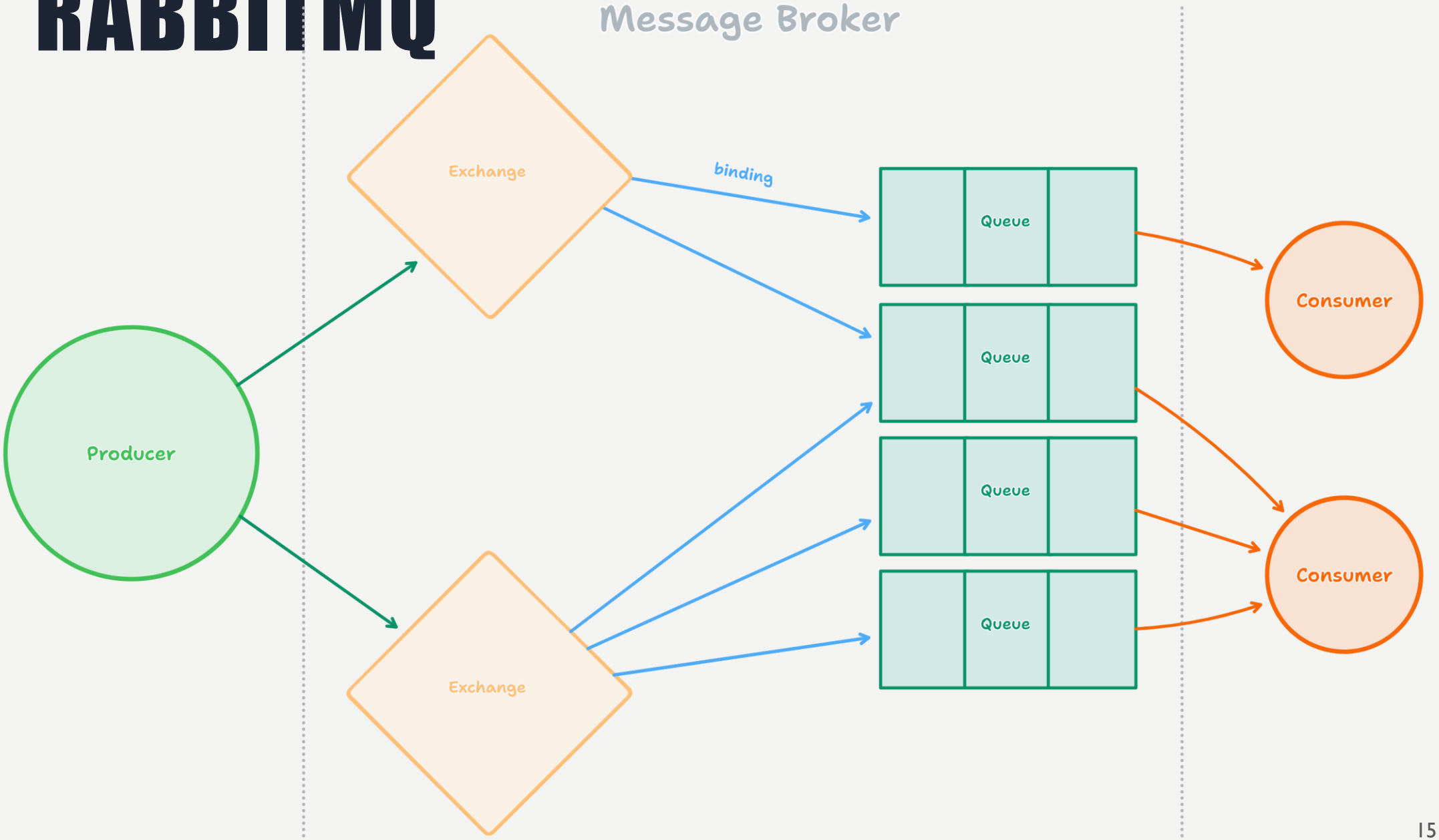
RABBITMQ

BRÈVE INTRODUCTION

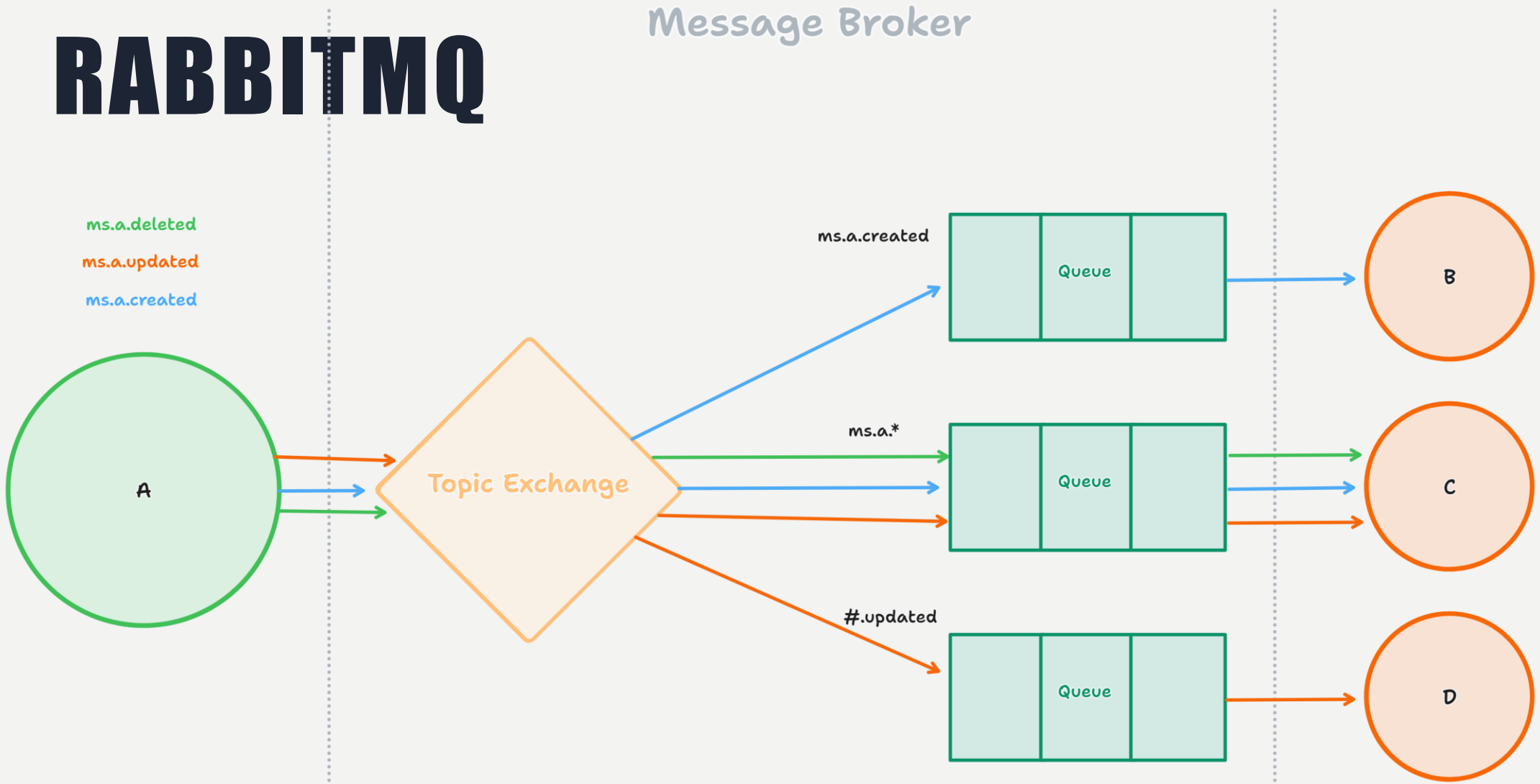
RABBITMQ

- **RabbitMQ** est un **Message Broker**
  - Gère des queues
    - Une queue est liée à un ou plusieurs exchanges
  - Gère des exchanges
    - Un exchange est lié à une ou plusieurs queues
  - Permet le rejeu
  - Permet la persistance d'une queue (si indisponible par exemple)

RABBITMQ


Message Broker
Exchange
binding
Queue
Queue
Queue
Queue
Consumer
Consumer
Producer
Exchange

RABBITMQ


Message Broker
ms.a.deleted
ms.a.updated
ms.a.created
A
Topic Exchange
ms.a.created
ms.a.*
#.updated
Queue
Queue
Queue
B
C
D

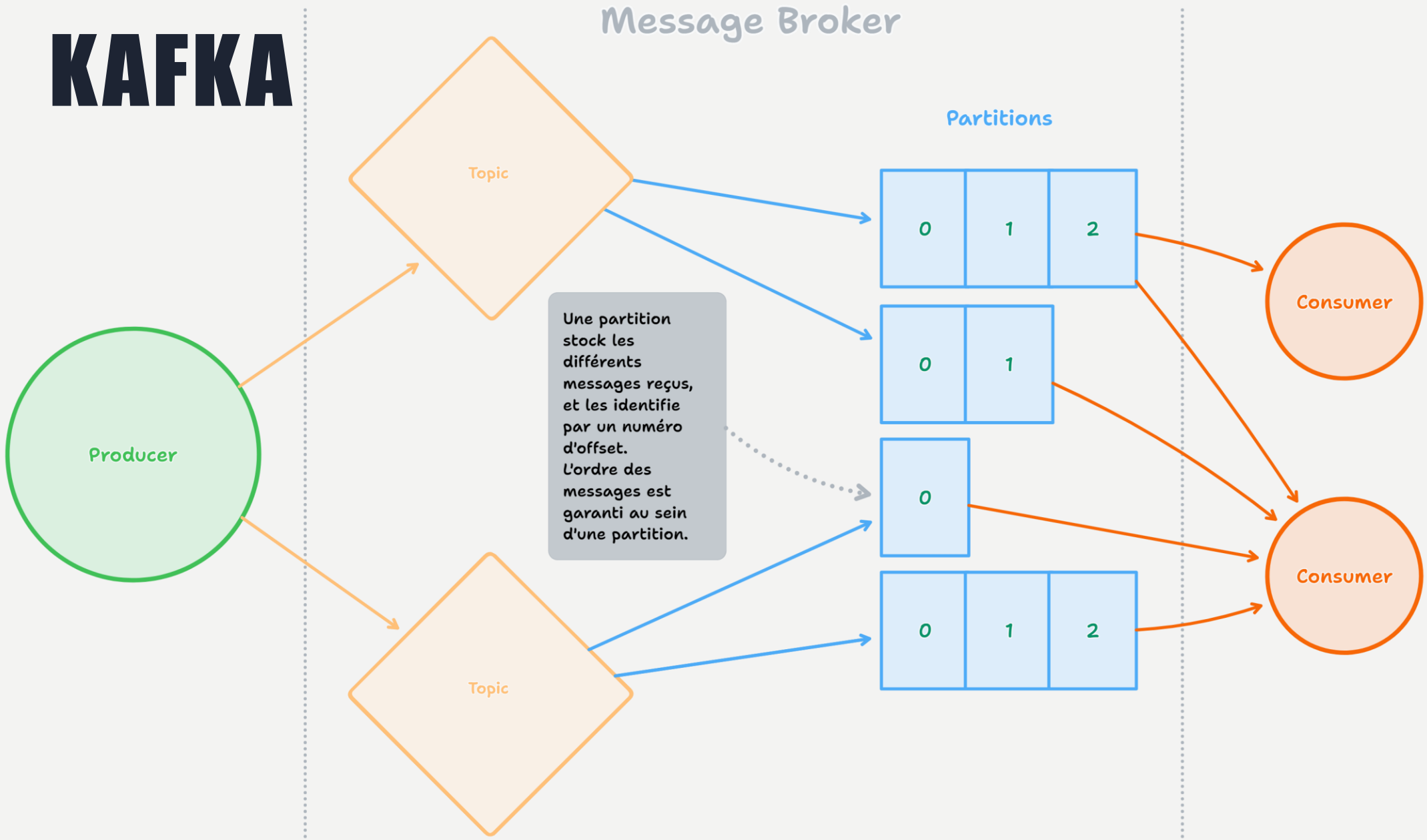
KAFKA

BRÈVE INTRODUCTION

KAFKA

- **Kafka** est un **Message Broker**
  - Gère des topics
    - Un topic possède une ou plusieurs partitions
    - Une partition contient des offsets dont le numéro s'incrémente
    - L'ordre des messages est garanti dans une partition, mais pas entre partitions
  - Les partitions dans un topic permettent se répartir un topic sur plusieurs nœuds d'un cluster **Kafka**
    - Sans clé dans le message, il sera attribué à une partition au hasard
  - Permet le rejeu
  - Permet la persistance des messages (une semaine par défaut)
  - Utilise **ZooKeeper** pour stocker les métadonnées
    - Qui ne sera plus utilisé dans la V4 et sera remplacé par **raft**, le système interne de **Kafka** – qu'on nomme aussi **Kraft**

KAFKA


Message Broker
Partitions
Topic
Topic
Producer
Une partition stock les différents messages reçus, et les identifie par un numéro d'offset.
L'ordre des messages est garanti au sein d'une partition.
0
1
2
0
1
0
0
1
2
Consumer
Consumer

SPRING BOOT

EXEMPLE D'IMPLÉMENTATIONS

SPRING BOOT

- Pour mettre en place **Kafka** avec **SPRING BOOT**
  - Utiliser la dépendance **kafka**, et les interfaces `KafkaTemplate` et `@KafkaListener`
    - Mais dans ce cas, on est lié à **Kafka**, et si on veut changer de **Message Broker**, il faudra adapter le code
  - Utiliser la dépendance **spring-cloud-stream** et choisir un **binder**, qui sera **Kafka** par exemple
    - En utilisant différentes interfaces
      - `StreamBridge` Permettra d'envoyer un message (déclenchement manuel)
      - `Consumer<?>` Permettra de recevoir un message (déclenchement automatique)
      - `Supplier<?>` Permettra d'envoyer un message (déclenchement automatique)
      - `Function<?, ?>` Permettra de recevoir et de renvoyer un message (automatique)

SPRING BOOT

- Ajouter la dépendance *spring-cloud-stream-binder-kafka*

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

- Configurer la connexion dans le fichier *application.properties*

```
spring.cloud.stream.kafka.binder.brokers = localhost:9092
```

SPRING BOOT – PRODUCER

- Injecter `StreamBridge` à l'endroit nécessaire (dans un contrôleur par exemple)
- Utiliser `StreamBridge` pour envoyer un évènement

```
this.streamBridge.send("nom-topic", commentaireCreatedCommand);
```

- Sans autre configuration, le nom du topic sera « nom-topic »

SPRING BOOT – PRODUCER

- Injecter `StreamBridge` à l'endroit nécessaire (dans un contrôleur par exemple)
- Utiliser `StreamBridge` pour envoyer un message

```
this.streamBridge.send("nom-evenement-out-0", commentaireCreatedCommand);
```

- En configurant de la sorte, le nom du topic sera « nom-topic »

```
spring.cloud.stream.bindings.nom-evenement-out-0.destination = nom-topic
```

SPRING BOOT – CONSUMER

- Déclarer un Consumer, ou un Supplier

```
@Bean
public Consumer<CommnentaireCreatedEvent> nomEvenement1() {
  return evt -> {
    System.out.println(evt);
  };
}
```

- Configurer les topics dans le fichier *application.properties*

```
spring.cloud.stream.bindings.nomEvenement1-in-0.destination = nom-topic
spring.cloud.stream.bindings.nomEvenement1-in-0.group = nomService
```

NOTE : Le nom du groupe est utilisé
pour éviter de recevoir plusieurs fois le même message
Si le service est exécuté 2 fois, seule une des 2 instances recevra le message

SPRING BOOT – CONSUMER

- Il est possible d'écouter plusieurs topics

```
spring.cloud.stream.bindings.nomEvenement1-in-0.destination = nom-topic,topic-2,test
```

SPRING BOOT – CONSUMER

- Dans le cas où il existe plusieurs `Consumer` ou `Supplier` ou `Function`
  - Il est important de la déclarer avec cette configuration

```
spring.cloud.function.definition = nomEvenement1;nomEvenement2
```

```
spring.cloud.stream.bindings.nomEvenement1-in-0.destination = nom-topic
spring.cloud.stream.bindings.nomEvenement1-in-0.group = nomService

spring.cloud.stream.bindings.nomEvenement2-in-0.destination = nom-topic-2
spring.cloud.stream.bindings.nomEvenement2-in-0.group = nomService
```

SPRING BOOT – CONSUMER

- Dans le cas où il existe plusieurs `Consumer`
  - L'on peut également configurer un « routing » (`functionRouter` n'existe pas en tant que **bean**)

```
spring.cloud.function.definition = functionRouter
spring.cloud.stream.bindings.functionRouter-in-0.destination = nom-topic,nom-topic-2
spring.cloud.stream.bindings.functionRouter-in-0.group = nomService

spring.cloud.function.routing-expression = 'nom' + headers['type']
```

  - Mais il faudra penser à adapter le **Producer** pour qu'il envoie l'en-tête « type »

```
this.streamBridge.send("nom-topic", MessageBuilder
  .withPayload(commentaireCreatedCommand)
  .setHeader("type", "Evenement1")
  .build()
);
```