

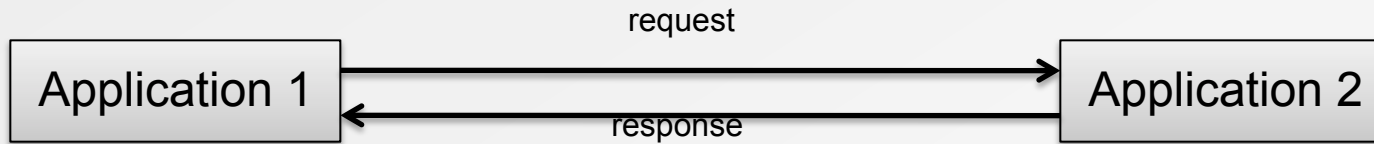
Messages Oriented Middleware

- L'API JMS
- Le standard AMQP
- Messaging AMQP / JMS
- L'approche Message-Driven (ou Event-Driven)
- Transaction, intégrité et fiabilité

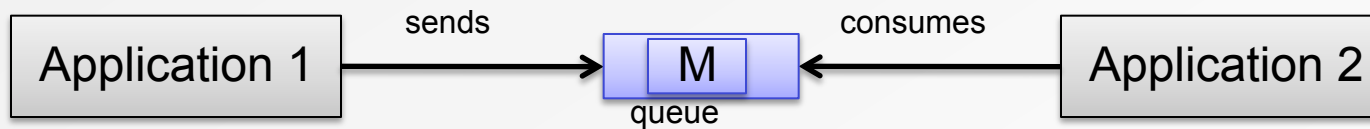
- Les MOM peuvent véhiculer n'importe quel type de messages
 - Texte
 - *XML*
 - *String*
 - Binaire
 - *Objet*
 - *Byte*
- Les messages contiennent des headers
 - Métiers
 - Techniques

Synchrone VS Asynchrone

- Echange synchrone (non supporté sur les MOM nativement)



- Echange asynchrone



- Producer
 - Celui qui produit le message
- Consumer
 - Celui qui consomme le message
- Les brokers découplent totalement les producers des consumers
 - Pas besoin d'être côté à côté
 - Pas besoin d'une réponse immédiate (potentiellement jamais)
 - Ils ne se connaissent pas
 - N'ont pas besoin d'être implémenter avec la même technologie (Java, Cobol)



JMS

L'API JMS (Java Message Service)

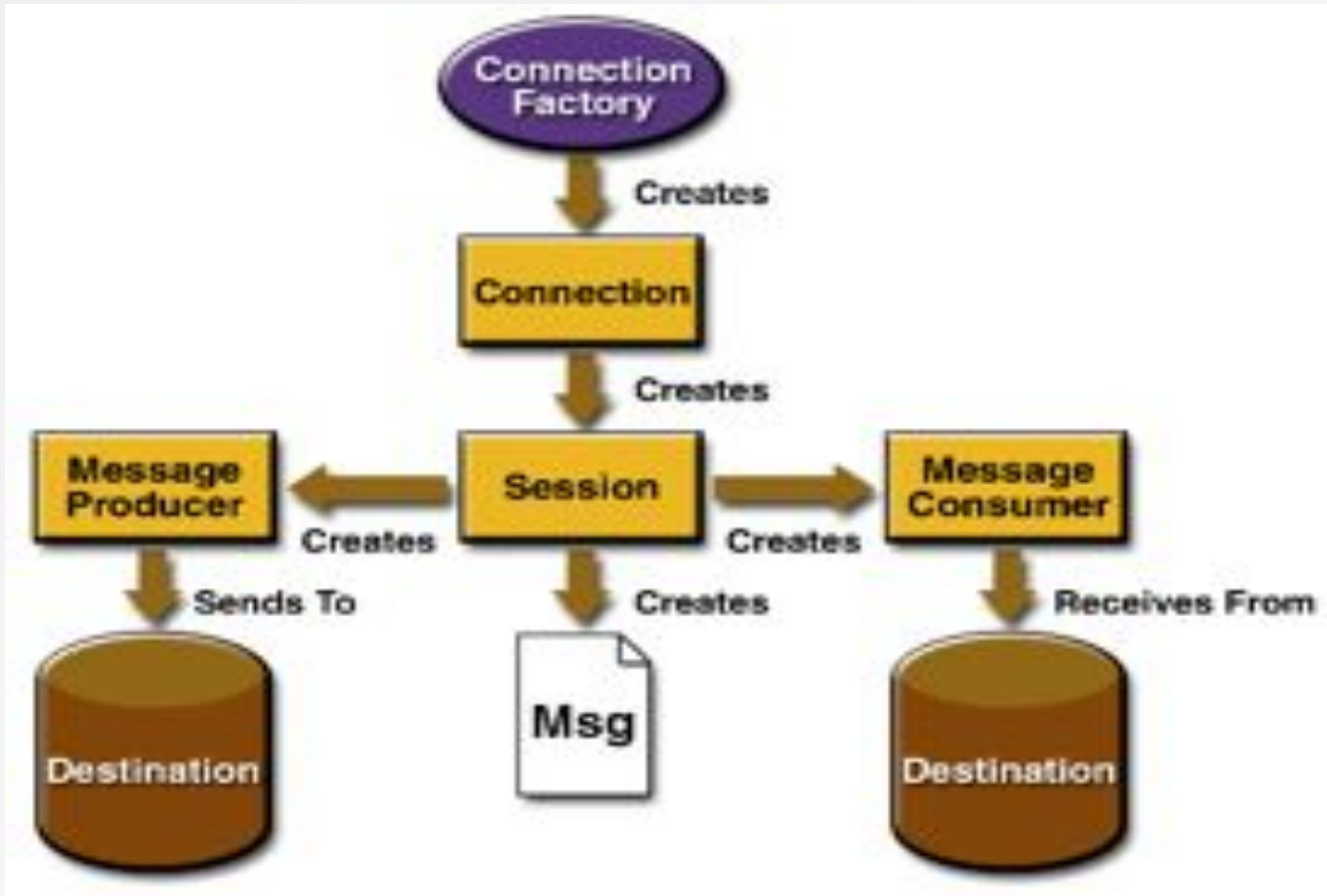
- JMS est une API JAVA
 - Ensemble d'interfaces
 - Ensemble de principes de fonctionnement du MOM
- Définition standard Java et abstraction du broker de messages
 - Très pratique pour changer d'implémentation / driver JMS

- Les messages sont stockés sur des files de messages
 - Queue
 - Topic
- La queue
 - Un seul consommateur reçoit le message
- Le topic (publish / subscribe)
 - Tous les consommateurs connectés reçoivent une copie du message

- JMS est une API JAVA
 - Ensemble d'interfaces
- Définition standard Java et abstraction du broker de messages
 - Très pratique pour changer d'implémentation / driver JMS
- Elle spécifie au niveau développement les conventions d'accès
 - Au broker
 - Au message
 - À la connexion

L'API JMS (Java Message Service)

- API « old school »



- Récupération d'une ConnectionFactory JMS
 - Généralement via un lookup ou bean Spring

```
@Resource  
private ConnectionFactory connectionFactory;
```

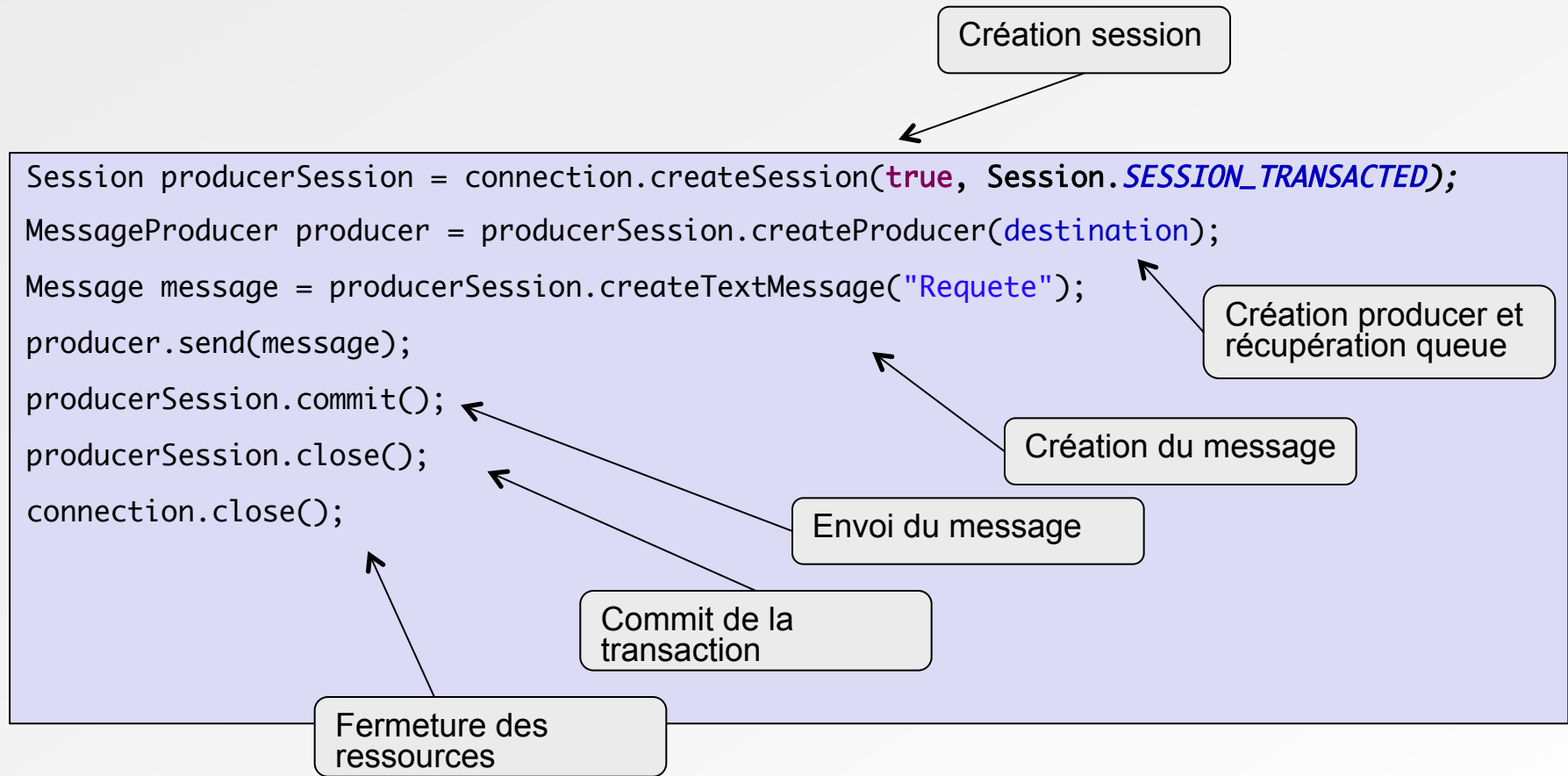
- Récupération de la queue / topic
 - Généralement via un lookup ou bean Spring

```
@Resource  
private Destination destination;
```

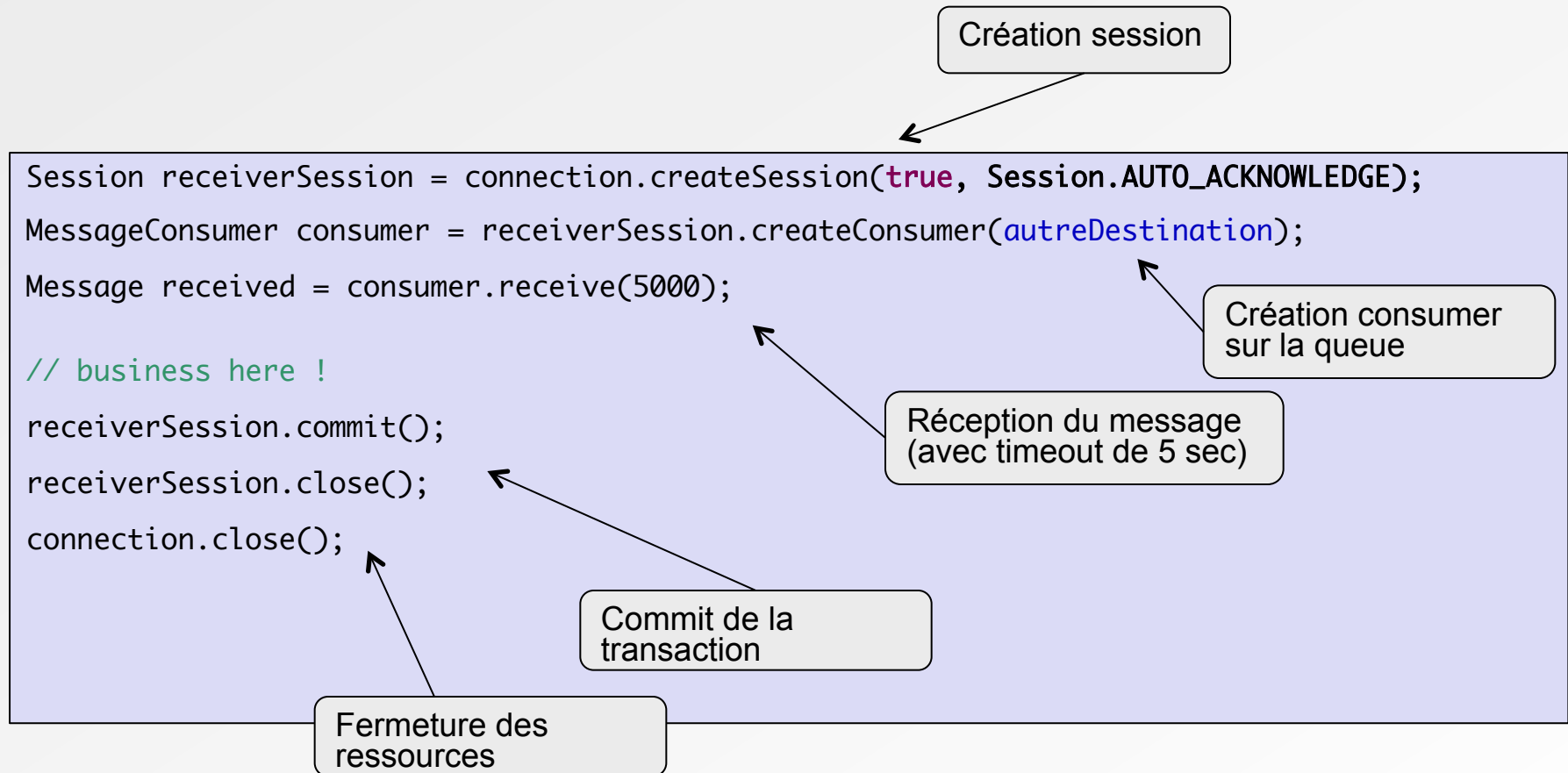
- Création et établissement d'une Connection

```
Connection connection = connectionFactory.createConnection();  
connection.start();
```

Envoi d'un message JMS



Réception d'un message JMS





AMQP

- Jusqu'en 2012 (!), aucun standard pour les MOM
 - Il existait bien JMS, mais ce n'est pas un standard

- En 2012 est sorti la version 1.0 du standard AMQP
- Pourquoi AMQP ?
 - Besoin d'intégrer des systèmes hétérogènes avec les propriétés des MOM
 - Se baser sur un protocole standard et ouvert
 - Le langage et l'implémentation n'importe pas



asynchrone

SMTP

?

synchrone

HTTP

IIOP

unreliable

reliable

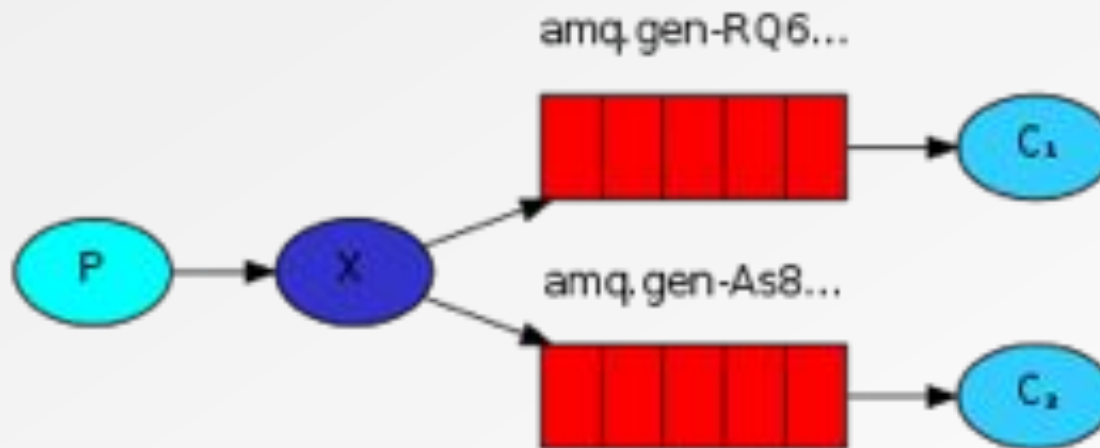
- En 2012 est sorti la version 1.0 du standard AMQP
- Pourquoi AMQP ?
 - Besoin d'intégrer des systèmes hétérogènes avec les propriétés des MOM
 - Se baser sur un protocole standard et ouvert
 - Le langage et l'implémentation n'importe pas

AMQP != JMS

RabbitMQ – Broker AMQP performant

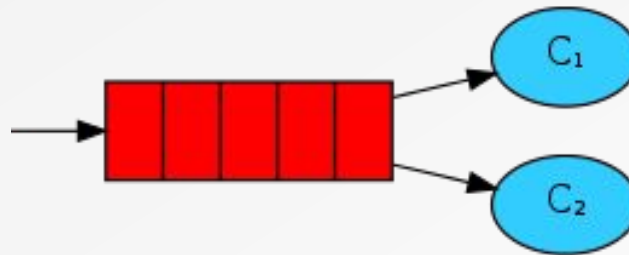
- Broker de messages multi-protocoles
 - AMQP, SMTP, STOMP, XMPP, etc.
- Construit autour du protocole AMQP
- Des « bindings » dans la plupart des langages
 - Java, .NET, Python, JavaScript, Ruby, PHP, etc.

Le fonctionnement d'AMQP

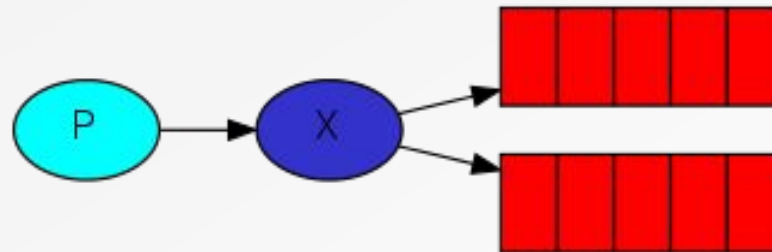


- Queue
 - Les messages sont consommés à partir des *queues*
 - Les messages sont stockés dans les *queues*
 - Les *queues* sont FIFO
- Exchange
 - Les messages ne sont pas directement envoyés sur les *queues*
 - Ils sont routés vers les *queues* au travers des *exchanges*
- Binding
 - Les *queues* sont connectées aux *exchanges* grâce aux bindings
 - Les *bindings* se font grâce à des *patterns*

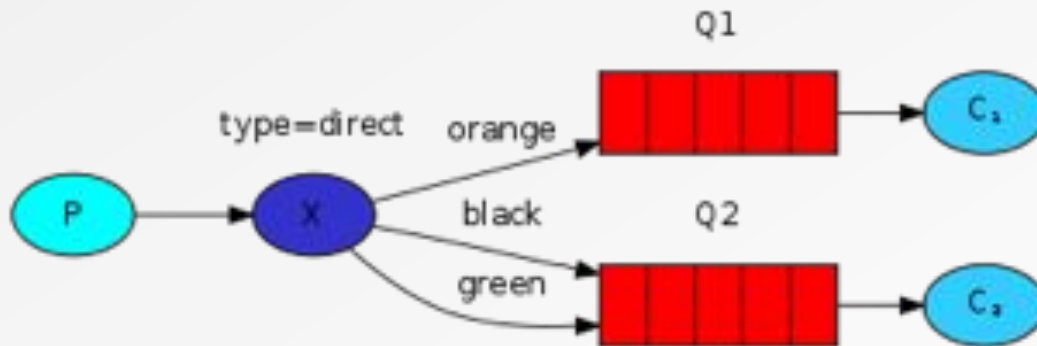
- Consumer
 - Les *consumers* sont directement connectés aux *queues*
 - Plusieurs *consumers* peuvent être connectés à la même *queue*
 - Dans ce cas, le message n'est délivré qu'à un seul *consumer*
 - Le mode de dispatch est round-robin



- Producer
 - Les *producers* ne sont pas connectés au *queue*
 - Ils sont connectés aux *exchanges*
 - Plusieurs *producers* peuvent être connectés au même *exchange*



- Le routage
 - Les *messages* sont publiés avec une *routing-key*
 - Les messages sont routés jusqu'aux *queues* par *matching* entre la *routing-key* et les patterns des *bindings*



- Les différents types d'Exchange
 - fanout
 - Pas de pattern, pas de routing-key, le lien est direct
 - direct
 - le pattern du binding est simplement le nom de la queue
 - topic
 - le pattern du binding est une expression qui vérifie le nom de la queue (pattern matching)
 - headers
 - Pattern appliqué sur un header défini

- Attention
 - Le topic en AMQP n'est pas un publish / subscribe !
 - Uniquement un type de routage
- Pour faire un mode pub/sub en AMQP
 - Chaque consumer doit avoir sa propre queue

Envoyer un message AMQP

- Il n'y a pas d'API autour d'AMQP, les codes sont donc dépendants des implémentations et des API des drivers des brokers
- Création de la connexion AMQP vers le broker RabbitMQ

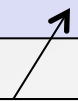
```
ConnectionFactory factory = new ConnectionFactory();  
factory.setUsername("guest");  
factory.setPassword("guest");  
factory.setVirtualHost("/");  
factory.setHost("localhost");  
factory.setPort(5672);  
Connection connection = factory.newConnection();
```

Envoyer un message AMQP

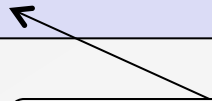
- Création du channel et envoi du message

```
Channel channel = connection.createChannel();  
byte[] message = "message".getBytes();  
channel.basicPublish("quotations", "nasq", null, message);
```

Nom de l'échange



Routing-key



Recevoir un message AMQP

- Création du channel et envoi du message

```
GetResponse response = channel.basicGet("quotations", true);
```

Nom de la queue

Auto acknowledge flag

Message driven

- Jusqu'à présent, nous avons vu uniquement des consommateurs « actifs », qui sont en attente de réception de message bloquante
- Il est évidemment possible de réaliser une attente de façon asynchrone, et d'être notifié lorsqu'un message arrive sur la queue.

- Réception asynchrone JMS

```
import javax.jms.Message;
import javax.jms.MessageListener;

public class JMSMessageDriven implements MessageListener {
    public void onMessage(Message message) {
        // business here
    }
}
```

- Une instance (ou n instances) est ensuite enregistré sur la connexion, et peut ainsi être notifié de façon totalement asynchrone
- Peut aussi être enregistré dans le contexte Spring

- Réception asynchrone AMQP (Spring AMQP)

```
import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessageListener;

public class AMQPMessageDriven implements MessageListener {
    public void onMessage(Message message) {
        // business here
    }
}
```

- Uniquement les imports Java changent (abstraction sur le broker)

Transactions

- Il est également possible de grouper des messages dans une même transactions
 - Plusieurs messages en mode consumer
 - Plusieurs messages en mode producer
 - Un mélange des deux !

Grâce aux transactions des MOM, on peut donc très simplement assurer une intégrité totale et une très grande fiabilité des traitements

→ les traitements critiques sont très souvent placés entre 1 paire de files

- Une des plus grandes forces des MOM est d'être transactionnel
- Au mettre titre que les opérations sur les bases de données, les opérations faites sur les brokers peuvent faire l'objet d'un :
 - Commit (validation)
 - Rollback (annulation)
- On peut donc par exemple :
 - Lire un message sur une file en mode transactionnel
 - Faire un traitement
 - Puis finalement commiter le message

Si un erreur survient lors du traitement, le message non commité sera automatiquement remis sur la file

- En AMQP, on parle plus d'Acknowledgement plutôt que de transaction, bien que les 2 existent
- Le système d'acknowledgement est plus souple et permet de grouper plusieurs messages ensemble (émission ou réception)
- Le système de transaction est utilisé lorsque l'on souhaite coupler dans la même transaction la réception et l'émission.

- Acknowledgement d'un message seul

```
boolean autoAck = false;
GetResponse response = channel.basicGet("queueName", autoAck);

if (response != null) {

    AMQP.BasicProperties props = response.getProps();
    byte[] body = response.getBody();
    long deliveryTag = response.getEnvelope().getDeliveryTag();

    // do business logic here

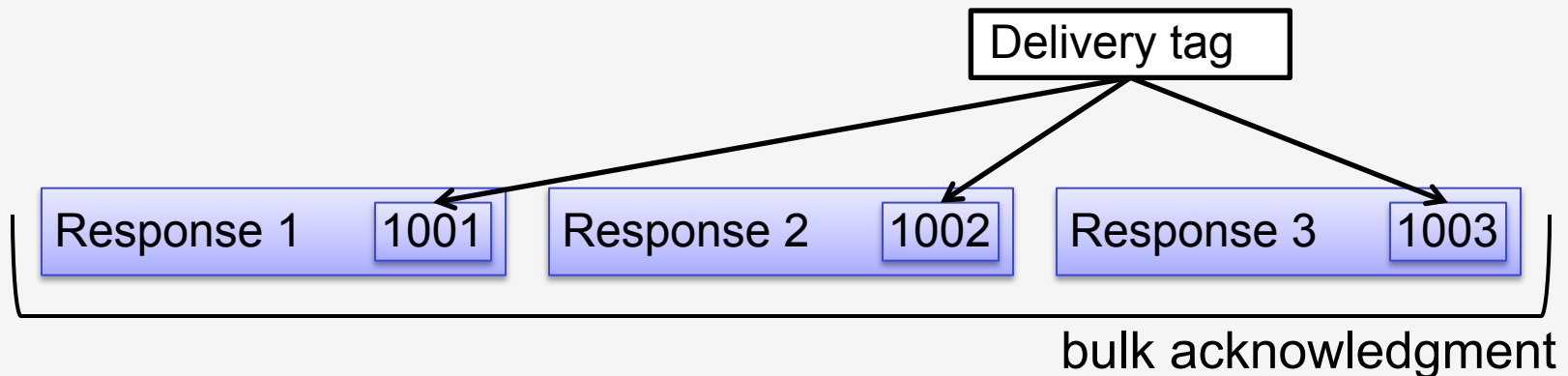
    boolean multiple = false;
    channel.basicAck(deliveryTag, multiple);

}
```

- Acknowledgement d'un groupe de messages

```
boolean autoAck = false;  
GetResponse response1 = channel.basicGet("queueName", autoAck);  
// do business with response1  
  
GetResponse response2 = channel.basicGet("queueName", autoAck);  
// do business with response2  
  
GetResponse response3 = channel.basicGet("queueName", autoAck);  
// do business with response3  
  
long latestDeliveryTag = response3.getEnvelope().getDeliveryTag();  
channel.basicAck(latestDeliveryTag, true);
```

- Tous les messages qui
 - Sont reçus sur le même channel
 - Pas encore reçu d'acknowledge
 - Avec un DeliveryTag inférieur ou égal au DeliveryTag passé à la méthode `basicAck()`



```
channel.basicAck(1003, true);
```

Multiple acks flag

