Java Technologies
Messaging

The Context

- In a distributed system components interact with each other over a network, in order to achieve a common goal:
  - *Message Passing* vs. *Shared Memory*
- Message passing can be performed:
  - *Synchronous* vs. *Asynchronous*
- The components may interact:
  - directly with each other → *tight-coupled*
  - through an *intermediary* → *loose-coupled*

What is Messaging?

- Method of peer-to-peer communication between software components or applications, using an *intermediate agent*.
- **Loosely coupled, Asynchronous, Reliable**
  - differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.
  - differs from electronic mail (email), as it is used for communication between software applications or software components.

Use Cases

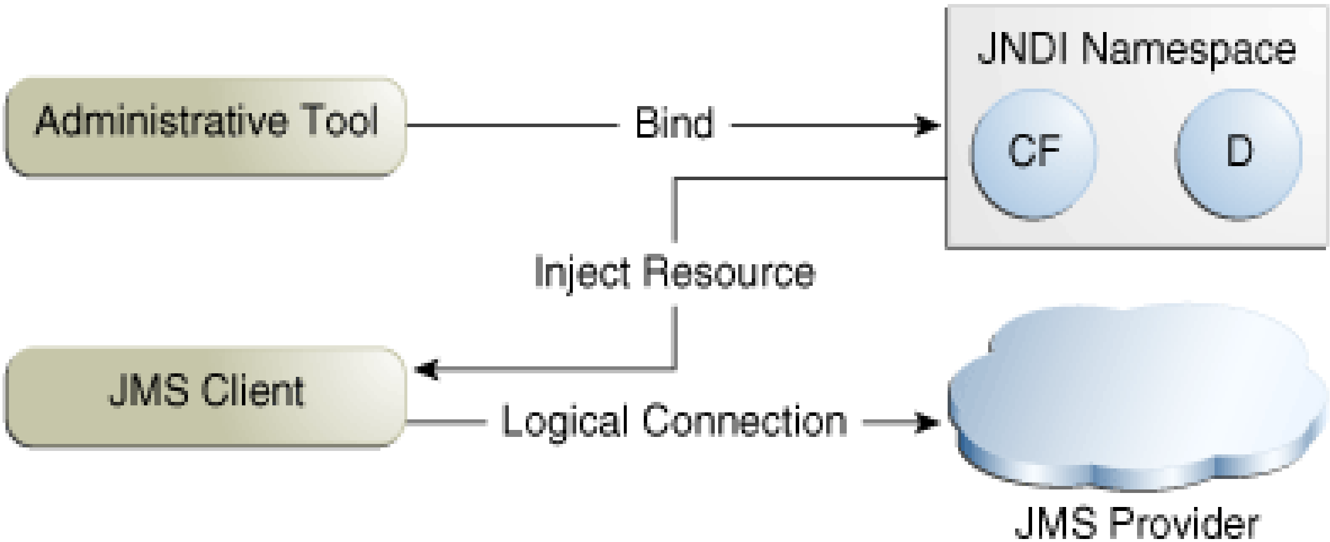
- **Loose coupling**: The provider wants the components not to depend on information about other components' interfaces, so components can be easily replaced
- **Fault-tolerance**: The provider wants the application to run whether or not all components are up and running simultaneously.
- **Asynchronous (delayed) communication**: The application business model allows a component to send information to another and to continue to operate without receiving an immediate response
- **Event-based communication**
- **Scalability**: Handling large number of operations

What is JMS?

Java Message Oriented Middleware (MOM) API

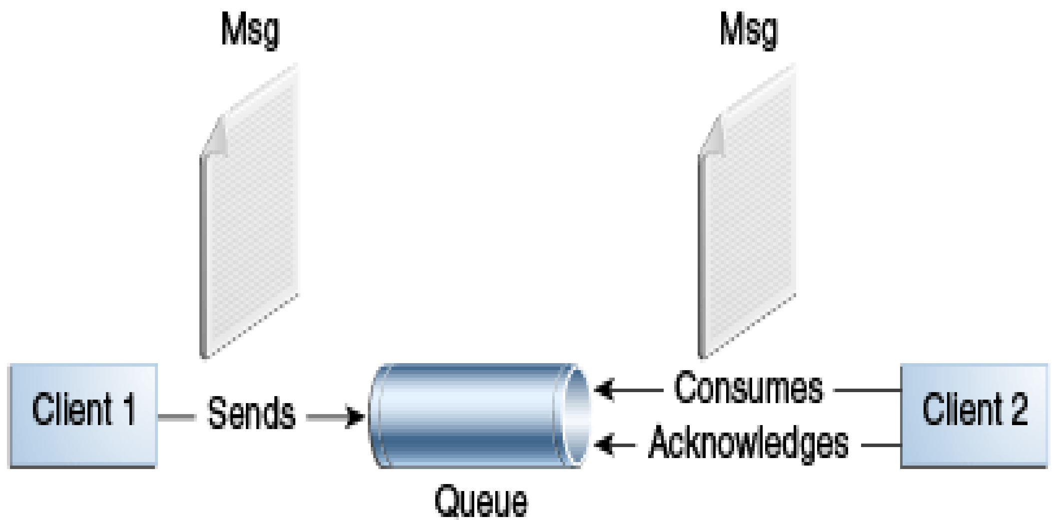
- The JMS API defines **a common set of interfaces** and associated semantics that allow programs written in the Java to communicate with other messaging implementations, regardless of the protocol.
- Allows applications to **create, send, receive,** and **read messages**.
- Offers **portability** of JMS applications across JMS providers.
- **Implementations:** OpenMQ (Oracle), ActiveMQ (Apache), RabbitMQ (Pivotal), JBoss Messaging, etc.
- Application servers usually have a JMS implementation included, by default.

JMS Architecture



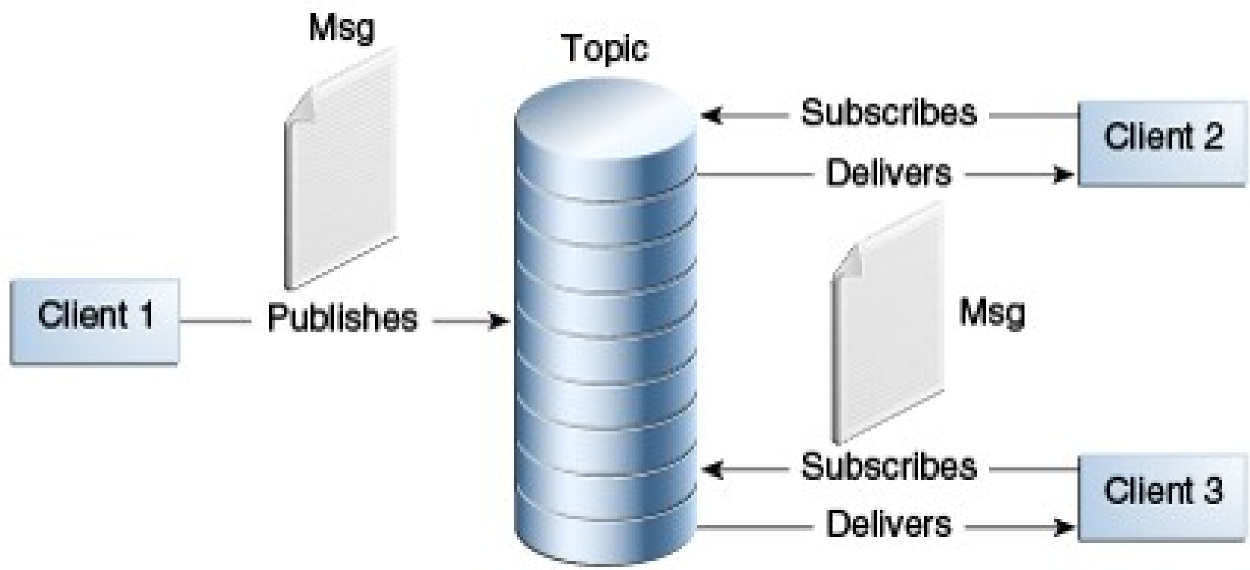
- A **JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features.
- **JMS clients** are the programs or components, that produce and consume messages. Any Java EE or SE application component can act as a JMS client.
- **Messages** are the objects that communicate information between JMS clients.
- **Administered objects** are JMS objects configured for the use of clients:
  - connection factories
  - destinations

Point-to-Point (PTP) Model



- PTP model is built on the concept of:
  - **message queues, senders, and receivers.**
- Each message is **addressed to a specific queue**, and receiving clients extract messages from the queues established to hold their messages.
- Queues retain all messages sent to them until the messages are consumed or expire.
- **Each message has only one consumer.**
- The receiver can fetch the message whether or not it was running when the client sent the message.

Publish/Subscribe Model

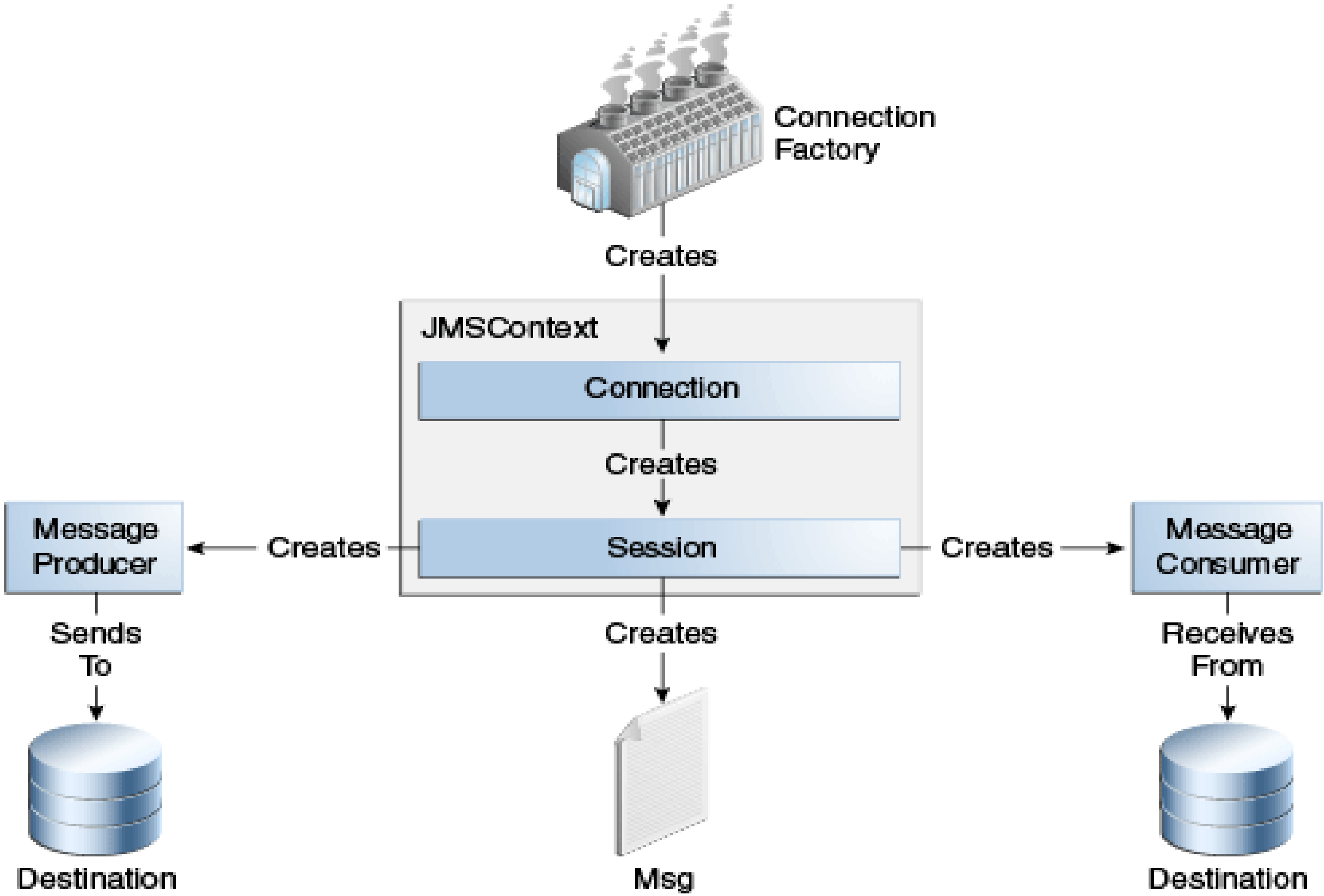


- Clients address messages to **a topic**.
- Publishers and subscribers can **dynamically publish or subscribe to the topic**.
- The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to subscribers.
- **A topic can have many consumers**, but a subscription has **only one subscriber**.
- A client that subscribes to a topic can consume only messages sent after the client has created a subscription, and the consumer must continue to be active in order for it to consume messages (except for duarable supscriptions).

Message Consumption

- **Synchronously / Blocking**: A consumer explicitly fetches the message from the destination by calling the *receive* method. The receive method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
- **Asynchronously / Non-blocking**: An application client or a Java SE client can register a *message listener* with a consumer. calling the listener's *onMessage* method, which acts on the contents of the message.

The Programming Model



JMS Administered Objects

- A **connection factory** is the object a client uses to create a connection to a provider.

```
@Resource(lookup = "MyConnectionFactory")
private ConnectionFactory connectionFactory;
```

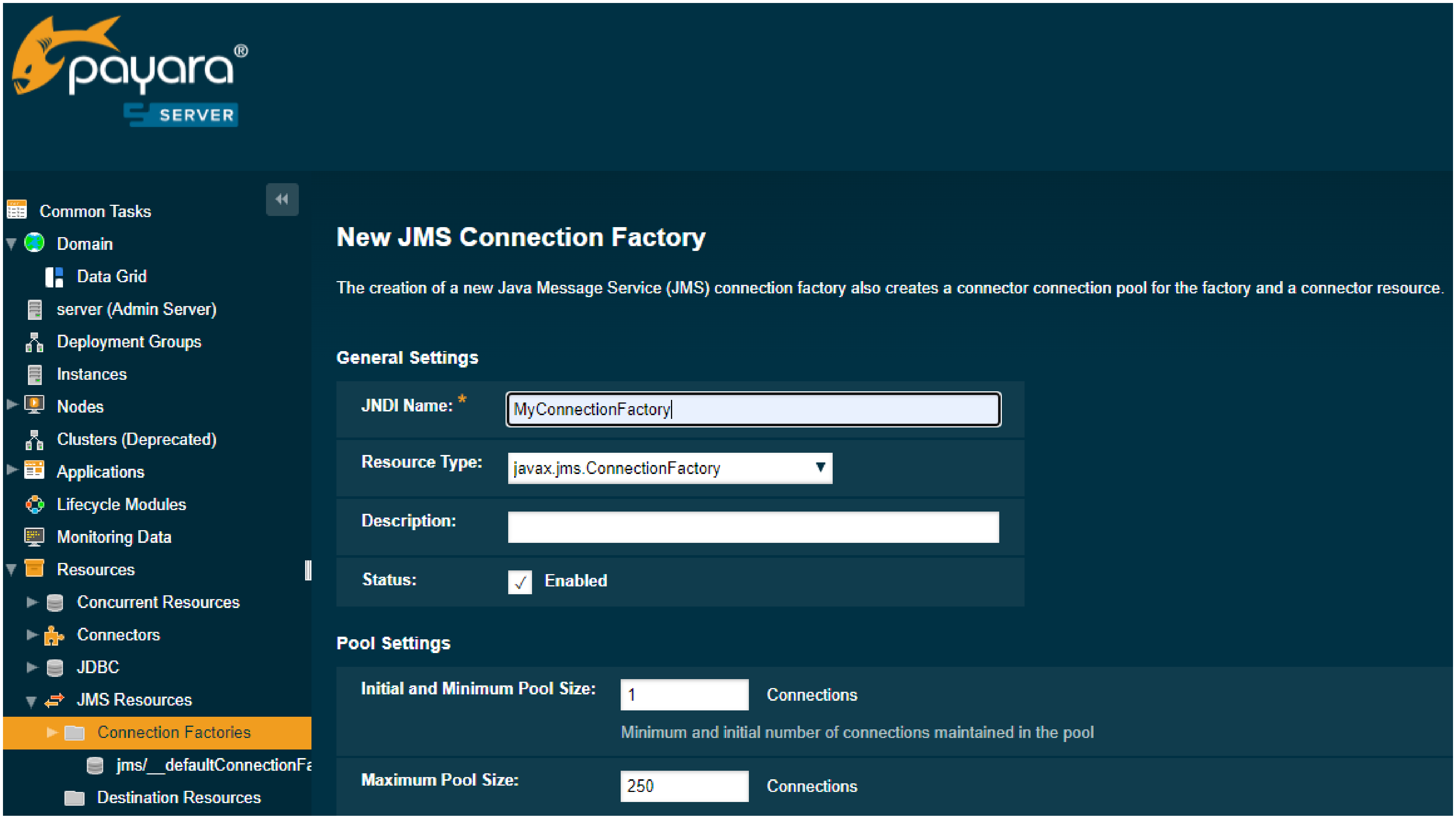
- A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes.

```
@Resource(lookup = "MyQueue")
private Queue queue;

@Resource(lookup = "MyTopic")
private Topic topic;
```

Example (Payara)

Payara already comes with OpenMQ



JMS Dependency

- Jakarta EE 8 Web Profile contains JMS
- Java EE 7 Web Profile does not contain JMS

```
<dependency>
    <groupId>javax.jms</groupId>
    <artifactId>javax.jms-api</artifactId>
    <version>2.0.1</version>
</dependency>
```

Connections, Sessions

- A **connection** encapsulates a virtual connection with a JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon.
- You use a connection to create one or more sessions
- A **session** is a single-threaded context for producing and consuming messages.
- A session provides a **transactional context** with which to group a set of sends and receives into an atomic unit of work.

JMS Context

- A **JMSContext object** combines a *connection* and a *session* in a single object.
- Responsible with creation of JMS objects:
  - Message producers
  - Message consumers
  - Messages
  - Queue browsers

```
JMSContext context = connectionFactory.createContext();
```

Message Producers / Consumers

- A **message producer** is an object used for *sending messages to a destination*.
- **A message consumer** is an object used for *receiving messages sent to a destination*.

```
try(JMSContext context = connectionFactory.createContext();) {
  JMSProducer producer = context.createProducer();
  Message message = create the message ;
  producer.send(dest, message); //dest is a queue or a topic
  ...
  JMSConsumer consumer = context.createConsumer(dest);
  Message message = consumer.receive();
  // or
  consumer.setMessageListener( new MssageListener() {
    public void onMessage(Message message) {
     // Cod specific de tratare a mesajului
    }});
}
```

Messages

A JMS message can have three parts:

- **Header**: values used by both clients and providers to identify and route messages:
  - JMSSessionId, JMSDestination, JMSType, JMSPriority, ...
- **Properties**: used to provide compatibility with other messaging systems, or to create *message selectors*.
- **Body**: [Text, Map, Bytes, Stream, Object] Message

```
TextMessage message = context.createTextMessage();
message.setText("Hello World!");
String body = message.getText();
```

Message Selectors

- A **selector** filters the messages it receives.
- It allows a message consumer to specify the messages it is interested in. Message selectors assign the work of filtering messages to the JMS provider rather than to the application.
- A message selector is a *String* that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. Examples:
  - `"NewsType = ’Sports’ OR NewsType = ’Opinion’"`
  - `"JMSType = 'car' AND color = 'blue' AND weight > 2500"`
- A selector can be specifed when you create a consumer.

  The message consumer then receives only messages whose headers and properties match the selector. A message selector cannot select messages on the basis of the content of the message body.

Queue Browsers

- *Messages sent to a queue remain in the queue* until the message consumer for that queue consumes them.
- The JMS API provides a QueueBrowser object that allows you to browse the messages in the queue and display the header values for each message.

```
QueueBrowser browser = context.createBrowser(queue);
```

Acknowledgement & Transactions

- **Acknowledgement**: until a JMS message has been acknowledged, it is not considered to be successfully consumed.
  - AUTO_ACKNOWLEDGE
  - CLIENT_ACKNOWLEDGE → message.acknowledge()
- A **transaction** groups a series of operations into an *atomic unit of work*.

```
JMSContext context =
   connectionFactory.createContext(JMSContext.SESSION_TRANSACTED);
…
context.commit(); // or context.rollback();
```

You can send multiple messages in a transaction, and *the messages will not be added to the queue or topic until the transaction is committed*. If you receive multiple messages in a transaction, *they will not be acknowledged until the transaction is committed*.

Sending Messages Asynchronously

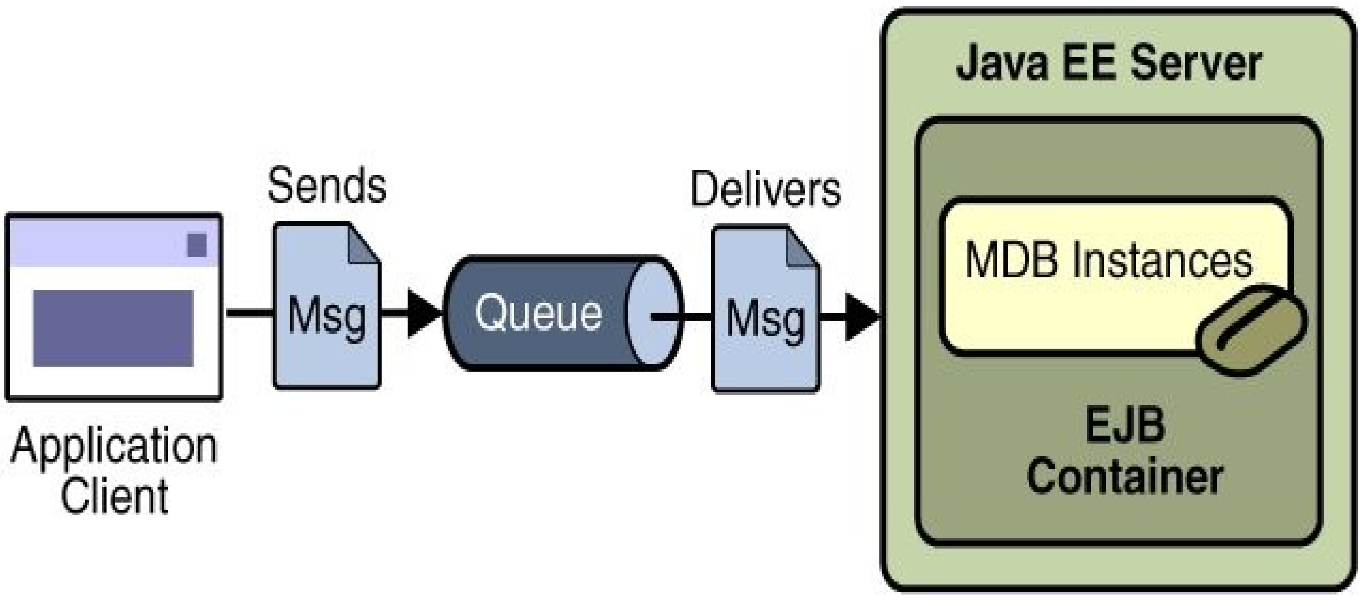
- The send method *blocks* until the JMS provider confirms that the message was sent successfully.
- Sending a message asynchronously involves supplying a callback object → *CompletionListener*

```
CompletionListener listener = new SendListener();
context.createProducer().setAsync(listener).
    send(dest, message);
```

```
public class SendListener implements CompletionListener {
  public void onCompletion(Message message) {
    System.out.println("Send has completed.");
  }
  public void onException(Message message, Exception e) {
    System.out.println("Send failed: " + e.toString());
    System.out.println("Unsent message is: \n" + message);
  }
}
```

Message-Driven Beans

- Process messages **asynchronously**.
- Similar to an event listener, implements *javax.jms.MessageListener* interface.



Creating a MDB

```
@MessageDriven(mappedName="jms/Queue")
public class SimpleMessageBean implements MessageListener {
  public void onMessage(Message inMessage) {
    TextMessage msg = null;
    try {
      if (inMessage instanceof TextMessage) {
        msg = (TextMessage) inMessage;
      } else {
        ...
      }
    } catch (JMSException e) {
      e.printStackTrace();
    }
  }
}
```

Sending a Message

```
@Resource(mappedName="jms/ConnectionFactory")
private ConnectionFactory connectionFactory;

@Resource(mappedName="jms/Queue")
private static Queue queue;

JMSContext context = connectionFactory.createContext();

producer = context.createProducer(queue);

Message message = context.createTextMessage();
message.setText("Hello");

producer.send(message);
```

the destination

Advanced JMS Features

- Specifying Message Persistence
  - DeliveryMode.PERSISTENT, NON-PERSISTENT
- Setting Message Priority Levels
  - from 0 (lowest) to 9 (highest), the default level is 4
- Allowing Messages to Expire
  - setTimeToLive
- Specifying a Delivery Delay
  - setDeliveryDelay
- Creating Temporary Destinations

JMS Disadvantages

- JMS is a **Java API** for messaging. In applications using microservices this may be a problem.
- JMS is not a messaging protocol (and does not enforce one). If you want to learn about protocols take a look at:
  - **STOMP**: Simple (or Streaming) Text Orientated Messaging Protocol.
  - **AMQP**: Advanced Message Queuing Protocol
- Scalability can become an issues in complex scenarios. JMS may not be well suited to large scale message processing applications, where there is a requirement for very high throughput.

Event Streaming

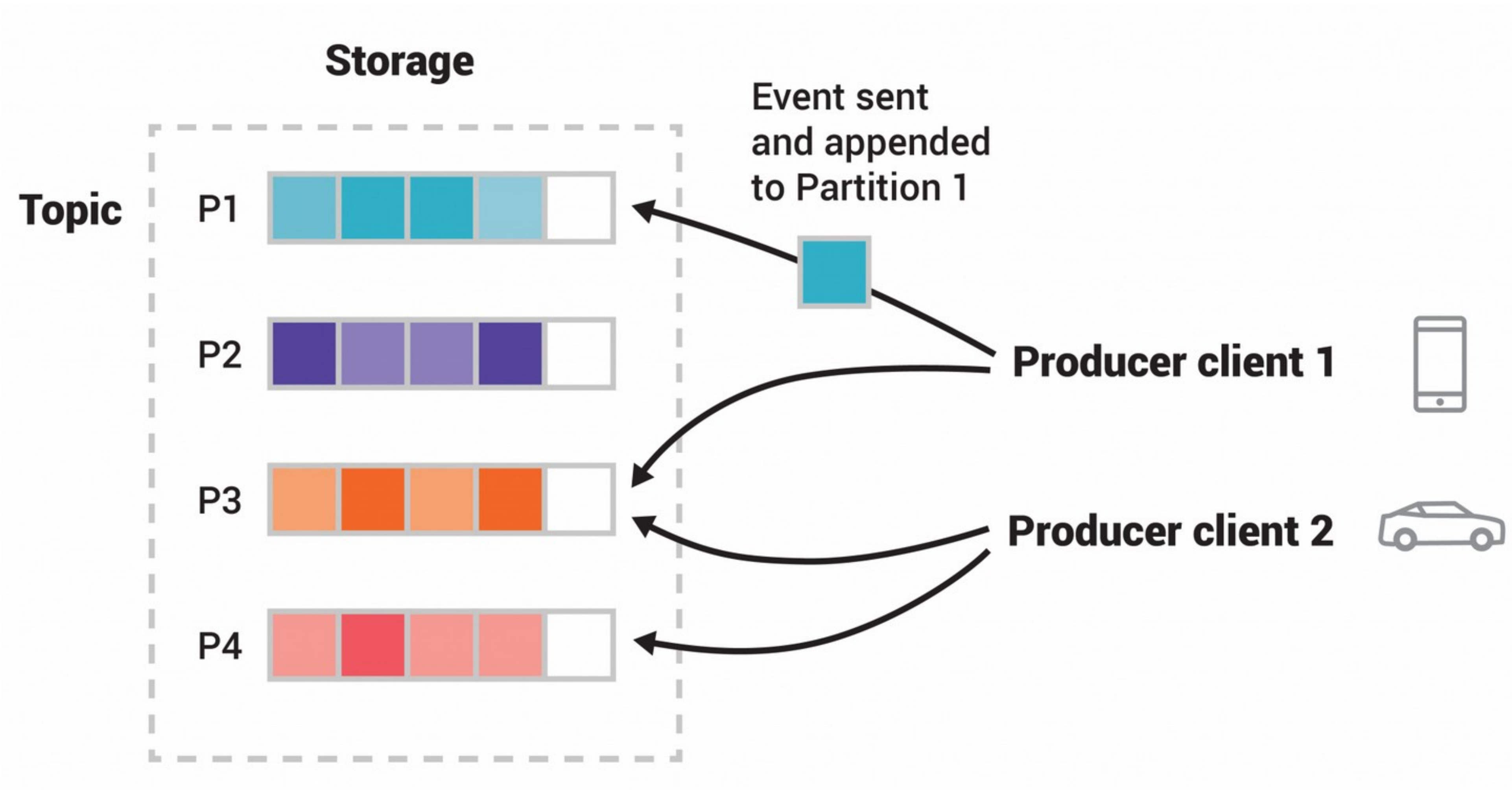
https://kafka.apache.org/documentation

- Event streaming is the practice of:
  - capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and other applications in the form of streams of events;
  - storing these event streams durably for later retrieval;
  - manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and
  - routing the event streams to different destination technologies as needed.
- Ensures a continuous flow and interpretation of data.
- Crucial for data platforms, event-driven architectures, and microservices.
- Better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

Key Concepts

- **Server**: *Brokers*: form the storage laye; *Connectors*: continuously import and export data as event streams; A server *cluster* should be highly scalable and fault-tolerant.
- **Client**: *Producers* and *Consumers;* Fully *decoupled* and *agnostic* of each other
- **Event:** An event has a key, value, timestamp, and optional metadata headers; (also called *Record* or *Message*);
- **Topic**: Multi-producer and multi-subscriber; Events are not deleted after consumption by default.
- **Partition**: A topic is spread over a number of "buckets" located on different brokers → important for scalability.
- **Replication**: There are always multiple brokers that have a copy of the data in topics → fault-tolerance and high-availability

Partitions



Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

Use Cases

Real-time, high volume, data centralization

- Traditional Messaging
- Website Activity Tracking
- Operational monitoring data (Metrics)
- Log Aggregation
- Stream Processing
  - Multiple stage processing pipelines
- Event Sourcing
  - state changes are logged as a time-ordered sequence of records
- Commit Log, etc.

Kafka API Example (Producer)

```
 static final String SERVER = "localhost:9092";
 static final String TOPIC = "myTopic";

Properties props = new Properties();
props.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                  SERVER);
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
          StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
          StringSerializer.class.getName());

try (Producer<String, String> producer =
                                  new KafkaProducer<>(props)) {
  ProducerRecord record =
                 new ProducerRecord(TOPIC, "Hey Kafka!");
  producer.send(record);
  producer.flush(); producer.close();
}
```

Serialization is the process of converting an object into a stream of bytes that are used for transmission.

Kafka API Example (Consumer)

```
Properties props = new Properties();
props.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, SERVER);
props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaExampleConsumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
          StringDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
          StringDeserializer.class.getName());

try (Consumer consumer = new KafkaConsumer(props)) {

  consumer.subscribe(Collections.singletonList(TOPIC));

  ConsumerRecords<Long, String> consumerRecords
                      = consumer.poll(Duration.ofSeconds(1));

  for (ConsumerRecord record : consumerRecords) {
    System.out.printf("Consumer Record:(%d, %s, %d, %d)\n",
                    record.key(), record.value(),
                    record.partition(), record.offset());
};

///message should not be considered as consumed
consumer.commitAsync();
```

MicroProfile Reactive Messaging

https://github.com/eclipse/microprofile-reactive-messaging

- Provides an easy and **standard** way to asynchronously send, receive, and process messages that are received as *continuous streams of events*.
- Provides a *Connector API* so that your methods can be connected to external messaging systems that produce and consume the streams of events, such as Apache Kafka.
- Implementations: Lightbend Alpakka, SmallRye Reactive Messaging, Open Liberty, Quarkus

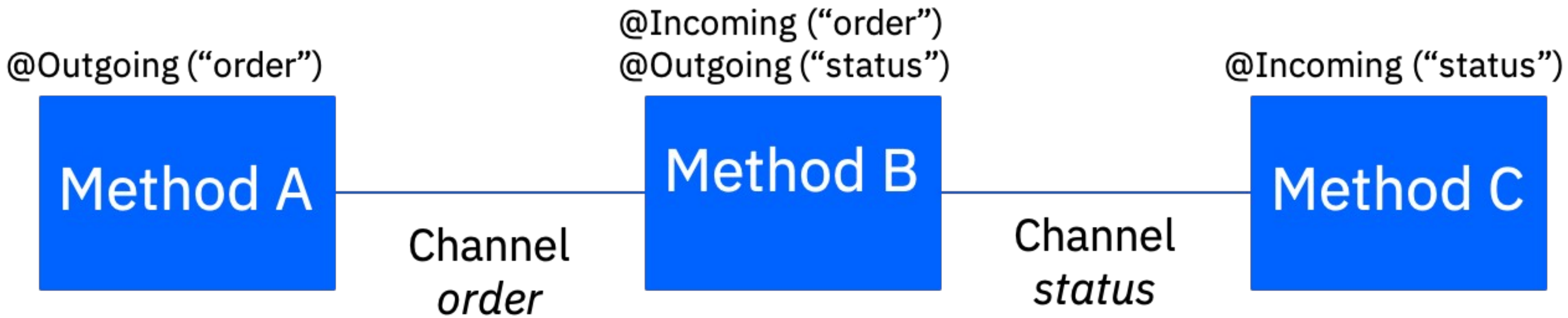
Architecture

- An application using Reactive Messaging is composed of CDI beans **consuming**, **producing** and **processing** messages. These messages can be internal to the application or can be sent and received via different message brokers.
- A **channel** is a name indicating which source or destination of messages is used.
  - Internal channels are local to the application. They allow implementing multi-step processing where several beans from the same application form *a chain of processing*.
  - Channels can be connected to remote brokers or various message transport layers such as Apache Kafka or to an AMQP broker. These channels are managed by connectors.
- A **message** is an envelope wrapping a payload. A message is sent to a specific channel and, when received and processed successfully, acknowledged.

Reactive Messaging API

https://download.eclipse.org/microprofile/microprofile-reactive-messaging-1.0/apidocs/

- *@Incoming*: Used to signify a subscriber to incoming messages.
- *@Outgoing*: Used to signify a publisher of outgoing messages.



https://download.eclipse.org/microprofile/microprofile-reactive-messaging-1.0/microprofile-reactive-messaging-spec.pdf

Example (Producer)

```
@ApplicationScoped
public class ProducerBean{
  private Random random= new Random();

  @Outgoing("bar")
  public Flowable<Integer> generatePrices() {
      return Flowable.interval(5, TimeUnit.SECONDS)
          .map(tick -> {
                int price = random.nextInt(1000);
                return price;
           });
  }
}
```

The io.reactivex.**rxjava3**.core.Flowable class implements the Reactive Streams Publisher Pattern and offers factory methods, intermediate operators and the ability to consume reactive dataflows.

Example (Consumer)

```
@ApplicationScoped
public class ConsumerBean {

    @Incoming ("foo")
    public void consume(int price) {
        System.out.println(
                    "Consumer recieved: " + price
                    + " @" + System.currentTimeMillis());
    }

}
```

microprofile-config.properties

```
mp.messaging.outgoing.bar.connector=liberty-kafka
mp.messaging.outgoing.bar.bootstrap.servers=localhost:9092
mp.messaging.outgoing.bar.topic = myTopic
mp.messaging.outgoing.bar.key.serializer=org.apache.kafka.common.serialization.StringSerializer
mp.messaging.outgoing.bar.value.serializer=org.apache.kafka.common.serialization.IntegerSerializer

mp.messaging.incoming.foo.connector=liberty-kafka
mp.messaging.incoming.foo.bootstrap.servers=localhost:9092
mp.messaging.incoming.foo.topic = myTopic
mp.messaging.incoming.foo.group.id=foo-reader
mp.messaging.incoming.foo.key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
mp.messaging.incoming.foo.value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer
```

Dependencies

<feature>microProfile-3.0</feature>
<feature>mpReactiveMessaging-1.0</feature>

```
<dependency>
  <groupId>org.eclipse.microprofile.reactive.messaging</groupId>
  <artifactId>microprofile-reactive-messaging-api</artifactId>
  <version>1.0</version>
  <scope>provided</scope>
</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.8.1</version>
</dependency>

<dependency>
    <groupId>io.reactivex.rxjava3</groupId>
    <artifactId>rxjava</artifactId>
    <version>3.1.2</version>
</dependency>
```