**How to start with JMS**

**First need to understand some**

1. **Apache ActiveMQ**

* The ‘Classic’ 5.x Broker
* The ‘next generation’ **Artemis Broker**

1. **JMS API**

* JMS API 1.2
* JMS API 2.0

Q. **What is MQ (Messaging Queue)?**

<https://medium.com/@Mohitdtumce/what-is-message-queue-b5468ff6db50>

Quite a mouthful huh. In layman terms, asynchronous communication means when an application ‘**A’** sends message ‘M’ to another application ‘**B’** but does not require an immediate response to continuing processing. Also ‘**B’** can be busy or disconnected. Once ‘**B’** is available, it can send back response to ‘**A’**. In the meantime ‘**A’** can perform some other tasks as well. So, where do we store ‘M’ in the meantime? Obviously we don’t want ‘M’ to be lost. **MQ provides temporary storage when the destination program is busy or not connected.**

Email is probably the best example of asynchronous messaging. When an email is sent, sender can continue processing other things without an immediate response from the receiver.

# **Message Queue: Message + Queue**

**Queue** contains sequence of messages, sent between applications, awaiting their turn to be processed. Messages placed onto queue are stored until consumers retrieve them.

Sender application is called **producer** and receiver application is called **consumer**. Producer produces messages and consumer consumes messages.

**Message** is the data to be sent from producer to consumer. It can be either a request or a response. MQ doesn’t process message and simply stores it. This way of handling messages **decouple** the producer from the consumer. The producer and the consumer of the message do not need to interact with the message queue at the same time.

# **Advantages of Message Queue**

Message Queues are important because they help in implementing “**decoupling**”. Two or more applications are decoupled if they can communicate to each other without being connected. Also, one application is unaware of implementation of other application. In other words, there is no dependencies between them.

**With decoupled applications:**

1. **Any changes made to one application doesn’t affect other application** as long as communication contract is not breached
2. We can break one monolith application into smaller applications which **reduces overall complexity.**
3. It becomes **easier to maintain and debug applications**.
4. We can have cross platform applications. Smaller applications can be **independently developed in any programming languages and scaled**.

With message queue, there is **increase in reliability and performance** of the system. Producers don’t have to wait for consumers to become available and can add requests in the queue. Consumers can process the message when they are available. No overhead in waiting. MQ are persisting the messages so even if different application components go down, no data will be lost and system will be more fault tolerant.

**<-----End----->**

<https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html>

**Message queuing allows applications to communicate by sending messages to each other. The message queue provides temporary message storage when the destination program is busy or not connected.**

A **queue**is a line of things waiting to be handled, starting at the beginning of the line and processing it in sequential order. A message queue is a queue of messages sent between applications. It includes a sequence of work objects that are waiting to be processed.

A **message** is the data transported between the sender and the receiver application; it's essentially a byte array with some headers at the top. An example of a message could be something that tells one system to start processing a task, it could contain information about a finished task or just be a plain message.

The basic architecture of a **message queue** is simple; there are client applications called **producers** that create messages and deliver them to the message queue. Another application, called a **consumer**, connects to the queue and gets the messages to be processed. Messages placed onto the queue are stored until the consumer retrieves them.

**Message queues**

A message queue provides an asynchronous communications protocol, which is a system that puts a message onto a message queue and does not require an immediate response to continuing processing. Email is probably the best example of asynchronous communication. When an email is sent, the sender continues to process other things without needing an immediate response from the receiver. This way of handling messages decouples the producer from the consumer so that they do not need to interact with the message queue at the same time.

**DECOUPLING AND SCALABILITY**

Decoupling describes how much one piece of a system relies on another piece of the system. Decoupling is the process of separating functions so that they are more self-contained.

A decoupled system is achieved when two or more systems are able to communicate without being connected. The systems can remain completely autonomous and unaware of other functions. Decoupling is often a sign of a computer system that is well structured because it is easier to maintain,

If one process in a decoupled system fails to process messages from the queue, other messages can still be added to the queue and be processed when the system has recovered. You can also use a message queue to delay processing - for example, a producer posts messages to a queue. At the appointed time, the consumers start and process the messages in the queue. A queued message can be stored-and-forwarded, and the message can be redelivered until it is processed.

Q. What is **ActiveMQ** ?

From Official Documentation

* Its an Apache product, that’s why it is called Apache ActiveMQ.
* It is the most popular open source, multi-protocol, Java-based **messaging server**.
* It supports industry standard protocols so users get the benefit of client choices across a broad range of languages and platforms.
* Connectivity from C, C++, Python, .Net, and more is available.
* We can integrate our multi-platform applications using ubiquitous (omni-present) **AMQP** protocol.
* Exchange messages between our web application using **STOMP** over web sockets.
* Manage our IoT devices using **MQTT.**

From Medium’s Blog

<https://blog.fabric8.io/fabric8-mq-not-just-a-message-broker-18ca5af1e0f>

* Enterprise message brokers like Apache **ActiveMQ** are designed to support many different patterns of messaging, from **point**-**2**-**point**, **publish/subscribe** and many flavours in between.
* They are resilient, support high availability, automatic failover, store and forward and different protocols from **MQTT**, **STOMP**, **OpenWire**, **AMQP** to **WebSockets**.
* Over 10 years of development, **ActiveMQ** has become extremely feature rich and flexible, however this flexibility can have some performance cost.
* However, traditional message brokers are not designed for the type of flexibility required for cloud, we need very high scalability and the ability to scale up and down the number of message brokers based on demand. The number of concurrent clients a message **broker** can support isn’t an exact science. It depends on what the clients are doing — how many destinations (Queues/Topics) are being used, and how much message load each client will produce. As a rough guide — the upper limit is going to be 10,000 clients at best.
* **ActiveMQ** supports high availability through a master/slave mechanism, where there is only one active member of a cluster and one or more passive slaves. In such a scenario, you can scale up and down brokers, but only the master will be active, so it’s not going to help horizontally scale up your message brokers.
* **ActiveMQ** also supports store and forward (or networks of brokers) — where messages are delivered from the client to the connected broker, and forwarded on to any other broker that has interested consumers for the messages. This is designed for passing messages over a WAN, but it can also be used to form a mesh of interconnected brokers on the cloud.
* The benefits are that this could provide a mechanism for scaling up and down brokers.

**There are currently two flavours of ActiveMQ.**

1. The ‘Classic’ 5.x broker (ActiveMQ5 classic)
2. The ‘next generation’ Artemis broker (ActiveMQ Artemis)

Once Artemis reaches a sufficient level of feature parity (consistency)

With the 5.x code-base, it will become **ActiveMQ 6**.

**What is a Message Broker?**

In its core, a message broker is “a program that translates a message to a formal messaging protocol of the sender, to the formal messaging protocol of the receiver”

**So, when do I need a Message Broker?**

A Message Broker is really good in one thing — processing messages. This means that when you have a lot of messages (think thousands, millions, billions of messages) that it could be worth looking into a Message Broker to create a centralized store/processor for these messages, so that other applications or users can work with these messages — your single source of truth.

**Which Message Brokers are out there to process my events?**

There are tons of message brokers out there like **ActiveMQ**, Kafka, RabbitMQ, OMS, JMS, Redis, Service Bus, etc.

Q. **What is Apache ActiveMQ Artemis ?**

* It’s one of the two flavours of ActiveMQ.
* First version 1.0.0 was released on date xx-xx-xxxx.
* It’s an open source project to build a multi-protocol, embeddable, very high performance, clustered, asynchronous **messaging system.**
* It’s an example of MoM (Message Oriented Middleware).

Q**. Why use Artemis?**

* 100% open source project.
* Written in Java, runs on any platform with a Java 8+ runtime, that’s everything from Windows desktops to IBM Mainframes.
* Amazing performance.
* Full feature set. All the features we’d expect in any serious messaging system, and others we won’t find anywhere else.
* Elegant, clean-cut design with minimal 3rd party dependencies.
* It can be run as stand-alone, can be integrated with our favourite Java EE application server, or even can be run by embedding inside our product. It’s up to us.
* Seamlessly high availability. It provides a HA (**High Availability**) solution with automatic client failover. So, we can guarantee zero message loss or duplication in event of server failure.

**Q. How do client applications (Producer/Consumer) interact with Messaging System in order to send and consume messages?**

**Producer ? Messaging ? Consumer**

**System**

**(MOM)**

Several Messaging systems provide their own proprietary APIs with

Which the client communicates with the Messaging systems.

**Various Messaging Systems are:**

* Apache ActiveMQ Artemis
* RabbitMQ

**Various Messaging APIs are**:

* JMS API

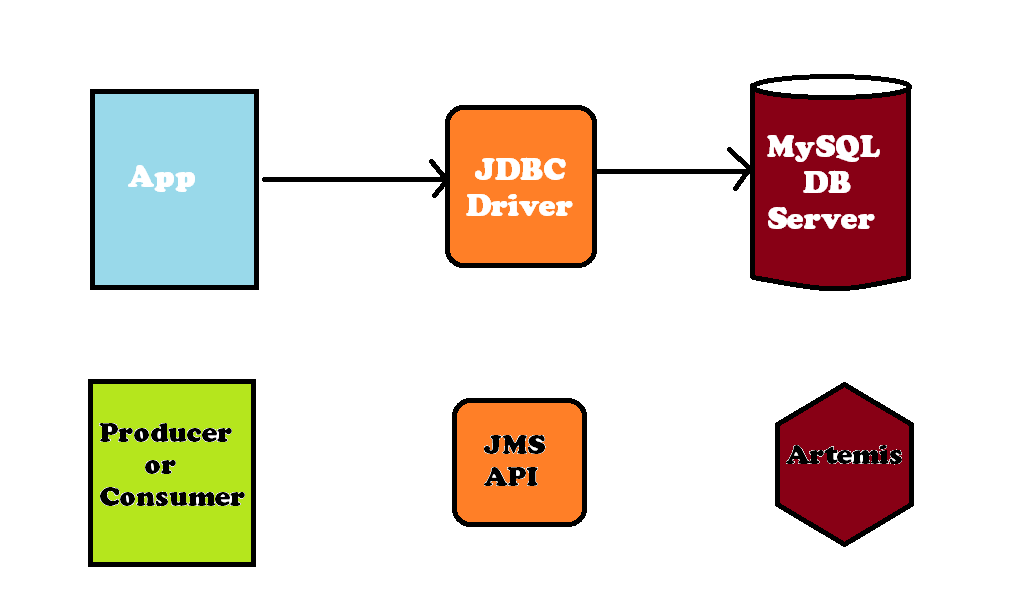
**Producer JMS API Messaging JMS API Consumer**

**System**

**(MOM)**

**Q What is JMS ?**

* It’s a part of Oracle’s Java EE Specification.
* It’s a Java API that encapsulate both Messaging Queue (**MQ**) and Publish-Subscribe Messaging patterns.
* JMS is a very popular API and is implemented by Most Messaging Systems. JMS is only available to clients running Java.
* We can think of JMS API like MySQL connector driver, which helps to communicate with MySQL Database server. In the same way JMS API helps us to communicate with Messaging System/Server (i.e. Artemis or Artemis Broker here).



* JMS API 1.2 or 2.0 will help the client to communicate with the Messaging System (i.e. Artemis).
* Artemis clients, potentially on different physical machines, interact with Artemis broker.
* Apache ActiveMQ Artemis currently ships two API implementations for messaging at client side.
  + - Core Client API
    - JMS 2.0 Client API

* Artemis also provides different protocol implementations on the server, so we can use respective clients for these protocols:
* **AMQP**
* **OpenWire**
* **MQTT**
* **STOMP**
* **HornetMQ**
* **Core(Artemis Core Protocols)**