*Message Queue - What it is ?*

*A Queue in this context could be define as a list of data items, commands, or task, stored so as to be retrieve in a specific order, usually the order of insertion.*

*Message on the other hand, is data transported between a sender and a receiver. A message should be something that tells a system to start processing a particular task, it could contain information about a finished task or just a simple message of a completed task.*

*A message queue is a form of asynchronous service-to-service communication used in microservices architectures. Messages are stored in the queue until they are processed and deleted. Each message is processed only once by a single consumer.*

*The basic structure or architecture of a message queue is simple, you have a client application called producers that create messages or make request to be delivered in a queue. Another application which is called a consumer, connect to the queue and get the messages to be processed. Messages placed in the queue could be stored in a system memory until the consumer processes them accordingly. The producer which is the client and the consumer of the message do not need to interact with the message queue synchronously.*

*Many producers and consumers can use the queue, but each message is processed only once by a single consumer. When a message needs to be processed by more than one consumer, message queues can be combined with Pub/Sub messaging in a fanout design pattern.*

***Use case scenario –***

*Imagine you have a web service or an API that receives many requests every second and all requests needs to be processed by a process that is time-consuming and your web service has to be highly available and ready to receive new request instead of being locked by the processing of previously received requests. In this case, it is ideal to put a queue between the web service and the processing service. Messages in a queue and the other process can take and handle messages in order. The two processes will be separated or decouple from each other and does not need to wait for each other. If you have a lot of requests coming in a short amount of time, the processing system will be able to process them all anyway. The queue will store or persist the requests if their number becomes really huge.*

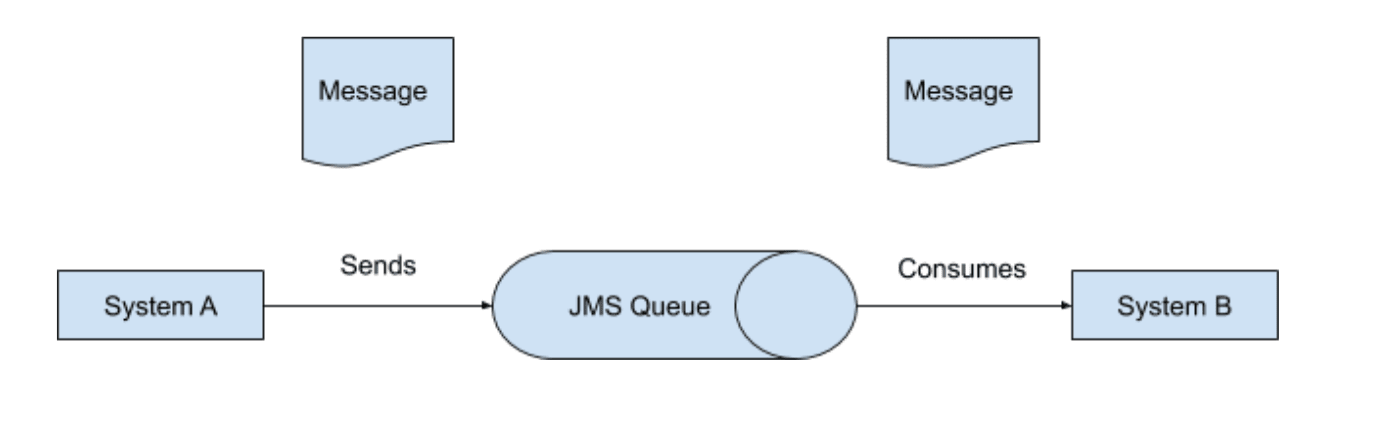
*Now if your workload is growing and you need to scale up your system. All that is needed to be done now is to add more workers, receivers, to work off the queues faster.*

*Another use case could be when working on an imaging system where people upload images and then you have a service that generates thumbnails for each uploaded message, in this case, your best solution would be to implement the concept of “message queue”.*

*Java Message Queue Messaging Models –*

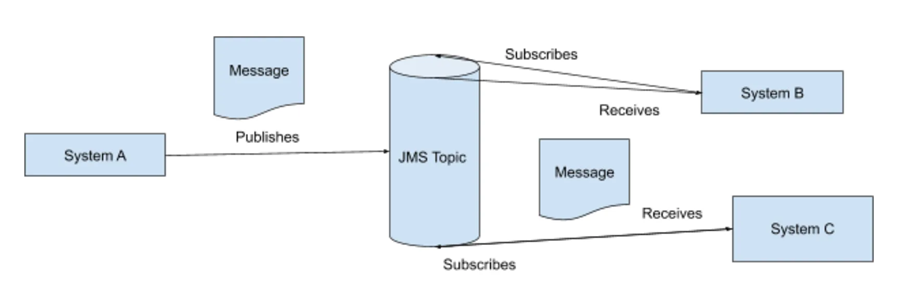
* *Point-to-Point (PTP) Messaging Model - In the Point-to-Point (PTP) style of messaging, one message is delivered to one receiver only. The model is built around the concept of a Queue which is used as a Message-Oriented Middleware (MOM).*

*Messages are stored in the queue by the producer application and the consumer retrieves the messages from the queue. This facilitates the storage and forwarding of messages to the consumer application in an asynchronous decoupled manner. In the PTP model, the message is delivered to the receiver only once. Once received and acknowledged by the consumer, it will then be removed from the queue.*

**

* *Publisher/Subscriber (Pub/Sub) Messaging Model*

*In the Publisher/Subscriber (Pub/Sub) pattern, you have the sender (publisher), the receiver (subscriber), and multiple queues called Topics. The publisher can send one message and it will be delivered to all the subscribers of the Topic just like broadcasting. Here, a Topic is used as a Message-Oriented Middleware (MOM) that is responsible for storing and delivering messages. The message will be removed from the queue only when all the subscribers have successfully read the message or when the retention policy expires.*

**

***Receive modes -***

*You can specify two different modes in which consumers can receive messages from Service Bus.*

* ***Receive and delete****.*

*In this mode, when Service Bus receives the request from the consumer, it marks the message as being consumed and returns it to the consumer application. This mode is the simplest model. It works best for scenarios in which the application can tolerate not processing a message if a failure occurs. To understand this scenario, consider a scenario in which the consumer issues the receive request and then crashes before processing it. As Service Bus marks the message as consumed, the application begins consuming messages upon restart. It will miss the message that it consumed before the crash. This process is often called at-most once processing.*

* ***Peek lock****.*

*In this mode, the receive operation becomes two-stage, which makes it possible to support applications that can't tolerate missing messages.*

*Finds the next message to be consumed, locks it to prevent other consumers from receiving it, and then, return the message to the application.*

*After the application finishes processing the message, it requests the Service Bus service to complete the second stage of the receive process. Then, the service marks the message as consumed.*

*If the application is unable to process the message for some reason, it can request the Service Bus service to abandon the message. Service Bus unlocks the message and makes it available to be received again, either by the same consumer or by another competing consumer. Secondly, there's a timeout associated with the lock. If the application fails to process the message before the lock timeout expires, Service Bus unlocks the message and makes it available to be received again.*

*If the application crashes after it processes the message, but before it requests the Service Bus service to complete the message, Service Bus redelivers the message to the application when it restarts. This process is often called at-least once processing. That is, each message is processed at least once. However, in certain situations the same message might be redelivered. If your scenario can't tolerate duplicate processing, add extra logic in your application to detect duplicates. For more information, see Duplicate detection, which is known as exactly once processing.*

*Advantages*

* *Load balancing: A message queue can be used to distribute workloads across multiple nodes or machines.*
* *Asynchronous processing: A message queue can be used to allow applications to perform tasks asynchronously, without waiting for a response.*
* *Scalability: A message queue can be used to scale up or down the number of application instances handling a particular task based on demand.*
* *Reliable message delivery: A message queue can be used to ensure that messages are delivered reliably, even in the face of network failures or other issues.*

*Disadvantages*

* *Complexity: Distributed message queues like Kafka can add complexity to an architecture, as they require careful management and coordination between services. This can lead to increased development and operational overhead.*
* *Latency: Message queues can introduce additional latency into an architecture, as messages need to be processed and delivered between services. This can be a concern for real-time systems or applications with strict latency requirements.*
* *Scalability: While message queues can improve scalability in an architecture, they also introduce the potential for bottlenecks and performance issues. For example, if a particular service is producing too many messages or consuming them too slowly, it can cause a backlog in the message queue and lead to slower processing times overall.*
* *Cost: Running and managing a distributed message queue like Kafka can be expensive, as it requires dedicated resources and infrastructure. This can be a challenge for startups or smaller companies with limited resources.*

*Message Queue Options*

*Some popular alternatives both open-source and commercial to Apache Kafka include:*

* *RabbitMQ*
* *Amazon Simple Queue Service (SQS)*
* *Apache ActiveMQ*
* *Microsoft Azure Service Bus*