Spring AMQP

Spring AMQP is the Spring implementation of AMQP-based messaging solutions. Spring AMQP provides us a "template" as a high-level abstraction for sending and receiving messages. We will look into following topics in this spring AMQP. tutorial.

1. What is AMQP?
2. Why we need AMQP?
3. Difference between JMS and AMQP
4. How Spring AMQP works?
5. Spring AMQP Modules

1. What is AMQP?

We have already discussed some JMS concepts and examples in my previous posts. In this post, we are going to discuss AMQP protocol and Spring AMQP Messaging.

AMQP stands for Advanced Message Queuing Protocol. AMQP is an open standard protocol for implementing MOMs (Message Oriented Middleware).

2. Why we need AMQP?

We have JMS API to develop Enterprise Messaging systems but why we need another Messaging standard.

The main drawback or limitation of JMS API is interoperability that means we can develop Messaging systems that will work only in Java-based applications. It does not support other languages.

AMQP solves the JMS API problem. The major advantage of AMQP is that it supports interoperability between heterogeneous platforms and messaging brokers. We can develop our Messaging systems in any language (Java, C++, C#, Ruby etc.) and in any operating system; still, they can communicate with each other by **using AMQP based message brokers.**

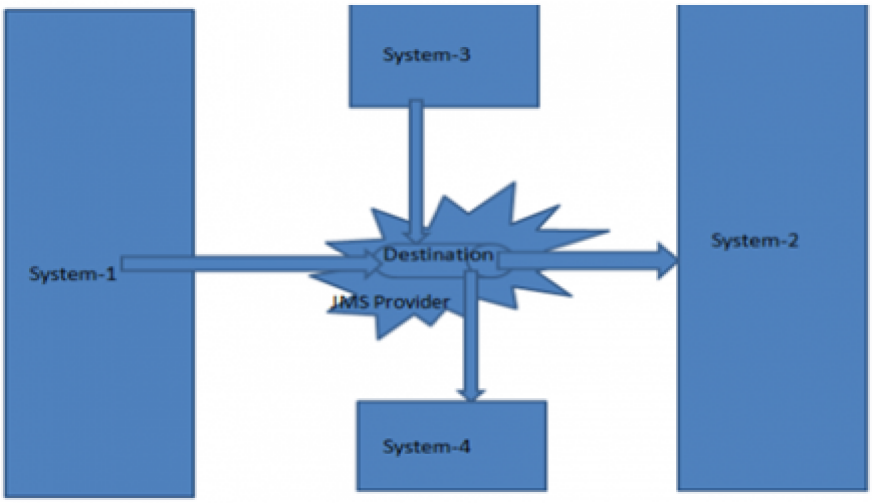
3. Difference between JMS and AMQP

In this section, we will discuss about what are the differences between JMS and AMQP protocols.

1. **Interoperability**

   JMS Application works in any OS environment, but it supports only Java platform. If we want to develop the below system architecture by using JMS API then all those systems should be developed by using Java language only.

   But if we use AMQP standard to develop the same system, then we can develop those systems in any language that means System-1(JAVA), System-2(C#), System-3(Ruby) and System-4(C++).



JMS API is specific for Java Platform only, but AMQP supports many technologies.

2. **Messaging Models**
   As we have already discussed, JMS API supports two kinds of messaging models: P2P(Peer-to-Peer) Model and PUB/SUB (Publisher/Subscriber) Model.

   AMQP supports five different Messaging models (or Exchange types)
   - Direct Exchange
   - Fanout Exchange
   - Topic Exchange
   - Headers Exchange
   - System Exchange
3. **Message Structure**
   JMS Message is divided into **3 parts: Header, Properties, and Body.**
   AMQP Message is divided into **4 parts: Header, Properties, Body, and Footer.**

4. **Message Types**
   MS API supports 5 types of messages as part of the Body section, but AMQP supports only one type of message – Binary (bytes) message.

4. How Spring AMQP works?

Spring Framework provides Spring AMQP API to integrate AMQP Message brokers with Spring applications to develop Enterprise Messaging Systems.

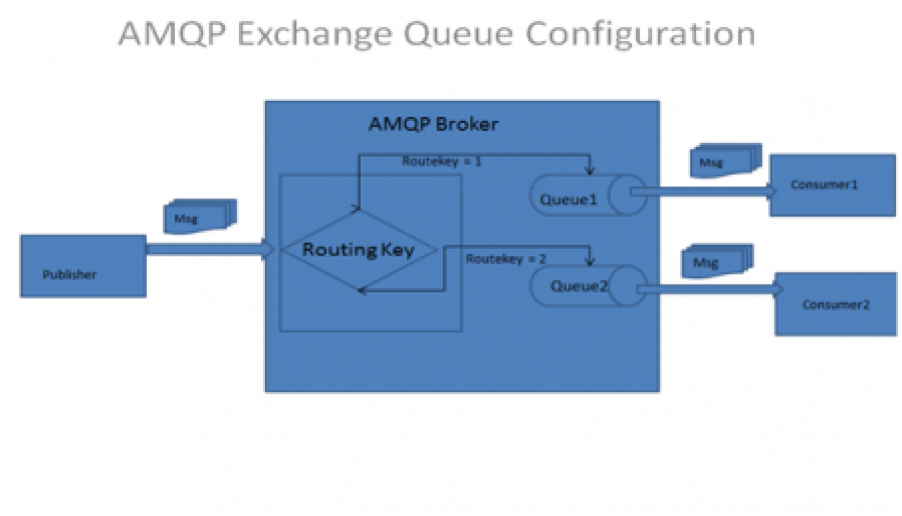
In AMQP Messaging systems, Message Publisher sends messages to Exchange. Publisher does not know which queue is configured to this Exchange and which Consumer is associated with this Queue.

While configuring Exchange, we will map it to one or more Queues by specifying some routing key.

AMQP Consumer is connected to Queue and listening to messages with some routing key.

When Publisher sends a message into Exchange, then Exchange uses this routing key and send

those messages to associated Queue. When the Queue receives messages, then associated Consumer receives those messages automatically.



For example, we have implemented our AMQP Messaging system by following above architecture.

If Publisher sends a message with Route key = 1, then this message is routed to Queue1, then to Consumer1.

If Publisher sends a message with Route key = 2, then this message is routed to Queue2, then to Consumer2.

NOTE: Most popular AMQP protocol implemented Message brokers or JMS Servers are:

1. Apache Active MQ server
2. Spring Rabbit MQ server

Spring AMQP Modules

Spring Framework provides two set of APIs to deal with AMQP protocol servers. Spring AMQP Projects are located at the following website:

https://spring.io/projects/spring-amqp

1. **Spring RabbitMQ AMQP API**
   Spring RabbitMQ AMQP API is used to implement messaging applications with Spring Rabbit MQ server. This API is developed in two Spring modules:
   - **spring-amqp:**
     It is base abstraction for AMQP Protocol implementation.
   - **spring-rabbit:**
     It is the RabbitMQ implementation.

2. **Spring ActiveMQ AMQP API**
   This API is used to implement messaging applications with Apache Active MQ server. This API uses Spring AMQP abstraction module with ActiveMQ API:

- spring-amqp:
  It is base abstraction for AMQP Protocol implementation.
- activemq-spring:
  It is the ActiveMQ implementation.

NOTE:

- Both **Spring AMQP and Spring RabbitMQ** Modules are from Pivotal Team (Spring Framework). RabbitMQ servers is also from Pivotal Team.
- ActiveMQ Server is from Apache Software Foundation. Apache ActiveMQ has released an API to support Spring Framework that is activemq-spring (It's not from Pivotal Team).

Spring RabbitMQ
Spring RabbitMQ is the message broker based on Spring AMQP protocol implementation.

Spring RabbitMQ
In my previous post, we have discussed about AMQP Protocol and Spring AMQP Module in detail theoretically. Before reading this post, Please read that post here: Spring AMQP.

In this post, we are going to discuss on two things mainly: "How to install RabbitMQ Server" and "How to setup Queue & Exchanges in RabbitMQ Server".

In my coming post, we will discuss and develop one Spring AMQP RabbitMQ Messaging application with one simple and useful example.

Spring AMQP Modules
Spring AMQP Projects are located at the following website: https://projects.spring.io/spring-amqp/

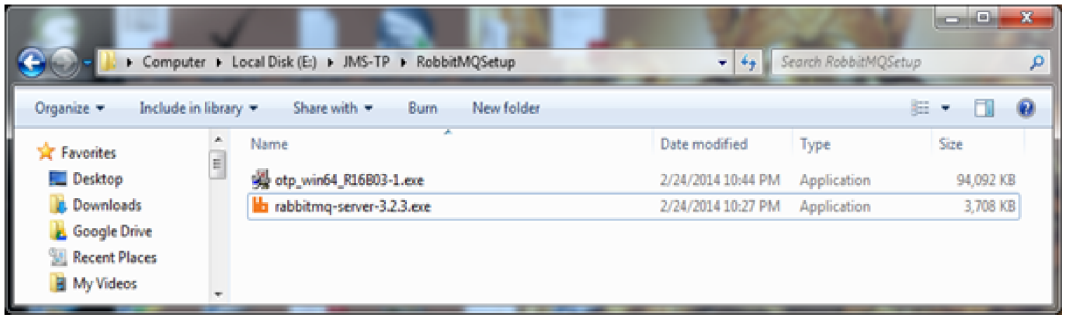
Spring Framework has two modules to support Spring AMQP RabbitMQ development.

- **spring-amqp:**
  It is base abstraction for AMQP Protocol implementation.
- **spring-rabbit:**
  It is the RabbitMQ implementation.

Install Spring RabbitMQ Server
Please use the followings steps to download and install RabbitMQ Server into local system.

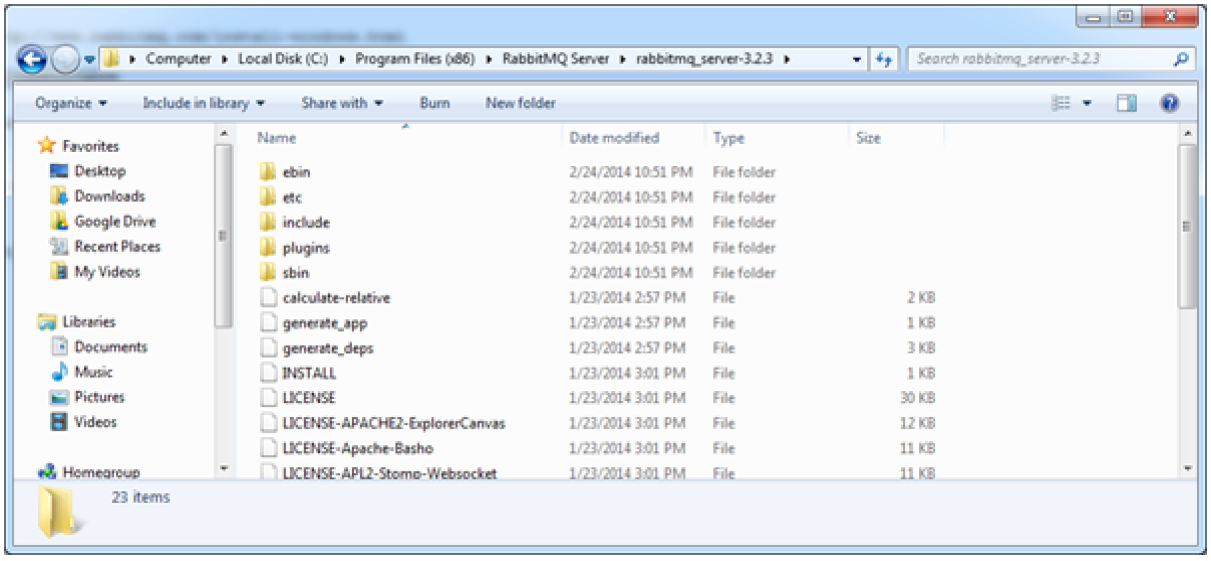
1. **Download Erlang and RabbitMQ Server softwares**
   We need to install both Erlang and Rabbit MQ server to start our Application Setup.

   - Download Erlang from http://www.erlang.org/download.html
   - Download Rabbit MQ from https://www.rabbitmq.com/install-windows.html



2. **Install Erlang Software**
   First install Erlang by double clicking "otp_win64_R16B03-1.exe" then follow default options by clicking "Next" then finally click on "Install" button.
3. **Install Rabbit MQ Server Software**
   First install Rabbit MQ Server by double clicking "rabbitmq-server-3.2.3.exe" then follow default options by clicking "Next" then finally click on "Install" button.

   Once we install Rabbit MQ Server, we can see it's HOME folder as shown below:



   Set the following SYSTEM variable if missing.
   RABBITMQ_HOME=C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.2.3

4. **Configure Rabbit MQ Server**
   By default, Rabbit MQ Server comes with no plug-ins that means we cannot use it as a JMS Provider. We need to perform the following steps:

Open CMD Prompt

CMD>CD to ${RABBITMQ_HOME}/sbin

Install Rabbit MQ Plug-ins by using below command

CMD> rabbitmq-plugins.bat enable rabbitmq_management

Restart Rabbit MQ Server from Windows Control Pannel –> Services

5.

Understanding AMQP

AMQP (Advanced Message Queueing Protocol) is an openly published wire specification for asynchronous messaging. Every byte of transmitted data is specified. This characteristic allows

libraries to be written in many languages, and to run on multiple operating systems and CPU architectures, which makes for a truly interoperable, cross-platform messaging standard.

Advantages of AMQP over JMS
AMQP is often compared to **JMS (Java Message Service),** the most common messaging system in the Java community. **A limitation of JMS is that the APIs are specified, but the message format is not. Unlike AMQP, JMS has no requirement for how messages are formed and transmitted. Essentially, every JMS broker can implement the messages in a different format. They just have to use the same API.**

Thus **Pivotal** has released a **JMS on Rabbit project**, a library that implements the JMS APIs but uses **RabbitMQ**, an **AMQP** broker, to transmit the messages.

AMQP publishes its specifications in a downloadable XML format. This availability makes it easy for library maintainers to generate APIs driven by the specs while also automating construction of algorithms to marshal and demarshal messages.

These advantages and the openness of the spec have inspired the creation of multiple brokers that support AMQP, including:

1. **RabbitMQ**
2. **ActiveMQ**
3. **Qpid**
4. **Solace**

AMQP and JMS terminology

- **JMS** has **queues** and **topics**. A message sent on a JMS queue is consumed by no more than one client. A message sent on a JMS topic may be consumed by multiple consumers. AMQP only has queues. While AMQP queues are only consumed by a single receiver, AMQP producers don't publish directly to queues. A message is published to an exchange, which through its bindings may get sent to one queue or multiple queues, effectively emulating JMS queues and topics.
- JMS and AMQP have an equivalent message header, providing the means to sort and route messages.
- JMS and AMQP both have brokers responsible for receiving, routing, and ultimately dispensing messages to consumers.
- AMQP has **exchanges, routes, and queues.** Messages are first published to exchanges. Routes define on which queue(s) to pipe the message. Consumers subscribing to that queue then receive a copy of the message. If more than one consumer subscribes to the same queue, the messages are dispensed in a round-robin fashion.

Messaging with Spring AMQP

1. Overview

In this article, we will explore Messaging-based communication over AMQP protocol using Spring AMQP framework. First, we'll cover some of the key concepts of messaging, and we'll move on to practical examples in section 5.

1.1. Maven Dependencies

To use spring-amqp and spring-rabbit in your project, just add these dependencies:

```
<dependencies>
   <dependency>
      <groupId>org.springframework.amqp</groupId>
      <artifactId>spring-amqp</artifactId>
      <version>1.6.6.RELEASE</version>
   </dependency>
   <dependency>
      <groupId>org.springframework.amqp</groupId>
      <artifactId>spring-rabbit</artifactId>
      <version>1.6.6.RELEASE</version>
   </dependency>
</dependencies>
```

You will find the newest versions in the Maven repository.

2. Messaging Based Communication

Messaging is a technique for inter-application communication that relies on asynchronous message-passing instead of synchronous request response-based architecture. Producer and consumer of messages are decoupled by an **intermediate messaging layer known as the message broker**. **Message broker provides features like persistent storage of messages, message filtering, message transformation, etc.**

In a case of intercommunication between applications written in Java, JMS (Java Message Service) API is commonly used for sending and receiving messages. For interoperability between different vendors and platforms, we won't be able to use JMS clients and brokers. This is where AMQP comes in handy.

3. AMQP – Advanced Message Queuing Protocol

AMQP is an open standard wire specification for asynchronous messaging based communication. It provides a description on how a message should be constructed. Every byte of transmitted data is specified.

3.1. How AMQP is Different From JMS

Since AMQP is a platform neutral binary protocol standard, this characteristic allows libraries to be written in different programming languages, and to run on different operating systems and CPU architectures.

There is no vendor based protocol lock in, as in the case of migrating from one JMS broker to another. For more details refer to JMS vs AMQP and Understanding AMQP. Some of the widely used AMQP brokers are RabbitMQ, OpenAMQ, and StormMQ.

3.2. AMQP Entities

AMQP entities comprise of Exchanges, Queues, and Bindings:

- **Exchanges** are like post offices or mailboxes and clients always publish a message to an AMQP exchange
- **Queues** bind to exchange using the binding key. A binding is "link" that you set up to bind a queue to an exchange
- Messages are sent to message broker/exchange with a routing key. The exchange then distributes copies of messages to queues. Exchange provides the abstraction to achieve different messaging routing topologies like fanout, hierarchical routing, etc.

3.3. Exchange Types

Exchanges are AMQP entities where messages are sent. Exchanges take a message and route it into zero or more queues. There are four built in exchange types:

- **Direct Exchange**
- **Fanout Exchange**
- **Topic Exchange**
- **Headers Exchange**

4. Spring AMQP

Spring AMQP comprise of two modules: spring-amqp and spring-rabbit, each represented by a jar in the distribution. spring-Spring-amqp is the base abstraction and you will find classes that represent the core AMQP model: Exchange, Queue and Binding.

We will be covering spring-rabbit specific features below in the article.

4.1. spring-amqp Features

- **Template-based abstraction** – **AMQPTemplate interface** defines all basic operations for sending/receiving messages, with **RabbitTemplate as the implementation**
- **Publisher Confirmations/Consumer** Acknowledgements support
- **AmqpAdmin** – tool that allows performing basic operations

4.2. spring-rabbit Features

AMQP model and related entities we discussed above are generic and applicable to all implementations. But there are some features which are specific to the implementation. A couple of those spring-rabbit features is explained below.

Connection Management Support – org.springframework.amqp.rabbit.connection.ConnectionFactory interface is the central component for managing the connections to RabbitMQ broker. The responsibility of CachingConnectionFactory implementation of ConnectionFactory interface is to provide an instance of org.springframework.amqp.rabbit.connection.Connection interface. Please note that spring-rabbit provides Connection interface as a wrapper over RabbitMQ client com.rabbitmq.client.Connection interface.

Asynchronous Message Consumption – **For asynchronous message consumption, two key concepts are a callback and a container.** The callback is where your application code will be integrated with the messaging system.
One of the ways to code a callback is to provide an implementation of the MessageListener interface:

```
public interface MessageListener {
```

```
    void onMessage(Message message);
}
```

In order to have a clear separation between application code and messaging API, Spring AMQP provides MessageListenerAdapter also. Here is a Spring bean configuration example for listener container:

```
MessageListenerAdapter listener = new MessageListenerAdapter(somePojo);
listener.setDefaultListenerMethod("myMethod");
```

Now that we saw the various options for the Message-listening callback, we can turn our attention to the container. Basically, the container handles the "active" responsibilities so that the listener callback can remain passive. The container is an example of a lifecycle component. It provides methods for starting and stopping.

When configuring the container, you are essentially bridging the gap between an AMQP Queue and the MessageListener instance. By default, the listener container will start a single consumer which will receive messages from the queues.

5. Send and Receive Messages Using Spring AMQP

Here are the steps to send and receive a message via Spring AMQP:

1. Setup and Start RabbitMQ broker – RabbitMQ installation and setup is straightforward, just follow the steps mentioned (https://www.rabbitmq.com/download.html)
2. Setup Java Project – Create a Maven based Java project with dependencies mentioned above
3. Implement Message Producer – We can use RabbitTemplate to send a "Hello, world!" message:

   ```
   AbstractApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
   AmqpTemplate template = ctx.getBean(RabbitTemplate.class);
   template.convertAndSend("Hello, world!");
   ```

4. Implement Message Consumer – As discussed earlier, we can implement message consumer as a POJO:

   ```
   public class Consumer {
      public void listen(String foo) {
         System.out.println(foo);
      }
   }
   ```

5. Wiring Dependencies – We will be using following spring bean configuration for setting up queues, exchange, and other entities. Most of the entries are self-explanatory. Queue named "myQueue" is bound to Topic Exchange "myExchange" using "foo.*" as the binding key. RabbitTemplate has been set up to send messages to "myExchange" exchange with "foo.bar" as the default routing key. ListenerContainer ensures an asynchronous delivery of messages from "myQueue" queue to listen() method of Foo class:

   ```
   <rabbit:connection-factory id="connectionFactory" host="localhost" username="guest"
   password="guest" />
   ```

```
<rabbit:template id="amqpTemplate" connection-factory="connectionFactory"
exchange="myExchange" routing-key="foo.bar" />

<rabbit:admin connection-factory="connectionFactory" />

<rabbit:queue name="myQueue" />

<rabbit:topic-exchange name="myExchange">
   <rabbit:bindings>
      <rabbit:binding queue="myQueue" pattern="foo.*" />
   </rabbit:bindings>
</rabbit:topic-exchange>

<rabbit:listener-container connection-factory="connectionFactory">
   <rabbit:listener ref="consumer" method="listen" queue-names="myQueue" />
</rabbit:listener-container>

<bean id="consumer" class="com.baeldung.springamqp.consumer.Consumer" />
```

Note: By default, in order to connect to local RabbitMQ instance, use username "guest" and password "guest".

6. Run the Application:

- The first step is to make sure RabbitMQ is running, default port being 5672
- Run the application by running Producer.java, executing the main() method
- Producer sends the message to the "myExchange" with "foo.bar" as the routing key
- As per the binding key of "myQueue", it receives the message
- Foo class which is a consumer of "myQueue" messages with listen() method as the callback receives the message and prints it on the console

6. Conclusion

In this article, we have covered messaging based architecture over AMQP protocol using Spring AMQP for communication between applications.

The complete source code and all code snippets for this article are available on the GitHub project.**(https://github.com/eugenp/tutorials/tree/master/spring-amqp)**

Difference between JMS and AMQP

- **Definition** :
  JMS : Java Message Service is an API that is part of Java EE for sending messages between two

or more clients. There are many JMS providers such as **OpenMQ** (glassfish's default), **HornetQ(Jboss), and ActiveMQ**.

RabbitMQ: is an open source message broker software which uses the **AMQP** standard and is written by Erlang.

- **Messaging Model:**
  JMS supports two models: **one to one and publish/subscriber. RabbitMQ** supports the **AMQP model which has 4 models : direct, fanout, topic, headers.**

- **Data types:**
  JMS supports 5 different data types but RabbitMQ supports only the **binary data type**.

- **Workflow strategy:**
  In AMQP, producers send to the exchange then the queue, but in JMS, producers send to the queue or topic directly.

- **Technology compatibility:**
  JMS is specific for java users only, but RabbitMQ supports many technologies.

- **Performance**:
  If you would like to know more about their performance, this benchmark is a good place to start, but look for others as well.

Understanding When to use RabbitMQ or Apache Kafka

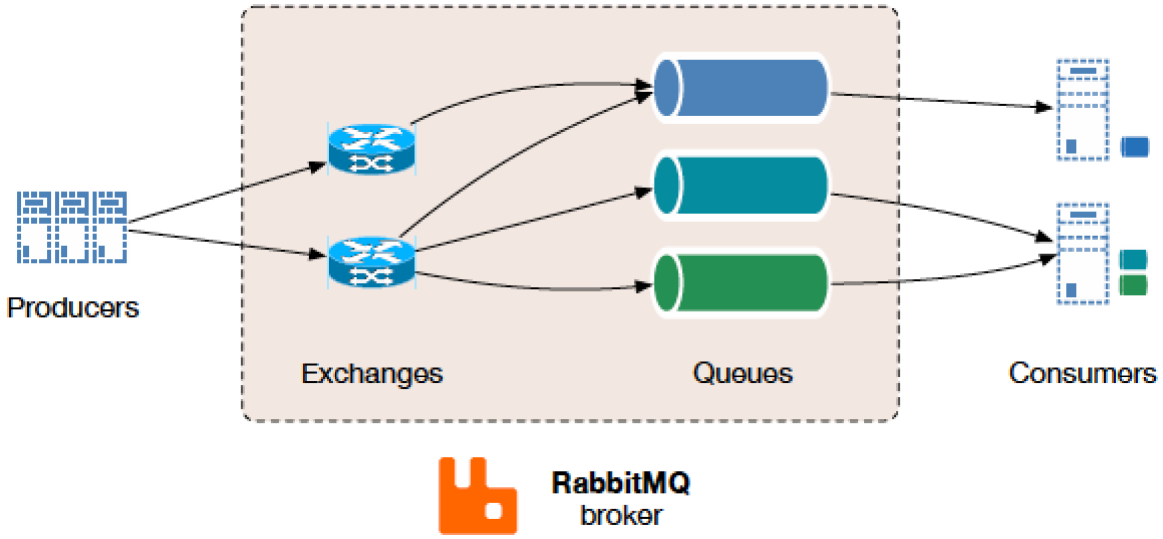
RabbitMQ is a "traditional" message broker that implements variety of messaging protocols. It was one of the first open source message brokers to achieve a reasonable level of features, client libraries,

dev tools, and quality documentation. **RabbitMQ was originally developed to implement AMQP,** an open wire protocol for messaging with powerful routing features. **While Java has messaging standards like JMS, it's not helpful for non-Java applications that need distributed messaging which is severely limiting to any integration scenario, microservice or monolithic**. With the advent of AMQP, cross-language flexibility became real for open source message brokers.

Apache Kafka is developed in **Scala** and started out at **LinkedIn** as a way to connect different internal systems. At the time, **LinkedIn** was moving to a more distributed architecture and needed to reimagine capabilities like data integration and realtime stream processing, breaking away from previously monolithic approaches to these problems. Kafka is well adopted today within the Apache Software Foundation ecosystem of products and is particularly useful in **event-driven architecture.**

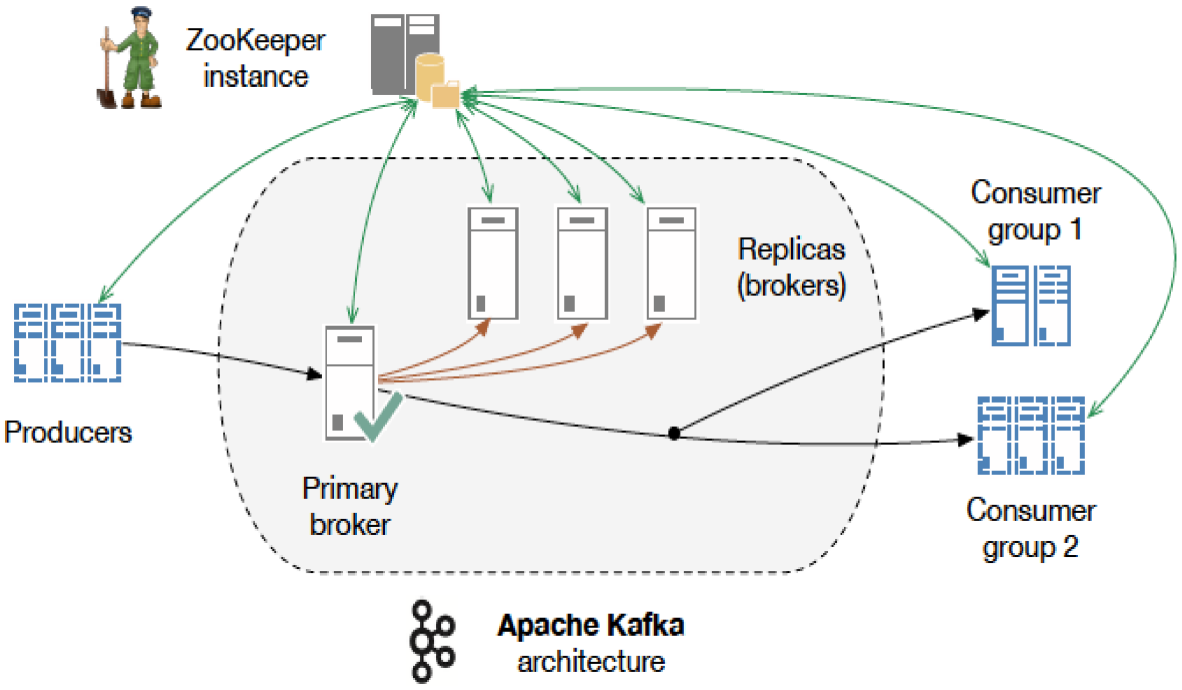
Architecture and Design

RabbitMQ is designed as a general purpose message broker, employing several variations of point to point, request/reply and pub-sub communication styles patterns. It uses a smart broker / dumb consumer model, focused on consistent delivery of messages to consumers that consume at a roughly similar pace as the broker keeps track of consumer state. It is mature, performs well when configured correctly, is well supported (client libraries Java, .NET, node.js, Ruby, PHP and many more languages) and has dozens of plugins available that extend it to more use cases and integration scenarios.



Communication in RabbitMQ can be either synchronous or asynchronous as needed. Publishers send messages to exchanges, and consumers retrieve messages from queues. Decoupling producers from queues via exchanges ensures that producers aren't burdened with hardcoded routing decisions. RabbitMQ also offers a number of distributed deployment scenarios (and does require all nodes be able to resolve hostnames). It can be setup for multi-node clusters to cluster federation and does not have dependencies on external services (but some cluster formation plugins can use AWS APIs, DNS, Consul, etcd).

Apache Kafka is designed for high volume publish-subscribe messages and streams, meant to be durable, fast, and scalable. At its essence, Kafka provides a durable message store, similar to a log, run in a server cluster, that stores streams of records in categories called topics.



Every message consists of a key, a value, and a timestamp. Nearly the opposite of RabbitMQ, Kafka employs a dumb broker and uses smart consumers to read its buffer. Kafka does not attempt to track which messages were read by each consumer and only retain unread messages; rather, Kafka retains all messages for a set amount of time, and consumers are responsible to track their location in each log (consumer state). Consequently, with the right developer talent creating the consumer code, Kafka can support a large number of consumers and retain large amounts of data with very little overhead. As the diagram above shows, Kafka does require external services to run - in this case Apache Zookeeper, which is often regarded as non-trivial to understand, setup and operate.

Requirements and Use Cases

Many developers begin exploring messaging when they realize they have to connect lots of things together, and other integration patterns such as shared databases are not feasible or too dangerous.

Apache Kafka includes the broker itself, which is actually the best known and the most popular part of it, and has been designed and prominently marketed towards stream processing scenarios. In addition to that, Apache Kafka has recently added Kafka Streams which positions itself as an alternative to streaming platforms such as Apache Spark, Apache Flink, Apache Beam/Google Cloud Data Flow and Spring Cloud Data Flow. The documentation does a good job of discussing popular use cases like Website Activity Tracking, Metrics, Log Aggregation, Stream Processing, Event Sourcing and Commit logs. One of those use cases it describes is messaging, which can generate some confusion. So let's unpack that a bit and get some clarity on which messaging scenarios are best for Kafka for, like:

- Stream from A to B without complex routing, with maximal throughput (100k/sec+), delivered in partitioned order at least once.
- When your application needs access to stream history, delivered in partitioned order at least

once. Kafka is a durable message store and clients can get a "replay" of the event stream on demand, as opposed to more traditional message brokers where once a message has been delivered, it is removed from the queue.
- Stream Processing
- Event Sourcing

RabbitMQ is a general purpose messaging solution, often used to allow web servers to respond to requests quickly instead of being forced to perform resource-heavy procedures while the user waits for the result. It's also good for distributing a message to multiple recipients for consumption or for balancing loads between workers under high load (20k+/sec). When your requirements extend beyond throughput, RabbitMQ has a lot to offer: features for reliable delivery, routing, federation, HA, security, management tools and other features. Let's examine some scenarios best for RabbitMQ, like:

- Your application needs to work with any combination of existing protocols like AMQP 0-9-1, STOMP, MQTT, AMQP 1.0.
- You need a finer-grained consistency control/guarantees on a per-message basis (dead letter queues, etc.) However, Kafka has recently added better support for transactions.
- Your application needs variety in point to point, request / reply, and publish/subscribe messaging
- Complex routing to consumers, integrate multiple services/apps with non-trivial routing logic

RabbitMQ can also effectively address several of Kafka's strong uses cases above, but with the help of additional software. RabbitMQ is often used with Apache Cassandra when application needs access to stream history, or with the LevelDB plugin for applications that need an "infinite" queue, but neither feature ships with RabbitMQ itself.

For a deeper dive on microservice - specific use cases with Kafka and RabbitMQ, head over to the Pivotal blog and read this short post by Fred Melo.

Developer Experience
RabbitMQ officially supports Java, Spring, .NET, PHP, Python, Ruby, JavaScript, Go, Elixir, Objective-C, Swift - with many other clients and devtools via community plugins. The RabbitMQ client libraries are mature and well documented.

Apache Kafka has made strides in this area, and while it only ships a Java client, there is a growing catalog of community open source clients, ecosystem projects, and well as an adapter SDK allowing you to build your own system integration. Much of the configuration is done via .properties files or programmatically.

The popularity of these two options has a strong influence on many other software providers who make sure that RabbitMQ and Kafka work well with or on their technology.

Security and Operations
Both are strengths of RabbitMQ. RabbitMQ management plugin provides an HTTP API, a browser-based UI for management and monitoring, plus CLI tools for operators. External tools like CollectD,

Datadog, or New Relic are required for longer term monitoring data storage. RabbitMQ also provides API and tools for monitoring, audit and application troubleshooting. Besides support for TLS, RabbitMQ ships with RBAC backed by a built-in data store, LDAP or external HTTPS-based providers and supports authentication using x509 certificate instead of username/password pairs. Additional authentication methods can be fairly straightforwardly developed with plugins.

These domains pose a challenge for Apache Kafka. On the security front, the recent Kafka 0.9 release added TLS, JAAS role based access control and kerberos/plain/scram auth, using a CLI to manage security policy. This made a substantial improvement on earlier versions where you could only lock down access at the network level, which didn't work well for sharing or multi-tenancy.

Kafka uses a management CLI comprised of shell scripts, property files and specifically formatted JSON files. Kafka Brokers, Producers and Consumers emit metrics via Yammer/JMX but do not maintain any history, which pragmatically means using a 3rd party monitoring system. Using these tools, operations is able manage partitions and topics, check consumer offset position, and use the HA and FT capabilities that Apache Zookeeper provides for Kafka. While many view the requirement for Zookeeper with a high degree of skepticism, it does confer clustering benefits for Kafka users.

For example, a 3-node Kafka cluster the system is functional even after 2 failures. However if you want to support as many failures in Zookeeper you need an additional 5 Zookeeper nodes as Zookeeper is a quorum based system and can only tolerate N/2+1 failures. These obviously should not be co-located with the Kafka nodes - so to stand up a 3 node Kafka system you need ~ 8 servers. Operators must take the properties of the ZK cluster into account when reasoning about the availability of any Kafka system, both in terms of resource consumption and design.

Performance

Kafka shines here by design: 100k/sec performance is often a key driver for people choosing Apache Kafka.

Of course, message per second rates are tricky to state and quantify since they depend on so much including your environment and hardware, the nature of your workload, which delivery guarantees are used (e.g. persistent is costly, mirroring even more so), etc.

20K messages per second is easy to push through a single Rabbit queue, indeed rather more than that isn't hard, with not much demanded in the way of guarantees. The queue is backed by a single Erlang lightweight thread that gets cooperatively scheduled on a pool of native OS threads - so it becomes a natural choke point or bottleneck as a single queue is never going to do more work than it can get CPU cycles to work in.

Increasing the messages per second often comes down to properly exploiting the parallelism available in one's environment by doing such things as breaking traffic across multiple queues via clever routing (so that different queues can be running concurrently). When RabbitMQ achieved 1 million message per second , this use case basically came down entirely to doing that judiciously - but was achieved using lot of resources, around 30 RabbitMQ nodes. Most RabbitMQ users enjoy excellent performance with clusters made up of anywhere from three to seven RabbitMQ nodes.

Making the call

Absorb some research on a few of the other top options on the market. If you want to go deeper with the most popular options, a master's thesis from Nicolas Nannoni inspired this article and it features a

side-by-side comparison table in Section 4.4 (page 39) but is a bit dated at this point. If you feel like plunking down $15.00 USD, this ACM report is also excellent and more recent.

While researching, loop back with the stakeholders and the business as often as possible. Understanding the business use case is the single largest factor in making the right choice for your situation. Then, if you are pop psychology fan, your best bet is sleep on it, let it percolate, and let your instincts take over. You got this.