Message-oriented Middleware (MOM) Providers

ActiveMQ

ActiveMQ is an Apache open-source implementation of the Java Message Service (JMS) specification v1.1. It provides high availability, performance, scalability, reliability, and security for enterprise messaging. Although ActiveMQ is written in Java, APIs for many languages other than Java are provided, including C/C++, .NET, Perl, PHP, Python, Ruby, and many more. The current release version of ActiveMQ is v5.14.5 and is licensed under the Apache Software License v2.0.

The goal of ActiveMQ is to provide standards-based, message-oriented application integration across as many languages and platforms as possible. ActiveMQ implements the JMS specification and offers additional features such as:

- OpenWire, a cross language wire protocol, for high performance clients in Java, C, C++, C#
- Simple Text Orientated Messaging Protocol (STOMP) v1.1 provides an interoperable wire format for communication between clients, written in C, Ruby, Perl, Python, PHP, and messaging brokers.
- Support of Advanced Message Queuing Protocol (AMQP) v1.0 an OASIS standard
- Support of MQ Telemetry Transport (MQTT) v3.1 – a machine-to-machine (M2M)/"Internet of Things" connectivity protocol.

ActiveMQ Artemis

Artemis is an ActiveMQ sub-project that is a complete rewrite on ActiveMQ v5 that incorporates the HornetQ code base and provides support of JMS specification v2.0. The HornetQ code base was donated by Red Hat to the Apache ActiveMQ community in 2014. It is possible that Artemis will eventually become the successor to ActiveMQ (possibly branded as ActiveMQ v6.x). The current release version of ActiveMQ Artemis is v2.1.0 and is licensed under the Apache Software License v2.0.

Amazon SNS

Amazon Simple Notification Service (SNS) is a fast, flexible, fully managed push notification service that lets you send individual messages or to fan-out messages to large numbers of recipients. Amazon SNS makes it simple and cost effective to send push notifications to mobile device users, email recipients or even send messages to other distributed services. Amazon SNS has the following features:

- Reliable - Amazon SNS runs within Amazon's proven network infrastructure and datacenters, so topics will be available whenever applications need them. To prevent messages from being lost, all messages published to Amazon SNS are stored redundantly across multiple servers and data centers.
- Scalable - Amazon SNS is designed to meet the needs of the largest and most demanding applications, allowing applications to publish an unlimited number of messages at any time.
- Simple - In most cases, developers can get started with Amazon SNS by using just three APIs: CreateTopic, Subscribe, and Publish. Additional APIs are available, which provide more advanced functionality.
- Flexible - Amazon SNS allows applications and end-users on different devices to receive notifications via Mobile Push notification (Apple, Google and Kindle Fire Devices), HTTP/HTTPS,

Email/Email-JSON, SMS or Amazon Simple Queue Service (SQS) queues, or AWS Lambda functions.

- Secure - Amazon SNS provides access control mechanisms to ensure that topics and messages are secured against unauthorized access. Topic owners can set policies for a topic that restrict who can publish or subscribe to a topic. Additionally, topic owners can ensure that notifications are encrypted by specifying that the delivery mechanism must be HTTPS.

Amazon SNS works with other AWS services such as Amazon SQS and EC2. For example, applications running in EC2 can publish event/information updates to Amazon SNS and have them immediately delivered to other applications or end-users. Additionally, subscribers can select Amazon SQS as a delivery protocol, and have notifications delivered to multiple SQS queues in parallel - providing persistence of messages and guaranteed delivery. Furthermore, messages can now be delivered to AWS Lambda functions for handling message customizations, enabling message persistence or communicating with other AWS services.

Amazon SQS

Amazon Simple Queue Service (Amazon SQS) is a web service that gives you access to message queues that store messages waiting to be processed. With Amazon SQS, you can quickly build message queuing applications that can run on any computer.

Amazon SQS offers a reliable, secure, and highly-scalable hosted queuing service for storing messages in transit between computers. With Amazon SQS, you can move data between diverse, distributed application components without losing messages and without requiring each component to be always available. You can exchange sensitive data between applications using Amazon SQS server-side encryption (SSE) integrated with the AWS Key Management Service (KMS).

Amazon SQS offers two queue types for different application requirements:

- Standard Queues - A standard queue, default queue type, lets you have a nearly-unlimited number of transactions per second. Standard queues guarantee that a message is delivered at least once. However, occasionally (because of the highly-distributed architecture that allows high throughput), more than one copy of a message might be delivered out of order. Standard queues provide best-effort ordering which ensures that messages are generally delivered in the same order as they are sent.
- FIFO Queues - Amazon recently added support of a FIFO queue that complements the standard queue. The most important features of this queue type are FIFO (first-in-first-out) delivery and exactly-once processing: the order in which messages are sent and received is strictly preserved and a message is delivered once and remains available until a consumer processes and deletes it; duplicates are not introduced into the queue. FIFO queues also support message groups that allow multiple ordered streams within a single queue. FIFO queues are limited to 300 transactions per second (TPS), but have all the capabilities of standard queues.

Amazon SQS employs a simple interface that is easy to use and highly flexible. The following requests are provided:

- Basic Message Operations
    - SendMessage - Send messages to a specified queue.

    - ReceiveMessage - Return one or more messages from a specified queue.
    - DeleteMessage - Remove a previously received message from a specified queue.
    - ChangeMessageVisibility - Change the visibility timeout of a previously received message.
- Batch Message Operations
    - SendMessageBatch - Send multiple messages to a specified queue.
    - DeleteMessageBatch - Remove multiple previously received messages from a specified queue.
    - ChangeMessageVisibilityBatch - Change the visibility timeout of multiple previously received messages.
- Basic Queue Management
    - CreateQueue - Create queues for use with your AWS account.
    - ListQueues - List your existing queues.
    - DeleteQueue - Delete one of your queues.
    - PurgeQueue - Delete all the messages in a queue.
- Advanced Queue Management
    - SetQueueAttributes - Control queue settings such as the visibility timeout (amount of time that messages are locked after being read so they cannot be read again), a delay value, or dead letter queue parameters.
    - GetQueueAttributes - Get information about a queue such as the visibility timeout, number of messages in the queue, or the maximum message size.
    - GetQueueUrl - Get the queue URL.
    - AddPermission - Add queue sharing for another AWS account for a specified queue.
    - RemovePermission - Remove an AWS account from queue sharing for a specified queue.
    - ListDeadLetterSourceQueues - List the queues attached to a dead letter queue.

Messages that are stored in Amazon SQS have a lifecycle that is easy to manage but ensures that all messages are processed.

1. A system that needs to send a message will select an Amazon SQS queue, and use SendMessage to send a new message to it.
2. A different system that processes messages needs more messages to process, so it calls ReceiveMessage, and this message is returned.
3. Once a message has been returned by ReceiveMessage, it will not be returned by any other ReceiveMessage until the visibility timeout has passed. This keeps multiple consumers from processing the same message at once.
4. If the system that processes messages successfully finishes working with this message, it calls DeleteMessage, which removes the message from the queue so no one else will ever process it. If this system fails to process the message, then it will be read by another ReceiveMessage call as soon as the visibility timeout passes.
5. If you have associated a Dead Letter Queue with a source queue, messages will be moved to the Dead Letter Queue after the maximum number of delivery attempts you specify have been reached.

Kafka

Kafka is an Apache open source, distributed, partitioned, and replicated commit-log-based publish-subscribe messaging system written in Java and Scala. The current release version of Kafka is v0.10.2.1 and is licensed under the Apache Software License v2.0.

Kafka is mainly designed with the following characteristics:

- Persistent messaging - To derive the real value from big data, any kind of information loss cannot be afforded. Apache Kafka is designed with O(1) disk structures that provide constant-time performance even with very large volumes of stored messages that are in the order of TBs. With Kafka, messages are persisted on disk as well as replicated within the cluster to prevent data loss.
- High throughput - Keeping big data in mind, Kafka is designed to work on commodity hardware and to handle hundreds of MBs of reads and writes per second from large number of clients.
- Distributed - Apache Kafka with its cluster-centric design explicitly supports message partitioning over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics. Kafka cluster can grow elastically and transparently without any downtime.
- Multiple client support - The Apache Kafka system supports easy integration of clients from different platforms such as Java, .NET, PHP, Ruby, and Python.
- Real time - Messages produced by the producer threads should be immediately visible to consumer threads; this feature is critical to event-based systems such as Complex Event Processing (CEP) systems.

Kafka provides a real-time, high-throughput, low-latency publish-subscribe solution that overcomes the challenges of consuming the real-time and batch data volumes that may grow in order of magnitude to be larger than the real data.

The Spring Kafka project applies core Spring concepts to the development of Kafka-based messaging solutions. It provides a "template" as a high-level abstraction for sending messages. It also provides support for Message-driven POJOs with @KafkaListener annotations and a "listener container". These libraries promote the use of dependency injection and declarative.

The main features of Spring Kafka are:

- KafkaTemplate - A high-level abstraction for sending Kafka messages.
- KafkaMessageListenerContainer – Receives Kafka messages and delegates to targeted listeners for message processing.
- @KafkaListener - Annotation that marks a method to be the target of a Kafka message listener on the specified topics.

RabbitMQ

RabbitMQ is an open source messaging broker written in Erlang that implements Advanced Message Queuing Protocol (AMQP) v0.9.1. The current release of RabbitMQ is v3.6.9 and is licensed under the Mozilla Public License (MPL).

RabbitMQ features include:

- Asynchronous messaging - Supports multiple messaging protocols, message queuing, delivery acknowledgement, flexible routing to queues, multiple exchange types.
- Multiple client support – RabbitMQ supports integration of clients written in Java, C/C++, Erlang, .Net, PHP, Python, JavaScript, and Ruby.
- Distributed deployment - Deploy as load balanced clusters for high availability and throughput; federate across multiple availability zones and regions.
- Enterprise & cloud ready - Pluggable authentication, authorization, supports TLS and LDAP. Lightweight and easy to deploy in public and private clouds.
- Tools & plugins - Diverse array of tools and plugins supporting continuous integration, operational metrics, and integration to other enterprise systems. Flexible plug-in approach for extending RabbitMQ functionality.
- Management & monitoring - HTTP-API, command line tool, and UI for managing and monitoring RabbitMQ.

Pivotal Software develops and maintains RabbitMQ and offers a commercial distribution called Pivotal RabbitMQ, as well as a version that integrates natively with Pivotal Cloud Foundry. These distributions include all of the features of the open source version, with RabbitMQ for Pivotal Cloud Foundry providing some additional management features. Support agreements for such distributions would fall under commercial licensing of Pivotal Software.

The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions. It provides a "template" as a high-level abstraction for sending and receiving messages. It also provides support for Message-driven POJOs with a "listener container". These libraries facilitate management of AMQP resources while promoting the use of dependency injection and declarative configuration.

The main features of Spring AMQP are:

RabbitTemplate – Provides support for sending and synchronous reception of AMQP messages.

SimpleMessageListenerContainer – A listener container for asynchronous processing of inbound AMQP messages.

@RabbitListener - Annotation that marks a method to be the target of a Rabbit AMQP message.

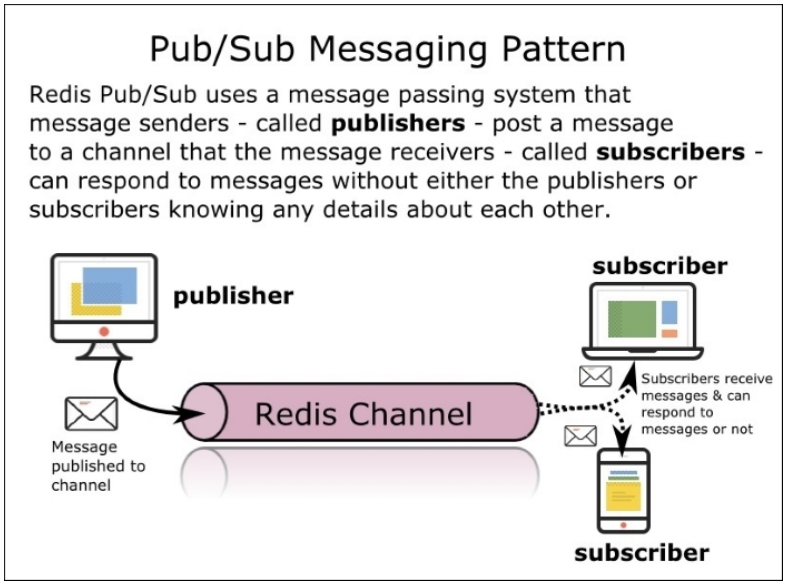
RabbitAdmin – Provides administrative operations for such things as declaring queues, exchanges and bindings.

Redis

Redis Publisher/Subscriber (Pub/Sub) is a messaging model that is fast and stable. Instead of processes sending messages directly to each other, a publisher or sender submits messages to one or more channels and the receivers or subscribers that have subscribed to a channel receive all messages posted to the specific channels. If you design your application being mindful of race conditions and the possibility of delivery failure, Pub/Sub offers fast messaging solutions.

Conceptually, Redis Pub/Sub is similar to Really Simple Syndication (RSS), or the atom formats used by websites to publish feeds for consumption by clients or readers. In either case, neither the website publisher nor the consuming client are directly sending messages to each other. The client connects and

consumes the content from feeds from those websites' publications. Like RSS, Redis Pub/Sub, and other publish/subscribe systems, the advantage of this messaging pattern is better scalability for the systems for more dynamic networks.



Spring Data provides dedicated messaging integration for Redis, very similar in functionality and naming to the JMS integration in Spring Framework; in fact, users familiar with the JMS support in Spring should feel right at home.

Redis messaging can be roughly divided into two areas of functionality, namely the production or publication and consumption or subscription of messages. The `RedisTemplate` class is used for message production. For asynchronous reception similar to Java EE's Message-driven Bean (MDB) style, Spring Data provides a dedicated message listener container that is used to create Message-Driven POJOs (MDPs) and for synchronous reception, the `RedisConnection` contract.

To publish a message, one can use, as with the other operations, either the low-level `RedisConnection` or the high-level `RedisTemplate`. Both entities offer the publish method that accepts as an argument the message that needs to be sent as well as the destination channel. While `RedisConnection` requires raw-data (array of bytes), the `RedisTemplate` allow arbitrary objects to be passed in as messages.

The `RedisTemplate` performs automatic serialization/deserialization between the given objects and the underlying binary data in the Redis store. By default, it uses Java serialization, `JdkSerializationRedisSerializer`, for its POJOs. For String intensive operations the dedicated `StringRedisTemplate` should be used. In a Spring Boot application, an instance of both the `RedisTemplate` and the `StringRedisTemplate` are configured and available in the application context.

In any messaging-based application, there are message publishers and messaging receivers. To create the message receiver, implement a receiver with a method to respond to a given message. In this example, we create a Person POJO to be the targeted message:

```
@AllArgsConstructor
```

```
@Getter
@ToString
public class Person implements Serializable {
  private int id;
  private String firstName;
  private String lastName;
}

@Component
class Receiver {
  public void receive(Person person) {
    // process person message
  }
}
```

The `Receiver` is a simple POJO that defines a method for receiving messages. We will need to configure a `MessageListener` and a `RedisMessageListenerContainer` to subscribe to publications of a `Person` message:

```
@Configuration
class RedisConfiguration {

  @Bean
  MessageListener messageListener(Receiver receiver) {
    MessageListenerAdapter messageListener = new
MessageListenerAdapter(receiver, "receive");
    messageListener.setSerializer(new JdkSerializationRedisSerializer());
    return messageListener;
  }

  @Bean
  RedisMessageListenerContainer container(RedisConnectionFactory
connectionFactory, MessageListener messageListener) {
    RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.addMessageListener(messageListener, new
PatternTopic("persons"));
    return container;
  }
}
```

This example uses Spring Boot's default `RedisConnectionFactory`, an instance of `JedisConnectionFactory` which is based on the Jedis Redis library. The connection factory will be injected into both the message listener container and the Redis template. The message listener is configured to use Java serialization since this is default setting of the underlying `RedisTemplate`. In the case of the listener container, the example adds the message listener to the 'persons' channel.

We use the Spring Boot's default `RedisTemplate` to publish a Person POJO:

```
Person person = new Person(123, "Jeff", "Edwards";
redisTemplate.convertAndSend("persons", person);
```

The first argument to `convertAndSend` operation is the targeted channel name to publish the `Person` POJO message.

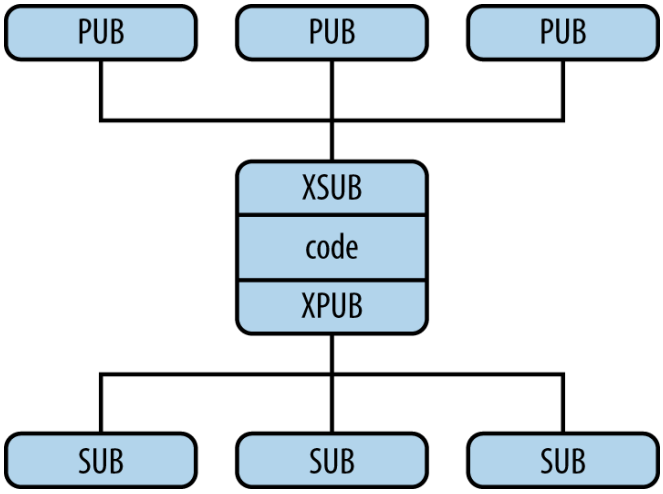
ZeroMQ

ZeroMQ (ØMQ) is an open source project focused on a decentralized messaging system. It is different from many other messaging systems in that it does not have the concept of a centralized broker. ØMQ comes as an embeddable library that allows elastic scaling of an application. The current release of ØMQ is v4.2.1 and is licensed under GNU Lesser General Public License V3.

ØMQ features include:

- Has bindings for languages such as Java, C/C++, .Net, PHP, Python, JavaScript, and Erlang
- Handles I/O asynchronously, in background threads. These communicate with application threads using lock-free data structures, so concurrent applications need no locks, semaphores, or other wait states.
- Components can come and go dynamically, and ØMQ will automatically reconnect. This means you can start components in any order. You can create Service-oriented Architectures (SOAs) where services can join and leave the network at any time.
- It queues messages automatically when needed. It does this intelligently, pushing messages as close as possible to the receiver before queuing them.
- It has ways of dealing with over-full queues (called the "high-water mark"). When a queue is full, ØMQ automatically blocks senders, or throws away messages, depending on the type of messaging.
- It lets your applications talk to each other over arbitrary transports: TCP, multicast, in-process, and inter-process. You don't need to change your code to use a different transport.
- It handles slow/blocked readers safely, using different strategies that depend on the messaging pattern.
- It lets you route messages using a variety of patterns, such as request-reply and publish-subscribe. These patterns are how you create the topology, the structure of your network.
- It lets you create proxies to queue, forward, or capture messages with a single call. Proxies can reduce the interconnection complexity of a network.
- It delivers whole messages exactly as they were sent, using a simple framing on the wire.
- It does not impose any format on messages. They are blobs of zero bytes to gigabytes large.
- It handles network errors intelligently - sometimes it retries, sometimes it tells you an operation failed.

A problem with the design of a larger distributed architecture is the dynamic discovery of messaging endpoints. There are several solutions to this problem, but the simplest is to add an intermediary; a static point in the network to which all other nodes connect. These intermediaries can be thought of as a simple stateless messaging switch. In ØMQ, an intermediary will open XSUB and XPUB sockets, binding each to a well-known IP address and port. These socket connections allow ØMQ to expose subscription information between messaging endpoints. The following illustrates such a use case:



Messaging with Spring Cloud Stream

If we look at software development and business needs as a developer or an architect, we are always looking at how to integrate internal and external components and systems into our architecture. We need components that are fully functional, highly available, and easy to maintain and enhance. Furthermore, we want to be able to opportunistically evolve and recompose our systems as business needs change.

The problem is that interleaving details of inter-component communication with business logic makes it difficult to leverage existing components and services in new ways. Business processes and components are often reusable in other contexts and are independent of:

- Wire protocol
- Serialization method
- Fault-tolerance aspects

We need an extensible solution where the integration architecture can be leveraged by other components. The use of Spring Cloud Stream provides a framework scaffold for design of message-driven applications that addresses these concerns. Spring Cloud Stream is founded on architectural principles that decouples messaging between producers and consumers, by creating bindings that can be used out of the box. You don't need to add broker-specific code in your application for producing or consuming messages. You just add the required binding dependencies to your application and Spring Cloud Stream will take care of messaging connectivity and communication.

The main components of Spring Cloud Stream are:

- Application - The application model is just a middleware-neutral core, which means that the application will communicate using input and output channels to external brokers (as a way of transporting messages) through binder specific implementations.
- Binder abstraction - Spring Cloud Stream provides at this moment the Kafka and RabbitMQ binder implementations. This abstraction makes it possible for Spring Cloud Stream apps to connect to the middleware. Spring Cloud Stream can dynamically choose at runtime the destinations based on channels.

- Persistent publish/subscribe - The application communication will be through the well-known publish/subscribe model. If Kafka were used, it would follow its own topic/subscriber model, and if RabbitMQ were used, it would create a topic exchange and the necessary bindings for each queue. This model reduces any complexity of producer and consumer.
- Consumer groups – Message consumers will need to be able to scale up at some point. Scalability can be done using the concept of a consumer group (this is similar to the Kafka consumer groups feature), where you can have multiple consumers in a group for a load-balancing scenario. This makes horizontal scaling needs very easy to set up.
- Partitioning support - Spring Cloud Stream support data partition, which allows multiple producers to send data to multiple consumers and ensure that common data is processed by the same consumer instances. This is a benefit for performance and consistency of data.
- Binder API - Spring Cloud Stream provides an API interface. It's actually a binder Service Provider Interface (SPI) where you can extend the core by modifying the original code, so it's easy to implement a specific binder, such as JMS, WebSockets, etc.