

Spring Integration is based on the concepts explained in the Enterprise Integration Patterns book. It is a lightweight messaging solution that will add integration capabilities to your Spring application. As a messaging strategy, it provides a way of sharing information quickly and with a high level of decoupling between involved components or applications. You will learn how to accomplish this while Spring handles any low-level infrastructure concern. This will allow you to focus on your business logic. One of the foundational patterns in this library is Messaging. The pattern is centred around messages – discrete payloads of data that move from an originating system or process to one or multiple systems or processes via predefined channels.

* Almost completely decouples the systems involved in the integration.
* Allows participant systems in the integration to be completely agnostic of each other’s underlying protocols, formatting, or other implementation details.
* Encourages the development and reuse of components involved in the integration.
* Loose coupling among components.
* Event oriented architecture.
* The integration logic (handled by the framework) is separated from the business logic.

**Goals and Principles**

Spring Integration is motivated by the following goals:

* Provide a simple model for implementing complex enterprise integration solutions.
* Facilitate asynchronous, message-driven behaviour within a Spring-based application.
* Promote intuitive, incremental adoption for existing Spring users.

Spring Integration is guided by the following principles:

* Components should be loosely coupled for modularity and testability.
* The framework should enforce separation of concerns between business logic and integration logic.
* Extension points should be abstract in nature (but within well-defined boundaries) to promote reuse and portability.

**Spring Integration's event-driven architecture:**

Event-driven architecture is one of the most powerful and successful patterns used for enterprise integration, and is the main focus of examples in this article. In an event-driven architecture, a system publishes events as they happen. Components within a given system listen for specific events, or types of events, occurring within that system. When an event of interest occurs, the components are alerted and can respond as necessary.

Event-driven architecture affords a high degree of loose coupling and enhances system scalability because message producers don't need to know anything about their consumers. This makes integrating a new component with an existing or legacy system relatively easy: existing systems publish events and new components are configured to listen for those events. Because all interactions in an event-driven architecture are asynchronous, components can process messages on their own time. If load increases substantially, a component may take longer to process a message, but it will eventually happen.

While an application may slow down, it should never go down.

Spring Integration's support for event-driven architecture rests on three core components:

**Messages** are objects sent from one component to another.

**Channels** are the means by which messages are sent, they can be synchronous or asynchronous.

**Adapters** route the output from one channel to the input of another one.

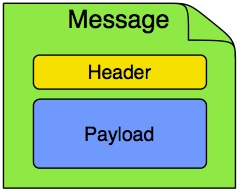
Spring Integration message channels store messages in memory by default. This is because memory is fast, easy to implement and it does not create extra network cost. However, in some cases, this can cause problem because all the messages will be lost if the application crashes or the server shuts down accidentally. For such situations, Spring Integration introduces JMS & AMQP backed message channels so the messages are stored within a JMS & AMQP broker instead of in memory.

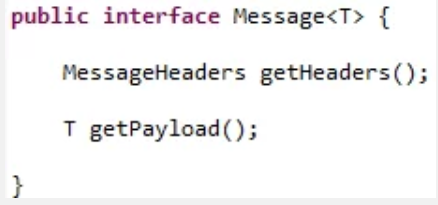
Advanced Message Queuing Protocol (AMQP) is an open standard for messaging protocol. It allows applications to communicate asynchronously, reliably, and securely. RabbitMQ is an open source message broker that supports the AMQP standard. One of the most important features of RabbitMQ is highly available queues.

In this article, Spring Integration’ s AMQP backed point-to-point message channel approach is explained by creating two messaging nodes and a RabbitMQ cluster covering two RabbitMQ Servers. Two messaging nodes start to process Order messages by using the RabbitMQ cluster. If First Messaging Node and First RabbitMQ Server are shut down accidentally, Second Messaging Node and Second RabbitMQ Server will continue to process Order messages so potential message loosing and service interruption problems can be prevented by using high available AMQP backed channel.

**Message:**

A message contains the information that will be shared among the different components of the application, or sent to an external system.





**Header:** It consists of a payload and headers. The payload can be of any type, and the headers hold commonly required information such as ID, timestamp, correlation ID, and return address. Headers are also used for passing values to and from connected transports. For example, when creating a message from a received file, the file name may be stored in a header to be accessed by downstream components. Likewise, if a message’s content is ultimately going to be sent by an outbound mail adapter, the various properties (to, from, cc, subject, and others) may be configured as message header values by an upstream component. Developers can also store any arbitrary key-value pairs in the headers. Contains meta-information about the message. If you check MessageHeaders class, you will see that it’s just a wrapper of a Map, but with its insertion operations marked as unsupported. The framework marks them like this because a message is considered to be immutable. Once the message has been created, you cannot modify it. You can add your own headers in the form of key-value pair, but they are mainly used to pass transport information. For example, if you want to send an e-mail, it will contain headers like to, subject, from…

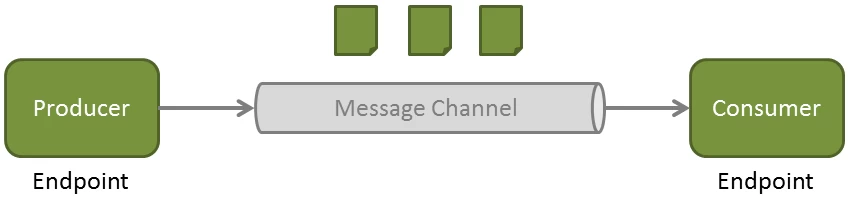
**Payload:** This is just a normal Java class that will contain the information you want to share. It can be any Java type.

**Message Endpoints:**

* Adapters (connect your channel to some other system)
* Filter (remove some messages from channels based on header, content, etc.)
* Transformer (convert a message content or structure)
* Enricher (add content to the message header or payload)
* Service activator (invoke service operations based on the arrival of a message)
* Gateway (connect your channels without SI coupling)
* A bridge in Spring Integration is used to connect two message channels or adapters if for any reason they can’t connect directly.

**Message Channel:**

A message channel may follow either point-to-point or publish-subscribe semantics. With a point-to-point channel, no more than one consumer can receive each message sent to the channel. Publish-subscribe channels, on the other hand, attempt to broadcast each message to all subscribers on the channel. Spring Integration supports both of these models.



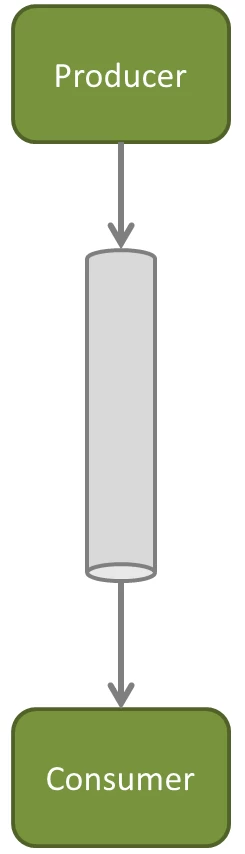
* Two general classifications of message channels
* Pollable Channel
* Subscribable Channel
* While there are many subtypes, they all implement at least one of these SI channel interfaces

see <http://docs.spring.io/spring-integration/reference/html/messaging-channels-section.html>

#### **Point-to-point or Pollable Channels:**

There’s only one receiver connected to the message channel. Well, that’s not strictly 100% true. If it is a subscribable channel you can have more than one receiver but only one will handle the message. For now, forget this since this is an advanced topic that will be seen later in this course (dispatcher configuration). This type of channel has several implementations.

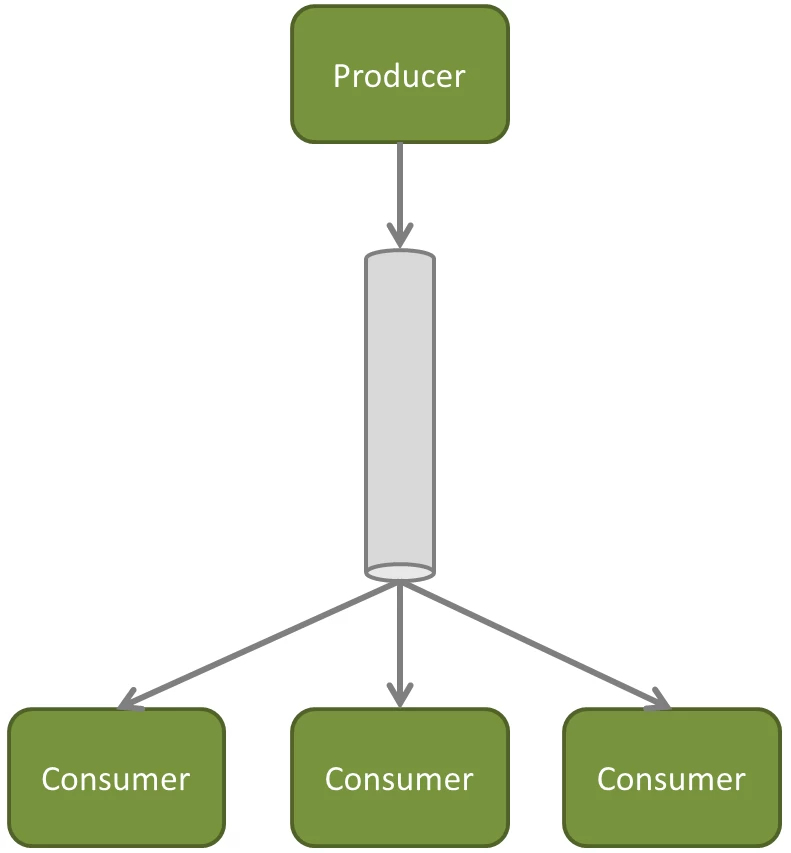
* May buffer its messages
  + Requires a queue to hold the messages
  + The queue has a designated capacity
* Waits for the consumer to get the messages
  + Consumers actively poll to receive messages
* Typically, a point-to-point channel
  + Only one receiver of a message in the channel
* Usually used for sending information or “document” messages between endpoints.



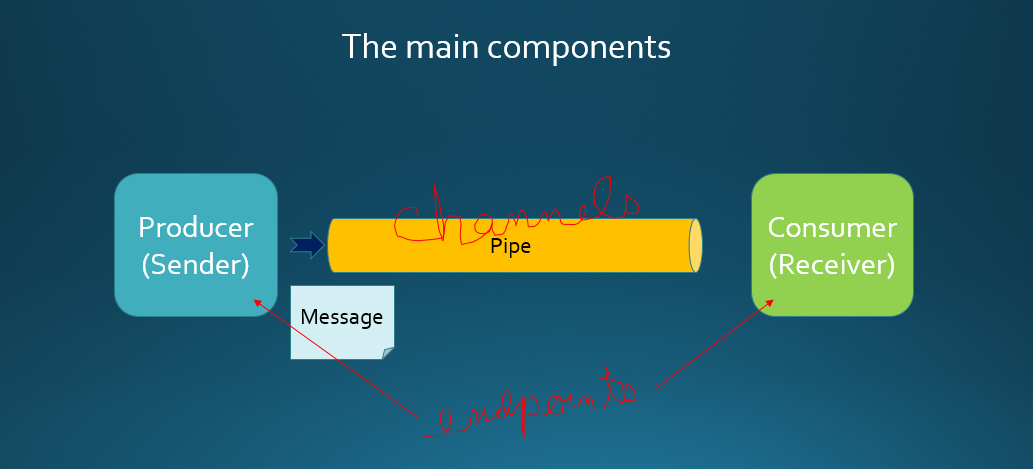
**DirectChannel:** Implements SubscribableChannel. The message is sent to the subscriber through the same receiver’s thread. This communication is synchronous and the producer block until a response is received. How it works:

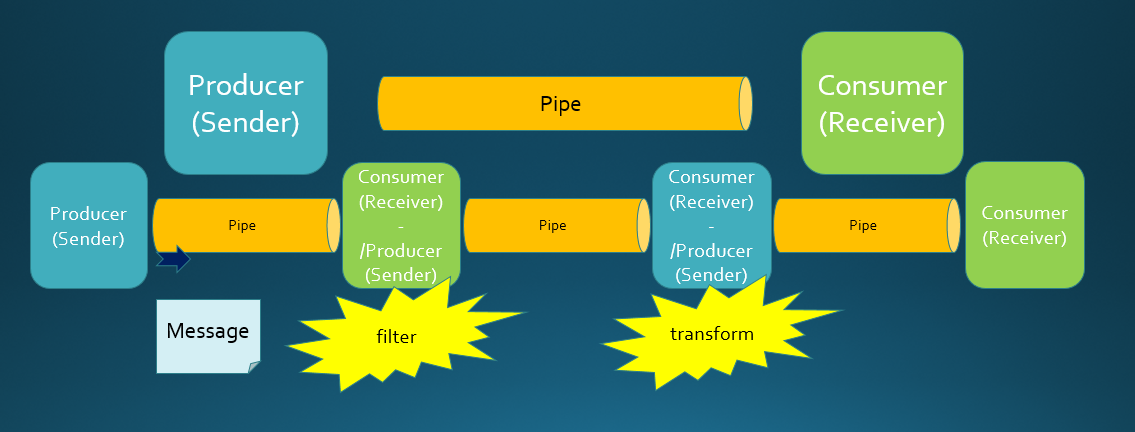
* The producer sends the message to the channel.
* The channel sends the message to its subscriber (passive subscriber).

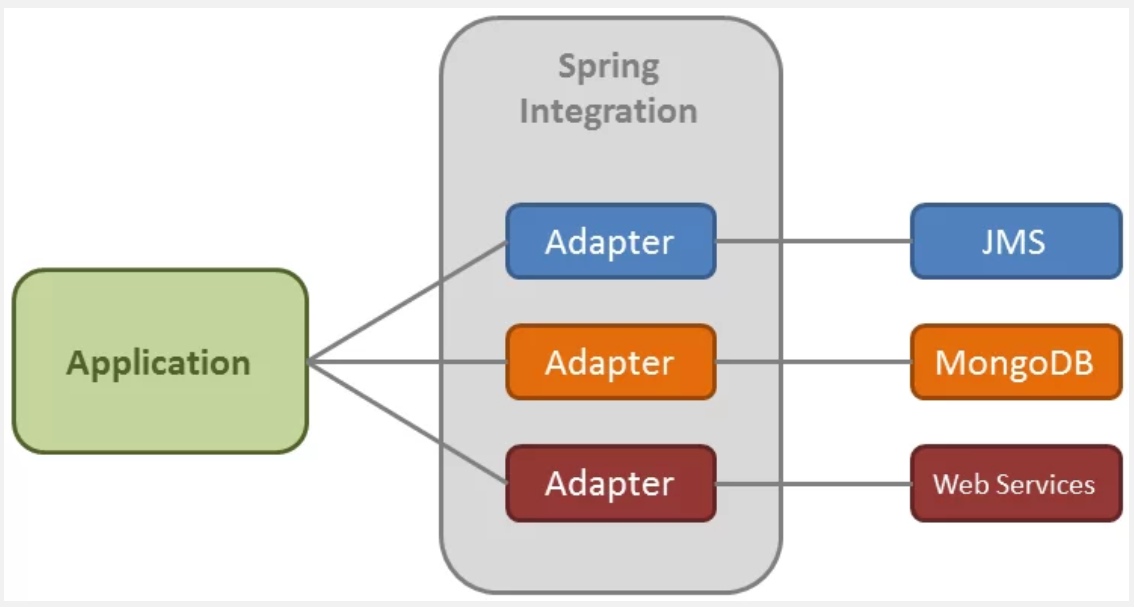
**Subscribable (Pub-Sub):**



* Allows multiple subscribers (or consumers) to register for its messages.
  + Messages are delivered to all registered subscribers on message arrival
  + It has to manage a list or registry of subscribers.
* Doesn’t buffer its messages
* Usually used for “event” messages
  + Notifying the subscribers that something happened and to take appropriate action.







<https://www.baeldung.com/spring-integration>

**Adapters fall into two broad categories — inbound and outbound.**

**Channel Adapters:**

* **Inbound channel adapter:** Unidirectional. It receives a message from an external system. It then enters to our messaging system through a message channel, where we will handle it.
* **Outbound channel adapter:** Unidirectional. Our message system creates a message and sends it to an external system.
* **Inbound gateway:** Bidirectional. A message enters into the application and expects a response. The response will be sent back to the external system.
* **Outbound gateway:** Bidirectional. The application creates a message and sends it to the external system. The gateway will then wait for a response.

**Transformer:**

This endpoint is used for payload conversion. It converts the type of the payload to another type. For example, from String to XML document. Just take into account that transforming the payload will result in a new message (remember that the message is immutable!). This type of endpoint increases loose-coupling between producers and consumers, because the consumer doesn’t need to know what is the created type of the producer. The transformer will take care of it and deliver the content type the consumer is waiting for.

Spring Integration provides several [implementations](http://docs.spring.io/spring-integration/docs/3.0.1.RELEASE/api/org/springframework/integration/transformer/package-summary.html) of Transformer. Here are some examples:

* **HeaderEnricher:** It permits to add header values to the message.
* **ObjectToMapTransformer:** Converts an Object to a Map, converting its attributes to map values.
* **ObjectToStringTransformer:** Converts an Object to a String. It converts it by calling its toString() operation.
* **PayloadSerializingTransformer / PayloadDeserializingTransformer:** Converts from Object to a byte array and the other way round.

**Filter:**

A filter is used to decide if a message should continue its way or on the contrary, dropped. To decide what to do, it is based on some criteria.

The following filter implementation will receive Order instances from the input channel and discard those with an invalid description.

**Router**

A router allows you to redirect a message to a specific message channel depending on a condition.

As usual, the framework provides some of the most basic implementations. The following example uses a payload type router. It will receive messages from the request channel, and depending of what type the payload is, it will send it to a different output channel:

Now let’s go back to our orders example and we are going to implement a router which will redirect messages depending on the order description.

**Gateways:**

A gateway is a type of channel adapter which can be used to:

* Provide an entry/exit mechanism to the messaging system. This way, the application can send a message to the messaging system, which will process it through its message endpoints.
* Send a message to an external system and wait for the response (output gateway)
* Receive a message from an external system and send a response after processing it (inbound gateway).

This example uses the first case. The application will send a message through a gateway and wait for the messaging system to process it. Here, we will use a synchronous gateway. Thus, the test application will send the message and block, waiting for the response.

**Error handling:**

This last section of this tutorial is going to explain the differences in error handling depending on which type of communication we have configured, synchronous or asynchronous.

In a synchronous communication, the sender blocks while the message is sent to the messaging system using the same thread. Obviously, if an exception is raised, it will go up reaching the application (our test in the example at the previous section).

But, in asynchronous communication, the consumer retrieves the message from a different thread. If it raises an exception it won’t reach the application. How does Spring Integration handle it? This is where the error channel comes in.

When an exception is raised, it is wrapped into a MessagingException, becoming the payload of a new message. This message is send to:

* An error channel: This channel is defined as a header named “errorChannel” in the original message header.
* A global error channel: If no error channel is defined in the message header, then it is sent to a global error channel. This channel is defined by default by Spring Integration.

**Splitter and Aggregator:**

The splitter’s target is to receive a message and partition it in several parts. These parts are then sent separately so they can be processed independently. This endpoint is usually combined with an aggregator.

The aggregator takes a list of messages and combines them into a single message. It is just the contrary of the splitter.

@Component("mySplitter")

public class MySplitter {

    public List<Order> splitOrderPackage(OrderPackage orderPackage) {

        return orderPackage.getOrders();

    }

}

The splitter returns a list of orders, but it can return any of the following:

* A collection or array of messages.
* A collection or array of Java Objects. Each list element will be included as a message payload.
* A message.
* A Java Object (will be included into the message payload).

**Poller:**

In Spring Integration there are two types of consumers:

* Active consumers
* Passive consumers

Passive components are those subscribed to a subscribable channel. This way, when a message is sent to this type of channel, the channel will invoke its subscribers. The consumer’s method will be invoked passively.

Active components are those connected to a pollable channel. This way, messages will be enqueued into the channel waiting for the consumer to actively retrieve them from the channel.

**Some Useful Links:**

<https://dzone.com/articles/integrating-with-rabbit-mq-using-spring-integratio>