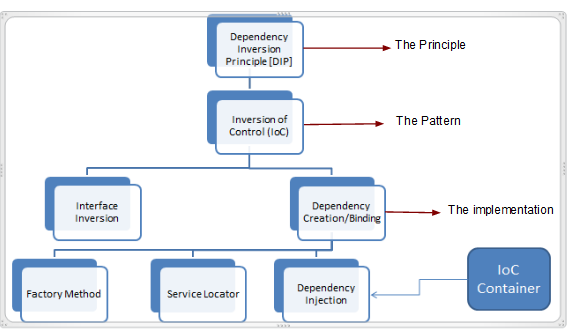
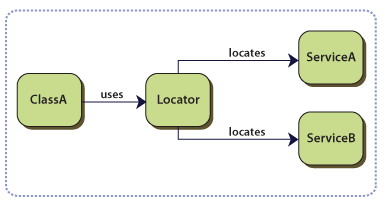
**<http://blog.christianposta.com/spring-integration/what-is-spring-integration/>**[**http://stackoverflow.com/questions/28262299/mule-esb-vs-spring-integration**](http://stackoverflow.com/questions/28262299/mule-esb-vs-spring-integration) **“what is Spring Integration”?** how does it relate to an ESB or SOA architecture, and if one were to do an analysis about Spring Integration vs its competition, what specifically is its competition? First, Spring Integration is NOT an ESB. It is a \*routing\* and \*mediation\* framework. When i say it’s a mediation framework, I mean that it allows two different systems with different messages and protocols to talk with each other by “mediating” the message: resolve/negotiate differences between the two so they can exchange data. This mediation and routing framework can be used anywhere and doesn’t need to be deployed into a heavyweight ESB container, or any ESB container. It can be deployed within in application (either stand alone application or part of a Java EE solution within an application server), within an ESB if you need, as part of a message broker, etc. It’s flexible regarding deployment. Spring Integration itself should not be compared to [ServiceMix](http://servicemix.apache.org/" \t "_blank), [MuleESB](http://www.mulesoft.org/what-mule-esb" \t "_blank), [TIBCO](http://www.tibco.com/), [IBM](http://www-01.ibm.com/software/websphere/products/appintegration/esb/) or [Oracle’s ESB](http://www.oracle.com/technetwork/middleware/service-bus/overview/index.html) solutions or [other ESBs](http://open-esb.java.net/). One open-source project that comes to mind that would be a fair comparison is[Apache’s Camel](http://camel.apache.org/)project which too is a mediation and routing engine. Apache Camel is also a very powerful and highly capable solution to the integration problem space and it also implements the patterns from the EIP book.

<http://www.javaworld.com/article/2142107/spring-framework/open-source-java-projects-spring-integration.html?page=2>

The Dependency Inversion Principle states that:

1. High level modules should not depend upon low level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

One of the key themes of the Spring Framework is inversion of control. It is done on below 3 types

1. Dependency injection relieves the components of the responsibility of locating or creating their dependencies.
2. Aspect-oriented programming relieves business components of generic cross-cutting concerns by modularizing them into reusable aspects
3. **Event Driven Architecture (EDA): Spring Integration** abstracts message sources and destinations and uses message passing and message manipulation to integrate various components within the application environment.

**What Spring Integrates?**

1. Applications built with Spring Integration are able to send messages between components, between 2 server in your environment through a message bus or
2. To another class within the same virtual machine.

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

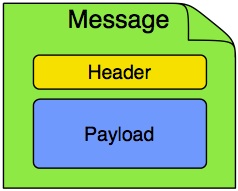
*Event-driven architecture* is one of the most powerful and successful patterns used for enterprise integration, which is followed in Spring Integration. In an event-driven architecture, a system publishes events as they happen. Components within a given system listen for specific events occurring within that system. When an event of interest occurs, the components are alerted and can respond as necessary.

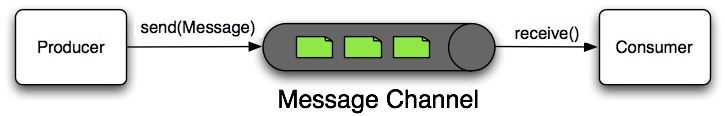
**Advantages of Event Driven Architecture**

1. 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.
2. 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.
3. 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.

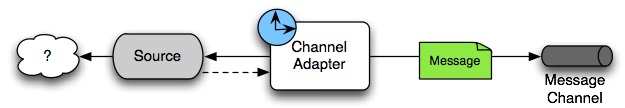
**How Spring Integration works**

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

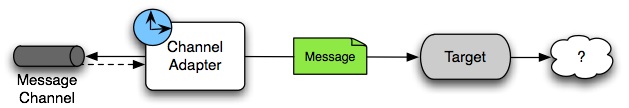
* + - * **Messages** are objects sent from one component to another.
      * **Channels** (Ex: Queues) are the means by which messages are sent, they can be synchronous or asynchronous.
        + A Message Channel may follow either Point-to-Point or Publish/Subscribe semantics
        + With a Point-to-Point channel, at most one consumer can receive each Message sent to the channel. Publish/Subscribe channels, on the other hand, will attempt to broadcast each Message to all of its subscribers.



1. **Adapters** route the output from one channel to the input of another one. A **Channel Adapter** is an endpoint that connects a Message Channel to some other system or transport.
2. There are 2 Channel Adapters
   1. For sending message – Outbound Channel Adapter
   2. For receiving message – Inbound Channel Adapter
      1. Message Driven. Endpoint is notified by queue on data.
      2. Polling. Endpoint calls queue every 5 or 10 seconds.



An inbound "Channel Adapter" endpoint connects a source system to a MessageChannel.



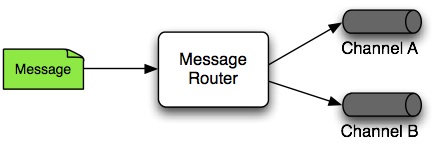
An outbound "Channel Adapter" endpoint connects a MessageChannel to a target system.

#### Messages, channels, and adapters

=================================================================================

### Router

A Message Router is responsible for deciding what channel or channels should receive the Message next (if any). Typically the decision is based upon the Message's content and/or metadata available in the Message Headers. A Message Router is often used as a dynamic alternative to a statically configured output channel on a Service Activator or other endpoint capable of sending reply Messages. Likewise, a Message Router provides a proactive alternative to the reactive Message Filters used by multiple subscribers as described above.

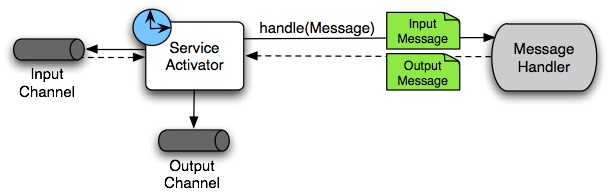


### Service Activator

1. The Service Activator gets input from a channel does process. Channel can be JMS, another Java or HTTP whoever implements org.springframework.integration.MessageChannel
2. The input Message Channel must be configured, and if the service method to be invoked is capable of returning a value, an output Message Channel may also be provided.

|  |  |
| --- | --- |
| [Note] | **Note** |
| The output channel is optional, since each Message may also provide its own 'Return Address' header. This same rule applies for all consumer endpoints. |

1. The Service Activator invokes an operation on some service object to process the request Message, extracting the request Message's payload and converting if necessary (if the method does not expect a Message-typed parameter).
2. Whenever the service object's method returns a value, that return value will likewise be converted to a reply Message if necessary (if it's not already a Message). That reply Message is sent to the output channel. If no output channel has been configured, then the reply will be sent to the channel specified in the Message's "return address" if available.



A request-reply "Service Activator" endpoint connects a target object's method to input and output Message Channels.

**Example Service Activator Configuration**

<service-activator input-channel="serviceReqChannel" output-channel="serviceResChannel" ref="servicePlatformProcessor" method="process" />

### Pipes and Filter : Pipes transport message between filters. Filters are components which capable of producing and consuming messages.

### Message Endpoint

One of the primary goals of Spring Integration is to simplify the development of enterprise integration solutions through inversion of control. This means that you should not have to implement consumers and producers directly, and you should not even have to build Messages and invoke send or receive operations on a Message Channel. Instead, you should be able to focus on your specific domain model with an implementation based on plain Objects. Then, by providing declarative configuration, you can "connect" your domain-specific code to the messaging infrastructure provided by Spring Integration.

A Message Endpoint represents the "filter" of a pipes-and-filters architecture. The Message Endpoint handles Messages. Message Endpoints are mapped to Message Channels. Message Endpoint isolate application code from the infrastructure.