AnswerHub

Transform Knowledge Into Success

KNOWLEDGE MANAGEMENT BUILT USING MODERN WEB TECHNOLOGIES

Delivers a Great Experience in Any Browser on Any Device

YOU CAN LAUNCH OUR SANDBOX IN SECONDS

TRY IT NOW!

DZone Refcardz

CONTENTS INCLUDE:

- Core Components
- Messaging Channels
- Message Routing
- Message Transformation
- Management and more...

Spring Integration

By: Soby Chacko and Chris Schaefer

ABOUT SPRING INTEGRATION

Spring Integration provides a Java-based framework with support for Enterprise Integration Patterns (http://www.eaipatterns.com) and is built on top of the popular Spring Framework. Spring Integration provides messaging capabilities within Spring applications as well as external system integration by way of integration adapters. By using Spring Integration in your application you benefit from a simple programming model for integration patterns, asynchronous message-driven behavior, and loose coupling for modularity and testing purposes. The Spring Integration project home page can be found at:

http://projects.spring.io/spring-integration

The most current Spring Integration documentation is located at:
http://docs.spring.io/spring-integration/docs/latest-ga/reference/html

And the source code can be found on GitHub at:
https://github.com/spring-projects/spring-integration

CORE COMPONENTS

Message-driven architectures are similar to application architectures but, rather than being "vertical" layers, message-driven architectures are more "horizontal". Messaging systems typically implement a "pipes-and-filters" approach where the "filters" are components that produce or consume messages and the "pipes" transport messages to achieve loose coupling.

Message

A message is simply a generic wrapper for any Java object and its associated metadata. A message is composed of a payload and its headers representing the metadata. The message payload can be of any type—often XML or JSON, and in some cases Plain Old Java Objects (POJO's). The message headers contain various pieces of information used by the messaging system such as a timestamp, a correlation ID, and so on. Message headers can also be set by the developer for more specific metadata in arbitrary key-value pairs.

Message Channel

In a messaging system, producers send messages to a channel and consumers receive those messages from a channel. In a message-driven architecture, a Message Channel represents the "pipe" of the "pipes-and-filters" model described above. Not only do Message Channels provide a place through which to produce and consume messages, they also provide the ability for other components to tap into the message flow for interception and monitoring of messages. Message Channels use either a Point-to-Point or Publish/Subscribe model. Point-to-Point messages are used when only one consumer should receive each message from a channel and the Publish/Subscribe channel model will broadcast the message to any subscriber listening to that channel. With Spring Integration, the channel is a "first class citizen". Typically endpoints don't know the type of channels they are subscribed to, or publish to. This makes it very easy to reconfigure applications as needs change. This is one of the most important distinctions between Spring Integration and other messaging frameworks.

Message Endpoint

Message Endpoints are components that connect your domain-specific code to the Spring Integration messaging infrastructure. Message Endpoints abstract away the need for application code to be aware of or construct any low-level objects such as Messages or Message Channels. As with the Message Channel, the Message Endpoint makes up the "filters" side of the "pipes-and-filters" architecture in the messaging system. Adapter endpoints provide one-way integration whereas gateway endpoints provide two-way request/response integration. This applies for both inbound and outbound endpoints.

MESSAGING CHANNELS

Messaging Channels decouple message producers from message consumers. Message Channels can be Pollable or Subscribable. In this section we look at the Message Channel types provided out of the box by Spring Integration.

Channels

To create a Channel that allows you to broadcast Messages to any of its subscribers, you use the publish-subscribe-channel element from the core Spring Integration XML name space.

```
<int:publish-subscribe-channel id="exampleChannel"/>
```

If you want to have a Channel in which a consumer can poll for messages, you create channel with a queue. Even if this channel has multiple consumers, only one of them will receive any Message sent to that channel. You can configure a queue channel in the following way.

```
<int:channel id="queueChannel">
    <queue capacity="25"/>
```

If you want to have Messages sent based on a priority, you can configure a priority channel.

```
<int:channel id="priorityChannel">
    <int:priority-queue capacity="20"/>
</int:channel>
```

A priority value in the message header is used for prioritizing messages. However, you can also provide a comparator to customize prioritization.

If you want to have a Channel with a queue associated with it, but at the same time want to also make sure that a Message is being handed off to a consumer before accepting new Messages, you need to create a rendezvous channel. The sending thread will be blocked until a consumer receives the messages. Here is how you may create one.

```
<int:channel id="rendezvousChannel"/>
    <int:rendezvous-queue/>
</int:channel>
```

The default channel provided by Spring Integration is called the Direct Channel. It allows you to dispatch any Messages sent to it to a single subscriber, thus blocking the sender thread until the Message is subscribed. It is fairly simple to create a Direct Channel.

```
<int:channel id="directChannel"/>
```

Direct Channel can also be configured with a load-balancing strategy (default uses a round-robin strategy) and supports failover.

In some cases, you want to have a Channel that can be subscribed, but at the same time can also scale. In this case, Spring Integration allows you to attach a task executor with the channel definition. This Channel then essentially becomes an Executor Channel. This allows for crafting send methods that don't block and therefore the handler invocation most likely will not occur in the sender's thread. An example is below.

```
<int:channel id="executorChannel">
    <int:dispatcher task-executor="someExecutor"/>
</int:channel>
```

MessageChannels can be further customized to restrict messages based on certain data types and scope. Here is an example of creating a Direct Channel that only dispatches messages of the type java.lang.String. Also, if a Spring conversion service is registered with the application context, an attempt will be made to convert the payload if an appropriate converter is available.

```
<int:channel id="numberChannel" datatype="java.lang.Number"/>
```

Data types can be grouped together also, as in the following.

```
<int:channel id="stringOrNumberChannel"
             datatype="java.lang.String,java.lang.Number"/>
```

Channel Adapter

A Channel Adapter is a Message Endpoint that connects a single sender or receiver to a Message Channel. Spring Integration provides adapters for a number of popular technologies, which we will briefly introduce later. However, there are two core channel adapters provided by core Spring Integration that provide you with Method-invoking Chanel Adapter support. On the inbound side, it is simply called the Inbound Chanel Adapter and on the outbound side, it is the Outbound Channel Adapter.

An Inbound Channel Adapter can invoke methods on a Spring-managed bean and send non-null values to a Message Channel as Spring Integration Messages. Here are some examples of configuring an Inbound Channel Adapter.

```
<int:inbound-channel-adapter ref="source1" method="method1"
   channel="channel1">
</int:inbound-channel-adapter>

<int:inbound-channel-adapter ref="source2" method="method2"
  channel="channel2">
</int:channel-adapter>
```

An Outbound Channel Adapter takes message payloads sent to a channel and invokes methods on a consumer POJO. Here is an example.

```
<int:outbound-channel-adapter channel="channel" ref="target"
  method="handle">
</int:outbound-channel-adapter>
```

All Channel Adapters can be created without a `channel` attribute, in which case it implicitly instantiates a Direct Channel. The Channel name will be the ID of the Channel Adapter. Therefore, either the `channel` attribute or ID is required on the Channel Adapter.

Messaging Bridge

Often times it is necessary to connect two types of Message Channels. For example, you might want to connect a PollableChannel to a SubscribableChannel and don't want the subscribing endpoint to deal with a Poller. To achieve this, you use a special type of endpoint called the Messaging Bridge, which will provide the Poller. By setting the max-messages-per-poll and other scheduling attributes appropriately, Messaging Bridge can be used to throttle inbound Messages. Configuring a Messaging Bridge is really simple. Here is an example of connecting a PollableChannel to a SubscribableChannel using a Messaging Bridge that is enabled with throttling by appropriately choosing Poller attributes.

```
<int:bridge input-channel="pollable" output-
channel="subscribable">
     <int:poller max-messages-per-poll="10" fixed-rate="5000"/>
</int:bridge>
```

MESSAGE ROUTING

Routers

Payload Type Router

A Payload Type Router allows you to send messages to a specific channel based on the payload type.

```
<int:payload-type-router input-channel="routingChannel">
  <int:mapping type="java.lang.String" channel="stringChannel" />
  <int:mapping type="java.lang.Integer" channel="integerChannel"/>
</int:payload-type-router>
```

Header Value Router

The Header Value Router allows you to route messages to a channel based on a specific header value. When a header value itself represents a channel, the mapping subelements may be omitted.

```
<int:header-value-router input-channel="routingChannel"
  header-name="testHeader">
    <int:mapping value="someHeaderValue" channel="channelA" />
    <int:mapping value="someOtherHeaderValue" channel="channelB" />
</int:header-value-router>
```

Recipient List Router

A Recipient List Router sends each message to a list of statically defined Message Channels. Recipients can also have an optional selector-expression, which can be used to determine which recipients will receive the message.

```
<int:recipient-list-router id="customRouter"
  input-channel="routingChannel">
    <int:recipient channel="channel1"/>
    <int:recipient channel="channel2"/>
</int:recipient-list-router>
```

Generic Router

A generic custom router can be created by extending a Spring Integration class called AbstractMessageRouter. In the following configuration, the `ref` attribute references the bean name of the custom Router implementation.

```
<int:router ref="customRouter" input-channel="input1"
  default-output-channel="defaultOutput1"/>
```

Exception Type Router

This type of Router can be used to route Exceptions based on their type to specific channels. A simple configuration could look like:

```
<int:exception-type-router input-channel="inputChannel"
  default-output-channel="defaultChannel">
    <int:mapping exception-type="java.lang.
IllegalArgumentException"
        channel="illegalChannel"/>
</int:exception-type-router>
```

Filter

Filter is another endpoint akin to a Router. However, a Filter will not make any decision as to where the Message should be routed; rather it determines whether the Message is sent to its output channel based on certain criteria. The following illustrates the common way of configuring a Filter.

```
<int:filter input-channel="input" output-channel="output"
         ref="exampleObject" method="someBooleanReturningMethod"/>
```

The referenced method must return a Boolean value. If it returns true, the message will be sent to the output channel; otherwise, it will not be sent. If it is false, you can set the `throw-exception-on-rejection` attribute to true so an exception will be thrown. Additionally if you want rejected messages

to be routed to a specific channel, you can provide a reference to the channel using the `discard-channel` attribute. You can use SpEL (Spring Expressions Language) expressions as well in a Filter to avoid using helper methods in a bean as in the following.

```
<int:filter input-channel="input" expression="payload.
equals('ok')"/>
```

Splitter

A Splitter is a component that splits a Message into several parts to be processed independently. In Spring Integration, any POJO can function as a Splitter, given that it has a method that takes a single input argument and return a single, collection or array of Message (or non-Message) objects. Although you can use the Spring Integration API to accomplish splitting, Splitter is the recommended approach as this decouples the application from any Spring Integration knowledge. Here is how to configure a Splitter using XML.

```
<int:splitter id="splitter" ref="messageSplitterBean"
  method="splitMessage"
  input-channel="inputChannel"
  output-channel="outputChannel" />
```

In this configuration, the splitMessage method in the object referenced by messageSplitterBean would split the incoming Message into different parts. If no ref or method attribute is provided it would be expected that the incoming payload is already a Collection.

Aggregator

An Aggregator is the reverse of a Splitter, but it is more complex than a Splitter as it needs to maintain state and know when Messages can be combined to form a single Message. Any POJO can act like an Aggregator as long as it has a method that can accept a single java.util.List. Below is an example of configuring an Aggregator using a Spring bean.

```
<int:aggregator
  input-channel="inputChannel"
  output-channel="outputChannel"
  discard-channel="throwAwayChannel"
  message-store="messageStore"
  ref="aggregatorBean"
  method="aggregate"
  expire-groups-upon-completion="false"/>
```

The message-store attribute is used to store messages until the message aggregation is completed. By default, a volatile in-memory store is used. When using a ref/bean combination for aggregation, the method must implement the logic for aggregation. By default, the aggregated Messages will be part of the output payload if no bean available to implement an aggregation logic. If the `expire-groups-upon-completion` attribute is set to true, the completed groups will be removed from the MessageStore.

Correlation and release strategies can be used with an Aggregator to combine Messages. With the default correlation strategy, Messages with the same CORRELATION_ID in the header are combined. You can override this behavior by implementing the CorrelationStrategy interface. The following is an example of using a correlation strategy with the provided method. For brevity, all other attributes are omitted.

```
<int:aggregator correlation-strategy="correlationStrategyBean"
                correlation-strategy-method="correlate" />
```

The following illustrates using a correlation strategy using a SpEL expression. Remember that you can use one or the other, not both.

```
<int:aggregator correlation-strategy="correlationStrategyBean"
               correlation-strategy-expression="headers['foo']" />
```

With the default release strategy, when all the messages in a sequence are present using the SEQUENCE_SIZE header, the Messages will be combined. If you want to customize how Messages are released, the ReleaseStrategy interface can be implemented. Here is how you may use the Release strategy.

```
<int:aggregator release-strategy="releaseStrategyBean"
                release-strategy-method="release" (16)
                release-strategy-expression="size() == 5 />
```

Similar to the correlation strategy, either the ref/bean or SpEL can be used, but not both at the same time.

You can add further statefulness to the aggregators by using the MessageGroupStore support available in Spring Integration.

Resequencer

A Resequencer is very similar to an Aggregator, but it does not do any processing on the Messages like an Aggregator does. It simply re-sequences the Messages based on the SEQUENCE-NUMBER header value. Both correlation and release strategy semantics are equivalent to that of the Aggregator. The XML configuration is mostly similar to that of the Aggregator. The namespace element used for resequencer is <int:resequencer>.

Message Handler Chain

The Message Handler Chain can be treated as a single endpoint. Using a Message handler chain, you can initiate a Spring Integration flow easily. It takes an input channel and if the final component is capable of producing a reply, an output channel can also be provided. Here is an example of a flow defined through a message handler chain. Some of the components used in this example will be explained later.

```
<int:chain input-channel="input" output-channel="output">
    <int:filter ref="filtRef" />
    <int:service-activator ref="someService"
method="someMethod"/>
 </int:chain>
```

As you can see, you can put several components together as a single large endpoint using the Message handler chain support. You can even call other chains from within an outer chain.

MESSAGE TRANSFORMATION

Transformer

Message transformers are endpoint types that perform transformation logic on messages, allowing components to produce messages in their native format and not be concerned with the type of message the consumer expects. Transformers can simply be placed between these components and the transformer will take the responsibility for message transformation. An XML example of a Transformer may look similar to:

```
<int:transformer id="myTransformer" ref="myTransformerBean"
    input-channel="input" output-channel="output"
method="transform"/>
```

In this example, you should have a Spring bean wired and referenced by the `ref` attribute along with a method implemented by the name of the `method` value.

SpEL is also supported using the `expression` attribute such as:

```
<int:transformer id="myTransformer" input-channel="input"
    output-channel="output" expression="payload.toUpperCase()"/>
```

Annotations are also supported by Transformers, by marking the method to be used as the transformation method with the @Transformer annotation. Method parameters can also be annotated with the @Header and @Headers annotations when values from the MessageHeaders should be mapped.

Spring integration also provides a number of commonly used Transformers out of the box, for example:

Namespace Element	Environment
object-to-string-transformer	Transforms an Object into its String representation.
payload-serializing-transformer	Serializes an Object to a byte array.
payload-deserializing-transformer	Serializes a byte array to an Object.
object-to-map-transformer	Transforms an Object into a Map.
object-to-json-transformer	Transforms an Object into its JSON representation.
json-to-object-transformer	Transforms a JSON message into its Object representation.

In the event that you need to do a simple transformation (for example, removing header names from the output Message), you may do so with a header-filter instead of a full transformer. For example, in XML removing the Social Security Number (SSN) and Date of Birth (DOB) headers would like similar to the following:

```
<int:header-filter input-channel="input" output-channel="output"
header-names="ssn, dob"/>
```

Content Enrichers

When receiving a message from an external source, you may find that additional information needs to be added to the message. Spring Integration provides this capability using the Content Enricher pattern and provides the following core enrichers as well as adapter-specific enrichers as part of the framework:

Enricher	Purpose
Header Enricher	Provides the ability to add Headers to a Message.
Payload Enricher	Provides the ability to enrich the Message payload itself.
XPath Header Enricher (XML Module Adapter)	Evaluates XPath expressions against the Message payload and inserts the result of the evaluation into the Message header.
Mail Header Enricher (Mail Module Adapter)	Provides the ability to enrich the headers of a Mail message.
XMPP Header Enricher (XMPP Module Adapter)	Provides the ability to enrich the headers of an XMPP message.

Header Enrichers

To simply add headers to a Message you can utilize the Header Enricher pattern support provided by Spring Integration. Adding headers to a Message using XML configuration could look similar to the following:

```
<int:header-enricher input-channel="input" output-channel="output">
    <int:header name="static" value="staticValue"/>
</int:header-enricher>
```

The `header enricher` namespace also provides support for well-known header names and can be configured using the following sub-elements:

```
<int:header-enricher ….>
    <int:error-channel ref="errorChannel"/>
    <int:reply-channel ref="replyChannel"/>
    <int:correlation-id value="12345"/>
    <int:priority value="HIGHEST"/>
    <int:header name="customHeader" ref="beanName"
      method="beanMethod"/>
</int:header-enricher>
```

As indicated by the enricher subelement `header`, a Spring bean can be referenced to dynamically determine the value for the header. Simply define and reference a Spring bean by the name referred to in the ref attribute along with the target method to invoke in the method attribute.

Payload Enricher

Payload enrichers allow you to enrich the payload with additional information. It passes a Message to the provided request channel and expects a reply message, which is used to enrich the target message. No request channel is needed if the Message payload only needs to be enriched with static values or expressions:

```
<int:enricher id="userLookupEnricher"
  input-channel="input"
  request-channel="requestChannel">
    <int:property name="firstName"
      expression="payload.firstName"/>
</int:enricher>
```

This configuration results in setting the payload value from requestChannel flow into the firstName property of the payload of target Message.

Claim Check

The Claim Check pattern allows you to store data in a known location, while only maintaining a pointer to that piece of data. The pointer to this location is stored in the payload of a Message, allowing components to request the data when it actually needs it. This pattern is useful when a Message payload is very large and would cause a performance bottleneck or, potentially, a security risk.

A basic configuration of an incoming Claim Check Transformer would look similar to the following:

```
<int:claim-check-in id="claimCheckIn"
  input-channel="input"
  output-channel="output
  message-store="messageStore"/>
```

Messages received on the input-channel will be persisted into the configured MessageStore implementation. Messages that are stored get assigned a generated ID (UUID) for Claim Check identification purposes. The returned payload will be the Claim Check ID that gets sent to the output-channel.

An Outgoing Claim Check Transformer is used to obtain the original Message payload. An example configuration would be similar to the following:

```
<int:claim-check-out id="claimCheckOut"
  input-channel="input"
  output-channel="output"
  message-store="messageStore"/>
```

The input-channel should have a Claim Check as its payload. The original payload will be looked up in the MessageStore and then sent back to the output-channel.

MESSAGE ENDPOINTS

Message Endpoints are responsible for connecting messaging components to channels. Messaging endpoints have two types of consumers, one being polling consumers, the other being event-driven consumers.

Polling

Spring Integration provides a PollingConsumer that allows polling to be scheduled by either a simple interval or a CRON expression. Pollers have many options allowing you as the developer to configure a fixed delay and rate, the number of messages processed per poll, receive timeout and a task executor to use, to name a few. Poller configuration is very simple and a very basic fixed rate example would like something similar to the following:

```
<int:transformer ….>
    <int:poller fixed-rate="1000"/>
</int:transformer>
```

Or in the case of a CRON expression:

```
<int:transformer ….>
    <int:poller cron="*/10 * * * * MON-FRI"/>
</int:transformer>
```

Pollers can also be configured as standalone, reusable components and referenced by their ID in the Poller's `ref` attribute. Global Pollers can be created by simply setting the `poller` attribute default to `true`.

Asynchronous Polling

Asynchronous polling can be achieved in the same way as the standard Poller configuration, but with the addition of a TaskExecutor. TaskExecutors can also be configured through the Spring task namespace. A simple asynchronous Poller configuration would like similar to the following:

```
<int:transformer ….>
    <int:poller fixed-rate="1000" task-executor="myTaskExecutor/>
</int:transformer>

<task:executor id="myTaskExecutor" pool-size="5" queue-
capacity="5"/>
```

Gateways

Message Gateways hide the underlying messaging API provided by Spring Integration to the client code. Using Message Gateways allows you to code to a standard Java interface rather than coupling your code to the Messaging API itself. By providing an interface, this allows your code to reference the interface and gives Spring the ability to create a proxy around it, supporting the Message Gateway behavior. A simple gateway

configuration could look similar to the following:

```
<int:gateway id="userService"
  service-interface="com.example.UserService"
  default-request-channel="requestChannel"/>
```

Methods of Gateway's service-interface can be configured via annotations by marking them with the @Gateway.

Asynchronous gateways are also supported and simply require the return value of the interface referenced by the `service-interface` attribute to return a standard `java.util.concurrent.Future`.

Service Activator

A service activator in Spring Integration simply connects any existing Spring-managed bean to a channel. These Spring service beans may or may not return a result. In the event that a result is returned from the service method, it can be returned on the configured output-channel. Service activator configuration is simple, and may look something similar to the following:

```
<int:service-activator
  input-channel="input"
  output-channel="output"
  ref="serviceBean"
  method="serviceBeanMethod"/>
```

This will configure a service activator component which receives messages from the specified input-channel, calls the method declared by the `method` attribute on the provided bean configured via the `ref` attribute and returns its result to the defined output-channel.

Delayer

A delayer is a simple, but useful component in Spring Integration that allows you to delay a Message flow by a certain defined interval. When a Delayer is used, the sender is not blocked and messages to be delayed are scheduled for delivery by a standard Spring TaskScheduler. There are a few ways that a Delayer can be configured, most commonly to delay all messages by the defined interval or on a per-Message basis.

Using XML, to configure all messages to be delayed by a specific interval, a typical configuration may look like the following:

```
<int:delayer id="delayer" input-channel="input"
  output-channel="output" default-delay="1000"/>
```

Using XML again, to use a Delayer on a per-Message basis, a simple configuration may look like the following:

```
<int:delayer id="delayer" input-channel="input"
  output-channel="output" default-delay="1000"
  delay-header-name="delay"/>
```

This configuration will create a delayer that delays any message by the value specified in the delay-header-name attribute. If this header value is not present, the delay amount will default to the value configured in the `default-delay` attribute.

MANAGEMENT

JMX Support

Spring Integration provides monitoring support through JMX. It comes with both inbound and outbound channel adapters for JMX for receiving and publishing JMX notifications. The notification listening channel adapter is an inbound channel adapter that is used to listen for events for which an MBean is publishing data. A JMX MBean has to be configured through an attribute called object-name. Here is an example of a basic Notification listening channel adapter.

```
<int-jmx:notification-listening-channel-adapter id="adapter"
  channel="channel"
  object-name="example.domain:name=publisher"/>
```

The Notification publishing channel adapter is pretty much the same as the inbound listening adapter, but it has to be aware of the MBeanExporter in the context. Here is an example of configuring a JMX publishing channel adapter.

```
<context:mbean:export/>

<int-jmx:notification-publishing-channel-adapter id="adapter"
  channel="channel"
  object-name="example.domain:name=publisher"/>
```

In addition to these, the JMX support also allows you to define an attribute polling channel adapter, which is used to periodically poll on values that are available through an MBean as a managed attribute.Additionally, there is an Operation-invoking channel adapter that allows you to invoke any managed operation exposed by an MBean. Using message payloads, you can pass arguments to the operations. Operation invoking outbound gateway allows you to create bi-directional gateways to extract the return value of the operation to a reply channel configured by the gateway.

Spring Integration components themselves can become MBeans and can be monitored directly. They will be exposed as MBeans when the IntegrationMBeanExporter is configured. Here is an example for exporting the SI components under the domain `com.data.domain` as MBeans.

```
<int-jmx:mbean-export default-domain="com.data.domain"
  server="mbeanServer"/>

<bean id="mbeanServer"
  class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true"/>
</bean>
```

Then you can start your application in the normal way that you start an application that needs JMX support and uses tools like JConsole to connect the MBeanServer and monitor SI components under the specified domain.

Message History

Message History is a great way to trace a Message, especially when you want to troubleshoot certain issues. Each time a Message goes through a tracked component, it will add a header to the Message. The history data will be maintained as a collection in the header of the Message. Here is an example of configuring Message History.

```
<int:message-history tracked-components="*Channel, router"/>
```

Message Store

Often times when you have components that save the message state, you probably want to persist those Messages using a message store. You can expose a message store through the message-store attribute on components that allow buffering. The default Message Store provided by Spring Integration is an in-memory-based one, but there are other persistent implementations available, such as JDBC, Redis, MongoDB, and Gemfire message stores.

Control Bus

Control Bus allows you to use the same messaging system for managing and monitoring messaging components such as channels and endpoints. You can therefore send a Message to invoke a method. Any Spring beans with methods annotated with either @ManagedAttribute or @ ManagedOperation can be invoked through a control bus. You can send messages with payload as a SpEL expression to initiate the operations. Here is how you may configure a control bus:

```
<int:control-bus input-channel="commandChannel"/>
```

The input channel contains Messages with a SpEL expression and the control bus will take this expression and invoke the corresponding operation.

Here is an example of how to build a Message that instructs to execute a method on a bean:

```
Message command = MessageBuilder.withPayload("@commandBean.
print()")
                .build();
commandChannel.send(command)
```

Spring Integration Adapters

Spring Integration provides several adapters for various systems, protocols, and third-party components. Most of them come with inbound and outbound channel adapters and gateway components wherever applicable. The following table lists the common adapters that are used widely.

Adapter	Summary
AMQP	Provides channel adapters for sending and receiving AMQP messages.
Spring	normal | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded
Feed	Provides support for syndication via inbound Feed adapters for formats such as RSS and ATOM.
File	Provides support for reading, writing, and transforming files. There are inbound and outbound channel adapters and an outbound gateway provided.
FTP/ FTPS	Provides inbound, outbound , outbound gateway, and session caching support for file transfer operations.
Gemfire	Provides inbound and outbound channel adapters for the Gemfire distributed caching system.
HTTP	Inbound and outbound gateways for HTTP request/response. Supports various HTTP methods.
TCP and UDP	Channel adapters for sending and receiving messages over TCP and UDP. Both of them provide one-way channel adapters and TCP provides a simple gateway support for two-way communication.
JDBC	Channel adapters for sending and receiving messages over JDBC. Provided components are inbound and outbound channel adapters, outbound gateway, stored-procedure inbound and outbound channel adapters, and stored-procedure outbound gateway.
JPA	Adapters for performing various database operations using JPA. Inbound and outbound channel adapters as well as updating and retrieving outbound gateways are provided.
JMS	Adapters for sending and receiving JMS messages. Two flavors of inbound channel adapter are provided: one using Spring's JMSTemplate and the other based on a message-driven behavior. An outbound channel adapter is also provided.

Adapter	Summary
Mail	Adapters for sending and receiving mail messages.
MongoDB	Inbound and outbound channel adapters for MongoDB.
Redis	Inbound and outbound channel adapters for Redis.
Resource	Inbound channel adapter built on top of Spring's Resource abstraction.
RMI	Provides RMI inbound and outbound gateways.
SFTP	Inbound and outbound channel adapters and outbound gateway support for secure FTP.
Stream	Inbound and outbound channel adapters for streams.
Twitter	Provides Twitter support through various types of inbound channel adapters, e.g. timeline, mentions, direct, search. Outbound channel adapter provides update and direct channel adapters.
Web Services	Provides outbound and inbound webservice gateways.
XML	XML support provides a wide array of components such as Marshalling and Unmarshalling transformers, XsltTransformer, and various xpath-based support.
XMPP	Inbound and outbound channel adapters for XMPP/XMPP presence.

In addition to out of the box adapters provided by Spring Integration, there is also a community supported Spring Integration Extensions project. This project provides integration touch points with various technologies. Details about this project can be found at:

https://github.com/spring-projects/spring-integration-extensions

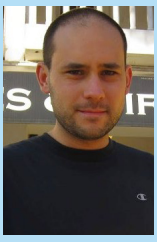
Spring Integration also has support for Scala and Groovy DSL's. You may find more details about these projects at the following URLs:

https://github.com/spring-projects/spring-integration-dsl-scala

https://github.com/spring-projects/spring-integration-dsl-groovy

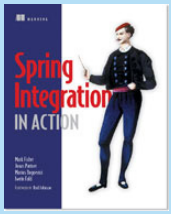
ABOUT THE AUTHOR

Soby Chacko is a Software Engineer with a background in designing and implementing distributed systems and large scale web applications. He is currently working as a consultant at Pivotal. He holds an MS in Computer Science from Villanova University, PA. Soby resides in Philadelphia, PA.



Chris Schaefer has been working in the software and systems industry for over 15 years. He works as a senior consultant implementing enterprise solutions focused on the Spring Framework portfolio ofprojects. Chris resides in Florida with his wife and cat. He is also the author of the Spring Batch refcard.

RECOMMENDED BOOK



Spring Integration in Action is a hands-on guide to Spring-based messaging and integration. After addressing the core messaging patterns, such as those used in transformation and routing, the book turns to the adapters that enable integration with external systems. Readers will explore real-world enterprise integration scenarios using JMS, Web Services, file systems, and email. They will also learn about Spring Integration's support for working with XML. The book concludes with a practical guide to advanced topics such as concurrency, performance, system-management, and monitoring.

BUY NOW!

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-83-7
ISBN-10: 1-936502-83-6
50795
9 781936 502837
$7.95

Version 1.0