Spring Integration

Jorge Simão, Ph.D.

Content

» Spring Integration Overview
» Channels & Interceptors
» Gateway & Service Activator
» Router, Filter, Transformer
» Splitter & Aggregator
» IntegrationFlow JAVA DSL
» Integration Components

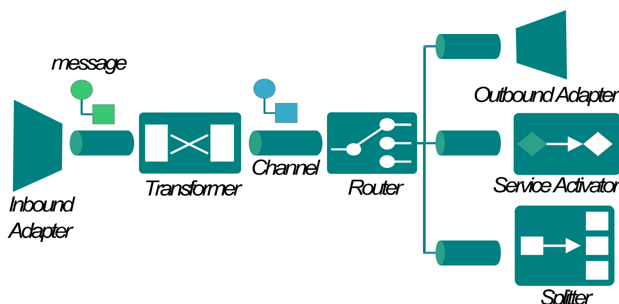
Spring Integration Overview

Spring Integration is an integration framework for **Java** that support the development of applications based on the principles of *Enterprise Integration Patterns* (EIP). It makes straightforward for applications to integrate with many types of protocols, systems, and data formats, using pipelines of *out-of-the-box* components.

Spring Integration is based on **Spring Framework** and support the same kinds of configuration styles, including XML, annotations, Java DSL, and programmatic use of the components API. Components are bundled in different modules, and each module has a dedicated XML namespace that can be used to configure the components. A core XML namespace defines the mostly used component (e.g. router, filter, transformer, service-activator, etc.), while the integration components come in dedicated namespaces (e.g. file-system, JMS, AMQP, FTP(S), HTTP, TCP/UDP/IP, JDBC, GemFire, etc.). Components can also be defined as Spring beans in factory methods of Java **@Configuration** classes, or using a Java DSL based on the concept of **IntegrationFlow**.

Enterprise Integration Patterns

In the EIP programming-model components interact by unidirectional *in-memory message passing*, rather than method invocation with return values. Data is exchanged between *endpoint* components through communication *channels*, with varied semantics. The overall architecture of the application is a network or pipeline of endpoints linked by channels (see fig. below).



Component Types

Components are broadly defined as *channels* or *endpoints*. Both channels and endpoints come in different kinds. Endpoint components that fetch/send data from/to an external system or transport protocol are designated *inbound/outbound adapters*, or if bidirectional *inbound/outbound gateways*. Several components are used inside pipelines to process data, including: *routers*, *filters*, *transformers*, *splitters* and *aggregators*. Some endpoints, such as the *service-activator* and the *gateway*, are used to mediate the interaction between pipelines and other Java/Spring beans.

Adapters	Inbound	Outbound
One-Way	Inbound Adapter	Outbound Adapter
Two-Way	Inbound Gateway	Outbound Gateway

Messages & Messaging Operations

In **Spring Integration** each *message* contains a *payload* object and a set of *headers* (key-value pairs), and its modelled by the generic interface **Message<T>**. A **Message** is a read-only (*immutable*) object – once it is created with some payload and headers it is not possible to modify it. This is intended to preserve the analogy with distributed message passing. The class **MessageBuilder** can be used to conveniently create a message using a fluent API and according to the *Builder design-pattern*.

» Example: Message API

```
public interface Message<T> {
  T getPayload();
  MessageHeaders getHeaders();
}
```

» Example: Creating Message w/ MessageBuilder

```
Message<Order> message = MessageBuilder.withPayload(order)
  .setHeader("status", OrderStatus.OPEN)
  .setExpirationDate(DateUtil.daysToExpire(30))
  .setHeader("description", "Deliver to Dr. J.")
  .build();
```

The utility class **MessagingTemplate** provides a low-

level programmatic API to send and receive messages to channels.

» Example: Send & Receive w/ MessagingTemplate

```
@Bean
public MessagingTemplate messagingTemplate() {
  return new MessagingTemplate();
}
```

```
@Autowired
private MessagingTemplate template;
```

```
Message<?> reply = template.sendAndReceive("orders", message);
if (reply.getHeaders().get("STATUS")==COMPLETED) { … }
```

Message Channels

A message **Channel** is used to connect endpoints. Channels come in two broad categories, according to the reception mode:

- **Poolable** – With active reception from a usually buffered channel – e.g. **QueueChannel**, **PriorityQueueChannel**.
- **Subscribable** – Message delivery is done via a callback or handler method – e.g. **DirectChannel**, **PublishSubscribeChannel**.

Subscribable channels are more light-weighted and are used more commonly. A **PoolableChannel** allows for more loose-coupling between endpoints, but requires active pooling via a **Poller**. Both the **Direct** & **QueueChannel** channels are used for *point-to-point communication* – each message is delivered to a single endpoint, while the **PublishSubscribableChannel** is used for dissemination of messages – all receivers get a “copy” of the message.

Channels can be defined as Spring beans using the core XML namespace of **Spring Integration**. In the Java DSL, the utility class **MessageChannels** provides factory methods and a fluent API to create and configure the different kinds of channels. Channels are light-weighted pure in-memory abstractions. But a **QueueChannel** can be made persistent by configuration of a **MessageStore**, or it can also be backed my a queue in an external message-broker (JMS or AMQP). Message delivery in **Subscribable** channels is by default node with synchronous hand-off (same thread), but can also be made asynchronous by configuring a **TaskExecutor**. When components are referenced by endpoints but not explicitly defined, the framework assume them to be auto-created **DirectChannel**.

» Example: Defining Message Channels in XML

```
<int:channel id="orderChannel">

<int:channel id="orderChannel">
    <int:queue capacity="100" />
</int:channel>

<int:publish-subscribe-channel id="pubsubChannel" task-
executor="someExecutor"/>

<task:executor id="taskExecutor"/>
```

» Example: Defining Message Channel in Java

```
@Bean
public MessageChannel fileContent() {
    return MessageChannels.direct("fileContent").get();
}
```

Table below summarizes the channels types.

Channel	Description
`DirectChannel`	Subscribable; Point-to-Point; Synchronous hand-off
`ExecuterChannel`	Subscribable; Point-to-Point; Asynchronous hand-off
`PublishScubscribChannel`	Subscribable; Dissemination; Synch or Asynch hand-off
`QueueChannel`	FIFO Buffer; Pollable; Point-to-Point; Asynchronous hand-off
`PriorityQueueChannel`	Priority Buffer; Pollable; Point-to-Point; Asynchronous hand-off
`RendezvousChannel`	Zero-capacity queue (blocks sender); Pollable; Point-to-Point; Async hand-off

Channel Interceptors

Message flow trough a **Channel** can be observed and acted upon using a **ChannelInterceptor**, whose API defines four callback methods that notify message exchange at different instants: **preSend()**, **postSend()**, **preReceive()**, **postReceive()**. A commonly used **ChannelInterceptor** is a **WireTap**, that copies message from a channel into another.

» Example: Setup ChannelInterceptor in Channel (XML)

```
<int:channel id="orderChannel">
   <int:interceptors>
       <ref bean="statsInterceptor"/>
   </int:interceptors>
</int:channel>
```

```
@Component
public class StatsInterceptor extends ChannelInterceptorAdapter {
  int nsent;

  @Override
  public Message<?> preSend(Message<?> message, MessageChannel channel) {
    nsent++;
    return message;
  }
}
```

» Example: Setup WireTap in Channel (XML)

```
<int:channel id="orderChannel">
    <int:interceptors>
        <int:wire-tap channel="logger"/>
```

```
    </int:interceptors>
</int:channel>
<int:channel id="logger"/>
```

Poller, Channel Adapter & Bridge

When using **PollableChannels**, like a **Queue**, endpoints need to active fetch messages. This can be done with a **Poller** component configured with a pooling policy. In XML, a Poller is defined with element **<poller>**, using attributes **fixed-rate**, **fixed-delay**, or **cron** to set the pooling policy. A **Poller** can be configured in an individual endpoint (as XML sub-element), or globally by setting attribute **default="true"**. Using the Java DSL, a **Poller** can be defined trough a bean of type **PollerMetaData**.

» Example: Defining a Default Polling Policy (XML)

```
<int:poller fixed-rate="1000" default="true"/>
```

» Example: Defining a Default Polling Policy (Java)

```
@Bean(name = PollerMetadata.DEFAULT_POLLER)
public PollerMetadata poller() {
  return Pollers.fixedRate(1000).get();
}
```

Pollers are also used to configure **ChannelAdapters** – endpoints that invokes a Java method and acts as a message source. The **Poller** define the rate at which the Java method is invoked.

» Example: Defining an Inbound Channel Adapter (XML)

```
<int:inbound-channel-adapter ref="getStatus"
  method="StatusService" channel="status">
    <int:poller fixed-rate="5000"/>
</int:inbound-channel-adapter>
```

A **Bridge** is a component that is used to move messages between two channels unchanged. This is useful when integrating two sub-networks – e.g. that have different expectations on the type or name of input/output channels, or to throttle the publisher sub-network via buffering in a queue.

» Example: Using a Bridge to Throttle a Publisher

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

<int:bridge input-channel="orders"
  output-channel="orderQueue"/>

<int:channel id="orderQueue" >
   <int:queue capacity="100" />
</int:channel>
```

Gateway Proxies

A **Gateway** is a *proxy-factory* component that allows Spring beans to integrated seemly with **Spring Integration** networks, by sending a message to a **Channel** when a method in the generated *proxy* is invoked. That is, rather than explicitly using the API of the **MessagingTemplate** to send messages, Spring managed components simply invoke a method defined in an interface. A **Gateway** is defined in XML with element **<gateway>** using the attribute **service-interface** to specify the interface the proxy should implement. Attribute **default-request-channel** specifies the **Channel** where the proxy sends the created messages. Annotation **@Gateway** can be used in the interface methods when the interface as several methods. When interface methods have a return value the attribute **default-reply-channel** can be used to specify the **Channel** where the proxy gets the response message from. If not specified, the proxy dynamically creates a (**Direct**)**Channel**. In both cases, the **replyTo** header of the request message is set to reference the **Channel** where the receiving endpoint should send the reply. The annotation **@IntegrationComponentScan** can also be used to automatically create **Gateway** proxies (without XML) by searching for **@Gateway** annotated methods in interfaces (on *classpath*).

» Example: Define Gateway in Java

```
<int:gateway id="orderService"
  service-interface="org.myshop.OrderService"
  default-request-channel="ordersChannel"
  default-reply-channel="confirmationChannel" />
```

» Example: Interface Implemented by Gateway Proxy

```
public interface OrderService {
      @Gateway
      Confirmation submitOrder(Order order);
}
```

» Example: Using a Proxy Gateway

```
@Service
public class ShoppingService {
  @Autowired
  private OrderService orderService;

  @Gateway
  public Confirmation processOrder(Order order) {
    //...
    return new Confirmation(order);
  }
}
```

Service Activators

A **ServiceActivator** is an endpoint that invokes a method in a Spring bean (usually in a service layer component) when a message arrives at an input channel in order to be processed. The return value of the invoked method is forward in a message to the output channel. A **ServiceActivator** is defined in XML with element **<service-activator>**, using the attributes

ref and **method** to specify the Java method to invoke. Annotation **@ServiceActivator** can also be used to specify which method to invoke. A third possibility, it to use a SPEL in-lined script, defined in XML attribute **expression**.

» Example: Defining a ServiceActivator (XML+Java)

```
<int:service-activator input-channel="in" output-channel="out"
  ref="OrderProcessor" method="processOrder"/>
```

```
@MessageEndpoint
public class OrderProcessor {
  @ServiceActivator
  Confirmation processOrder(Order order) {
    return new Confirmation(order);
  }
}
```

» Example: Define ServiceActivator w/ SPEL Expression

```
<int:service-activator input-channel="in" output-channel="out"
expression="@orderProcessor.process(payload,headers['type'])"/>
```

Message Routing

A **Router** is an endpoint that forwards incoming messages to one among several output channels. A routing criteria is defined – in Java or in SPEL – to select the output channel for each individual message. The content of the message payload and/or headers is usually used to make this decision. A generic router is defined in XML with element **<router>**, using the attributes **ref** and **method** to specify the Java method for the routing criteria. Alternatively, annotation **@Router** be used to specify which methods define the routing criteria. Additionally, the stereotype annotation **@MessageEndpoint** is often used to define criteria beans. SPEL scripts can also be used to specify the routing criteria with XML attribute **expression**.

» Example: Defining a Router (XML) & Routing Criteria

```
<int:router input-channel="orderChannel" ref="orderRouter" />
```

```
@MessageEndpoint
public class OrderRouter  {
  @Router
  public String routeOrder(Message<Order> message) {
    return isUrgent(message) ? "vipChannel" : "altChannel";
  }
}
```

» Example: Defining Routing Criteria w/ SPEL

```
<int:router input-channel="orderChannel"
  expression="headers['type']"/>
```

```
<int:router input-channel="orderChannel"
 expression="Orders.isXml(payload)?'xmlChannel':'csvChannel'"/>
```

More specialized kinds of routers are also available, such as the **PayloadTypeRouter** that routes messages to channels according to the type of the payload, or the **HeaderValueRouter** that routes based on the value on an header.

» Example: Routing w/ PayloadTypeRouter (XML)

```
<int:payload-type-router input-channel="orderChannel" default-
output-channel="miscOrderChannel" >
   <int:mapping type="org.w3c.dom.Document"
                channel="xmlOrderChannel" />
   <int:mapping type="java.lang.String"
                channel="csvOrderChannel" />
</int:payload-type-router>
```

» Example: Routing w/ HeaderValueRouter (XML)

```
<int:header-value-router input-channel="orderChannel"
     header-name="status" >
    <int:mapping value="CHECKED_OUT" channel="stockChannel" />
    <int:mapping value="COMPLETED" channel="invoiceChannel" />
    <int:mapping value="CANCELED" channel="cancelChannel" />
</int:header-value-router>
```

Message Filtering

A message **Filter** is an endpoint that relays messages from an input to an output channel conditionally. They are used to discard messages based on payload or header values, and detect invalid or non-conforming messages. A filter is defined in XML with element **<filter>**, using attributes **input-channel** and **output-channel** to specify the channel from where message are received and relayed, respectively, and **ref** and **method** for the filtering criteria. Alternatively, annotation **@Filter** be used to specify which method defines the routing criteria as well as the **Filter** configuration. A SPEL script can also be used as filtering criteria with XML attribute **expression**.

By default, filtered-out messages – those for who the criteria method returns **false** – are discarded. It is also possible to make filtered-out messages throw an **Exception**, with XML attribute **throw-exception-on-rejection="true".** Or, alternatively, to be diverted to an alternative channel with attribute **discard-channel**.

» Example: Defining Filter (XML) & Filtering Criteria

```
<int:filter ref="orderSelector" />
```

```
@MessageEndpoint
public class OrderSelector {
    @Filter(inputChannel="in", outputChannel="out")
    boolean filter(Order order) {
        return OrderUtil.isValid(order);
    }
}
```

» Example: Diverting Filtered Out Messages

```
<int:filter input-channel="in" output-channel="out"
            ref="orderValidator" discard-channel="invalid" />
```

Message Transformation

A message **Transformer** is an endpoint that transforms

messages payload and/or headers, and can be used to perform arbitrary transformations on payloads, convert payload types, and add/change/remove headers. The XML element **<transformer>** can be used to define a transformer. The transformation function can be defined as Spring Bean implementing interface **MessageProcessor**. Alternatively, a POJO can be used when XML attribute **method** or Java annotation **@Transformer** specifies the transformation method.

» Example: Define & Configure Transformer (XML+Java)

```
<int:transformer id="invoiceTransformer"
  input-channel="orderChannel" output-channel="invoiceChannel"
  ref="orderTransform" method="createInvoice" />
```

```
@MessageEndpoint
public class InvoiceCreator {
  @Transformer
  public Invoice createInvoce(Order order) {
    return new Invoice(order);
  }
}
```

More specialized kinds of transformer are also available such as the **ObjectToStringTransformer** that maps an object to its **String** representation, or transformers to perform XML or JSON (un)marshalling.

Message Splitting

A **Splitter** is an endpoint that produces multiples output messages out of a single input message. It is used to decompose messages with composite payloads (e.g. a collection or a structured object with child objects), into separated messages one for each of component parts of the payload (e.g. an **Order** may be split into a collection of **OrderItem)**. This is useful for processing the component parts separately, followed possibly by latter (re)aggregation.

The XML element **<splitter>** is be used to define a **Splitter**, with attribute **ref** naming a Spring bean implementing interface **SplittingCriteria**. Alternatively, a POJO can be used when XML attribute **method** or Java annotation **@Splitter** specifies the splitting criteria. The method takes a **Message** or POJO (for the payload) as parameter, and should return a **List** of **Message** or payload objects.

» Example: Define & Configure Splitter (XML + Java)

```
<int:splitter id="orderSplitter"
  input-channel="orderChannel" output-channel="itemChannel"
  ref="orderSplitterBean"  />
```

```
@MessageEndpoint("orderSplitterBean")
public class OrderSplitter {
    @Splitter
    public List<OrderItem> split(Order order) {
        return order.getItems();
    }
}
```

A **Splitter** copies the headers of the input message to the output message when POJO payloads are used in the splitting method. Additionally, it sets the following headers:

- **CORRELATION_ID** – is set with the value of the header **ID** of the input message.
- **SEQUENCE_SIZE** – is set with the number of output messages produced from the input message.
- **SEQUENCE_NUMBER** – is set with a ordering number from **1** to **SEQUENCE_SIZE**.

These headers are set so that it possible to track the origin of the message. For example, to reconstruct the original messages after the component elements are further processed by intermediate endpoints.

Message Aggregation

An **Aggregator** is an endpoint that combines multiple input messages into a single output message (i.e. the converse of a **Splitter**). A *correlation strategy* is used to decided which input messages are to be aggregated with which other messages (i.e. belong to the same *message group* or “bucket”). A *release strategy* is used to determine when the message group is complete and a single message should be produced out of all the messages in the group. An *aggregation criteria* defines how to assemble a single output message out of a group of correlated messages. A typical use-case for an **Aggregator** (but not the only one) is to reconstruct a composite message out of multiple “child” messages that were earlier produced by a **Splitter** (e.g. an **Order** (re)created out of a collection of **OrderItem)**. Messages that have the same correlation value are collected together, and when the release is done, the aggregation criteria is called to produce the combined output message.

The XML element **<aggregator>** is be used to define an **Aggregator**, with attribute **ref** naming a Spring bean implementing interface **AggregationCriteria**. Alternatively, a POJO can be used when XML attribute **method** or Java annotation **@Aggregator** specifies the aggregation criteria. The method takes a a **List** of **Message** or payload objects as parameter, and should return a single **Message** or POJO (for the combined payload).

The correlation and release strategies are specified with attributes **correlation-strategy** and, **release-strategy**, respectively, implementing interfaces **CorrelationStrategy** and **ReleaseStrategy**. Alternatively, POJO can be used if attributes **correlation-strategy-method** and **release-strategy-method**

are specified. In the common case where the aggregation is done over messages split earlier by a **Splitter** both strategies are optional. This is because the default strategy is to combined messages by the value of the **CORRELATION_ID**, and to release them when **SEQUENCE_SIZE** equals the number of available messages.

» Example: Aggregator w/ Default Correlation&Release

```
<int:aggregator id="orderAggregator"
  input-channel="itemChannel" output-channel="orderChannel"
  ref="orderAggregatation" />

@MessageEndpoint
public class OrderAggregation {
  @Aggregator
  public List<OrderItem> createOrder(Order order) {
    return order.getItems();
  }
}
```

» Example: Aggregator with Full Custom Configuration

```
<int:aggregator id="orderAggregator"
  input-channel="itemChannel" output-channel="orderChannel"
  ref="orderAggregatation" method="createOrder"
  correlation-strategy="correlationStrategyBean"
  correlation-strategy-method="correlate"
  release-strategy="releaseStrategyBean"
  release-strategy-method="release" />
```

Endpoint Chain

When several endpoints are connected in a serial pipeline, such that the output channel of previous component is the input channel of the following one, it is possible to simplify the configuration by defining a composite component that makes this relationship explicit. The XML element **<chain>** defines an endpoint **Chain**, that automatically wires the endpoints part of the pipeline including the creation of intermediate **DirectChannel** to connect the endpoints. The channel named in the attribute **input-channel** is wired as input channel to the first component in the pipeline, and the channel named in the attribute **output-channel** (if specified) is wired as output channel of the last component.

» Example: Defining a Chain as a Endpoint Pipeline

```
<int:chain input-channel="input" output-channel="output">
    <int:filter ref="orderValidator" />
    <int:header-enricher>
        <int:header name="valid" value="true"/>
    </int:header-enricher>
    <int:service-activator ref="OrderService" method="submit"/>
</int:chain>
```

Integration Flow (Java DSL)

A Java DSL can also be used to create pipelines based on the **IntegrationFlow** abstraction – similarly to the **Chain** – but allowing Java *lambda expressions* to be used to define the criteria strategies. An **IntegrationFlow** is defined as a Spring Bean and created with a **IntegrationFlowBuilder** that provides a fluent API to define the pipeline. The **IntegrationFlow** is a meta-data descriptor used only during configuration. To actually create the components from the meta-data the annotation **@EnableIntegration** should be used.

» Example: Using the IntegrationFlow DSL

```
@Bean
public IntegrationFlow orderFilesFlow() {
  return IntegrationFlows
    .from("orderFiles")
    .filter((File f) -> f.length()<=100000)
    .transform(Transformers.fileToString())
    .transform(Transformers.fromJson(Order.class))
    .filter((Order o) -> o.isValid())
    .handle(OrderService::submit)
    .get();
}
```

XML Payloads

In the special case where messages payload is XML, it is desirable to use XML specific technologies to process the messages. **XPath** expressions – which are used to select the sub-trees of an XML document that match a particular expression – can be used to define strategies for a **Spliter**, a **Router**, or a **Filter** (e.g. **XPath** expression **//item** returns the list of all **<item>** elements in a XML document, so it can be used as a splitting criteria). The element **<xpath-expression>** from the **spring-integration-xml** namespace is used to define **XPath** expression.

An XSLT script define an XML transformation, so it can also be naturally to used to define a **Transformer** for XML payloads. The XML element **<xslt-transformer>** defines a XSLT transformer, with attribute **xsl-resource** naming the script file.

» Example: Splitting XML Payload via XPath

```
<int-xml:xpath-splitter id="orderItemSplitter"
  input-channel="orderChannel" output-channel="itemsChannel">
    <int-xml:xpath-expression expression="//item"/>
</int-xml:xpath-splitter>
```

» Example: XLST Transformer & XSLT Script

```
<int-xml:xslt-transformer id="xsltTransformerWithResource"
  input-channel="withResourceIn" output-channel="output"
  xsl-resource="order-version-migration.xsl"/>

<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/order">
    <xsl:attribute name="priority" value="normal" />
  </xsl:template>
</xsl:stylesheet>
```

JMS Integration

JMS integration is done trough four main components from the **spring-integration-jms** XML namespace: **<inbound-channel-adapter>**, to relay messages sent from a **MessageChannel** to a JMS destination, **<outbound-channel-adapter>** to receive JMS messages and relay them to a channel, and for two-way communication the **<inbound-gateway>** to process JMS messages and send a response, and the **<outbound-gateway>** to send a SMS message and wait for a response.

» Example: Sending Channel Messages to JMS

```
<int-jms:inbound-channel-adapter id="jmsIn"
  destination="ordersQueue" channel="ordersChannel" />
```

File Integration

Integration with the file-system (reading & writing files) is done with a few components from XML namespace **spring-integration-file** such an **<inbound-channel-adapter>** that pool a directory for new files and emits messages with the **File** payload (the filename). Transformer class **FileToStringTransformer** can be use to read the content of the file as a message.

» Example: Pooling Directory & Reading File Content

```
<int-file:inbound-channel-adapter id="files"
  directory="file:input" filename-pattern="*.csv">
  <int:poller id="poller" fixed-rate="5000" />
</int-file:inbound-channel-adapter>

@Bean
@Transformer(inputChannel = "files",
             outputChannel = "fileContentChannel")
public FileToStringTransformer fileToStringTransformer() {
  return new FileToStringTransformer();
}
```

Other Integration Options

Spring Integration provides many additional integration components – inbound and outbound adaptors and gateways – for a variety of transport protocols and data stores, including: FTP, AMQP, Redis Pub/Sub, GemFire, JDBC, HTTP, TCP/UDP/IP, among other.

Resources

- Spring Integration Project home page: http://projects.spring.io/spring-integration/
- Spring Integration Reference Manual: http://docs.spring.io/spring-integration/docs/5.0.0.BUILD-SNAPSHOT/reference/html/
- Enterprise Integration Patterns: http://www.enterpriseintegrationpatterns.com/

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consulting) at **EInnovator**. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.

Spring Integration Training

Enterprise Spring is a 4-day trainer lead course that teaches how to use **Spring Framework**, **Spring Integration**, **Spring Batch**, and **Spring XD** to build enterprise integration solutions.
Completion of this training prepares participants to take a certification exam and become a **Pivotal** certified **Enterprise Integration Specialist**.
Book for a training event in a date&location of your choice: **www.einnovator.org/course/enterprise-spring**

++ QuickGuides » EInnovator.org

» Spring Dependency-Injection, Spring MVC
» RabbitMQ, Redis
» Cloud Foundry, Spring Cloud
» and much more...

++ Courses » EInnovator.org

» Core Spring, Spring Web
» RabbitMQ, Redis, CloudFoundry
» BigData and Hadoop, Spring XD, Spark
» and much more...

EInnovator – Software Engineering School

EInnovator.org offers the best Software Engineering resources and education, in partnership with the most innovative companies in the IT-world. Focusing on both foundational and cutting-edge technologies and topics, at EInnovator software engineers and data-scientists can get all the skills needed to become top-of-the-line on state-of-the-art IT professionals.

Contacts

Training – Bookings & Inquiries
training@einnovator.org

Consultancy – Partnerships & Inquiries
consulting@einnovator.org

General Info
info@einnovator.org

EInnoVator

Copyright © 2016 EInnovator.org. All rights reserved.