spring

SPRING INTEGRATION FOR EAI

Enterprise Application Integration Ignited

XAVIER PADRO

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Spring Integration for EAI

Contents

Preface

Spring Integration is an open source framework for enterprise application integration. It is a lightweight framework that builds upon the core Spring framework. It is designed to enable the development of integration solutions typical of event-driven architectures and messaging-centric architectures.

Spring Integration extends the Spring programming model to support the well-known Enterprise Integration Patterns. Enables lightweight messaging within Spring-based applications and supports integration with external systems via declarative adapters. Those adapters provide a higher-level of abstraction over Spring's support for remoting, messaging, and scheduling.

In this book, you are introduced to Enterprise Application Integration patterns and how Spring Integration addresses them. Next, you delve into the fundamentals of Spring Integration, like channels, transformers and adapters.

Furthermore, you will learn how Spring Integration works hand in hand with Web Services and Messaging Queues and finally you will develop a full-blown application from scratch.

About the Author

Xavier is a software engineer who has always considered programming to be his passion. He doesn't feel it is just a job but also a way to bring his ideas to life. He enjoys sharing his passion with the technology by writing his thoughts on his personal blog. Although he programmed in several languages, he currently earns his living by developing web applications in Java. He currently lives near the sunny Barcelona.

He has a degree in computer technical engineering (B.Eng) and he has kept forming himself in his main specialities, like Java (Sun Certified Java Programmer and Sun Certified Web Component Developer) and the Spring framework (SpringSource Certified Spring Professional and SpringSource Certified Enterprise Integration Specialist).

Regarding his job, he is specialized in Java EE and web development. His experience includes not only the development of web applications but also being involved in the design and implementation of architectural solutions. Some of the projects he were involved include the development of a SOA architecture and the design of a backend with enterprise integration patterns.

Since Java is his main programming language, he is knowledgeable about core aspects of the language. Among his skills there are several technologies from the Java Enterprise platform like JAX-RS, JAX-WS, JMS and Java Server Faces to name a few. He also has extensive experience developing web applications using the Spring framework. This includes not only core aspects of the framework but also some of its related projects like Spring Web Flow and Spring Integration.

Chapter 1

Introduction to Enterprise Application Integration

1.1 Introduction

This is the first part of the Spring Integration course. This part introduces you to what enterprise integration patterns are and how different strategies can be applied to design integration solutions. The reason behind why you should acquire a basic knowledge of these patterns in this course is because the Spring Integration project was designed upon these patterns. The following parts of this course will get into the Spring Integration project and show practical examples of how these patterns are applied.

1.2 Enterprise integration patterns

Enterprise Integration Patterns is a group of 65 design patterns which are compiled in the book under the same name and written by Gregor Hohpe and Bobby Woolf back in 2003. The target of defining these design patterns is in view of the need to standardize procedures and establish a reference for the developers in order to deal with building integration solutions.

This group of integration design patterns is the result of a recompilation of practices used by experienced developers during years, and each of them describe the base solution to a specific design problem related to the communication among different systems.

Once we move forward through the course, you will see how Spring Integration's API is based on these enterprise integration patterns, since its design was inspired on the concepts explained in the above book. This first tutorial will make a brief introduction to these concepts in order to make you feel familiar with them when you see how Spring Integration is built.

1.3 Integration strategies

The task of integration between applications or systems has always been pretty difficult. Some of the reasons are that applications can be written in different programming languages (the communication among them is impossible since one system does not understand the other) or use a different data format (the message is incompatible). During the years, different approaches have been implemented in order to face these problems and accomplish the challenge of integrating applications. There are basically four strategies which are briefly described below:

1.3.1 File transfer

This strategy involves applications sharing information by using files. You can have one or more applications which produce a file with information (the producer), and other applications will consume this data (consumers). One of the most important things to take into account is to decide which standard format will the data within the file have, because all involved applications should know how to deal with it. One of the most accepted formats is the use of XML.

Once the data format is set, there may be several applications which use the information in that file in a different manner. For that purpose, the consuming application will need an interceptor with the target to transform the format in which the file has been generated so that it can be adapted to the application's requirements.

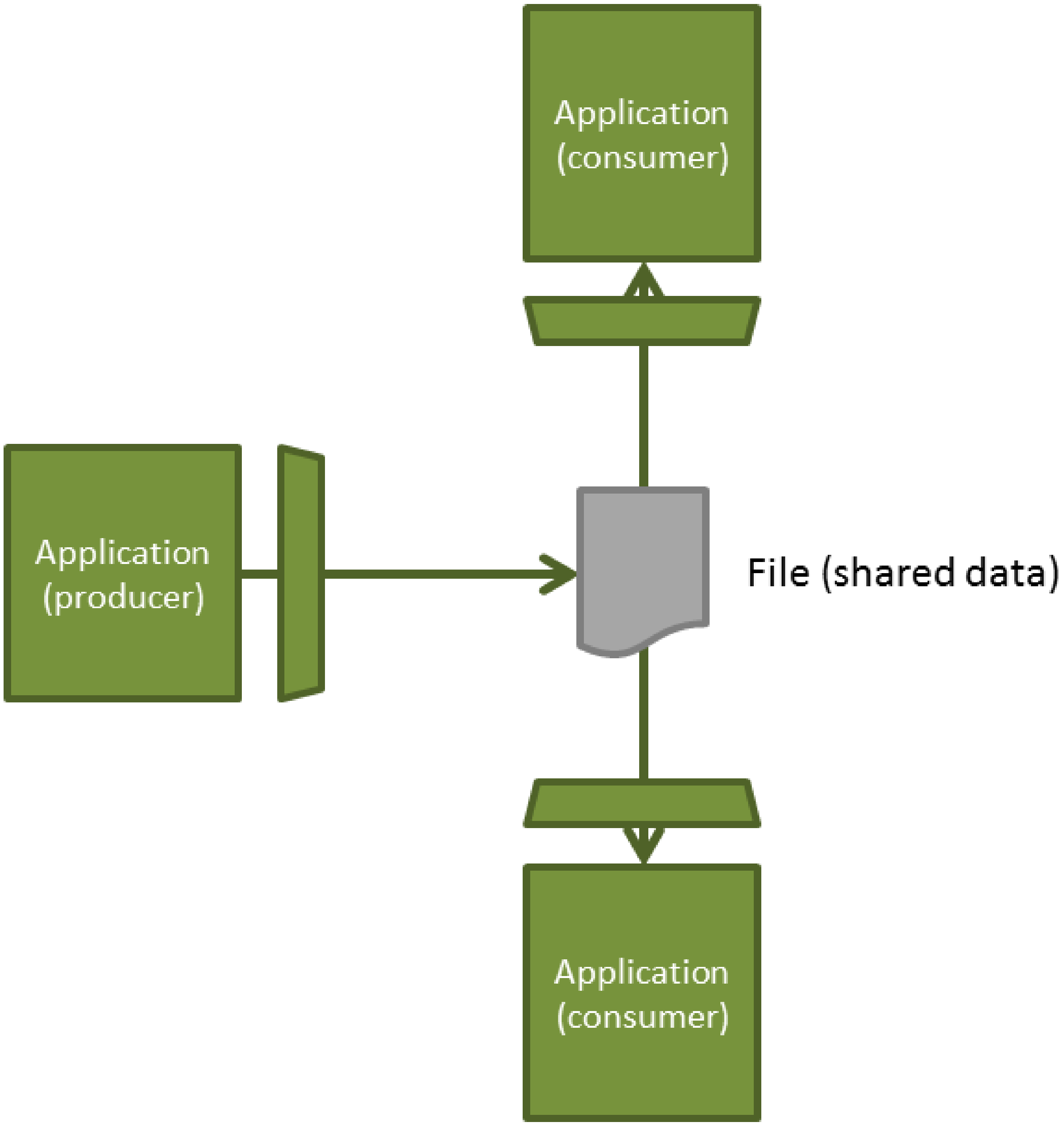


Figure 1.1: screenshot

The main advantage is that it decouples applications from each other. The consuming application does not need to know the internals of the producing application. Interceptors will deal with files, so changes in any of the involved applications don't affect the others as long as they keep the same file format.

On the other side, this approach usually takes time, so it may not be ideal if you need the information too frequently. Some applications may need to show updated information as quickly as possible. In this case, a shared database may be a better option. Another aspect to consider is that file transfer strategy is quite unsafe, since it is not transactional and can have concurrency issues.

1.3.2 Shared database

This solution is based upon having a central database that stores all the information that needs to be shared. This way, different applications will be able to simultaneously access the same data, provided you use transaction management. By using the same database, the retrieved information will be consistent and up to date. Also, the information can be quickly accessed by the consumers, making sure you don't get stale data. This would be one of the drawbacks you would have to face if file transfer was used.

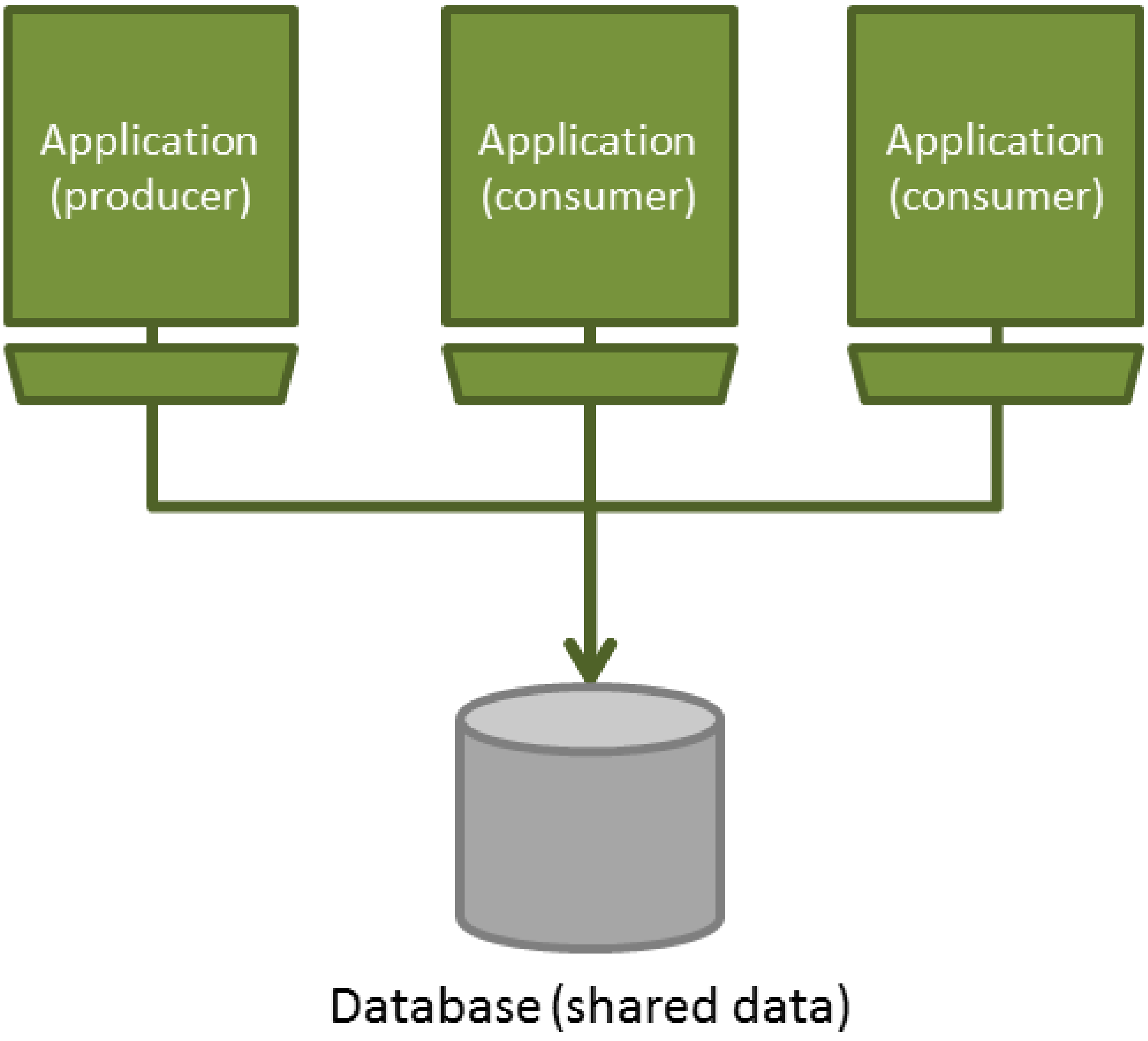


Figure 1.2: screenshot

You have to keep in mind though, that there can be performance issues if multiple applications access the same data. Some applications may be blocked trying to modify data locked by another application.

Another difficulty is found when designing the database schema. The resulting schema should be suitable for all the involved applications. Additionally, you will have to take into account that any change in the schema would affect them. Nowadays, this may not be an issue if you take the decision of using a NoSQL database like MongoDB or Apache Cassandra because these types of databases use a schemaless data structure. Considering the advantages or disadvantages of each type is beyond the scope of this tutorial.

1.3.3 Remote Procedure Invocation

In previously discussed approaches, a producer produces information (stored into a file or database) and others can consume it. But, what if you need to interact with the other application depending on the shared data? There's a problem here, since the producer does not know the internals of the consuming application. You need some kind of abstraction mechanism. This is where remote procedure invocation comes in.

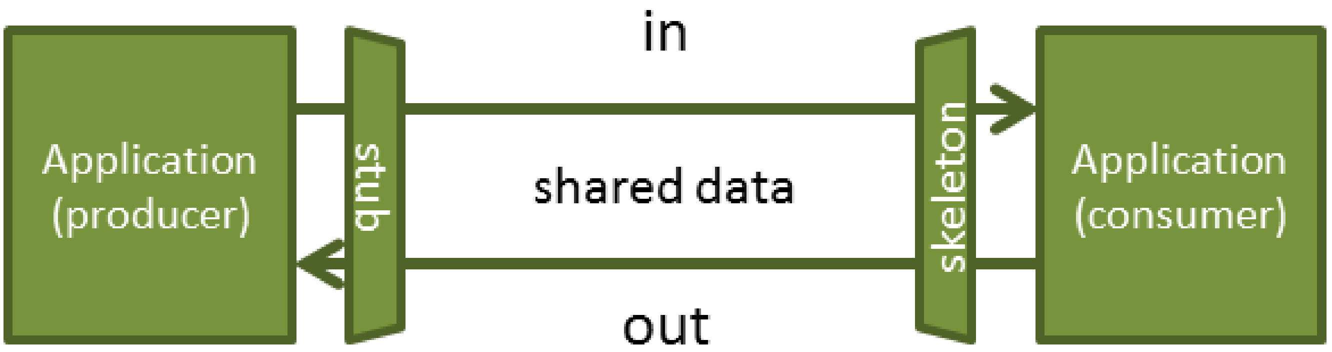


Figure 1.3: screenshot

The remote procedure invocation consists of an application interacting directly with another application through stubs. The client calls a stub (client stub) through a local procedure call, and the stub sends the message to the server, where another stub (skeleton) will receive it and call the server procedure.

The cons of this approach are that applications get tightly coupled and their calls are slow. This brings us to the last strategy, messaging.

1.3.4 Messaging

Messaging is a better fit if you need to exchange small amounts of information between applications. The great benefit of messaging is that components (producers and consumers) are decoupled. The producer can send the message without knowing who is on the other side. There may be one or more consumers that will receive the message, but that's not relevant to the producer.

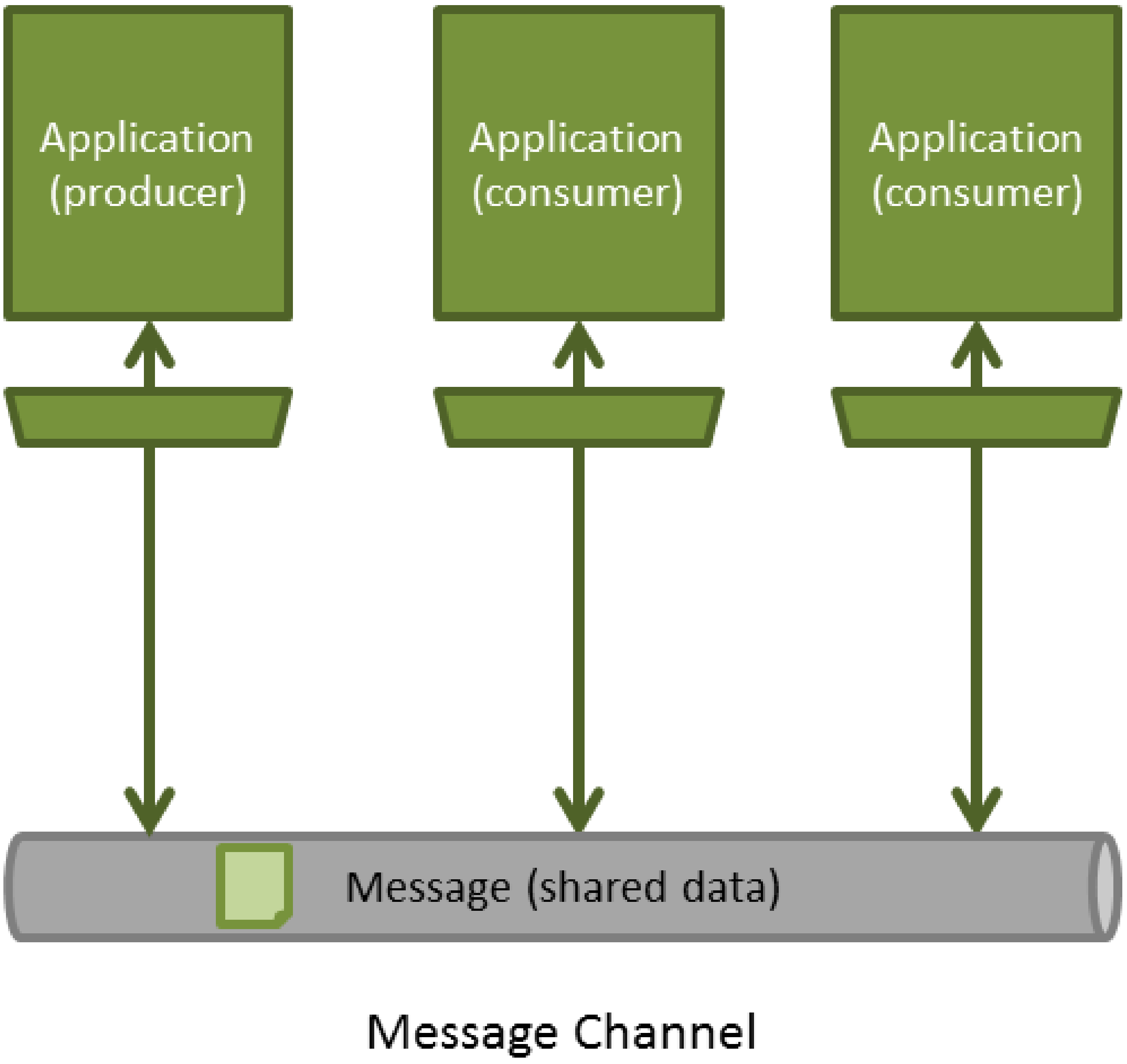


Figure 1.4: screenshot

Another important feature is that messaging can be done asynchronously. This means the producer can send the message and continue with his logic without having to block in order to wait for the consumer to return the response. Once the consumer has processed the message and sent a response, the producer will be notified.

The main downside of this approach is its complexity, especially when dealing with asynchronous messaging.

This strategy is considered by the book authors of enterprise integration patterns generally the best approach for integration enterprise applications, and the Spring Integration project is based upon this same strategy. For this reason, the rest of the tutorial will be based on this strategy.

An architecture based on this strategy is called message driven architecture. The next section explains its basic concepts, which will be largely used through this course.

1.4 Message driven architecture

A message driven architecture is an architecture based on the messaging strategy that you have seen in the previous section. The basic concepts on which this strategy is built are explained below:

- **Message**: Amount of information that is shared among applications or among different components in the same application. This message is a data structure composed by a header containing meta-information about the message, and a body that contains the information that we want to share.
- **Producer**: A component which creates (produces) a message and sends it to a message channel. The message can be sent synchronously so the producer will block its thread and wait until a response is received. But, if the processing may take time, there's a better option; the producer can send the message asynchronously.
- **Message channel**: A message channel is some kind of pipe or queue that connects the producer to one or several consumers.
- **Consumer**: A component that retrieves (consumes) the message from the message channel and processes it. Optionally, a response is sent back to the producer.

This message driven approach loosely couples applications. An asynchronous communication connects both applications in a way that one application does not need to know if the other is active. The producer can send the message and forget about it, continuing with its own work. If the sending requires a response, the producer will be notified in order to handle the result.

1.5 Synchronous and asynchronous communication

Synchronous communication allows for a real time conversation where both parts are active. The sender sends the message and waits for the receiver to process it and return a response. This is useful when the producer needs an immediate response in order to continue with its tasks. This type of communication has its drawbacks though; for example, the next task that needs to be done by the sender will be delayed if the receiver processing takes time. Or even worse, the consumer may be inactive. A common solution is to establish a timeout and handle it if the response takes too much time.

Asynchronous communication allows decoupling of both parts, each one possibly acting at a different time (the receiver could not be active at the moment of the sending). This approach is common when the sender does not need the response immediately. It will continue processing its tasks until a response is received. Asynchronous communication can be adequate when the receiver processing takes time.

Spring Integration allows both types of communication, each one with its advantages and disadvantages. During the following tutorials of this course, you will see how to accomplish this and be able to decide which one is more adequate in each situation.

Chapter 2

Spring Integration Fundamentals

2.1 Introduction

In this second tutorial, you will learn the basic concepts that form the core of Spring Integration. After these concepts are explained, we will review the different components that are shipped with the project. This revision is based upon the 3.0.1 release. Considering that the 4.0.0 release will be released soon, you may find some new components that are not explained in this tutorial. Anyway, you will gain enough knowledge of the framework to understand the behavior of future components.

To conclude this tutorial, you will learn how different types of communication (asynchronous and synchronous) are supported by Spring Integration, and how this decision can affect your design. One special case is error handling, which is explained in the last section.

This tutorial is composed by the following sections:

- Introduction
- What is Spring Integration?
- Core concepts of Spring Integration messaging system
- Components
- Synchronous and asynchronous communication
- Error handling

2.2 What is Spring Integration?

As explained in the previous section, 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.

Currently, Spring Integration configuration is mainly xml based, although some annotations are starting to be included. Examples shown in this tutorial will also be xml based, though I will show its respective annotation when possible.

Having explained that, a question arises: what can you do with Spring Integration? The framework basically allows you to do the following:

- It allows communication between components within your application, based on in-memory messaging. This allows these application components to be loosely coupled with each other, sharing data through message channels.

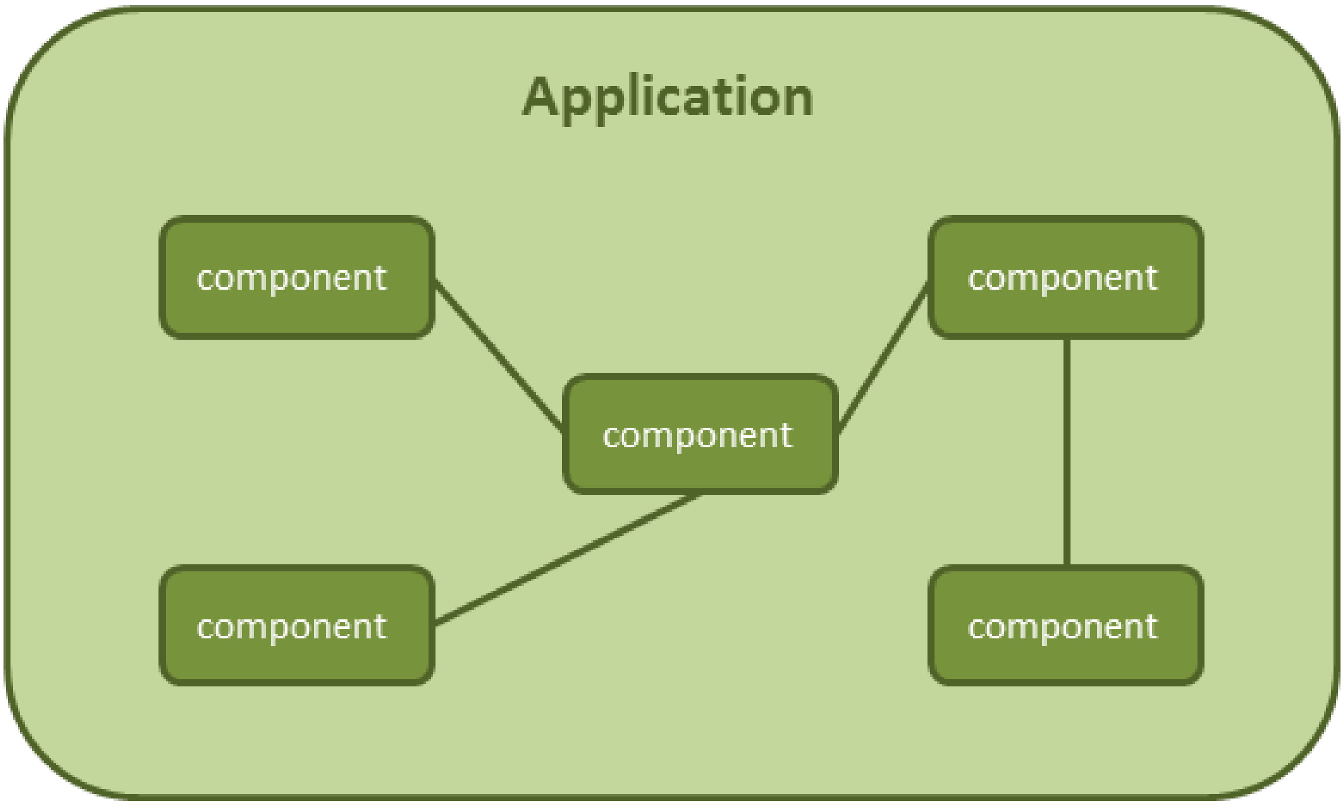


Figure 2.1: screenshot

- It allows communication with external systems. You just need to send the information; Spring Integration will handle sending it to the specified external system and bring back a response if necessary. Of course, this works the other way round; Spring Integration will handle incoming calls from the external system to your application. This will be explained later in this tutorial.

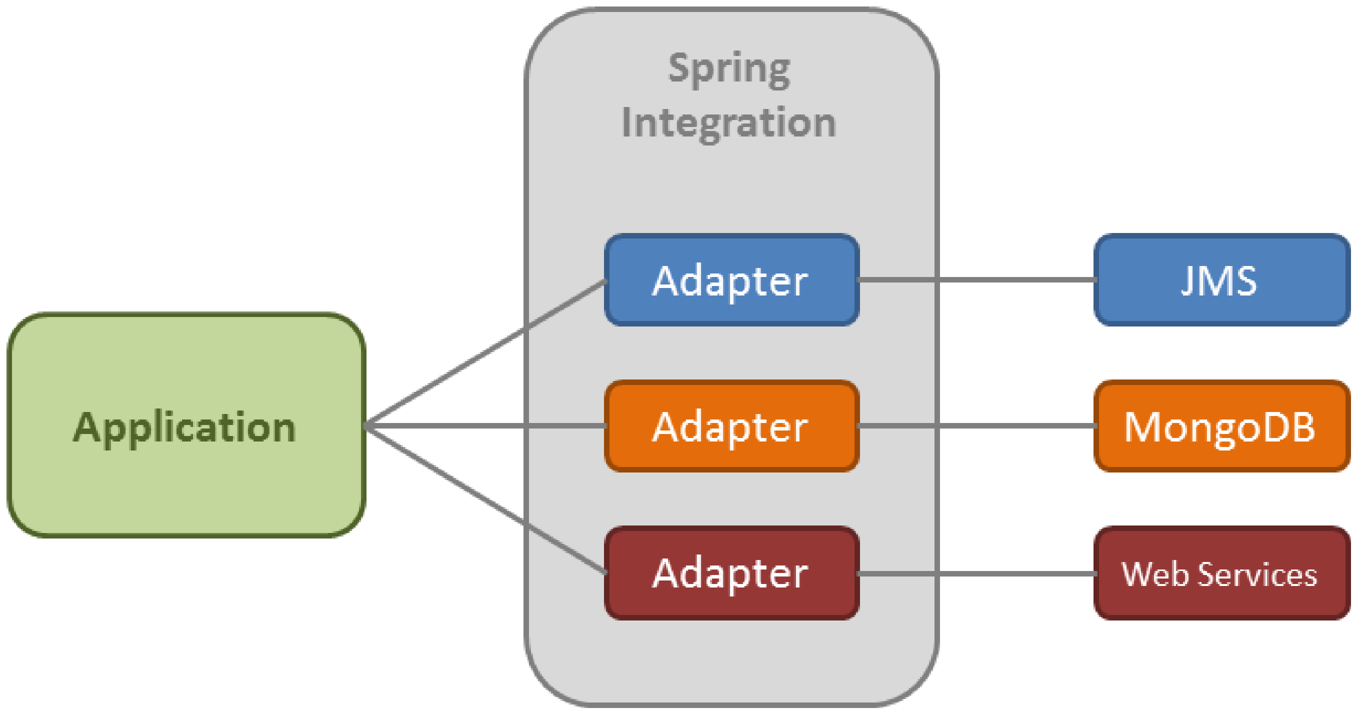


Figure 2.2: screenshot

Spring Integration targets the best practices of the Spring framework like programming with interfaces or composition over inheritance technique. Its main benefits are:

- Loose coupling among components.
- Event oriented architecture.
- The integration logic (handled by the framework) is separated from the business logic.

During the next section, you will learn what the three basic concepts are in which this messaging system is based.

2.3 Core concepts of Spring Integration messaging system

The basic concepts of a message-driven architecture are: message, message channel and message endpoint.

The API is pretty simple:

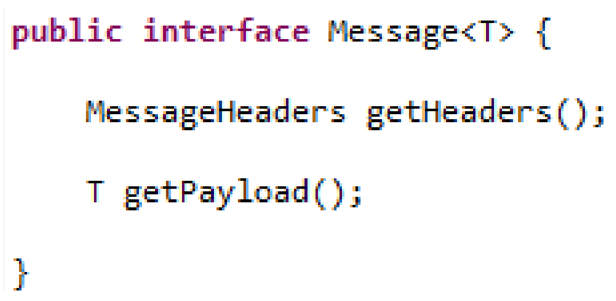
- A **message** is sent to an **endpoint**
- **Endpoints** are connected among them through **MessageChannels**
- An **endpoint** can receive **messages** from a **MessageChannel**

2.3.1 Message

A message contains the information that will be shared among the different components of the application, or sent to an external system. But, what is a message? A message is structured as follows:

```
 .screenshot
image::images/ch2_pic3.png["screenshot",link="http://academy.javacodegeeks.com/wp- ↩
    content/uploads/2014/04/ch2_pic3.png"]
```

As you can see in the following snippet, a message is an interface with GenericMessage as its main implementation (also provided by the framework):



```
public interface Message<T> {

    MessageHeaders getHeaders();

    T getPayload();

}
```

Figure 2.3: screenshot

- **Header**: 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.

If you want to create a message, you have two choices. The first one involves using a builder class (MessageBuilder).

```
Message<String> message = MessageBuilder
                .withPayload("my message payload")
                .setHeader("key1", "value1")
                .setHeader("key2", "value2")
                .build();
```

You have to set the payload and required headers before building it, since once the message is created you won't be able to do it unless you create a new message.

The other option is by using the implementation provided by the framework:

```
Map<String, Object> headers = new HashMap<>();
headers.put("key1", "value1");
headers.put("key2", "value2");

Message<String> message = new GenericMessage<String>("my message payload", headers);
```

2.3.2 Message Channel

A message channel is a pipe that connects endpoints, and where messages travel through. Producers send messages to a channel and consumers receive them from a channel. With this mechanism, you don't need any kind of broker.

A message channel can also be used as an interception point or for message monitoring.

```
 .screenshot
image::images/ch2_pic5.png["screenshot",link="http://academy.javacodegeeks.com/wp- ↩
    content/uploads/2014/04/ch2_pic5.png"]
```

Depending on how a message is consumed, message channels are classified as follows:

2.3.2.1 Point-to-point

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:

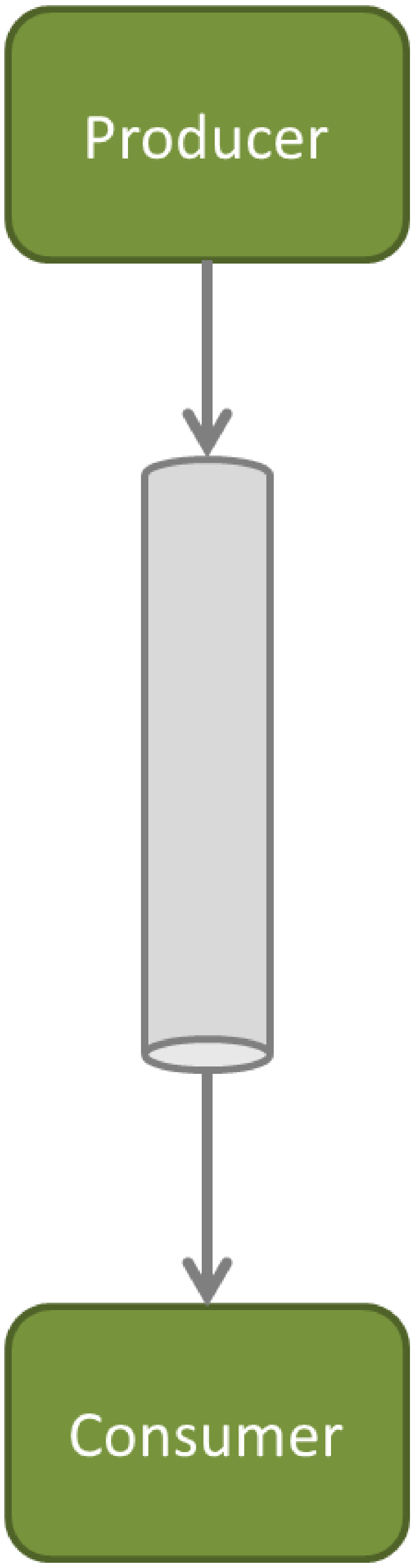


Figure 2.4: screenshot

- **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).
- **QueueChannel**: Implements PollableChannel. There's one endpoint connected to the channel, no subscribers. This communication is asynchronous; the receiver will retrieve the message through a different thread. How it works:
  - The producer sends the message to the channel.
  - The channel queues the message.
  - The consumer actively retrieves the message (active receiver).
- **ExecutorChannel**: Implements `SubscribableChannel`. Sending is delegated to a TaskExecutor. This means that the send() method will not block.
- **PriorityChannel**: Implements `PollableChannel`. Similar to the QueueChannel but messages are ordered by priority instead of FIFO.
- **RendezvousChannel**: Implements `PollableChannel`. Similar to the QueueChannel but with zero capacity. The producer will block until the receiver invokes its receive() method.

2.3.2.2 Publish-subscribe

The channel can have several endpoints subscribed to it. Thus, the message will be handled by different receivers.

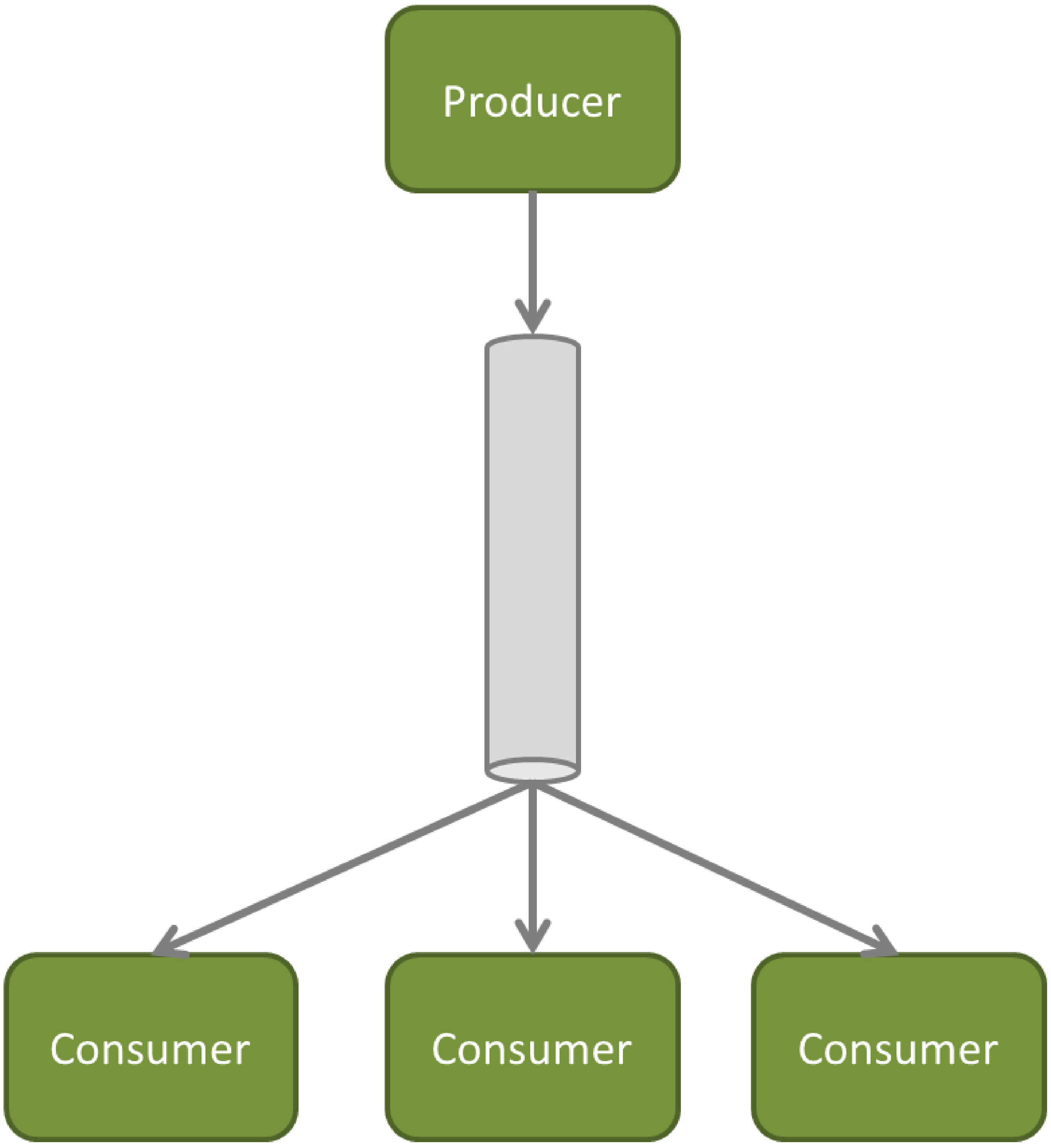


Figure 2.5: screenshot

- **PublishSubscribeChannel**: Implements `SubscribableChannel`. Subscribed receivers can be invoked consecutively through the producer's thread. If we specify a TaskExecutor, receivers will be invoked in parallel through different threads.

2.3.2.3 Temporary channels

This is a special type of channel which is created automatically by endpoints that have no output channel explicitly defined. The channel created is a point-to-point anonymous channel. You can see it defined in the message header under the name `replyChannel`.

These types of channels are automatically deleted once the response is sent. It is recommended that you don't explicitly define an output channel if you don't need it. The framework will handle it for you.

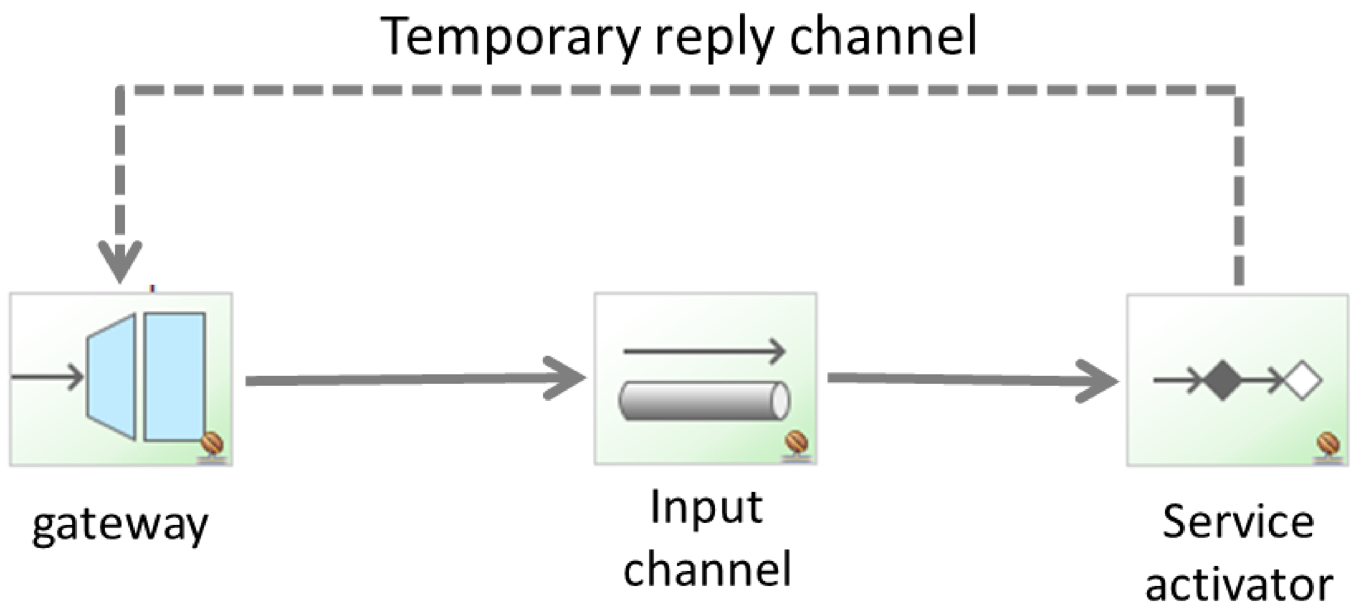


Figure 2.6: screenshot

2.3.3 Message Endpoint

Its target is to connect in a non-invasive manner, the application with the messaging framework. If you are familiar with Spring MVC, the endpoint will deal with messages the same way that an MVC controller deals with an HTTP request. The endpoint will be mapped to a message channel in the same way the MVC controller is mapped to a URL pattern.

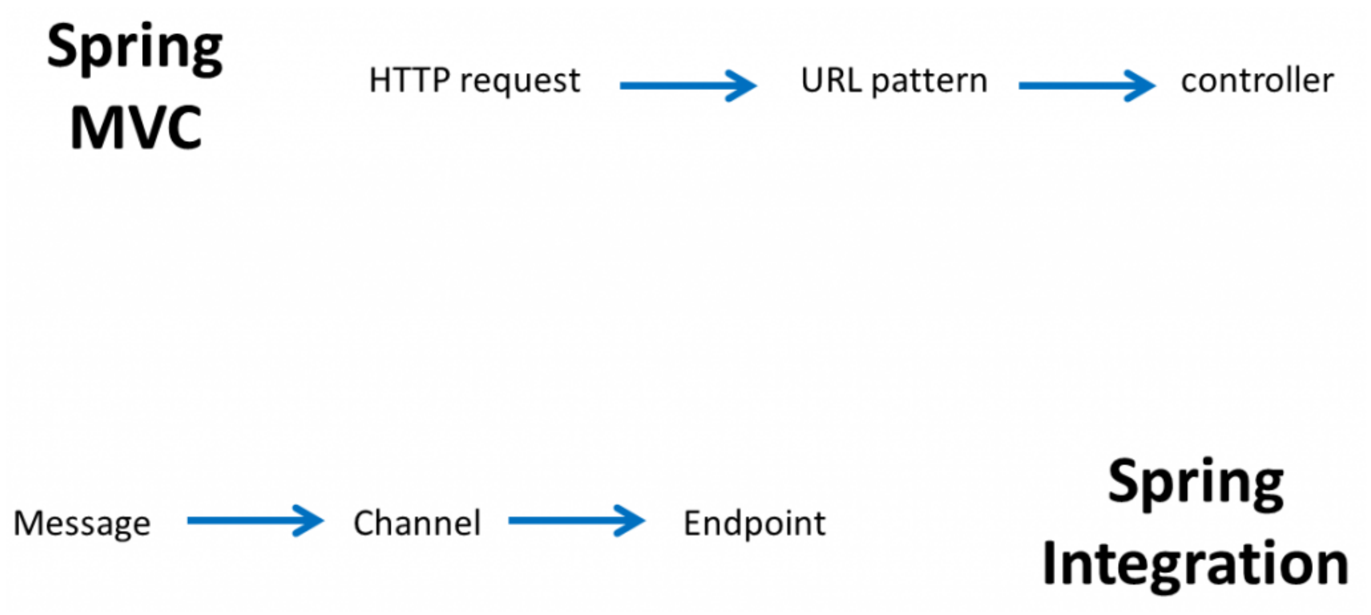


Figure 2.7: screenshot

Below is a list with a brief description of the available message endpoints:

- Channel adapter: Connects the application to an external system (unidirectional).
- Gateway: Connects the application to an external system (bidirectional).
- Service Activator: Can invoke an operation on a service object.
- Transformer: Converts the content of a message.
- Filter: Determines if a message can continue its way to the output channel.
- Router: Decides to which channel the message will be sent.
- Splitter: Splits the message in several parts.
- Aggregator: Combines several messages into a single one.

The next section of this tutorial explains each one of these endpoints.

2.4 Components

In this section you are going to learn what the different endpoints are and how you can use them in Spring Integration.

2.4.1 Channel Adapters

The channel adapter is the endpoint that allows your application to connect with external systems. If you take a look at the reference you will see the provided types like connecting to JMS queues, MongoDB databases, RMI, web services, etc.

There are four types of 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.

2.4.2 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 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.

Let's take a look at a couple of examples:

Assuming that we have the following model:

```java
public class Order implements Serializable {
    private static final long serialVersionUID = 1L;

    private int id;
    private String description;

    public Order() {}

    public Order(int id, String description) {
        this.id = id;
        this.description = description;
    }

    @Override
    public String toString() {
        return String.valueOf(this.getId());
    }

    //Setters & Getters
}
```

When this is sent to a message channel named “requestChannel”, the following code snippet will automatically transform the Order instance into a String by calling its toString() method:

```xml
<int:object-to-string-transformer input-channel="requestChannel" output-channel=" ↩
    transformedChannel"/>
```

The resulting String will be sent to the output channel named `transformedChannel`.

If you need a more customized transformation, you can implement your own transformer, which is a common bean. You will need to specify the referred bean in the transformer element as follows:

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

The transformer will invoke the “transform” method of a bean named “myTransformer”. This bean is shown below:

```
@Component("myTransformer")
public class MyTransformer {

    public Order transform(Order requestOrder) {
        return new Order(requestOrder.getId(), requestOrder.getDescription()+"_modified");
    }
}
```

In this example, the `method` attribute of the transformer element is not necessary, since the transformer only has one method. If it had several methods, you would need to set the “method” attribute to tell the framework which method to invoke. Or if you prefer annotations, you could specify the method using @Transformer annotation at method level:

```
@Component("myTransformer")
public class MyTransformer {

    @Transformer
    public Order transform(Order requestOrder) {
        return new Order(requestOrder.getId(), requestOrder.getDescription()+"_modified");
    }

    public Order doOtherThings(Order requestOrder) {
        //do other things
    }
}
```

2.4.3 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. Valid orders will be sent to the output channel:

```
<int:filter ref="myFilter" method="filterInvalidOrders" input-channel="requestChannel"  ↩
    output-channel="filteredChannel"/>
```

A filter method returns a boolean type. If it returns false, the message will be discarded:

```
@Component("myFilter")
public class MyFilter {

    public boolean filterInvalidOrders(Order order) {
        if (order == null || "invalid order".equals(order.getDescription())) {
            return false;
        }

        return true;
    }
}
```

As with the transformer, the `method` attribute will only be necessary if more than one method is defined in the filter bean. To specify the method you want to invoke, use the @Filter annotation:

```
@Filter
public boolean filterInvalidOrders(Order order) {
```

Spring Expression Language

If your filter is going to be very simple, you can skip any Java class to implement a filter. You can define a filter by using SpEL. For example, the following code snippet will implement the same filter as above but without java code:

```
<int:filter expression="!payload.description.equals('invalid order')" input-channel=" ↩
    requestChannel" output-channel="filteredChannel"/>
```

Discarding messages

With the default configuration, discarded messages are simply silently dropped. We can change that and, if we decide to do this we have two options:

We may don't want to lose any message. In that case we can throw an exception:

```
<int:filter expression="!payload.description.equals('invalid order')" input-channel=" ↩
    requestChannel" output-channel="filteredChannel"
        throw-exception-on-rejection="true"/>
```

We want to register all dropped messages. We can configure a discard channel:

```
<int:filter expression="!payload.description.equals('invalid order')" input-channel=" ↩
    requestChannel" output-channel="filteredChannel"
        discard-channel="discardedOrders"/>
```

2.4.4 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:

```
<int:payload-type-router input-channel="requestChannel">
    <int:mapping type="String" channel="stringChannel"/>
    <int:mapping type="Integer" channel="integerChannel"/>
</int:payload-type-router>
```

You can check the full list here.

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.

```
<int:router ref="myRouter" input-channel="requestChannel" default-output-channel=" ↩
    genericOrders"/>
```

The router implementation contains a method that returns the name of the message channel to where the message will be redirected:

```
@Component("myRouter")
public class MyRouter {

    public String routeOrder(Order order) {
        String returnChannel = "genericOrders";

        if (order.getDescription().startsWith("US-")) {
            returnChannel = "usOrders";
        }
        else if (order.getDescription().startsWith("EU-")) {
            returnChannel = "europeOrders";
        }
        return returnChannel;
    }
}
```

If you have several methods, you can use the `@Router` annotation:

```
@Router
public String routeOrder(Order order) {
```

In the same way as with the filter, you could route messages based on a Spring Expression Language.

2.4.5 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.

You will better see this with an example:

We are going to modify the order example so the Splitter receives an order package. This package contains several related orders that the splitter will separate. The splitter takes an order package and returns a list of orders:

```
<int:splitter input-channel="requestChannel" ref="mySplitter" output-channel="splitChannel" ↩
    />
```

The splitter implementation is very simple:

```
@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).

Following with the example, there is an aggregator endpoint which is connected to the "splitChannel" channel. This aggregator takes the list and combines its orders to form an order confirmation, adding the quantity of each order:

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

<int:aggregator ref="myAggregator" input-channel="splitChannel" output-channel=" ↩
    outputChannel"/>
```

`The aggregator implementation:`

```
@Component("myAggregator")
public class MyAggregator {

    public OrderConfirmation confirmOrders(List<Order> orders) {
        int total = 0;

        for (Order order:orders) {
            total += order.getQuantity();
        }

        OrderConfirmation confirmation = new OrderConfirmation("3");
        confirmation.setQuantity(total);

        return confirmation;
    }
}
```

2.4.5.1 Correlation and release strategies

When a message is split by the splitter endpoint, two headers are set:

- MessageHeaders.CORRELATION_ID
- MessageHeaders.SEQUENCE_SIZE

These headers are used by the aggregator endpoint to be able to combine messages correctly. It will hold messages until a set of messages with the same correlation id is ready. And when will it be ready? It will be ready when the sequence size is reached.

Correlation strategy Allow to group messages. By default, it will group all messages with the same value in `CORRELATION_ID` header. There are several strategies to choose from.

Release strategy By default, a group of messages will be considered complete when its size reaches the value specified by the message header `SEQUENCE_SIZE`.

2.4.6 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.

Pollers are used to specify how the active consumer retrieves these messages. Here are a couple of examples:

Basic poller configuration

It will poll the message channel in a one second interval

```
<int:service-activator method="processOrder" input-channel="pollableChannel" ref=" ↩
    orderProcessor">
    <int:poller fixed-rate="1000"/>
</int:service-activator>
```

Poller configured using a Cron expression

It will poll the message channel every 30 minutes

```
<int:service-activator method="processOrder" input-channel="pollableChannel" ref=" ↩
    orderProcessor">
    <int:poller cron="0 0/30 * * * ?"/>
</int:service-activator>
```

One thing to take into account is that if a consumer is connected to a pollable channel, it will need a poller. If not, an exception will be raised. If you don't want to configure a poller for each active consumer, you can define a default poller:

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

Don't forget to set the `default` and `id` attributes.

2.4.7 Messaging bridge

This type of endpoint connects two message channels or two channel adapters. For example, you can connect a `SubscribableChannel` channel to a `PollableChannel` channel.

Here is a sample:

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

<int:bridge input-channel="requestChannel" output-channel="pollableChannel"/>

<int:channel id="pollableChannel">
    <int:queue capacity="5"/>
</int:channel>

<int:service-activator method="processOrder" input-channel="pollableChannel" ref=" ↩
    orderProcessor"/>

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

In this example, the messaging bridge receives a message from the input channel and it publishes it to an output channel. In this case, we have a service activator connected to the output channel. The order processor (service activator) will be polling the message channel in one second intervals.

2.4.8 Message Handler Chain

The message handler chain is used to simplify the configuration when you have several message handlers connected in a linear way. The following example shows you a messaging configuration that will be simplified with the handler chain:

```
<int:channel id="requestChannel"/>
<int:channel id="responseChannel"/>

<int:filter ref="myFilter" method="filterInvalidOrders" input-channel="requestChannel"  ↩
    output-channel="filteredChannel"/>

<int:channel id="filteredChannel"/>

<int:transformer ref="myTransformer" method="transform"
    input-channel="filteredChannel" output-channel="transformedChannel"/>

<int:channel id="transformedChannel"/>

<int:service-activator method="processOrder" input-channel="transformedChannel" ref=" ↩
    orderProcessor" output-channel="responseChannel"/>
```

The message goes through a filter, then it will reach a transformer, and finally the message will be processed by a service activator. Once done, the message will be sent to the output channel “responseChannel”.

Using a message filter chain, the configuration will be as simplified as this:

```
<int:channel id="requestChannel"/>
<int:channel id="responseChannel"/>

<int:chain input-channel="requestChannel" output-channel="responseChannel">
    <int:filter ref="myFilter" method="filterInvalidOrders"/>
    <int:transformer ref="myTransformer" method="transform"/>
    <int:service-activator ref="orderProcessor" method="processOrder"/>
</int:chain>
```

2.5 Synchronous and asynchronous communication

As explained in the first tutorial of this course, communication can be performed synchronously or asynchronously. This section shows how to change this communication.

2.5.1 Message channels

Depending on how you configure message channels, messages will be retrieved synchronously or asynchronously. There are not many things to change, just the configuration.

For example, imagine we have a point-to-point direct channel like the one below:

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

The message sent to this channel will immediately be delivered to the passive consumer (subscriber). If a response is expected, the sender will wait until it is sent to him. In order to change this we just need to add a queue:

```
<int:channel id="requestChannel">
    <int:queue capacity="5"/>
</int:channel>
```

That’s it. Now the channel can queue up to five messages. The consumer will actively retrieve the message queued in this channel from a different thread than the sender.

Now, what about publish-subscribe channels? Let’s take a similar example by configuring a synchronous channel:

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

In this case we will change its behavior by using a task executor:

```
<int:publish-subscribe-channel id="mySubscribableChannel" task-executor="myTaskExecutor"/>

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

2.5.2 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.

The interface

All invocations to its `sendOrder` method will be caught by the gateway. See that there's no implementation of this interface. The gateway will wrap it to intercept those calls.

```
public interface OrderService {
    @Gateway
    public OrderConfirmation sendOrder(Order order);
}
```

The configuration

The gateway is linked to the interface in order to intercept its calls and send the message into the messaging system.

```
<int:gateway default-request-channel="requestChannel"
    service-interface="xpadro.spring.integration.service.OrderService"/>

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

The test

The service interface (the gateway) is injected to the application. A call to the `"sendOrder"` method will send the Order object to the messaging system, wrapped into a message.

```
@Autowired
private OrderService service;

@Test
public void testSendOrder() {
    OrderConfirmation confirmation = service.sendOrder(new Order(3, "a correct order"));
    Assert.assertNotNull(confirmation);
    Assert.assertEquals("confirmed", confirmation.getId());
}
```

In this other example, the test class will block until an order confirmation is sent back. Now we are going to configure it to make it asynchronous:

The interface

The only change here is to return a Future

```
public interface OrderService {
    @Gateway
    public Future<OrderConfirmation> sendFutureOrder(Order order);
}
```

The test

Now the test must handle a Future object that will be returned from the gateway.

```
@Autowired
private OrderService service;

@Test
public void testSendCorrectOrder() throws ExecutionException {
    Future<OrderConfirmation> confirmation = service.sendFutureOrder(new Order(3, "a ←
        correct order"));
    OrderConfirmation orderConfirmation = confirmation.get();
    Assert.assertNotNull(orderConfirmation);
    Assert.assertEquals("confirmed", orderConfirmation.getId());
}
```

2.6 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.

Global error channel

This channel is a publish-subscribe channel. This means we can subscribe our own endpoints to this channel and receive any error that is raised. In fact, Spring Integration already subscribes an endpoint: a logging handler. This handler logs the payload of any message sent to the global error channel.

To subscribe another endpoint in order to handle the exception, we just need to configure it as follows:

```
<int:service-activator input-channel="errorChannel" ref="myExceptionHandler" method=" ↩
    handleInvalidOrder"/>

<bean id="myExceptionHandler" class="xpadro.spring.integration.activator.MyExceptionHandler ↩
    "/>
```

The `handleInvalidOrder` method of our service activator endpoint will receive the messaging exception:

```
public class MyExceptionHandler {
    @ServiceActivator
    public void handleInvalidOrder(Message<MessageHandlingException> message) {
        //Retrieve the failed order (payload of the source message)
        Order requestedOrder = (Order) message.getPayload().getFailedMessage().getPayload() ↩
            ;

        //Handle exception
        ...
    }
}
```

Chapter 3

Spring Integration and Web Services

3.1 Introduction

In this tutorial you are going to see the first example of an application enhanced with Spring Integration. In order to accomplish it, this example will focus on the integration with external web services.

First, I will explain what are the necessary adapters that will allow us to invoke a web service from Spring Integration. Next, we will go through a brief explanation of a Spring Web Services project, which will be the external web service that will be invoked from our application. Finishing with the main part of the tutorial, we will implement an application that will invoke the web service.

To conclude the tutorial, we will complete our application with some features provided by Spring Integration, like adding time-outs, using interceptors and learning how to retry a failed invocation.

This tutorial is composed by the following sections:

- Introduction
- Explaining web service channel adapters
- Creating a Spring Web Services project
- Implementing a Spring Integration flow
- Adding client timeouts
- Using interceptors
- Web Service retry operations

3.2 Explaining web service channel adapters

The communication with external web services is done by Spring Integration with gateways. As explained in the previous tutorial, you can find two types of gateways: inbound and outbound. In this tutorial we will be using a special type of gateway: an outbound web service gateway. In this section we are going to focus on this type.

In order to use a web service gateway, you will need to specify a new namespace:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:int="http://www.springframework.org/schema/integration"
    xmlns:int-ws="http://www.springframework.org/schema/integration/ws"
    xsi:schemaLocation="
```

```
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/ ↩
            beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/integration http://www.springframework.org/ ↩
            schema/integration/spring-integration.xsd
        http://www.springframework.org/schema/integration/ws http://www.springframework.org ↩
            /schema/integration/ws/spring-integration-ws.xsd">
```

With the new namespace set, we can now use the web service gateway:

```
<int-ws:outbound-gateway id="aGateway"
        request-channel="requestChannel" reply-channel="responseChannel"
        uri="http://localhost:8080/spring-ws-tickets/tickets" marshaller="marshaller"
        unmarshaller="marshaller"/>
```

So, what is the behavior of this gateway? The execution of the flow would be as follows:

- A message is sent to the channel `requestChannel`.
- This message is then sent to the web service gateway, which is subscribed to the channel by setting its `request-channel` attribute.
- The gateway sends a request to an external web service, explained in the next section. The `uri` attribute specifies the destination.
- The gateway waits for the external web service until it returns a response.
- A response is returned and marshalled by the specified marshaller.
- The response is wrapped into a message and sent to the channel `responseChannel`, specified by the `reply-channel` attribute.

As you see, you just need to define the flow (request and reply channels) and where to call. The infrastructure details required to send the message are handled by Spring Integration.

3.2.1 Additional attributes

There are some more attributes that are available for customizing the invocation with the gateway. Below, there is a brief description of the main attributes:

- `*Destination provider:*` This can be used instead of providing the “uri” attribute. In this way, you can implement your own class that will resolve dynamically which endpoint is invoked. You should provide a bean with this interface:

```
public class MyDestinationProvider implements DestinationProvider {
    @Override
    public URI getDestination() {
        //resolve destination
    }
}
```

In the gateway definition, we can use this provider instead of providing the URI directly:

```
<int-ws:outbound-gateway id="aGateway"
    request-channel="requestChannel" reply-channel="responseChannel" destination-provider=" ↩
        myDestinationProvider"
    marshaller="marshaller" unmarshaller="marshaller"/>
```

- `*Message sender:*` Allows us to define a `WebServiceMessageSender`. We will use this to define a client timeout later in this tutorial.
- `*Interceptor/Interceptors:*` You can define client interceptors. This will also be explained in a later section of this tutorial.

3.2.2 Inbound web service gateway

This section is just a quick reference to the inbound service gateway in order to know how it generally works, since we will not use it in this tutorial.

This gateway will receive a request from an external service, wrap the request into a message and send it into our messaging system. When we have processed the request, a message will be sent back to the gateway in order to deliver a response that the web service is waiting for.

The syntax is similar to the outbound web service gateway:

```
<int-ws:inbound-gateway id="anotherGateway" request-channel="requestChannel"
    marshaller="marshaller" unmarshaller="marshaller"/>
```

As you may recall from previous tutorial, the response will reach the gateway through a temporary message channel. Don't explicitly define a channel if it is not necessary.

3.3 Creating a Spring Web Services project

This section explains the project that will expose the web service that will be used by our application. It consists in a web application implemented using the Spring Web Services project.

The application is quite simple. It consists in a service interface that allows the user to order tickets from a cinema service. When an order is requested, the service will process it and a `TicketConfirmation` is returned.

The diagram below shows how it is structured:

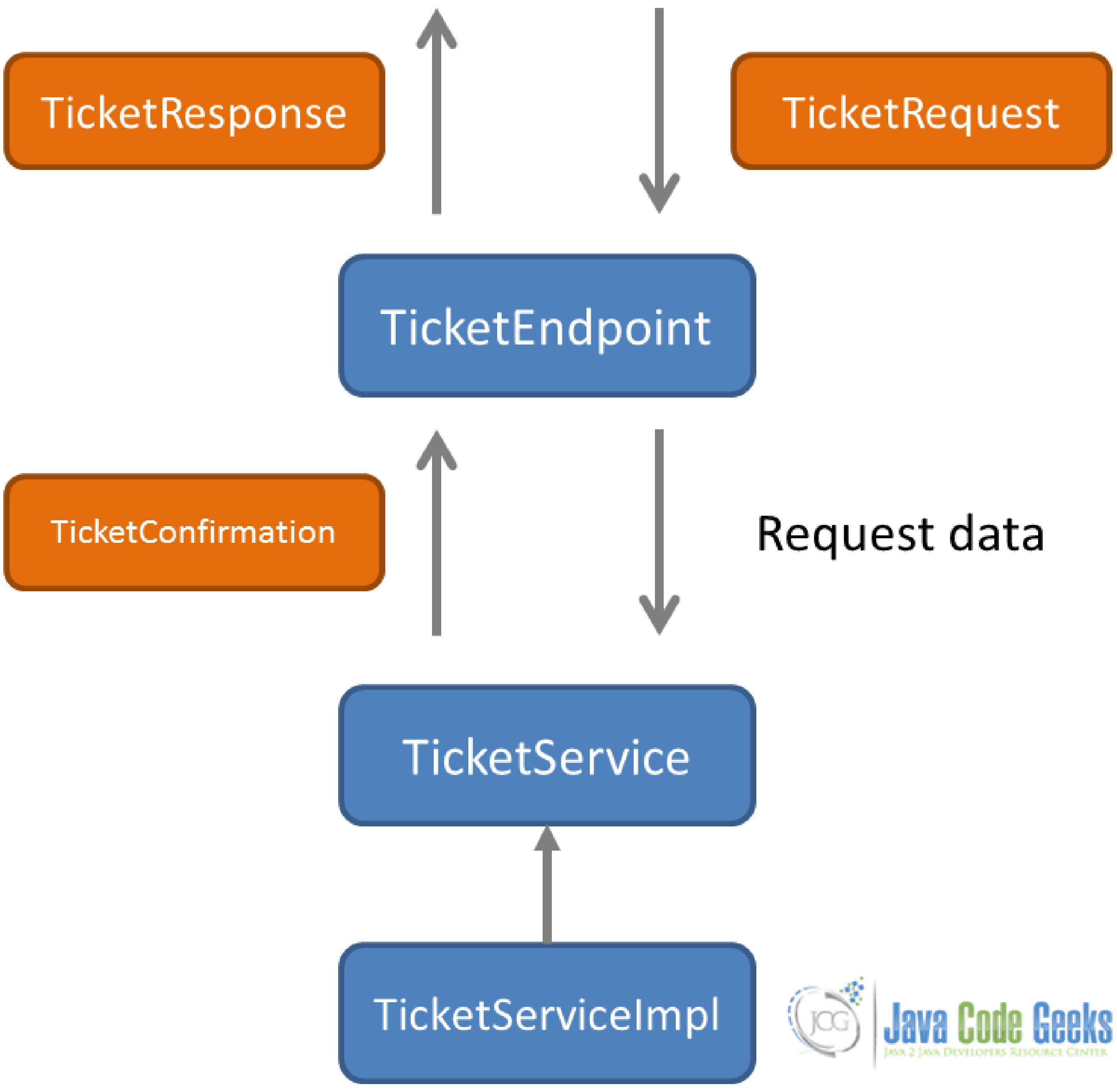


Figure 3.1: screenshot

We will explain it from bottom to top.

3.3.1 The ticket Service interface

Here is the service interface and implementation:

```
public interface TicketService {

    public TicketConfirmation order(String filmId, Date sessionDate, int quantity);
}
```

The implementation builds a `TicketConfirmation` instance from the data provided.

```
@Service
public class TicketServiceimpl implements TicketService {

    @Override
    public TicketConfirmation order(String filmId, Date sessionDate, int quantity) {
        float amount = 5.95f * quantity;
        TicketConfirmation confirmation = new TicketConfirmation(filmId, sessionDate,  ←
            quantity, amount);

        return confirmation;
    }
}
```

The `TicketConfirmation` object is an immutable class that will be used to read the confirmation data:

```
public final class TicketConfirmation {

        private String confirmationId;
        private String filmId;
        private int quantity;
        private Date sessionDate;
        private float amount;

        public TicketConfirmation(String filmId, Date sessionDate, int quantity, float  ←
            amount) {
                this.confirmationId = UUID.randomUUID().toString();
                this.filmId = filmId;
                this.sessionDate = new Date(sessionDate.getTime());
                this.quantity = quantity;
                this.amount = amount;
        }


        public String getConfirmationId() {
                return confirmationId;
        }

        public String getFilmId() {
                return filmId;
        }

        public int getQuantity() {
                return quantity;
        }

        public Date getSessionDate() {
                return new Date(sessionDate.getTime());
        }

        public float getAmount() {
                return amount;
        }
}
```

3.3.2 The ticket endpoint

The endpoint is responsible from receiving requests and delegating the order processing to the `Ticket` service:

```
@Endpoint
public class TicketEndpoint {

    @Autowired
    private TicketService ticketService;

    @PayloadRoot(localPart="ticketRequest", namespace="http://www.xpadro.spring.samples.com ↩
        /tickets")
    public @ResponsePayload TicketResponse order(@RequestPayload TicketRequest  ↩
        ticketRequest) throws InterruptedException {
        Calendar sessionDate = Calendar.getInstance();
        sessionDate.set(2013, 9, 26);

        TicketConfirmation confirmation = ticketService.order(
                ticketRequest.getFilmId(), DateUtils.toDate(ticketRequest.getSessionDate()) ↩
                    , ticketRequest.getQuantity().intValue());

        return buildResponse(confirmation);
    }

    private TicketResponse buildResponse(TicketConfirmation confirmation) {
        TicketResponse response = new TicketResponse();
        response.setConfirmationId(confirmation.getConfirmationId());
        response.setFilmId(confirmation.getFilmId());
        response.setSessionDate(DateUtils.convertDate(confirmation.getSessionDate()));
        BigInteger quantity = new BigInteger(Integer.toString(confirmation.getQuantity()));
        response.setQuantity(quantity);
        BigDecimal amount = new BigDecimal(Float.toString(confirmation.getAmount()));
        response.setAmount(amount);

        return response;
    }
}
```

The service will receive requests sent with the namespace "`http://www.xpadro.spring.samples.com/tickets`" and with a `ticketRequest` request element.

3.3.3 The service configuration

In the spring configuration we define the web service components:

```
<!-- Detects @Endpoint since it is a specialization of @Component -->
<context:component-scan base-package="xpadro.spring.ws"/>

<!-- detects @PayloadRoot -->
<ws:annotation-driven/>

<ws:dynamic-wsdl id="ticketDefinition" portTypeName="Tickets"
                 locationUri="http://localhost:8080/spring-ws-tickets">
    <ws:xsd location="/WEB-INF/schemas/xsd/ticket-service.xsd"/>
</ws:dynamic-wsdl>
```

The `web.xml` file exposes the `MessageDispatcherServlet`:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:xpadro/spring/ws/config/root-config.xml</param-value>
</context-param>

<listener>
```

```
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<servlet>
    <servlet-name>Ticket Servlet</servlet-name>
    <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet- ↩
        class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:xpadro/spring/ws/config/servlet-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>Ticket Servlet</servlet-name>
    <url-pattern>/tickets/*</url-pattern>
</servlet-mapping>
```

We now just need to deploy it into the server and it will be ready to serve ticket order requests.

3.4 Implementing a Spring Integration flow

Our Spring Integration application starts with a simple flow.

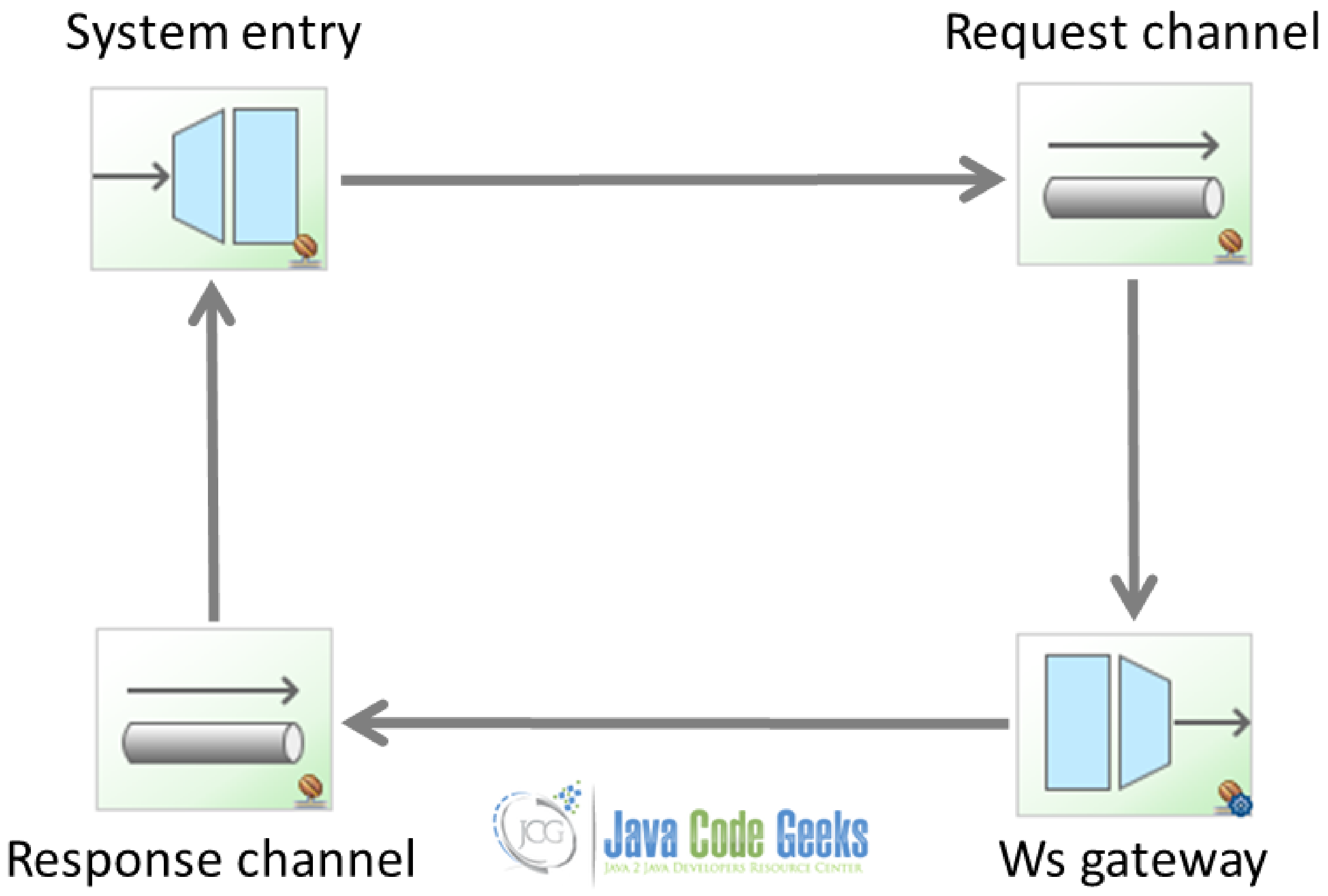


Figure 3.2: screenshot

The request message will come through the system entry gateway. The message will then be delivered to the web service outbound gateway that will send it to the endpoint and wait for the response. Once received, it will send the response through the response channel and back to the system entry gateway, which will then deliver it to the client.

The client application sends the `TicketRequest` to the `TicketService` interface. This interface is intercepted by the gateway. In this way, the `TicketRequest` object is wrapped into a Spring Integration message and sent to the messaging system.

```
public interface TicketService {
    /**
     * Entry to the messaging system.
     * All invocations to this method will be
     * intercepted and sent to the SI "system entry" gateway
     *
     * @param request
     */
    @Gateway
    public TicketResponse invoke(TicketRequest request);
}
```

Looking at the gateway configuration, we can see that we linked it to the `TicketService` interface:

```
<int:gateway id="systemEntry" default-request-channel="requestChannel"
    default-reply-channel="responseChannel"
    service-interface="xpadro.spring.integration.ws.gateway.TicketService" />
```

We have also defined the request and reply channels.

The request message will be sent to the `requestChannel` channel where a web service outbound gateway is subscribed:

```
<int-ws:outbound-gateway id="marshallingGateway"
    request-channel="requestChannel" reply-channel="responseChannel"
    uri="http://localhost:8080/spring-ws-tickets/tickets" marshaller="marshaller"
    unmarshaller="marshaller"/>
```

The `responseChannel` is configured as its reply channel, where the system entry gateway is subscribed. In this way, the client will receive the response.

The full flow is configured using direct channels. This means that the flow is synchronous; the client will block waiting the web service to respond:

```
<context:component-scan base-package="xpadro.spring.integration" />

<!-- Entry to the messaging system -->
<int:gateway id="systemEntry" default-request-channel="requestChannel" default-reply- ↩
    channel="responseChannel"
    service-interface="xpadro.spring.integration.ws.gateway.TicketService" />

<int:channel id="requestChannel"/>

<int-ws:outbound-gateway id="marshallingGateway"
    request-channel="requestChannel" reply-channel="responseChannel"
    uri="http://localhost:8080/spring-ws-tickets/tickets" marshaller="marshaller"
    unmarshaller="marshaller"/>

<oxm:jaxb2-marshaller id="marshaller" contextPath="xpadro.spring.integration.ws.types" />

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

The system is set; we didn't have to implement any Java class. All is configured through configuration.

Finishing with the example, let's see the test that executes this flow:

```
@ContextConfiguration({"classpath:xpadro/spring/integration/ws/test/config/int-ws-config. ↩
    xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TestInvocation {

    @Autowired
    private TicketService service;

    @Test
    public void testInvocation() throws InterruptedException, ExecutionException {
        TicketRequest request = new TicketRequest();
        request.setFilmId("aFilm");
        request.setQuantity(new BigInteger("3"));
        request.setSessionDate(DateUtils.convertDate(new Date()));

        TicketResponse response = service.invoke(request);

        assertNotNull(response);
        assertEquals("aFilm", response.getFilmId());
        assertEquals(new BigInteger("3"), response.getQuantity());
    }
}
```

During the next sections, we will add some features to this example application.

3.5 Adding client timeouts

Checking at the gateway's namespace, we can see that there's no configuration for setting the invocation timeout. Regardless of this, we can use a message sender.

A message sender is an implementation of the `WebServiceMessageSender`. One interesting implementation provided by the Spring Web Services project is the `HttpComponentsMessageSender` class. This class will allow us to add authentication or connection pooling to the invocation by internally using the `Apache HttpClient`. And what's more, we will also be able to define read and connection timeouts.

Following with the example, let's add it the timeouts.

First, we need to define a bean with the above mentioned class. This will be our message sender:

```
<bean id="messageSender" class="org.springframework.ws.transport.http. ↩
    HttpComponentsMessageSender">
    <property name="connectionTimeout" value="5000"/>
    <property name="readTimeout" value="10000"/>
</bean>
```

Next, we will configure the message sender in our web service gateway:

```
<int-ws:outbound-gateway id="marshallingGateway"
    request-channel="requestChannel" reply-channel="responseChannel"
    uri="http://localhost:8080/spring-ws-tickets/tickets" marshaller="marshaller"
    unmarshaller="marshaller" message-sender="messageSender"/>
```

That's it. Now, a `WebServiceIOException` will be thrown if the timeout is reached.

3.6 Using interceptors

Another feature included in the namespace of the web service gateway is the possibility to configure client interceptors. These client interceptors are a feature of the Spring Web Services project, and refer to endpoint interceptors on the client side. A `ClientInterceptor` implementation has the following methods:

```
public interface ClientInterceptor {

    boolean handleRequest(MessageContext messageContext) throws WebServiceClientException;

    boolean handleResponse(MessageContext messageContext) throws WebServiceClientException;

    boolean handleFault(MessageContext messageContext) throws WebServiceClientException;
}
```

- `handleRequest`: This method is invoked before the endpoint is called.
- `handleResponse`: This method is invoked after the endpoint has successfully returned.
- `handleFault`: If the endpoint throws a fault, this method is invoked.

Notice that these methods can manipulate a `MessageContext`, which contains the request and response.

Let's see this with an example. We are going to implement our custom client interceptor to intercept the invocation before calling the endpoint and we are going to change a request value.

The interceptor implements `ClientInterceptor` interface:

```
public class MyInterceptor implements ClientInterceptor {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Override
    public boolean handleRequest(MessageContext messageContext) throws  ↩
        WebServiceClientException {
        WebServiceMessage message = messageContext.getRequest();
        DOMSource source = (DOMSource) message.getPayloadSource();

        Node quantityNode = source.getNode().getAttributes().getNamedItem("quantity");
        String oldValue = quantityNode.getNodeValue();
        quantityNode.setNodeValue("5");

        logger.info("Before endpoint invocation. Changed quantity old value {} for {}",  ↩
            oldValue, 5);

        return true;
    }

    @Override
    public boolean handleResponse(MessageContext messageContext) throws  ↩
        WebServiceClientException {
        logger.info("endpoint invocation succeeds");

        return true;
    }

    @Override
    public boolean handleFault(MessageContext messageContext) throws  ↩
        WebServiceClientException {
        logger.info("endpoint returned a fault");

        return true;
    }
}
```

Now, we need to add our interceptor to the gateway configuration:

```
<int-ws:outbound-gateway id="marshallingGateway"
    request-channel="requestChannel" reply-channel="responseChannel"
    uri="http://localhost:8080/spring-ws-tickets/tickets" marshaller="marshaller"
    unmarshaller="marshaller" message-sender="messageSender" interceptor="myInterceptor" />

<bean id="myInterceptor" class="xpadro.spring.integration.ws.interceptor.MyInterceptor" />
```

The web service gateway namespace also allows us to define an `interceptors` attribute. This lets us to configure a list of client interceptors.

The test will validate that the request value has been modified:

```
@ContextConfiguration({"classpath:xpadro/spring/integration/ws/test/config/int-ws-config. ↩
    xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TestInvocation {

    @Autowired
    private TicketService service;

    @Test
    public void testInvocation() throws InterruptedException, ExecutionException {
        TicketRequest request = new TicketRequest();
        request.setFilmId("aFilm");
        request.setQuantity(new BigInteger("3"));
        request.setSessionDate(DateUtils.convertDate(new Date()));

        TicketResponse response = service.invoke(request);

        assertNotNull(response);
        assertEquals("aFilm", response.getFilmId());
        assertEquals(new BigInteger("5"), response.getQuantity());
    }
}
```

Before implementing your custom interceptor, take into account that the Spring Web Services project provides several implementations:

- `PayloadValidatingInterceptor`: Validates that the payload of the web service message by using a schema. If the validation is not passed, the processing will be cancelled.
- `Wss4jSecurityInterceptor`: Web service security endpoint interceptor based on Apache's WSS4J.
- `XwsSecurityInterceptor`: Web service security endpoint interceptor based on Sun's XML and Web Services Security package.

3.7 Web Service retry operations

Sometimes, we may want to invoke a service and it is temporarily down or maybe the service is online only on certain days. If this happens, we may want to retry the invocation later. Spring Integration offers the possibility to start retrying the service invocation until a condition is met. This condition may be that the service finally responded or we reached a maximum number of attempts. For this feature, Spring Integration offers a retry advice. This advice is backed by the Spring Retry project.

The retry advice is included in the web service outbound gateway. In this way, the gateway delegates the web service invocation to the retry advice. In case the service invocation fails, the advice will keep retrying the operation based on its configuration.

3.7.1 Defining the retry advice

We have to define a new bean with the `RequestHandlerRetryAdvice` class:

```
<bean id="retryAdvice" class="org.springframework.integration.handler.advice. ↩
    RequestHandlerRetryAdvice" >
    <property name="retryTemplate">
        <bean class="org.springframework.retry.support.RetryTemplate">
            <property name="backOffPolicy">
                <bean class="org.springframework.retry.backoff.FixedBackOffPolicy">
                    <property name="backOffPeriod" value="5000" />
                </bean>
            </property>
            <property name="retryPolicy">
                <bean class="org.springframework.retry.policy.SimpleRetryPolicy">
                    <property name="maxAttempts" value="5" />
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

We have defined an advice that, in case of a failed invocation, it will keep retrying every five seconds until the service responds or until it has tried five times. We will later see what are these policies defined in the advice.

3.7.2 Adding the advice to the gateway

Once the advice is defined, we need to include it into the gateway. The Spring Integration Web Services namespace already offers an element for that:

```
<int-ws:outbound-gateway id="marshallingGateway"
    request-channel="requestChannel" reply-channel="responseChannel"
    uri="http://localhost:8080/spring-ws-tickets/tickets" marshaller="marshaller"
    unmarshaller="marshaller" message-sender="messageSender" interceptor="myInterceptor" >

    <int-ws:request-handler-advice-chain>
        <ref bean="retryAdvice" />
    </int-ws:request-handler-advice-chain>
</int-ws:outbound-gateway>
```

We have integrated the advice into the gateway. Now, let's modify our example to see how this works.

3.7.3 Modifying the web service endpoint

We are going to modify our endpoint in order to keep failing until a specified number of retries is tried. In this case, two times until it returns a response.

```
@PayloadRoot(localPart="ticketRequest", namespace="http://www.xpadro.spring.samples.com/ ↩
    tickets")
public @ResponsePayload TicketResponse order(@RequestPayload TicketRequest ticketRequest)  ↩
    throws InterruptedException {
    Calendar sessionDate = Calendar.getInstance();
    sessionDate.set(2013, 9, 26);

    TicketConfirmation confirmation = ticketService.order(
            ticketRequest.getFilmId(), DateUtils.toDate(ticketRequest.getSessionDate()),  ↩
                ticketRequest.getQuantity().intValue());

    TicketResponse response = buildResponse(confirmation);
    retries++;
    if (retries < 3) {
        throw new RuntimeException("not enough retries");
    }
    else {
        retries = 0;
    }

    return response;
}
```

Now, we will launch the test and use our previously defined interceptor to see how it logs the attempts:

```
2014-03-26 08:24:50,535|AbstractEndpoint|started org.springframework.integration.endpoint. ↩
    EventDrivenConsumer@392044a1
2014-03-26 08:24:50,626|MyInterceptor|Before endpoint invocation. Changed quantity old  ↩
    value 3 for 5
2014-03-26 08:24:51,224|MyInterceptor|endpoint returned a fault
2014-03-26 08:24:56,236|MyInterceptor|Before endpoint invocation. Changed quantity old  ↩
    value 3 for 5
2014-03-26 08:24:56,282|MyInterceptor|endpoint returned a fault
2014-03-26 08:25:01,285|MyInterceptor|Before endpoint invocation. Changed quantity old  ↩
    value 3 for 5
2014-03-26 08:25:01,377|MyInterceptor|endpoint invocation succeeds
```

The gateway has kept trying the invocation until the service has responded, since the retry advice have a superior number of retry times (five).

3.7.4 Retry advice policies

The Spring Integration retry advice is backed up on Spring Retry project policies. These policies are explained below:

Back off policy

It establishes a period of time between retries or before the initial retry. The BackOffPolicy interface defines two methods:

<code>BackOffContext start(RetryContext context);

void backOff(BackOffContext backOffContext) throws BackOffInterruptedException;</code>

The `start` method allows defining an initial behavior. For example, an initial time delay. The `backoff` method allows defining a pause between retries.

The Spring Retry project provides several implementations of the back off policy:

- Stateless back off policies: Maintain no state between invocations.
  - `FixedBackOffPolicy`: It pauses for a specified time between retries. No initial delay is set.
  - `NoBackOffPolicy`: Retries are executed with no pause at all between them.
- Stateful back off policies: Maintain a state between invocations.
  - `ExponentialBackOffPolicy`: Starting with a specified amount of time, it will be multiplied on each invocation. By default it doubles the time. You can change the multiplier factor.
  - `ExponentialRandomBackOffPolicy`: Extends `ExponentialBackOffPolicy`. The multiplier is set in a random manner.

Retry policy

It allows defining how many times will the retry advice execute the web service invocation before giving up. The `RetryPolicy` interface defines several methods:

<code>boolean canRetry(RetryContext context);

RetryContext open(RetryContext parent);

void close(RetryContext context);

void registerThrowable(RetryContext context, Throwable throwable);</code>

The `canRetry` method returns if the operation can be retried. This could happen, for example, if we haven't reached the maximum number of retries. The `open` method is used to acquire all the necessary resources, to keep track of the number of attempts or if an exception was raised during the previous retry. The `registerThrowable` method is called after every failed invocation.

The Spring Retry project provides several implementations of the retry policy:

- `SimpleRetryPolicy`: Retries the invocation until a maximum number of retries is reached.
- `TimeoutRetryPolicy`: It will keep retrying until a timeout is reached. The timeout is started during the open method.
- `NeverRetryPolicy`: It just tries the invocation once.
- `AlwaysRetryPolicy`: canRetry method always returns true. It will keep retrying until the service responds.
- `ExceptionClassifierRetryPolicy`: It defines a different maximum number of attempts depending on the exception thrown.
- `CompositeRetryPolicy`: It contains a list of retry policies that will be executed in order.

3.7.5 Retry operations using a poller

Available retry policies are implemented using time delays, which are fine for most situations, but in this section we are going to implement a custom solution that will let us use a poller that will be configured using a Cron Expression.

Since the invocation may fail, the gateway won't return the result. We will make the flow asynchronous in order to allow the client to send the service request and proceed. In this way, the flow will keep retrying from another thread until a result is handled by a service activator or the retry limit is reached.

The gateway is as follows:

```java
public interface AsyncTicketService {
    @Gateway
    public void invoke(TicketRequest request);
}
```

The gateway does not define a reply channel, since no response will be sent. Since it is an asynchronous request, the request channel contains a queue. This will allow its consumer to actively poll the message from another thread:

```xml
<int:gateway id="systemEntry" default-request-channel="requestChannel"
    service-interface="xpadro.spring.integration.ws.gateway.AsyncTicketService" />

<int:channel id="requestChannel">
    <int:queue />
</int:channel>
```

We have included a poller to the web service gateway, since now it will poll for messages:

```
<int-ws:outbound-gateway id="marshallingGateway"
    request-channel="requestChannel" reply-channel="responseChannel"
    uri="http://localhost:8080/spring-ws-tickets/tickets" marshaller="marshaller"
    unmarshaller="marshaller" interceptor="myInterceptor" >

    <int:poller fixed-rate="500" />
</int-ws:outbound-gateway>
```

The previous invocation can result in three different results: a correct invocation, a failed invocation that needs to be retried, and a final failed invocation that needs to be logged.

Service invocation correctly invoked

We have a service activator subscribed to the response channel. It is a simple example, so it will just log the result:

```
<!-- Service is running - Response received -->
<int:channel id="responseChannel" />
<int:service-activator ref="clientServiceActivator" method="handleServiceResult" input- ↩
    channel="responseChannel" />
```

Service invocation failed. Retry the operation

If something went wrong, and since it is an asynchronous request, the exception will be wrapped into a `MessageHandlingException` and sent to the error channel, which is configured by default by Spring Integration.

At this point, we have a router subscribed to the error channel. This router handles how many retries have been tried and based on this, it will redirect the failed message to the appropriate channel. If the operation is to be retried, it will send the message to the retry channel:

```
@Component("serviceRouter")
public class ServiceRouter {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private int maxRetries = 3;
    private int currentRetries;

    public String handleServiceError(Message<?> msg) {
        logger.info("Handling service failure");

        if (maxRetries > 0) {
            currentRetries++;
            if (currentRetries > maxRetries) {
                logger.info("Max retries [{}] reached", maxRetries);
                return "failedChannel";
            }
        }

        logger.info("Retry number {} of {}", currentRetries, maxRetries);
        return "retryChannel";
    }
}
```

The configuration of the router is shown below:

```
<!-- Service invocation failed -->
<int:router ref="serviceRouter" method="handleServiceError" input-channel="errorChannel"/>
<int:channel id="retryChannel" />
<int:channel id="failedChannel" />
```

Next, we have these endpoints that are explained below:

```
<!-- Retry -->
<int:service-activator ref="clientServiceActivator" method="retryFailedInvocation" input- ↩
    channel="retryChannel" />
```

```
<int:inbound-channel-adapter id="retryAdapter" channel="requestChannel"
    ref="clientServiceActivator" method="retryInvocation" auto-startup="false">

    <int:poller cron="0/5 * * * * *"/>
</int:inbound-channel-adapter>

<!-- Log failed invocation -->
<int:service-activator ref="clientServiceActivator" method="handleFailedInvocation" input- ↩
    channel="failedChannel" />
```

The `retryAdapter` inbound channel adapter will keep polling the request channel but, notice that the attribute `auto-startup` is set to false. This means this adapter is disabled until someone activates it. We need to do this or otherwise it would start polling from the beginning, and we want to activate it only if a failed invocation occurs.

The service activator will start or stop the adapter depending on the result of the service invocation. When it fails, it will start the adapter in order to start retrying. If the maximum number of retries is reached, the router will redirect the message to the failed channel where the service activator will disable the adapter to stop it from polling. If the invocation finally succeeds, it will log the message and stop the adapter.

```
@Component("clientServiceActivator")
public class ClientServiceActivator {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    @Qualifier("retryAdapter")
    private AbstractEndpoint retryAdapter;

    private Message<?> message;

    public void handleServiceResult(Message<?> msg) {
        logger.info("service successfully invoked. Finishing flow");
        retryAdapter.stop();
    }

    public void retryFailedInvocation(Message<?> msg) {
        logger.info("Service invocation failed. Activating retry trigger...");
        MessageHandlingException exc = (MessageHandlingException) msg.getPayload();
        this.message = exc.getFailedMessage();
        retryAdapter.start();
    }

    public Message<?> retryInvocation() {
        logger.info("Retrying service invocation...");

        return message;
    }

    public void handleFailedInvocation(MessageHandlingException exception) {
        logger.info("Maximum number of retries reached. Finishing flow.");
        retryAdapter.stop();
    }
}
```

The test class has been modified in order to not to expect a result:

```
@ContextConfiguration({"classpath:xpadro/spring/integration/ws/test/config/int-ws-async- ↩
    config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TestAsyncInvocation {

    @Autowired
    private AsyncTicketService service;

    @Test
    public void testInvocation() throws InterruptedException, ExecutionException {
        TicketRequest request = new TicketRequest();
        request.setFilmId("aFilm");
        request.setQuantity(new BigInteger("3"));
        request.setSessionDate(DateUtils.convertDate(new Date()));

        service.invoke(request);
        Thread.sleep(80000);
    }
}
```

That’s it. Obviously there is no need to implement all this flow knowing that we can use the retry advice of the Spring Retry project, but the purpose of this example is to gain more knowledge on how to build more complex flows, using activation and deactivation of adapters, router redirections and other features to accomplish your needs.

3.8 Download source code

You can download the source code regarding the spring integration and web services from here: Spring_Integration_Sample.zip and Spring_WS_Sample.zip

Chapter 4

Enterprise Messaging

4.1 Introduction

This tutorial is focused on explaining how we can integrate our application with Spring Integration and JMS messaging. For this purpose, I will first show you how to install Active MQ, which will be our broker during this tutorial. Next sections will show examples of sending and receiving JMS messages by using the Spring Integration JMS channel adapters. Following these examples, we will see some ways of customizing these invocations by configuring message conversion and destination resolution.

The last part of this tutorial shows briefly how to use Spring Integration with the AMQP protocol. It will go through the installation of RabbitMQ and end up with a basic example of messaging.

This tutorial is composed by the following sections:

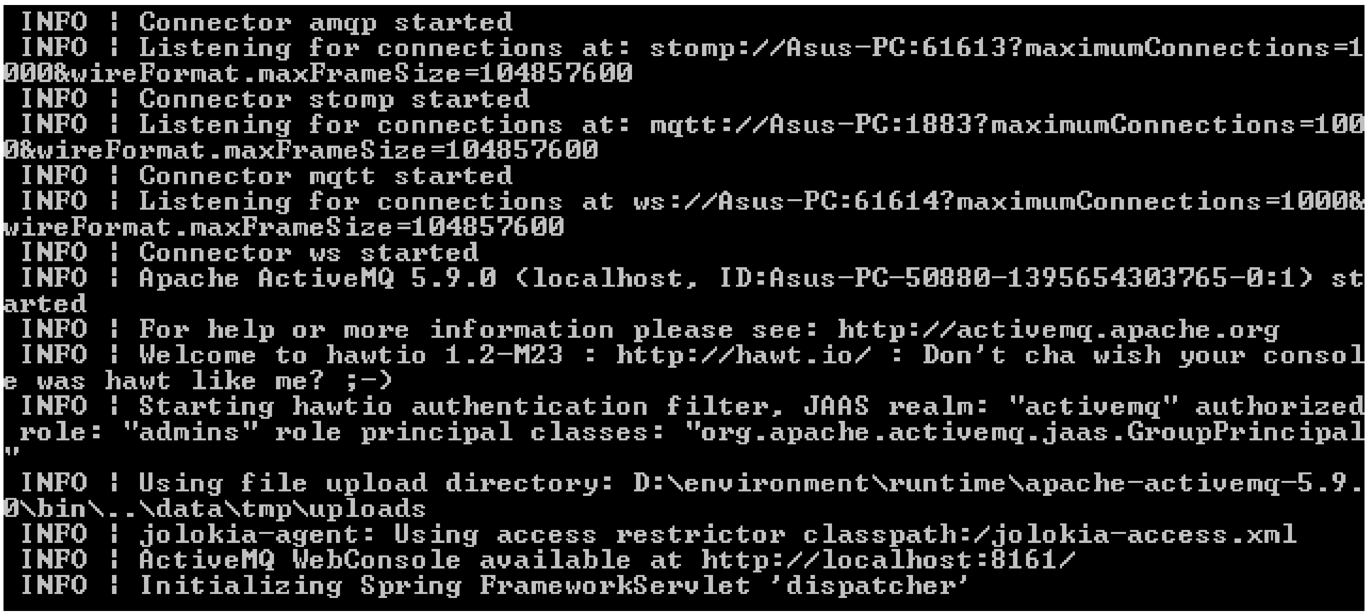
- Introduction
- Preparing the environment
- JMS Adapters: Reception
- JMS Adapters: Sending
- Using Gateways
- Message conversion
- JMS backed message channels
- Dynamic destination resolution
- AMQP integration

4.2 Preparing the environment

If you want to send messages through JMS, you will first need a broker. The examples included in this tutorial are executed over Active MQ, an open source messaging broker. In this section I will help you install the server and implement a simple Spring application that will test it is all set correctly. The explanation is based on a Windows system. If you already have a server installed just skip this section.

The first step is to download the Apache MQ server from Apache.org. Once, downloaded, just extract it into a folder of your choice.

To start the server you just need to execute the file activemq which is located in the apache-activemq-5.9.0bin folder.



```
 INFO | Connector amqp started
 INFO | Listening for connections at: stomp://Asus-PC:61613?maximumConnections=1
000&wireFormat.maxFrameSize=104857600
 INFO | Connector stomp started
 INFO | Listening for connections at: mqtt://Asus-PC:1883?maximumConnections=100
0&wireFormat.maxFrameSize=104857600
 INFO | Connector mqtt started
 INFO | Listening for connections at ws://Asus-PC:61614?maximumConnections=1000&
wireFormat.maxFrameSize=104857600
 INFO | Connector ws started
 INFO | Apache ActiveMQ 5.9.0 (localhost, ID:Asus-PC-50880-1395654303765-0:1) st
arted
 INFO | For help or more information please see: http://activemq.apache.org
 INFO | Welcome to hawtio 1.2-M23 : http://hawt.io/ : Don't cha wish your consol
e was hawt like me? ;-)
 INFO | Starting hawtio authentication filter, JAAS realm: "activemq" authorized
 role: "admins" role principal classes: "org.apache.activemq.jaas.GroupPrincipal
"
 INFO | Using file upload directory: D:\environment\runtime\apache-activemq-5.9.
0\bin\..\data\tmp\uploads
 INFO | jolokia-agent: Using access restrictor classpath:/jolokia-access.xml
 INFO | ActiveMQ WebConsole available at http://localhost:8161/
 INFO | Initializing Spring FrameworkServlet 'dispatcher'
```

Figure 4.1: screenshot

Ok, the server is running. Now we just need to implement the application. We are going to create a producer, a consumer, a spring configuration file and a test.

The producer

You can use any Java class instead of my `TicketOrder` object.

```java
public class JmsProducer {
    @Autowired
    @Qualifier("jmsTemplate")
    private JmsTemplate jmsTemplate;

    public void convertAndSendMessage(TicketOrder order) {
        jmsTemplate.convertAndSend(order);
    }

    public void convertAndSendMessage(String destination, TicketOrder order) {
        jmsTemplate.convertAndSend(destination, order);
    }
}
```

The consumer

```java
public class SyncConsumer {
    @Autowired
    private JmsTemplate jmsTemplate;

    public TicketOrder receive() {
        return (TicketOrder) jmsTemplate.receiveAndConvert("test.sync.queue");
    }
}
```

Spring configuration file

```xml
<bean id="consumer" class="xpadro.spring.integration.consumer.SyncConsumer"/>
<bean id="producer" class="xpadro.spring.integration.producer.JmsProducer"/>
```

```
<!-- Infrastructure -->
<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<bean id="cachingConnectionFactory" class="org.springframework.jms.connection. ↩
    CachingConnectionFactory">
    <property name="targetConnectionFactory" ref="connectionFactory"/>
</bean>

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="cachingConnectionFactory"/>
    <property name="defaultDestination" ref="syncTestQueue"/>
</bean>

<!-- Destinations -->
<bean id="syncTestQueue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="test.sync.queue"/>
</bean>
```

The test

```
@ContextConfiguration(locations = {"/xpadro/spring/integration/test/jms-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TestJmsConfig {
    @Autowired
    private JmsProducer producer;

    @Autowired
    private SyncConsumer consumer;

    @Test
    public void testReceiving() throws InterruptedException, RemoteException {
        TicketOrder order = new TicketOrder(1, 5, new Date());
        //Sends the message to the jmsTemplate's default destination
        producer.convertAndSendMessage(order);

        Thread.sleep(2000);

        TicketOrder receivedOrder = consumer.receive();
        assertNotNull(receivedOrder);
        assertEquals(1, receivedOrder.getFilmId());
        assertEquals(5, receivedOrder.getQuantity());
    }
}
```

If the test passes, you have all correctly set. We can now move to the next section.

4.3 JMS Adapters: Reception

Spring Integration provides several adapters and gateways to receive messages from a JMS queue or topic. These adapters are briefly discussed below:

- **Inbound Channel Adapter**: It internally uses a JmsTemplate to actively receive messages from a JMS queue or topic.
- **Message Driven Channel Adapter**: It internally uses a Spring MessageListener container to passively receive messages.

4.3.1 Inbound Channel Adapters: Active reception

This section explains how to use the first adapter described in the previous section.

The JMS inbound channel adapter actively polls a queue to retrieve messages from it. Since it uses a poller, you will have to define it in the Spring configuration file. Once the adapter has retrieved a message, it will be sent into the messaging system through the specified message channel. We can then process the message using endpoints like transformers, filters, etc... Or we can send it to a service activator.

This example retrieves a ticket order message from a JMS queue and sends it to a service activator, which will process it and confirm the order. The order is confirmed by sending it to some kind of repository which has a simple List with all the registered orders.

We are using the same producer as in the section “2 preparing the environment”:

```
<bean id="producer" class="xpadro.spring.integration.producer.JmsProducer"/>

<!-- Infrastructure -->
<!-- Connection factory and jmsTemplate configuration -->
<!-- as seen in the second section -->

<!-- Destinations -->
<bean id="toIntQueue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="int.sync.queue"/>
</bean>
```

The test will use the producer to send a message to the “toIntQueue”. Now we are going to set up the Spring Integration configuration:

integration-jms.xml

```
<context:component-scan base-package="xpadro.spring.integration"/>

<int-jms:inbound-channel-adapter id="jmsAdapter" destination="toIntQueue" channel=" ↩
    jmsChannel"/>

<int:channel id="jmsChannel"/>

<int:service-activator method="processOrder" input-channel="jmsChannel" ref=" ↩
    ticketProcessor"/>

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

The JMS inbound channel adapter will use the defined poller to retrieve messages from the “toIntQueue”. You have to configure a poller for the adapter or it will throw a runtime exception. In this case, we have defined a default poller. This means that any endpoint that needs a poller will use this one. If you don’t configure a default poller, you will need to define a specific poller for each endpoint that retrieves messages actively.

The consumer

The service activator is just a bean (auto detected by component scan):

```
@Component("ticketProcessor")
public class TicketProcessor {
    private static final Logger logger = LoggerFactory.getLogger(TicketProcessor.class);
    private static final String ERROR_INVALID_ID = "Order ID is invalid";

    @Autowired
    private OrderRepository repository;

    public void processOrder(TicketOrder order) {
        logger.info("Processing order {}", order.getFilmId());

        if (isInvalidOrder(order)) {
            logger.info("Error while processing order [{}]", ERROR_INVALID_ID);
            throw new InvalidOrderException(ERROR_INVALID_ID);
        }

        float amount = 5.95f * order.getQuantity();

        TicketConfirmation confirmation = new TicketConfirmation("123", order.getFilmId(),  ↩
            order.getOrderDate(), order.getQuantity(), amount);
        repository.confirmOrder(confirmation);
    }

    private boolean isInvalidOrder(TicketOrder order) {
        if (order.getFilmId() == -1) {
            return true;
        }
        return false;
    }
}
```

In the previous code snippet, the `processOrder` method receives a `TicketOrder` object and directly processes it. However, you could instead define Message<?> or Message<TicketOrder> in order to receive the message. In this way, you will have both access to the payload of the message and to its headers.

Notice also, that the method returns void. We don't need to return anything since the messaging flow ends here. If needed, you could also define a reply channel to the service adapter and return the confirmation. Additionally, we would then subscribe an endpoint or a gateway to this reply channel in order to send the confirmation to another JMS queue, send it to a web service or store it to a database, for example.

Finally, let's take a look at the test to see how it is all executed:

```
@ContextConfiguration(locations = {"/xpadro/spring/integration/test/jms-config.xml",
        "/xpadro/spring/integration/test/int-jms-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TestIntegrationJmsConfig {
    @Autowired
    private JmsProducer producer;

    @Autowired
    private OrderRepository repository;

    @Test
    public void testSendToIntegration() throws InterruptedException, RemoteException {
        TicketOrder order = new TicketOrder(1, 5, new Date());
        //Sends the message to the jmsTemplate's default destination
        producer.convertAndSendMessage("int.sync.queue", order);

        Thread.sleep(4000);

        assertEquals(1, repository.getConfirmations().size());
        assertNotNull(repository.getConfirmations().get(0));
        TicketConfirmation conf = repository.getConfirmations().get(0);
        assertEquals("123", conf.getId());
    }
}
```

I have put a `Thread.sleep` of four seconds to wait for the message to be sent. We could have used a while loop to check if the message have been received until a timeout is reached.

4.3.2 Inbound Channel Adapters: Passive reception

This second part of the JMS reception section uses a message driven channel adapter. In this way, as soon as a message is sent to the queue, it will be delivered to the adapter, without the need of using pollers. It is the message channel we delivers the message to its subscribers.

The example is very similar to the one seen in the previous section. I will just show what the changes are made in the configuration.

The only thing I have changed from the previous example is the spring integration configuration:

```
<context:component-scan base-package="xpadro.spring.integration"/>

<int-jms:message-driven-channel-adapter id="jmsAdapter" destination="toIntQueue" channel=" ↩
    jmsChannel" />

<int:channel id="jmsChannel"/>

<int:service-activator method="processOrder" input-channel="jmsChannel" ref=" ↩
    ticketProcessor"/>
```

I deleted the poller and changed the JMS inbound adapter for a message driven channel adapter. That's it; the adapter will passively receive messages and deliver them to the `jmsChannel`.

Take into account that the message listener adapter needs at least one of the following combinations:

- A message listener container.
- A connection factory and a destination.

In our example, we have used the second option. The destination is specified in the adapter configuration and the connection factory is defined in the jms-config file, which is also imported by the test.

4.4 JMS Adapters: Sending

In the previous section we have seen how to receive messages sent to a JMS queue by an external system. This section shows you outbound channel adapters, which let you to send JMS messages outside of our system.

In contrast to inbound adapters, there is only one type of outbound adapter. This adapter uses a `JmsTemplate` internally to send the message, and in order to configure this adapter you will need to specify at least one of the following:

- A JmsTemplate.
- A connection factory and a destination.

As in the inbound example, we are using the second option to send a message to a JMS queue. The configuration is as follows:

For this example, we are going to create a new queue to the jms configuration (jms-config.xml). This is where our Spring Integration application will send the message to:

```
<bean id="toJmsQueue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="to.jms.queue"/>
</bean>
```

Ok, now we configure the integration configuration with the JMS outbound adapter:

```
<context:component-scan base-package="xpadro.spring.integration"/>

<int:gateway default-request-channel="requestChannel"
    service-interface="xpadro.spring.integration.service.TicketService"/>

<int:channel id="requestChannel"/>

<int-jms:outbound-channel-adapter id="jmsAdapter" channel="requestChannel" destination=" ↩
    toJmsQueue"/>
```

We are using a gateway as an entry to our messaging system. The test will use this interface to send a new `TicketOrder` object. The gateway will receive the message and place it into the `requestChannel` channel. Since it is a direct channel, it will be sent to the JMS outbound channel adapter.

The adapter receives a Spring Integration Message. It then can send the message in two ways:

- Convert the message into a JMS message. This is done by setting the adapter's attribute "extract-payload" to true, which is the default value. This is the option we have used in the example.
- Send the message as it is, a Spring Integration Message. You can accomplish this by setting the "extract-payload" attribute to false.

This decision depends on what type of system is expecting your message. If the other application is a Spring Integration application, you can use the second approach. Otherwise, use the default. In our example, there's a simple Spring JMS application on the other side. Thus, we have to choose the first option.

Continuing with our example, we now have a look at the test, which uses the gateway interface to send a message and a custom consumer to receive it. In this test, the consumer will play the role of a JMS application which uses a `jmsTemplate` to retrieve it from the JMS queue:

```
@ContextConfiguration(locations = {"/xpadro/spring/integration/test/jms-config.xml",
        "/xpadro/spring/integration/test/int-jms-out-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TestIntegrationJmsOutboundConfig {
    @Autowired
    private SyncConsumer consumer;

    @Autowired
    private TicketService service;

    @Test
    public void testSendToJms() throws InterruptedException, RemoteException {
        TicketOrder order = new TicketOrder(1, 5, new Date());
        service.sendOrder(order);

        TicketOrder receivedOrder = consumer.receive("to.jms.queue");
        assertNotNull(receivedOrder);
        assertEquals(1, receivedOrder.getFilmId());
        assertEquals(5, receivedOrder.getQuantity());
    }
}
```

4.5 Using Gateways

Additionally to channel adapters, Spring Integration provides inbound and outbound gateways. As you may recall from previous tutorials, gateways provide a bidirectional communication with external systems, meaning send and receive or receive and reply operations. In this case, it allows request or retry operations.

In this section, we are going to see an example using a JMS outbound gateway. The gateway will send a JMS message to a queue and wait for a reply. If no reply is sent back, the gateway will throw a MessageTimeoutException.

Spring Integration configuration

```
<context:component-scan base-package="xpadro.spring.integration"/>

<int:gateway id="inGateway" default-request-channel="requestChannel"
    service-interface="xpadro.spring.integration.service.TicketService"/>

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

```
<int-jms:outbound-gateway id="outGateway" request-destination="toAsyncJmsQueue"
    request-channel="requestChannel" reply-channel="jmsReplyChannel"/>

<int:channel id="jmsReplyChannel"/>

<int:service-activator method="registerOrderConfirmation" input-channel="jmsReplyChannel"  ↩
    ref="ticketProcessor"/>
```

The flow is as follows:

- A TicketOrder wrapped into a Spring Integration Message will enter the messaging system through the “inGateway” gateway.
- The gateway will place the message into the “requestChannel” channel.
- The channel sends the message to its subscribed endpoint, the JMS outbound gateway.
- The JMS outbound gateway extracts the payload of the message and wraps it to a JMS message.
- The gateway sends the message and waits for a reply.
- When the reply comes, in the form of a TicketConfirmation wrapped into a JMS message, the gateway will get the payload and wrap it into a Spring Integration message.
- The message is sent to the “jmsReplyChannel” channel, where a service activator (TicketProcessor) will process it and register to our OrderRepository.

The order processor is quite simple. It receives the TicketConfirmation and adds it to the ticket repository:

```
@Component("ticketProcessor")
public class TicketProcessor {
    @Autowired
    private OrderRepository repository;

    public void registerOrderConfirmation(TicketConfirmation confirmation) {
        repository.confirmOrder(confirmation);
    }
}
```

The test

```
@RunWith(SpringJUnit4ClassRunner.class)
public class TestIntegrationJmsOutGatewayConfig {
    @Autowired
    private OrderRepository repository;

    @Autowired
    private TicketService service;

    @Test
    public void testSendToJms() throws InterruptedException, RemoteException {
        TicketOrder order = new TicketOrder(1, 5, new Date());
        service.sendOrder(order);

        Thread.sleep(4000);

        assertEquals(1, repository.getConfirmations().size());
        assertNotNull(repository.getConfirmations().get(0));
        TicketConfirmation conf = repository.getConfirmations().get(0);
        assertEquals("321", conf.getId());

    }
}
```

The external system

To fully understand the example, I will show you what happens when the message is delivered to the JMS queue.

Listening to the queue where the message has been sent by Spring Integration, there's a listener `asyncConsumer`:

```
<bean id="toAsyncJmsQueue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="to.async.jms.queue"/>
</bean>

<!-- Listeners -->
<jms:listener-container connection-factory="connectionFactory">
    <jms:listener destination="to.async.jms.queue" ref="asyncConsumer"/>
</jms:listener-container>
```

The listener receives the message, creates a new message with the ticket confirmation and replies. Notice that we have to set the correlation ID of the reply message with the same value as the request message. This will allow the client to know which message we are responding to. Also, we are setting the destination to the reply channel configured in the request message.

```
@Component("asyncConsumer")
public class AsyncConsumer implements MessageListener {
    @Autowired
    private JmsTemplate template;

    @Override
    public void onMessage(Message order) {
        final Message msgOrder = order;
        TicketOrder orderObject;
        try {
            orderObject = (TicketOrder) ((ObjectMessage) order).getObject();
        } catch (JMSException e) {
            throw JmsUtils.convertJmsAccessException(e);
        }
        float amount = 5.95f * orderObject.getQuantity();
        TicketConfirmation confirmation = new TicketConfirmation("321", orderObject. ↩
            getFilmId(), orderObject.getOrderDate(), orderObject.getQuantity(), amount);

        try {
            template.convertAndSend(msgOrder.getJMSReplyTo(), confirmation, new  ↩
                MessagePostProcessor() {
                public Message postProcessMessage(Message message) throws JMSException {
                    message.setJMSCorrelationID(msgOrder.getJMSCorrelationID());

                    return message;
                }
            });
        } catch (JmsException | JMSException e) {
            throw JmsUtils.convertJmsAccessException((JMSException) e);
        }
    }
}
```

4.6 Message conversion

Both message channel adapters and gateways use a message converter to convert incoming messages to Java types, or the opposite way. A converter must implement the MessageConverter interface:

```
public interface MessageConverter {

    <P> Message<P> toMessage(Object object);
    <P> Object fromMessage(Message<P> message);

}
```

Spring Integration comes with two implementations of the `MessageConverter` interface:

MapMessageConverter

Its `fromMessage` method creates a new HashMap with two keys:

- payload: The value is the payload of the message (`message.getPayload`).
- headers: The value is another HashMap with all the headers from the original message.

"toMessage" method expects a Map instance with the same structure (payload and headers keys) and constructs a Spring Integration message.

SimpleMessageConverter

This is the default converter used by the adapters and gateways. You can see from the source code that it converts from/to an Object:

```
public Message<?> toMessage(Object object) throws Exception {
    if (object == null) {
        return null;
    }
    if (object instanceof Message<?>) {
        return (Message<?>) object;
    }
    return MessageBuilder.withPayload(object).build();
}

public Object fromMessage(Message<?> message) throws Exception {
    return (message != null) ? message.getPayload() : null;
}
```

Anyway, if you need your own implementation, you can specify your custom converter at the channel adapter or gateway configuration. For example, using a gateway:

```
<int-jms:outbound-gateway id="outGateway" request-destination="toAsyncJmsQueue"
    request-channel="requestChannel" reply-channel="jmsReplyChannel"
    message-converter="myConverter"/>
```

Just remember that your converter should implement MessageConverter:

```
@Component("myConverter")
public class MyConverter implements MessageConverter {
```

4.7 JMS backed message channels

Channel adapters and gateways are used to communicate with external systems. JMS backed message channels are used to send and receive JMS messages between consumers and producers that are located within the same application. Although we can still use channel adapters in this situation, it is much simpler to use a JMS channel. The difference from an integration message channel is that the JMS channel will use the JMS broker to send the message. This means that the message will not be just stored in an in-memory channel. Instead, it will be sent to the JMS provider, making it possible to also use transactions. If you use transactions, it will work as follows:

- The producer sending the message to the JMS backed channel will not write it if the transaction is rolled back.

- The consumer subscribed to the JMS backed channel will not remove the message from it if the transaction is rolled back.

For this feature, Spring Integration provides both channels: point to point and publish/subscribe channels. They are configured below:

point to point direct channel

```
<int-jms:channel id="jmsChannel" queue="myQueue"/>
```

publish/subscribe channel

```
<int-jms:publish-subscribe-channel id="jmsChannel" topic="myTopic"/>
```

In the following example, we can see a simple application with two endpoints communicating with each other using a JMS backed channel.

Configuration

Messages sent to the messaging system (`TicketOrder` objects) arrive to a service activator, the ticket processor. This processor then sends the order (`sendJMS`) to the JMS backed message. Subscribed to this channel, there is the same processor that will receive the message (`receiveJms`), process it creating a `TicketConfirmation` and registering it to the ticket repository:

```
<context:component-scan base-package="xpadro.spring.integration"/>

<int:gateway default-request-channel="requestChannel"
    service-interface="xpadro.spring.integration.service.TicketService"/>

<int:channel id="requestChannel"/>

<int:service-activator method="sendJms" input-channel="requestChannel" output-channel=" ↩
    jmsChannel" ref="ticketJmsProcessor"/>

<int-jms:channel id="jmsChannel" queue="syncTestQueue"/>

<int:service-activator method="receiveJms" input-channel="jmsChannel" ref=" ↩
    ticketJmsProcessor"/>
```

The processor

Implements both methods: `sendJms` and `receiveJms`:

```
@Component("ticketJmsProcessor")
public class TicketJmsProcessor {
    private static final Logger logger = LoggerFactory.getLogger(TicketJmsProcessor.class);

    @Autowired
    private OrderRepository repository;

    public TicketOrder sendJms(TicketOrder order) {
        logger.info("Sending order {}", order.getFilmId());
        return order;
    }

    public void receiveJms(TicketOrder order) {
        logger.info("Processing order {}", order.getFilmId());

        float amount = 5.95f * order.getQuantity();

        TicketConfirmation confirmation = new TicketConfirmation("123", order.getFilmId(),  ↩
            order.getOrderDate(), order.getQuantity(), amount);
        repository.confirmOrder(confirmation);
    }
}
```

The test

```
@ContextConfiguration(locations = {"/xpadro/spring/integration/test/jms-config.xml",
                "/xpadro/spring/integration/test/int-jms-jms-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TestIntegrationJmsToJmsConfig {
        @Autowired
        private OrderRepository repository;

        @Autowired
        private TicketService service;

        @Test
        public void testSendToJms() throws InterruptedException, RemoteException {
                TicketOrder order = new TicketOrder(1, 5, new Date());
                service.sendOrder(order);

                Thread.sleep(4000);

                assertEquals(1, repository.getConfirmations().size());
                assertNotNull(repository.getConfirmations().get(0));
                TicketConfirmation conf = repository.getConfirmations().get(0);
                assertEquals("123", conf.getId());

        }
}
```

JMS backed channels offer different possibilities like configuring the queue name instead of the queue reference or using a destination resolver:

```
<int-jms:channel id="jmsChannel" queue-name="myQueue"
    destination-resolver="myDestinationResolver"/>
```

4.8 Dynamic destination resolution

A destination resolver is a class which allows us to resolve destination names into JMS destinations. Any destination resolver must implement the following interface:

```
public interface DestinationResolver {
    Destination resolveDestinationName(Session session, String destinationName, boolean  ↩
        pubSubDomain)
            throws JMSException;
}
```

Destination resolvers can be specified on JMS channel adapters, JMS gateways and JMS backed channels. If you don't configure a destination resolver explicitly, Spring will use a default implementation, which is DynamicDestinationResolver. This resolver is explained below as the other implementations provided by Spring:

- **DynamicDestinationResolver**: Resolves destination names as dynamic destinations by using the standard JMS Session.createTopic and Session.createQueue methods.
- **BeanFactoryDestinationResolver**: It will look up at the Spring context for a bean with a name like the destination name provided and expecting it to be of type javax.jms.Destination. If it can't find it, it will throw a DestinationResolutionException.
- **JndiDestinationResolver**: It will assume that the destination name is a JNDI location.

If we don't want to use the default dynamic resolver, we can implement a custom resolver and configure it in the desired endpoint. For example, the following JMS backed channel uses a different implementation:

```
<int-jms:channel id="jmsChannel" queue-name="myQueue"
    destination-resolver="myDestinationResolver"/>
```

4.9 AMQP integration

4.9.1 Installation

To install and start the RabbitMQ server you just need to follow the steps described below. If you already have the server installed, just skip this section.

- The first step is to install erlang, required for the RabbitMQ server. Go to the following URL, download your system version and install it:
  - http://www.erlang.org/download.html
- The next step is to download and install RabbitMQ. Download version 3.2.4 release if you want to use the same version as used in this tutorial:
  - http://www.rabbitmq.com/
- Now, open the command prompt. If you are a windows user, you can go directly by clicking on the start menu and selecting RabbitMQ Command Prompt in the RabbitMQ folder.
- Activate the management plugin

```
> rabbitmq-plugins enable rabbitmq_management
```

- Start the server

```
> rabbitmq-server.bat
```

Ok, now we will test that RabbitMQ is correctly installed. Go to http://localhost:15672 and log using 'guest' as both username and password. If you are using a version prior to 3.0, then the port will be 55672.

If you see the web UI then is all set.

4.9.2 Demo application

In order to use AMQP with Spring Integration, we will need to add the following dependencies to our pom.xml file:

Spring AMQP (for rabbitMQ)

```
<dependency>
   <groupId>org.springframework.amqp</groupId>
   <artifactId>spring-rabbit</artifactId>
   <version>1.3.1.RELEASE</version>
</dependency>
```

Spring Integration AMQP endpoints

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-amqp</artifactId>
    <version>3.0.2.RELEASE</version>
</dependency>
```

Now we are going to create a new configuration file amqp-config.xml that will contain rabbitMQ configuration (like the jms-config for JMS we used previously in this tutorial).

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rabbit="http://www.springframework.org/schema/rabbit"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/ ↩
            beans/spring-beans.xsd
        http://www.springframework.org/schema/rabbit http://www.springframework.org/schema/ ↩
            rabbit/spring-rabbit.xsd">

    <rabbit:connection-factory id="connectionFactory" />

    <rabbit:template id="amqpTemplate" connection-factory="connectionFactory" />

    <rabbit:admin connection-factory="connectionFactory" />

    <rabbit:queue name="rabbit.queue" />

    <rabbit:direct-exchange name="rabbit.exchange">
        <rabbit:bindings>
            <rabbit:binding queue="rabbit.queue" key="rabbit.key.binding" />
        </rabbit:bindings>
    </rabbit:direct-exchange>
</beans>
```

The next file is the Spring Integration file which contains channels and channel adapters:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:int="http://www.springframework.org/schema/integration"
    xmlns:int-amqp="http://www.springframework.org/schema/integration/amqp"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/ ↩
            beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema ↩
            /context/spring-context-3.0.xsd
        http://www.springframework.org/schema/integration http://www.springframework.org/ ↩
            schema/integration/spring-integration.xsd
        http://www.springframework.org/schema/integration/amqp http://www.springframework. ↩
            org/schema/integration/amqp/spring-integration-amqp.xsd">

    <context:component-scan base-package="xpadro.spring.integration.amqp"/>

    <int:gateway default-request-channel="requestChannel"
        service-interface="xpadro.spring.integration.amqp.service.AMQPService"/>

    <int:channel id="requestChannel"/>

    <int-amqp:outbound-channel-adapter
        channel="requestChannel" amqp-template="amqpTemplate" exchange-name="rabbit. ↩
            exchange"
        routing-key="rabbit.key.binding"/>

    <int-amqp:inbound-channel-adapter channel="responseChannel"
        queue-names="rabbit.queue" connection-factory="connectionFactory" />

    <int:channel id="responseChannel"/>

    <int:service-activator ref="amqpProcessor" method="process" input-channel=" ↩
        responseChannel"/>
```

```
</beans>
```

The flow is as follows:

- The test application sends a message, which will be a simple String, to the gateway.
- From the gateway, it will reach the outbound channel adapter through the “requestChannel” channel.
- The outbound channel adapter sends the message to the “rabbit.queue” queue.
- Subscribed to this “rabbit.queue” queue we have configured an inbound channel adapter. It will receive messages sent to the queue.
- The message is sent to the service activator through the “responseChannel” channel.
- The service activator simply prints the message.

The gateway that serves as an entry point to the messaging system contains a single method:

```
public interface AMQPService {
    @Gateway
    public void sendMessage(String message);
}
```

The service activator amqpProcessor is very simple; it receives a message and prints its payload:

```
@Component("amqpProcessor")
public class AmqpProcessor {

    public void process(Message<String> msg) {
        System.out.println("Message received: "+msg.getPayload());
    }
}
```

To finish with the example, here is the application that initiates the flow by invoking the service wrapped by the gateway:

```
@ContextConfiguration(locations = {"/xpadro/spring/integration/test/amqp-config.xml",
        "/xpadro/spring/integration/test/int-amqp-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TestIntegrationAMQPConfig {

    @Autowired
    private AMQPService service;

    @Test
    public void testSendToJms() throws InterruptedException, RemoteException {
        String msg = "hello";

        service.sendMessage(msg);

        Thread.sleep(2000);
    }
}
```

Chapter 5

Spring Integration Full Example

5.1 Introduction

This tutorial will go through a complete example of an application that uses several of the components provided by Spring Integration in order to provide a service to its users. This service consists of a system prompting the user to choose among different theaters. After his selection, the system will make a request to the external system of the chosen theater and return the list of its available films. Each cinema offers its service through a different API; we will see this in the sections that explain each external system (sections three and four).

5.2 Overview of the system

The following activity diagram shows a high level view of the system.

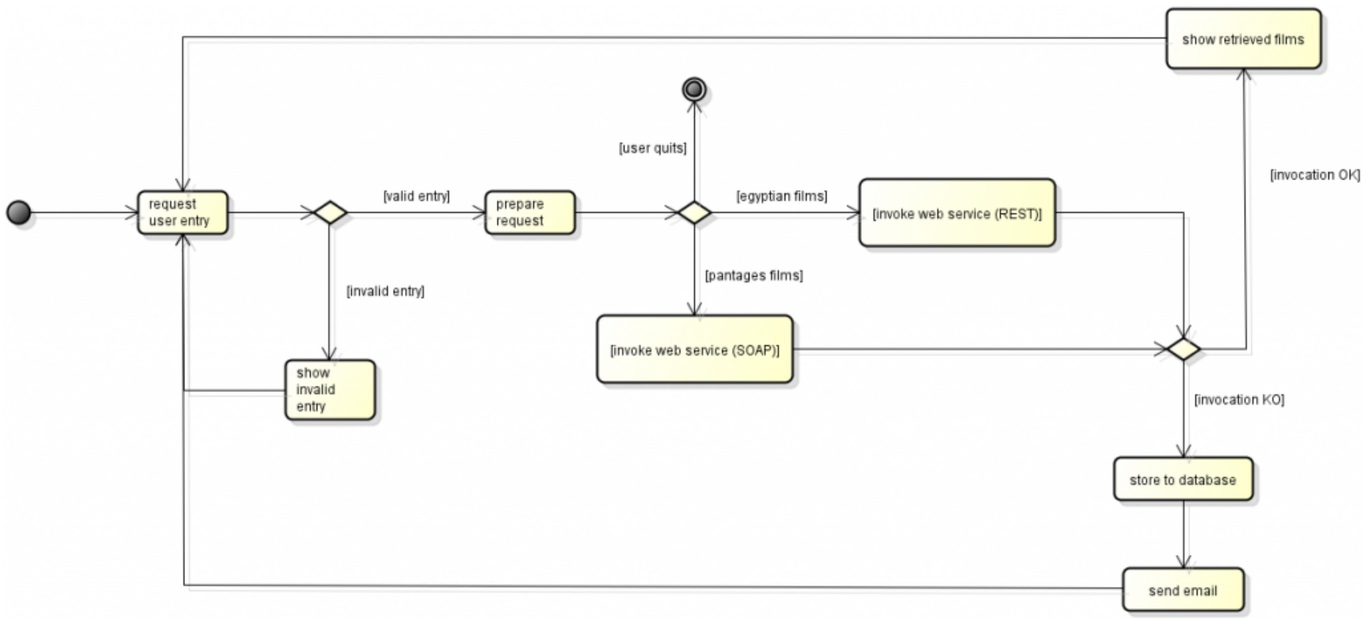


Figure 5.1: screenshot

- **User interface**: On the left side of the diagram, we have the start of the flow; the request user entry. This entry request shown by the system along with the system responses to the user, are both examples of integration with streams.

- **Web service invocations**: Depending on the selection made by the user, the system will retrieve the films list from a different external system. The Egyptian Theater exposes its service through HTTP, while the Pantages Theater exposes its service through SOAP.
- **Error handling**: If there is an error during the flow, maybe because of an unexpected exception or because a web service is not available, the system will send information to another two external systems: a noSQL database (MongoDB) and an email address.

Next sections will enter more deeply in each of the parts of this system.

5.3 The Egyptian Theater service

The Egyptian Theater system exposes his service through HTTP. In this section we are going to take a quick look at the application. It is a Spring 4 MVC application that contains a RESTful web service.

The controller will serve requests to retrieve all available films that are being showed in the theater:

```
@RestController
@RequestMapping(value="/films")
public class FilmController {
    FilmService filmService;

    @Autowired
    public FilmController(FilmService service) {
        this.filmService = service;
    }

    @RequestMapping(method=RequestMethod.GET)
    public Film[] getFilms() {
        return filmService.getFilms();
    }
}
```

The API is simple, and for this example the service will return some dummy values, since the focus of this section is to just provide a little bit more of detail of what external system is being called:

```
@Service("filmService")
public class FilmServiceImpl implements FilmService {

    @Override
    public Film[] getFilms() {
        Film film1 = new Film(1, "Bladerunner", "10am");
        Film film2 = new Film(2, "Gran Torino", "12pm");

        return new Film[]{film1, film2};
    }
}
```

The Spring configuration is based on annotations:

```
<!-- Detects annotations like @Component, @Service, @Controller, @Repository,  ↩
    @Configuration -->
<context:component-scan base-package="xpadro.spring.mvc.films.controller,xpadro.spring.mvc. ↩
    films.service"/>

<!-- Detects MVC annotations like @RequestMapping -->
<mvc:annotation-driven/>
```

The web.xml file configures the web application:

```
<!-- Root context configuration -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:xpadro/spring/mvc/config/root-context.xml</param-value>
</context-param>

<!-- Loads Spring root context, which will be the parent context -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- Spring servlet -->
<servlet>
    <servlet-name>springServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:xpadro/spring/mvc/config/app-context.xml</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>springServlet</servlet-name>
    <url-pattern>/spring/*</url-pattern>
</servlet-mapping>
```

So, the system will serve requests sent to `http://localhost:8080/rest-films/spring/films`, with `rest-films` being the context path of the application.

5.4 The Pantages Theater service

The Pantages Theater service exposes its service through SOAP. Like the Egyptian Theater, it consists in a web application but in this case, it is implemented with Spring Web Services.

The endpoint will serve `filmRequest` requests with the namespace `http://www.xpadro.spring.samples.com/films`. The response is built from the result received from a film service:

```
@Endpoint
public class FilmEndpoint {
    @Autowired
    private FilmService filmService;

    @PayloadRoot(localPart="filmRequest", namespace="http://www.xpadro.spring.samples.com/ ↩
        films")
    public @ResponsePayload FilmResponse getFilms() {
        return buildResponse();
    }

    private FilmResponse buildResponse() {
        FilmResponse response = new FilmResponse();

        for (Film film : filmService.getFilms()) {
            response.getFilm().add(film);
        }

        return response;
    }
}
```

The film service is also a dummy service that will return some default values:

```
@Service
public class FilmServiceImpl implements FilmService {

    @Override
    public List<Film> getFilms() {
        List<Film> films = new ArrayList<>();

        Film film = new Film();
        film.setId(new BigInteger(("1")));
        film.setName("The Good, the Bad and the Uggly");
        film.setShowtime("6pm");
        films.add(film);

        film = new Film();
        film.setId(new BigInteger(("2")));
        film.setName("The Empire strikes back");
        film.setShowtime("8pm");
        films.add(film);

        return films;
    }
}
```

The Spring configuration is shown below:

```
<!-- Detects @Endpoint since it is a specialization of @Component -->
<context:component-scan base-package="xpadro.spring.ws"/>

<!-- detects @PayloadRoot -->
<ws:annotation-driven/>

<ws:dynamic-wsdl id="filmDefinition" portTypeName="Films"
                 locationUri="http://localhost:8080/ws-films">
    <ws:xsd location="/WEB-INF/schemas/xsd/film-service.xsd"/>
</ws:dynamic-wsdl>
```

Finally, the `web.xml` file:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:xpadro/spring/ws/config/root-config.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<servlet>
    <servlet-name>Films Servlet</servlet-name>
    <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet- ↩
        class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:xpadro/spring/ws/config/servlet-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>Films Servlet</servlet-name>
    <url-pattern>/films/*</url-pattern>
```

```
</servlet-mapping>
```

Based on this configuration, the Pantages Theater application will serve requests sent to `http://localhost:8080/ws-films/films`, where `ws-films` is the context path of the application.

5.5 The user interface

Once we have seen what the external systems are that will communicate with our Spring Integration application, let's move on to see how this application is built.

The standalone application starts with a main method that bootstraps the Spring context, which contains all our integration components. Next, it prompts the user to enter his selection:

```
public class TheaterApp {
    private static Logger logger = LoggerFactory.getLogger("mainLogger");
    static AbstractApplicationContext context;

    public static void main(String[] args) {

        context = new ClassPathXmlApplicationContext("classpath:xpadro/spring/integration/ ↩
            config/int-config.xml");
        context.registerShutdownHook();

        logger.info("\nSelect your option (1-Egyptian Theater / 2-Pantages Theater / 0-quit ↩
            ):\n");
    }

    public static void shutdown() {
        logger.info("Shutting down...");
        context.close();
    }
}
```

The application also implements a shutdown method that will be invoked by the user. We will see this in more detail in a later section.

Ok, now how will the user selection get into the messaging system? This is where the integration with streams comes into play. The system entry to the messaging system is implemented with an inbound channel adapter that will read from stdin (`System.in`).

```
<!-- System entry -->
<int-stream:stdin-channel-adapter id="consoleIn" channel="systemEntry">
    <int:poller fixed-delay="1000" max-messages-per-poll="1" />
</int-stream:stdin-channel-adapter>
```

By using a poller, the inbound channel adapter will try to read from System.in each second and place the result into the `systemEntry` channel.

Now, we received a Spring Integration Message with the user entry as its payload. We can use our endpoints to transform the data and send it to the required system. Example: The console will prompt the user to choose among the different theaters:

```
2014-04-11 13:04:32,959|AbstractEndpoint|started org.springframework.integration.config. ↩
    ConsumerEndpointFactoryBean#7

Select your option (1-Egyptian Theater / 2-Pantages Theater / 0-quit):
```

5.6 Logging each user request

The first thing the system does is logging the user selection. This is done by using a wire tap. This wire tap is an implementation of an interceptor that will intercept messages that travel across the channel, in our case, the `systemEntry` channel. It will not alter the flow; the message will continue to its destination, but the wire tap will also send it to another channel, usually for monitoring.

In our application, the message is also sent to a logging channel adapter.

```
<int:channel id="systemEntry">
    <int:interceptors>
        <int:wire-tap channel="requestLoggingChannel"/>
    </int:interceptors>
</int:channel>

<int:logging-channel-adapter id="requestLoggingChannel"
    expression="'User selection: '.concat(payload)" level="INFO"/>
```

The logging channel adapter is implemented by the `http://docs.spring.io/spring-integration/docs/3.0.2.RELEASE/api/org/springframework/integration/config/xml/LoggingChannelAdapterParser.html[LoggingChannelAdapterParser]`. It basically creates a `http://docs.spring.io/spring-integration/docs/3.0.2.RELEASE/api/org/springframework/integration/handler/LoggingHandler.html[LoggingHandler]` that will log the payload of the message using the Apache Commons Logging library. If you want to log the full message instead of only its payload, you can add the `log-full-message` property to the logging channel adapter.

Example: The log shows the selection made by the user

```
Select your option (1-Egyptian Theater / 2-Pantages Theater / 0-quit):

1
2014-04-11 13:06:07,110|LoggingHandler|User selection: 1
```

5.7 Discarding invalid entries

Taking a glance at the user prompt, we see that the system accepts three valid entries:

```
logger.info("\nSelect your option (1-Egyptian Theater / 2-Pantages Theater / 0-quit):\n")
```

The treatment for invalid entries is very simple; the system will filter invalid entries, preventing them to move forward through the flow. These discarded messages will then be sent to a `discards` channel.

Subscribed to the `invalidEntries` discards channel, there is another stream channel adapter, in this case, an outbound adapter:

```
<int:filter input-channel="systemEntry" output-channel="validEntriesChannel" ref=" ↩
    entryFilter"
    discard-channel="invalidEntries"/>

<!-- Invalid entries (show on console) -->
<int:chain input-channel="invalidEntries">
    <int:transformer ref="discardsTransformer"/>
    <int-stream:stdout-channel-adapter id="consoleOut" append-newline="true" />
</int:chain>
```

The function of this adapter is to write to stdout (`System.out`), so the user will receive that he entered an invalid request on the console.

One thing to keep in mind is that we didn't create the `invalidEntries` channel. The adapter will be connected to the filter through an anonymous temporary direct channel.

Example: The console shows the user entered an invalid selection.

```
Select your option (1-Egyptian Theater / 2-Pantages Theater / 0-quit):

8
2014-04-11 13:07:41,808|LoggingHandler|User selection: 8
Invalid entry: 8
```

5.8 Choosing which theater to request

Valid entries that successfully pass the previous filter will be sent to a router:

```
<!-- Valid entries (continue processing) -->
<int:channel id="validEntriesChannel" />
<int:router input-channel="validEntriesChannel" ref="cinemaRedirector"/>
```

This router is responsible of deciding which external system is the one the user is requiring information from. It will also detect when the user wants to shut down the application:

```
@Component("cinemaRedirector")
public class CinemaRedirector {
    private static final String CINEMA_EGYPTIAN_CHANNEL = "egyptianRequestChannel";
    private static final String CINEMA_PANTAGES_CHANNEL = "pantagesRequestChannel";
    private static final String QUIT_REQUEST_CHANNEL = "quitRequestChannel";

    @Router
    public String redirectMessage(Message<String> msg) {
        String payload = msg.getPayload();

        if ("1".equals(payload)) {
            return CINEMA_EGYPTIAN_CHANNEL;
        }
        else if ("2".equals(payload)) {
            return CINEMA_PANTAGES_CHANNEL;
        }

        return QUIT_REQUEST_CHANNEL;
    }
}
```

So, here the flow splits in three different channels, a request to each theater and a request to finish the flow.

5.9 Requesting the Egyptian Theater

To communicate with the Egyptian Theater system, we need to send an HTTP request. We accomplish this by using an HTTP outbound gateway.

```
<int-http:outbound-gateway url="http://localhost:8080/rest-films/spring/films"
        expected-response-type="java.lang.String" http-method="GET" charset="UTF-8"/>
```

This gateway is configured with several properties:

- `expected-response-type`: The return type returned by the web service will be a String containing the JSON with the films listing.
- `http-method`: We are making a GET request.
- `charset`: the charset for converting the payload to bytes.

Once the response is received, we will use a transformer to convert the returned JSON to a Java Object. In our case, we will convert the response to a `Film` array:

```
<int:json-to-object-transformer type="xpadro.spring.integration.model.Film[]"/>
```

Next, a service activator will go through the array and build a String more appropriate to the user.

```
<int:service-activator ref="restResponseHandler"/>
```

The implementation is shown below:

```
@Component("restResponseHandler")
public class RestResponseHandler {
    private static final String NEW_LINE = "\n";

    @ServiceActivator
    public String handle(Message<Film[]> msg) {
        Film[] films = msg.getPayload();

        StringBuilder response = new StringBuilder(NEW_LINE);
        if (films.length > 0) {
            response.append("Returned films:" + NEW_LINE);
        }
        else {
            response.append("No films returned" + NEW_LINE);
        }

        for (Film f:films) {
            response.append(f.getName()).append(NEW_LINE);
        }

        return response.toString();
    }
}
```

Finally, we will show the response to the user by printing it on the console. We are using the same channel adapter as the one used to show the user its invalid entries. The adapter will write to System.out:

```
<int-stream:stdout-channel-adapter id="consoleOut" append-newline="true" />
```

The next code snippet shows the complete request. Since I did not want to create message channels for each interaction between these endpoints, I used a message handler chain. This type of endpoint simplifies the required XML configuration when we have several endpoints in a sequence. By using a message handler chain, all its endpoints are connected through anonymous direct channels.

```
<!-- Egyptian Theater request -->
<int:chain input-channel="egyptianRequestChannel">
    <int-http:outbound-gateway url="http://localhost:8080/rest-films/spring/films"
        expected-response-type="java.lang.String" http-method="GET" charset="UTF-8"/>
    <int:json-to-object-transformer type="xpadro.spring.integration.model.Film[]"/>
    <int:service-activator ref="restResponseHandler"/>
    <int-stream:stdout-channel-adapter id="consoleOut" append-newline="true" />
</int:chain>
```

Example: The Egyptian Theater film listing is shown to the user.

```
1
2014-04-11 14:26:20,981|LoggingHandler|User selection: 1

Returned films:
Bladerunner
Gran Torino
```

5.10 Requesting the Pantages Theater

We are going to need a web service gateway to interact with the Pantages Theater system, but first, we have to build a request object of the type `filmRequest` in order to be served by the Egyptian web service endpoint.

We are using a transformer to change the message to a film web service request:

```
<int:transformer ref="soapRequestTransformer"/>
```

The implementation:

```
@Component("soapRequestTransformer")
public class SoapRequestTransformer {

    @Transformer
    public Message<?> createRequestMessage(Message<String> msg) {
        return MessageBuilder.withPayload(new FilmRequest()).copyHeadersIfAbsent(msg. ↩
            getHeaders()).build();
    }
}
```

We got the request object so we can now invoke the web service using a web service gateway:

```
<int-ws:outbound-gateway uri="http://localhost:8080/ws-films/films"
        marshaller="marshaller" unmarshaller="marshaller"/>
```

As seen in a previous tutorial, a marshaller will be required to convert the request and response. For this task we are going to use the oxm namespace:

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="xpadro.spring.integration.ws.types" />
```

When we receive the web service response it will be in the form of a FilmResponse. The next endpoint in the sequence will adapt the response and return a String in order to be shown to the user in the next phase:

```
<int:service-activator ref="soapResponseHandler"/>
```

The implementation:

```
@Component("soapResponseHandler")
public class SoapResponseHandler {
    private static final String NEW_LINE = "\n";

    @ServiceActivator
    public String handle(Message<FilmResponse> msg) {
        FilmResponse response = msg.getPayload();

        StringBuilder resp = new StringBuilder(NEW_LINE);
        if (response.getFilm().size() > 0) {
            resp.append("Returned films:" + NEW_LINE);
        }
        else {
            resp.append("No films returned" + NEW_LINE);
        }

        for (Film f : response.getFilm()) {
            resp.append(f.getName()).append(NEW_LINE);
        }

        return resp.toString();
    }
}
```

Like in the Egyptian request, this request also ends with another stream outbound channel adapter:

```
<int-stream:stdout-channel-adapter id="consoleOut" append-newline="true" />
```

Since we have another sequence of endpoints, we use a message handler chain to minimize the quantity of configuration required:

```
<!-- Pantages Theater request -->
<int:chain input-channel="pantagesRequestChannel">
    <int:transformer ref="soapRequestTransformer"/>
    <int-ws:outbound-gateway uri="http://localhost:8080/ws-films/films"
        marshaller="marshaller" unmarshaller="marshaller"/>
    <int:service-activator ref="soapResponseHandler"/>
    <int-stream:stdout-channel-adapter id="consoleOut" append-newline="true" />
</int:chain>
```

Example: The Pantages Theater film listing is shown to the user.

```
2
2014-04-11 14:27:54,796|LoggingHandler|User selection: 2

Returned films:
The Good, the Bad and the Uggly
The Empire strikes back
```

5.11 Handling errors

When anything goes wrong, we need to register it in some way. When this happens, the application will do two things:

- Store the message to a database.
- Send an email to a specified address.

The Spring Integration message channel called `errorChannel` will receive errors of type `http://docs.spring.io/spring-integration/docs/3.0.2.RELEASE/api/org/springframework/integration/MessagingException.html[MessagingException]` thrown by message handlers. This special channel is a publish-subscribe channel, which means that it can have several subscribers to it. Our application subscribes two endpoints in order to execute the two previously specified actions.

5.11.1 Storing the message to the database

The following service activator is subscribed to the error channel, thus it will receive the MessagingException:

```
<int:service-activator ref="mongodbRequestHandler"/>
```

What this activator will do is to resolve what theater was the request sent to and then it will build a `FailedMessage` object, containing the information we want to log:

```
@Component("mongodbRequestHandler")
public class MongodbRequestHandler {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    private TheaterResolver theaterResolver;

    public FailedMessage handle(MessagingException exc) {
        logger.error("Request failed. Storing to the database");

        String theater = theaterResolver.resolve(exc);
        FailedMessage failedMsg = new FailedMessage(new Date(), exc.getMessage(), theater);

        return failedMsg;
    }
}
```

The failed message structure contains basic information:

```
public class FailedMessage implements Serializable {
    private static final long serialVersionUID = 4411815706008283621L;

    private final Date requestDate;
    private final String messsage;
    private final String theater;

    public FailedMessage(Date requestDate, String message, String theater) {
        this.requestDate = requestDate;
        this.messsage = message;
        this.theater = theater;
    }


    public Date getRequestDate() {
        return new Date(requestDate.getTime());
    }

    public String getMessage() {
        return this.messsage;
    }

    public String getTheater() {
        return this.theater;
    }
}
```

Once the message is built, we store it to the database by using a mongodb outbound channel adapter:

```
<int-mongodb:outbound-channel-adapter id="mongodbAdapter" collection-name="failedRequests"  ↩
    mongodb-factory="mongoDbFactory" />
```

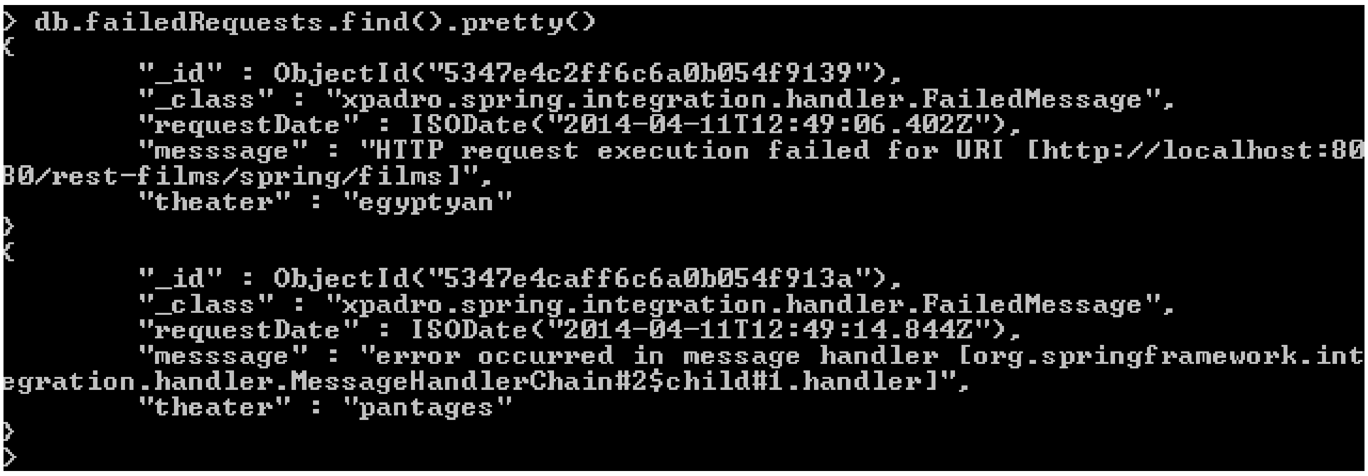
The complete code for this part of the flow is shown below:

```
<import resource="mongodb-config.xml"/>

<int:chain input-channel="errorChannel">
    <int:service-activator ref="mongodbRequestHandler"/>
    <int-mongodb:outbound-channel-adapter id="mongodbAdapter" collection-name=" ↩
        failedRequests" mongodb-factory="mongoDbFactory" />
</int:chain>
```

The `mongodb-config.xml` file contains information specific to MongoDB configuration:

```
<bean id="mongoDbFactory" class="org.springframework.data.mongodb.core.SimpleMongoDbFactory ↩
    ">
    <constructor-arg>
        <bean class="com.mongodb.Mongo"/>
    </constructor-arg>
    <constructor-arg value="jcgdb"/>
</bean>

<bean id="mongoDbMessageStore" class="org.springframework.integration.mongodb.store. ↩
    ConfigurableMongoDbMessageStore">
    <constructor-arg ref="mongoDbFactory"/>
</bean>
```

Example: The following screenshot shows the collection after two failed requests, one for each type of theater:



```
> db.failedRequests.find().pretty()
{
        "_id" : ObjectId("5347e4c2ff6c6a0b054f9139"),
        "_class" : "xpadro.spring.integration.handler.FailedMessage",
        "requestDate" : ISODate("2014-04-11T12:49:06.402Z"),
        "messsage" : "HTTP request execution failed for URI [http://localhost:80
80/rest-films/spring/films]",
        "theater" : "egyptyan"
}
{
        "_id" : ObjectId("5347e4caff6c6a0b054f913a"),
        "_class" : "xpadro.spring.integration.handler.FailedMessage",
        "requestDate" : ISODate("2014-04-11T12:49:14.844Z"),
        "messsage" : "error occurred in message handler [org.springframework.int
egration.handler.MessageHandlerChain#2$child#1.handler]",
        "theater" : "pantages"
}
>
```

Figure 5.2: screenshot

5.11.2 Sending an email to the person in charge

Another endpoint subscribed to the error channel is the mail request handler.

```
<int:service-activator ref="mailRequestHandler"/>
```

This handler is responsible of creating a MailMessage in order to send it to the mail gateway:

```
@Component("mailRequestHandler")
public class MailRequestHandler {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    private TheaterResolver theaterResolver;

    @ServiceActivator
    public MailMessage handle(MessagingException exc) {
        logger.error("Request failed. Sending mail");

        MailMessage mailMsg = new SimpleMailMessage();
        mailMsg.setFrom("Theater.System");
        mailMsg.setTo("my.mail@gmail.com");
        mailMsg.setSubject("theater request failed");

        String theater = theaterResolver.resolve(exc);
        StringBuilder textMessage = new StringBuilder("Invocation to ").append(theater). ↩
            append(" failed\n\n")
                .append("Error message was: ").append(exc.getMessage());
        mailMsg.setText(textMessage.toString());

        return mailMsg;
    }
}
```

The theater resolver checks the user selection and returns the name of the requested theater.

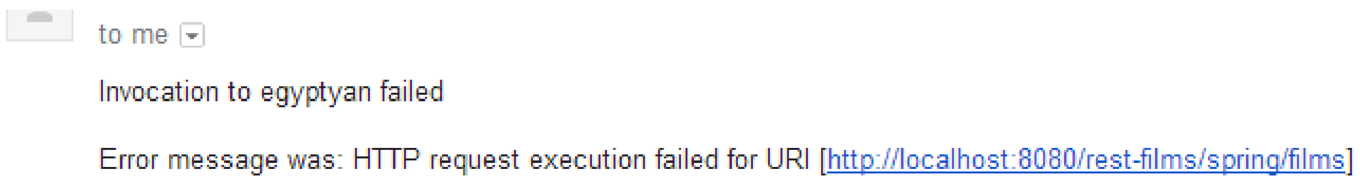
The complete configuration is as follows:

```
<int:chain input-channel="errorChannel">
    <int:service-activator ref="mailRequestHandler"/>
    <int-mail:outbound-channel-adapter mail-sender="mailSender" />
</int:chain>

<import resource="mail-config.xml"/>
```

The `mail-config.xml` contains the configuration of the Spring mail sender:

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host" value="smtp.gmail.com" />
  <property name="port" value="465" />
  <property name="username" value="my.mail@gmail.com" />
  <property name="password" value="myPassword" />
  <property name="javaMailProperties">
     <props>
        <prop key="mail.smtp.starttls.enable">true</prop>
        <prop key="mail.smtp.auth">true</prop>
        <prop key="mail.smtp.socketFactory.class">javax.net.ssl.SSLSocketFactory</prop>
     </props>
  </property>
</bean>
```

Example: The message sent to the gmail account:


to me

Invocation to egyptyan failed

Error message was: HTTP request execution failed for URI [http://localhost:8080/rest-films/spring/films]

Figure 5.3: screenshot

5.12 Shutting down the application

When prompted, the user can decide to finish with the execution of the application by entering a zero. When he decides to do that, the router will redirect the message to the `quitRequestChannel` channel (see section 8). Subscribed to this channel, we have configured a message handler chain:

```
<!-- Quit message (shutdown application) -->
<int:chain input-channel="quitRequestChannel">
    <int:service-activator ref="shutdownActivator"/>
    <int-event:outbound-channel-adapter/>
</int:chain>
```

The service activator will create a `ShutdownEvent` and return it in order to be handled by the next endpoint of the message handler chain:

```
@Component("shutdownActivator")
public class ShutdownActivator {

    @ServiceActivator
    public ShutdownEvent createEvent(Message<String> msg) {
        return new ShutdownEvent(this);
    }
}
```

The `ShutdownEvent` is an instance of `http://docs.spring.io/spring/docs/3.2.8.RELEASE/javadoc-api/org/springframework/context/ApplicationEvent.html[ApplicationEvent]`.

```
public class ShutdownEvent extends ApplicationEvent {
    private static final long serialVersionUID = -198696884593684436L;

    public ShutdownEvent(Object source) {
        super(source);
    }

    public ShutdownEvent(Object source, String message) {
        super(source);
    }

    public String toString() {
        return "Shutdown event";
    }
}
```

The event outbound channel adapter will publish as an `ApplicationEvent`, any message sent to the channel where it is subscribed. In this way, they will be handled by any `http://docs.spring.io/spring/docs/3.2.8.RELEASE/javadoc-api/org/springframework/context/ApplicationListener.html[ApplicationListener]` instance registered in the application context. However, if the payload of the message is an instance of `ApplicationEvent`, it will be passed as-is. As seen in the previous code, our `ShutdownEvent` is an instance of `ApplicationEvent`.

Listening to this kind of events, we have registered a listener:

```
@Component("shutdownListener")
public class ShutdownListener implements ApplicationListener<ShutdownEvent> {

    @Override
    public void onApplicationEvent(ShutdownEvent event) {
        TheaterApp.shutdown();
    }
}
```

The listener is used to shut down the application. If you remember the shutdown method in section five, the `Theater` application closes the application context.

5.13 Taking a glance at the complete flow

I have put all integration elements within the same file just for ease of clarity, but we could consider splitting it in smaller files:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:int="http://www.springframework.org/schema/integration"
    xmlns:int-jms="http://www.springframework.org/schema/integration/jms"
    xmlns:int-stream="http://www.springframework.org/schema/integration/stream"
    xmlns:int-event="http://www.springframework.org/schema/integration/event"
    xmlns:int-http="http://www.springframework.org/schema/integration/http"
    xmlns:int-ws="http://www.springframework.org/schema/integration/ws"
    xmlns:int-mongodb="http://www.springframework.org/schema/integration/mongodb"
    xmlns:int-mail="http://www.springframework.org/schema/integration/mail"
    xmlns:oxm="http://www.springframework.org/schema/oxm"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/ ↩
            beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema ↩
            /context/spring-context-3.0.xsd
        http://www.springframework.org/schema/integration http://www.springframework.org/ ↩
            schema/integration/spring-integration-3.0.xsd
        http://www.springframework.org/schema/integration/jms http://www.springframework. ↩
            org/schema/integration/jms/spring-integration-jms-3.0.xsd
        http://www.springframework.org/schema/integration/stream http://www.springframework ↩
            .org/schema/integration/stream/spring-integration-stream-3.0.xsd
        http://www.springframework.org/schema/integration/event http://www.springframework. ↩
            org/schema/integration/event/spring-integration-event-3.0.xsd
        http://www.springframework.org/schema/integration/http http://www.springframework. ↩
            org/schema/integration/http/spring-integration-http-3.0.xsd
        http://www.springframework.org/schema/integration/ws http://www.springframework.org ↩
            /schema/integration/ws/spring-integration-ws-3.0.xsd
        http://www.springframework.org/schema/integration/mongodb http://www. ↩
            springframework.org/schema/integration/mongodb/spring-integration-mongodb-3.0. ↩
            xsd
        http://www.springframework.org/schema/integration/mail http://www.springframework. ↩
            org/schema/integration/mail/spring-integration-mail-3.0.xsd
        http://www.springframework.org/schema/oxm http://www.springframework.org/schema/oxm ↩
            /spring-oxm-3.0.xsd">

    <context:component-scan base-package="xpadro.spring.integration"/>

    <!-- System entry -->
    <int-stream:stdin-channel-adapter id="consoleIn" channel="systemEntry">
        <int:poller fixed-delay="1000" max-messages-per-poll="1" />
    </int-stream:stdin-channel-adapter>

    <int:channel id="systemEntry">
        <int:interceptors>
            <int:wire-tap channel="requestLoggingChannel"/>
        </int:interceptors>
    </int:channel>

    <int:logging-channel-adapter id="requestLoggingChannel"
        expression="'User selection: '.concat(payload)" level="INFO"/>

    <int:filter input-channel="systemEntry" output-channel="validEntriesChannel" ref=" ↩
        entryFilter"
        discard-channel="invalidEntries"/>


    <!-- Invalid entries (show on console) -->
    <int:chain input-channel="invalidEntries">
        <int:transformer ref="discardsTransformer"/>
        <int-stream:stdout-channel-adapter id="consoleOut" append-newline="true" />
    </int:chain>


    <!-- Valid entries (continue processing) -->
    <int:channel id="validEntriesChannel" />
    <int:router input-channel="validEntriesChannel" ref="cinemaRedirector"/>


    <!-- Quit message (shutdown application) -->
    <int:chain input-channel="quitRequestChannel">
        <int:service-activator ref="shutdownActivator"/>
        <int-event:outbound-channel-adapter/>
    </int:chain>


    <!-- Continue processing (get data) -->
    <!-- Pantages Theater request -->
    <int:chain input-channel="pantagesRequestChannel">
        <int:transformer ref="soapRequestTransformer"/>
        <int-ws:outbound-gateway uri="http://localhost:8080/ws-films/films"
            marshaller="marshaller" unmarshaller="marshaller"/>
        <int:service-activator ref="soapResponseHandler"/>
        <int-stream:stdout-channel-adapter id="consoleOut" append-newline="true" />
    </int:chain>

    <oxm:jaxb2-marshaller id="marshaller" contextPath="xpadro.spring.integration.ws.types"  ↩
        />


    <!-- Egyptian Theater request -->

    <int:chain input-channel="egyptianRequestChannel">
        <int-http:outbound-gateway url="http://localhost:8080/rest-films/spring/films"
            expected-response-type="java.lang.String" http-method="GET" charset="UTF-8"/>
        <int:json-to-object-transformer type="xpadro.spring.integration.model.Film[]"/>
        <int:service-activator ref="restResponseHandler"/>
        <int-stream:stdout-channel-adapter id="consoleOut" append-newline="true" />
    </int:chain>


    <!-- Error handling -->
    <import resource="mongodb-config.xml"/>

    <int:chain input-channel="errorChannel">
        <int:service-activator ref="mongodbRequestHandler"/>
        <int-mongodb:outbound-channel-adapter id="mongodbAdapter" collection-name=" ↩
            failedRequests" mongodb-factory="mongoDbFactory" />
    </int:chain>

    <int:chain input-channel="errorChannel">
        <int:service-activator ref="mailRequestHandler"/>
        <int-mail:outbound-channel-adapter mail-sender="mailSender" />
    </int:chain>

    <import resource="mail-config.xml"/>

</beans>
```

5.14 Technology versions

For this application, I have used the 3.2.8 release version of the Spring framework and the latest 3.0.2 release version of Spring Integration.

The full dependency list is shown here:

```
<properties>
    <spring-version>3.2.8.RELEASE</spring-version>
    <spring-integration-version>3.0.2.RELEASE</spring-integration-version>
    <slf4j-version>1.7.5</slf4j-version>
    <jackson-version>2.3.0</jackson-version>
    <javax-mail-version>1.4.1</javax-mail-version>
</properties>

<dependencies>
    <!-- Spring Framework - Core -->
    <dependency>
```

```
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring-version}</version>
    </dependency>

    <!-- Spring Framework - Integration -->
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-core</artifactId>
        <version>${spring-integration-version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-jms</artifactId>
        <version>${spring-integration-version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-stream</artifactId>
        <version>${spring-integration-version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-event</artifactId>
        <version>${spring-integration-version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-http</artifactId>
        <version>${spring-integration-version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-ws</artifactId>
        <version>${spring-integration-version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-mongodb</artifactId>
        <version>${spring-integration-version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-mail</artifactId>
        <version>${spring-integration-version}</version>
    </dependency>

    <!-- javax.mail -->
    <dependency>
        <groupId>javax.mail</groupId>
        <artifactId>mail</artifactId>
        <version>${javax-mail-version}</version>
    </dependency>

    <!-- Jackson -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>${jackson-version}</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>${jackson-version}</version>
    </dependency>

    <!-- Logging -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>${slf4j-version}</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>${slf4j-version}</version>
    </dependency>
</dependencies>
```

Chapter 6

Monitoring and Management

6.1 Introduction

After having experimented with the main components provided by Spring Integration and seen how it integrates well with other systems like JMS queues or web services, this chapter finishes the course by showing different mechanisms of monitoring or gathering more information about what is going on within the messaging system.

Some of these mechanisms consist of managing or monitoring the application through MBeans, which are part of the JMX specification. We will also learn how to monitor messages to see which components were involved during the messaging flow and how to persist messages for components that have the capability to buffer messages.

Another mechanism discussed in this chapter is how we will implement the EIP idempotent receiver pattern using a metadata store.

Finally, the last mechanism described is the control bus. This will let us send messages that will invoke operations on components in the application context.

6.2 Publishing and receiving JMX notifications

The JMX specification defines a mechanism that allows MBeans to publish notifications that will be sent to other MBeans or to the management application. The Oracle documentation explains how to implement this mechanism.

Spring Integration supports this feature by providing channel adapters that are both able to publish and receive JMX notifications. We are going to see an example that uses both channel adapters:

- A notification listening channel adapter
- A notification publishing channel adapter

6.2.1 Publishing a JMX notification

In the first part of the example, the messaging system receives a String message (a message with a payload of type String) through its entry gateway. It then uses a service activator (notification handler) to build a `http://docs.oracle.com/javase/7/docs/api/javax/management/Notification.html[javax.management.Notification]` and sends it to the notification publishing channel adapter, which will publish the JMX notification.

The flow of this first part is shown below:

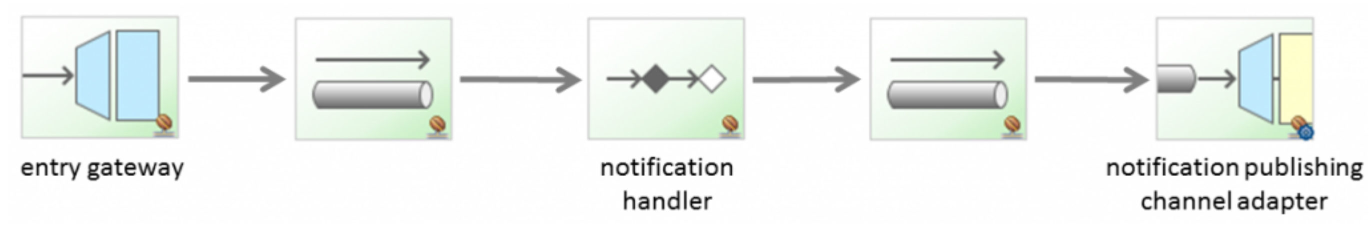


Figure 6.1: screenshot

The xml configuration equivalent to the previous graphic:

```
<context:component-scan base-package="xpadro.spring.integration.jmx.notification"/>

<context:mbean-export/>
<context:mbean-server/>

<!-- Sending Notifications -->
<int:gateway service-interface="xpadro.spring.integration.jmx.notification. ↩
    JmxNotificationGateway" default-request-channel="entryChannel"/>

<int:channel id="entryChannel"/>

<int:service-activator input-channel="entryChannel" output-channel="sendNotificationChannel ↩
    "
    ref="notificationHandler" method="buildNotification"/>

<int:channel id="sendNotificationChannel"/>

<int-jmx:notification-publishing-channel-adapter channel="sendNotificationChannel"
    object-name="xpadro.spring.integration.jmx.adapter:type=integrationMBean,name= ↩
        integrationMbean"/>
```

The gateway is as simple as in previous examples. Remember that the `@Gateway` annotation is not necessary if you have just one method:

```
public interface JmxNotificationGateway {

    public void send(String type);
}
```

A `Message` will reach the service activator, which will build the message with the JMX notification:

```
@Component("notificationHandler")
public class NotificationHandler {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private static final String NOTIFICATION_TYPE_HEADER = "jmx_notificationType";

    public void receive(Message<Notification> msg) {
        logger.info("Notification received: {}", msg.getPayload().getType());
    }

    public Message<Notification> buildNotification(Message<String> msg) {
        Notification notification = new Notification(msg.getPayload(), this, 0);

        return MessageBuilder.withPayload(notification)
                .copyHeadersIfAbsent(msg.getHeaders()).setHeader(NOTIFICATION_TYPE_HEADER,  ↩
                    "myJmxNotification").build();
    }
}
```

Notice that we have set a new header. This is necessary to provide the notification type or the JMX adapter will throw an `IllegalArgumentException` with the message “No notification type header is available, and no default has been provided”.

Finally, we just need to return the message in order to be sent to the publishing adapter. The rest is handled by Spring Integration.

6.2.2 Receiving a JMX notification

The second part of the flow consists in a notification listening channel adapter that will receive our previously published notification.

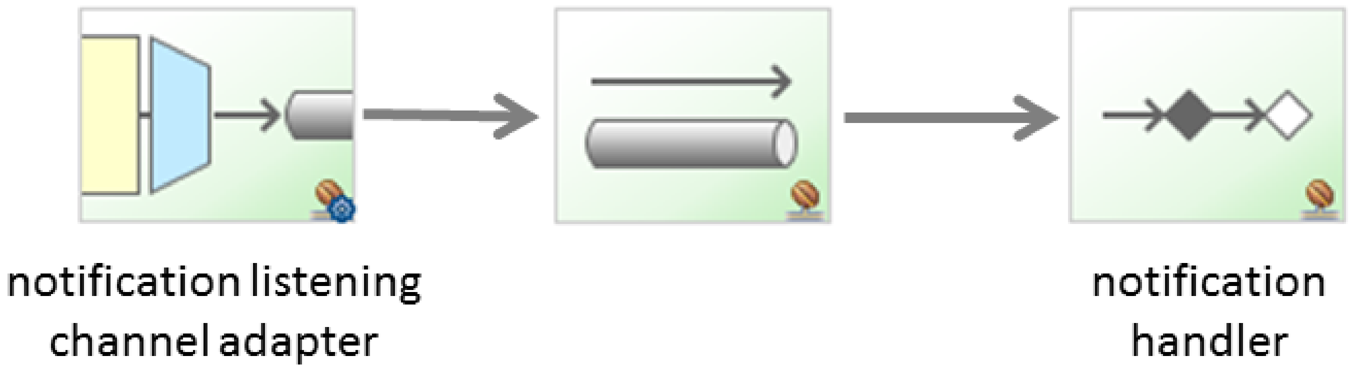


Figure 6.2: screenshot

The xml configuration:

```
<!-- Receiving Notifications -->
<int-jmx:notification-listening-channel-adapter channel="receiveNotificationChannel"
    object-name="xpadro.spring.integration.jmx.adapter:type=integrationMBean,name= ↩
        integrationMbean"/>

<int:channel id="receiveNotificationChannel"/>

<int:service-activator input-channel="receiveNotificationChannel"
    ref="notificationHandler" method="receive"/>
```

We will just receive the notification and log it:

```
public void receive(Message<Notification> msg) {
    logger.info("Notification received: {}", msg.getPayload().getType());
}
```

The application that runs the example:

```
public class NotificationApp {
    public static void main(String[] args) throws InterruptedException {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("classpath: ↩
            xpadro/spring/integration/jmx/config/int-notification-config.xml");

        JmxNotificationGateway gateway = context.getBean(JmxNotificationGateway.class);
        gateway.send("gatewayNotification");
        Thread.sleep(1000);
        context.close();
    }
}
```

6.3 Polling managed attributes from an MBean

Imagine we have an `MBean` that is monitoring some feature. With the attribute polling channel adapter, your application will be able to poll the `MBean` and receive the updated data.

I have implemented an `MBean` that generates a random number every time is asked. Not the most vital feature but will serve us to see an example:

```
@Component("pollingMbean")
@ManagedResource
public class JmxPollingMBean {

    @ManagedAttribute
    public int getNumber() {
        Random rnd = new Random();
        int randomNum = rnd.nextInt(100);
        return randomNum;
    }
}
```

The flow couldn't be simpler; we need an attribute polling channel adapter specifying the type and name of our `MBean`. The adapter will poll the `MBean` and place the result in the result channel. Each result polled will be shown on console through the stream stdout channel adapter:

```
<context:component-scan base-package="xpadro.spring.integration.jmx.polling"/>

<context:mbean-export/>
<context:mbean-server/>

<!-- Polling -->
<int-jmx:attribute-polling-channel-adapter channel="resultChannel"
        object-name="xpadro.spring.integration.jmx.polling:type=JmxPollingMBean,name= ↩
            pollingMbean"
        attribute-name="Number">
    <int:poller max-messages-per-poll="1" fixed-delay="1000"/>
</int-jmx:attribute-polling-channel-adapter>

<int:channel id="resultChannel"/>

<int-stream:stdout-channel-adapter channel="resultChannel" append-newline="true"/>
```

The application that runs the example:

```
public class PollingApp {
    public static void main(String[] args) throws InterruptedException {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("classpath: ↩
            xpadro/spring/integration/jmx/config/int-polling-config.xml");
        context.registerShutdownHook();
        Thread.sleep(5000);
        context.close();
    }
}
```

And the console output:

```
2014-04-16 16:23:43,867|AbstractEndpoint|started org.springframework.integration.config. ↩
    ConsumerEndpointFactoryBean#0
82
72
20
47
```

```
21
2014-04-16 16:23:48,878|AbstractApplicationContext|Closing org.springframework.context. ↩
    support.ClassPathXmlApplicationContext@7283922
```

6.4 Invoking MBean operations

The next mechanism allows us to invoke an operation of an `MBean`. We are going to implement another bean that contains a single operation, our old friend hello world:

```
@Component("operationMbean")
@ManagedResource
public class JmxOperationMBean {

    @ManagedOperation
    public String hello(String name) {
        return "Hello " + name;
    }
}
```

Now, we can use a channel adapter if the operation does not return a result, or a gateway if so. With the following xml configuration, we export the `MBean` and use the gateway to invoke the operation and wait for the result:

```
<context:component-scan base-package="xpadro.spring.integration.jmx.operation"/>

<context:mbean-export/>
<context:mbean-server/>

<int:gateway service-interface="xpadro.spring.integration.jmx.operation.JmxOperationGateway ↩
    " default-request-channel="entryChannel"/>

<int-jmx:operation-invoking-outbound-gateway request-channel="entryChannel" reply-channel=" ↩
    replyChannel"
    object-name="xpadro.spring.integration.jmx.operation:type=JmxOperationMBean,name= ↩
        operationMbean"
    operation-name="hello"/>

<int:channel id="replyChannel"/>

<int-stream:stdout-channel-adapter channel="replyChannel" append-newline="true"/>
```

In order to work, we have to specify the type and name of the `MBean`, and the operation we want to invoke. The result will be sent to the stream channel adapter in order to be shown on the console.

The application that runs the example:

```
public class OperationApp {
    public static void main(String[] args) throws InterruptedException {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("classpath: ↩
            xpadro/spring/integration/jmx/config/int-operation-config.xml");

        JmxOperationGateway gateway = context.getBean(JmxOperationGateway.class);
        gateway.hello("World");
        Thread.sleep(1000);
        context.close();
    }
}
```

6.5 Exporting components as MBeans

This component is used to export message channels, message handlers and message endpoints as MBeans so you can monitor them.

You need to put the following configuration into your application:

```
<int-jmx:mbean-export id="integrationMBeanExporter"
    default-domain="xpadro.integration.exporter" server="mbeanServer"/>

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

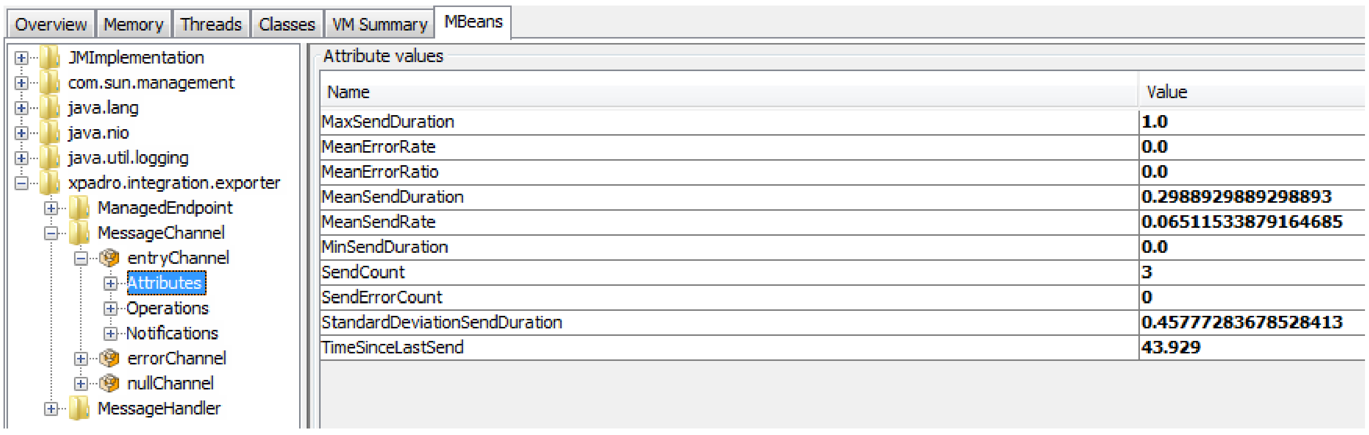
And set the following VM arguments as explained in the Spring documentation:

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=6969
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

The application that runs the example sends three messages:

```
public class ExporterApp {
    public static void main(String[] args) throws InterruptedException {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("classpath: ↩
            xpadro/spring/integration/jmx/config/int-exporter-config.xml");
        context.registerShutdownHook();

        JmxExporterGateway gateway = context.getBean(JmxExporterGateway.class);
        gateway.sendMessage("message 1");
        Thread.sleep(500);
        gateway.sendMessage("message 2");
        Thread.sleep(500);
        gateway.sendMessage("message 3");
    }
}
```

Once the application is running, you can see information about the components. The following screenshot is made on the JConsole:



Name	Value
MaxSendDuration	1.0
MeanErrorRate	0.0
MeanErrorRatio	0.0
MeanSendDuration	0.2988929889298893
MeanSendRate	0.06511533879164685
MinSendDuration	0.0
SendCount	3
SendErrorCount	0
StandardDeviationSendDuration	0.45777283678528413
TimeSinceLastSend	43.929

Figure 6.3: screenshot

You can notice that the `sendCount` attribute of the entry channel has the value 3, because we have sent three messages in our example.

6.6 Trailing a message path

In a messaging system, components are loosely coupled. This means that the sending component does not need to know who will receive the message. And the other way round, the receiver is just interested in the message received, not who sent it. This benefit can be not so good when we need to debug the application.

The message history consists in attaching to the message, the list of all components the message passed through.

The following application will test this feature by sending a message through several components:

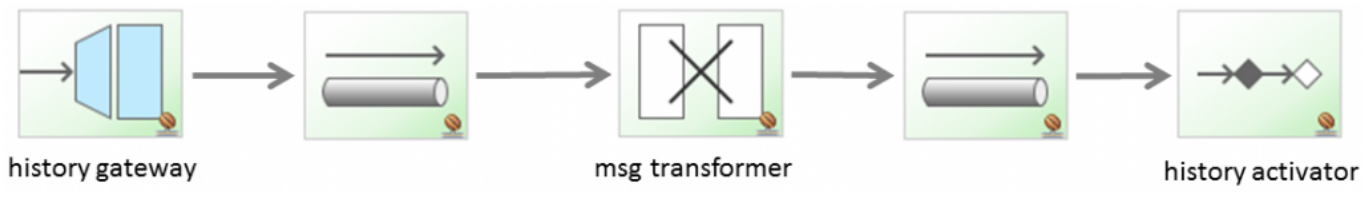


Figure 6.4: screenshot

The key element of the configuration is not visible in the previous graphic: the `message-history` element:

```
<context:component-scan base-package="xpadro.spring.integration.msg.history"/>

<int:message-history/>

<int:gateway id="historyGateway" service-interface="xpadro.spring.integration.msg.history. ↩
    HistoryGateway"
    default-request-channel="entryChannel"/>

<int:channel id="entryChannel"/>

<int:transformer id="msgTransformer" input-channel="entryChannel"
    expression="payload + 'transformed'" output-channel="transformedChannel"/>

<int:channel id="transformedChannel"/>

<int:service-activator input-channel="transformedChannel" ref="historyActivator"/>
```

With this configuration set, the service activator at the end of the messaging flow will be able to retrieve the list of visited components by looking at the header of the message:

```
@Component("historyActivator")
public class HistoryActivator {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    public void handle(Message<String> msg) {
        MessageHistory msgHistory = msg.getHeaders().get(MessageHistory.HEADER_NAME,  ↩
            MessageHistory.class);
        if (msgHistory != null) {
            logger.info("Components visited: {}", msgHistory.toString());
        }
    }
}
```

The application running this example:

```
public class MsgHistoryApp {
    public static void main(String[] args) throws InterruptedException {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("classpath: ↩
            xpadro/spring/integration/msg/history/config/int-msg-history-config.xml");
        HistoryGateway gateway = context.getBean(HistoryGateway.class);
        gateway.send("myTest");
        Thread.sleep(1000);
        context.close();
    }
}
```

The result will be shown on the console:

```
2014-04-16 17:34:52,551|HistoryActivator|Components visited: historyGateway,entryChannel, ↩
    msgTransformer,transformedChannel
```

6.7 Persisting buffered messages

Some of the components in Spring Integration can buffer messages. For example, a queue channel will buffer messages until consumers retrieve them from it. Another example is the aggregator endpoint; as seen in the second tutorial, this endpoint will gather messages until the group is complete basing its decision on the release strategy.

These integration patterns imply that if a failure occurs, buffered messages can be lost. To prevent this, we can persist these messages, for example storing them into a database. By default, Spring Integration stores these messages in memory. We are going to change this using a message store.

For our example, we will store these messages into a MongoDB database. In order to do that, we just need the following configuration:

```
<bean id="mongoDbFactory" class="org.springframework.data.mongodb.core.SimpleMongoDbFactory ↩
    ">
    <constructor-arg>
        <bean class="com.mongodb.Mongo"/>
    </constructor-arg>
    <constructor-arg value="jcgdb"/>
</bean>

<bean id="mongoDbMessageStore" class="org.springframework.integration.mongodb.store. ↩
    ConfigurableMongoDbMessageStore">
    <constructor-arg ref="mongoDbFactory"/>
</bean>
```

Now, we are going to create an application to test this feature. I have implemented a flow that receives through a gateway, a message with a String payload. This message is sent by the gateway to a queue channel that will buffer the messages until the service activator `msgStoreActivator` retrieves it from the queue. The service activator will poll messages every five seconds:

```
<context:component-scan base-package="xpadro.spring.integration.msg.store"/>

<import resource="mongodb-config.xml"/>

<int:gateway id="storeGateway" service-interface="xpadro.spring.integration.msg.store. ↩
    MsgStoreGateway"
    default-request-channel="entryChannel"/>

<int:channel id="entryChannel">
    <int:queue message-store="myMessageStore"/>
</int:channel>

<int:service-activator input-channel="entryChannel" ref="msgStoreActivator">
    <int:poller fixed-rate="5000"/>
</int:service-activator>
```

Maybe you have noticed the `myMessageStore` bean. In order to see how the persisting messages mechanism works, I have extended the `http://docs.spring.io/spring-integration/docs/3.0.2.RELEASE/api/org/springframework/integration/mongodb/store/ConfigurableMongoDbMessageStore.html[ConfigurableMongoDBMessageStore]` class to put logs in it and debug the result. If you want to try this, you can delete the MongoDB messageStore bean in `mongodb-config.xml` since we are no longer using it.

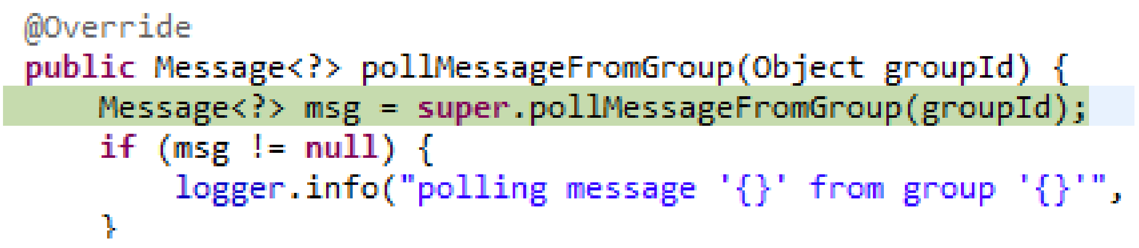
I have overwritten two methods:

```
@Component("myMessageStore")
public class MyMessageStore extends ConfigurableMongoDbMessageStore {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private static final String STORE_COLLECTION_NAME = "messageStoreCollection";

    @Autowired
    public MyMessageStore(MongoDbFactory mongoDbFactory) {
        super(mongoDbFactory, STORE_COLLECTION_NAME);
        logger.info("Creating message store '{}'", STORE_COLLECTION_NAME);
    }

    @Override
    public MessageGroup addMessageToGroup(Object groupId, Message<?> message) {
        logger.info("Adding message '{}' to group '{}'", message.getPayload(), groupId);
        return super.addMessageToGroup(groupId, message);
    }

    @Override
    public Message<?> pollMessageFromGroup(Object groupId) {
        Message<?> msg = super.pollMessageFromGroup(groupId);
        if (msg != null) {
            logger.info("polling message '{}' from group '{}'", msg.getPayload(), groupId);
        }
        else {
            logger.info("Polling null message from group {}", groupId);
        }

        return msg;
    }
}
```

This mechanism works as follows:

- When a message reaches the queue channel, which have our message store configured, it will invoke the `addMessageToGroup' method.This method will insert a document with the payload to the MongoDB collection specified in the constructor.This is done by using a ` http://docs.spring.io/spring-data/mongodb/docs/1.4.x/api/org/springframework/data/mongodb/core/MongoTemplate.html[MongoTemplate].`
- When the consumer polls the message, the `pollMessageFromGroup` will be invoked, retrieving the document from the collection.

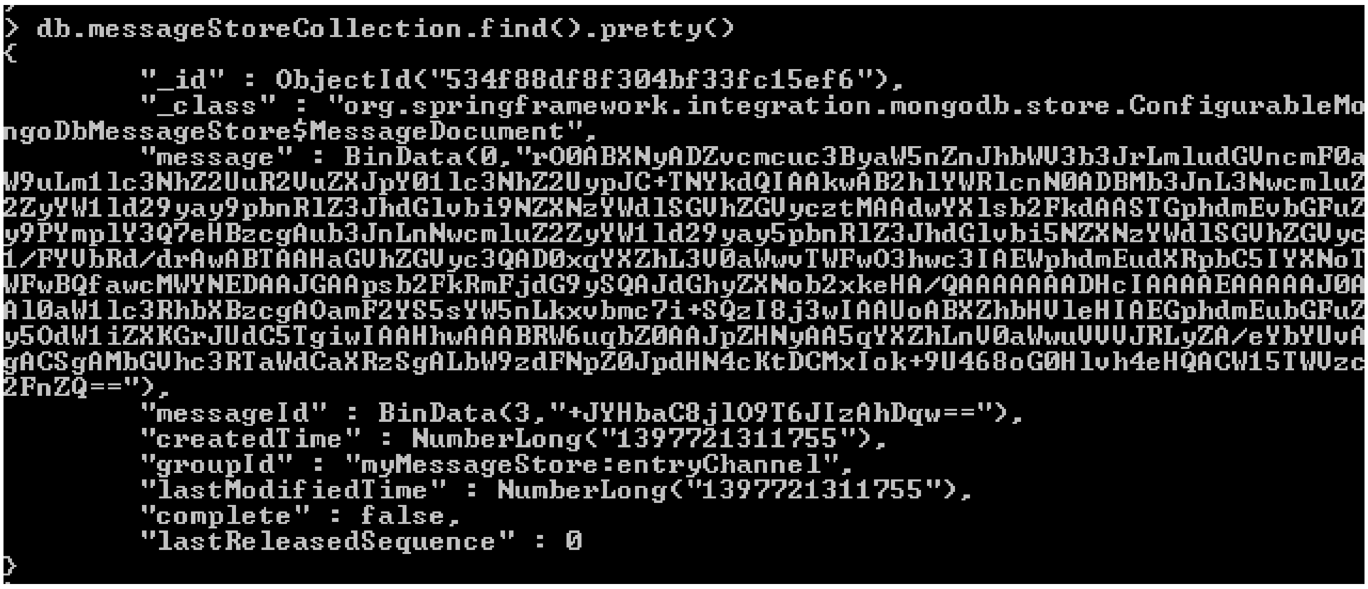
Let's see how it works by debugging the code. We will stop just before polling the message to see how it is stored in the database:



```
@Override
public Message<?> pollMessageFromGroup(Object groupId) {
    Message<?> msg = super.pollMessageFromGroup(groupId);
    if (msg != null) {
        logger.info("polling message '{}' from group '{}'",
    }
```

Figure 6.5: screenshot

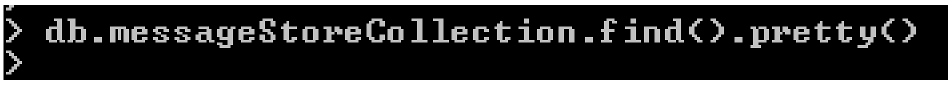
At this moment, we can take a look at the database:



```
> db.messageStoreCollection.find().pretty()
{
        "_id" : ObjectId("534f88df8f304bf33fc15ef6"),
        "_class" : "org.springframework.integration.mongodb.store.ConfigurableMo
ngoDbMessageStore$MessageDocument",
        "message" : BinData(0,"rO0ABXNyADZvcmcuc3ByaW5nZnJhbWV3b3JrLmludGVncmF0a
W9uLm1lc3NhZ2UuR2VuZXJpY01lc3NhZ2UypJC+TNYkdQIAAkwAB2hlYWRlcnN0ADBMb3JnL3NwcmluZ
2ZyYW1ld29yay9pbnRlZ3JhdGlvbi9NZXNzYWdlSGVhZGVyczt MAAdwYXlsb2FkdAASTGphdmEvbGFuZ
y9PYmplY3Q7eHBzcgAub3JnLnNwcmluZ2ZyYW1ld29yay5pbnRlZ3JhdGlvbi5NZXNzYWdlSGVhZGVyc
1/FYVbRd/drAwABTAAHaGVhZGVyc3QAD0xqYXZhL3V0aWwvTWFwO3hwc3IAEWphdmEudXRpbC5IYXNoT
WFwBQfawcMWYNEDAAJGAApsb2FkRmFjdG9ySQAJdGhyZXNob2xkeHA/QAAAAAAADHcIAAAAEAAAAAJ0A
Al0aW1lc3RhbXBzcgAOamF2YS5sYW5nLkxvbmc7i+SQzI8j3wIAAUoABXZhbHVleHIAEGphdmEubGFuZ
y5OdW1iZXKGrJUdC5TgiwIAAHhwAAABRW6uqbZ0AAJpZHNyAA5qYXZhLnV0aWwuVVVJRLyZA/eYbYUvA
gACSgAMbGVhc3RTaWdCaXRzSgALbW9zdFNpZ0JpdHN4cKtDCMxIok+9U468oG0Hlvh4eHQACW15TWVzc
2FnZQ=="),
        "messageId" : BinData(3,"+JYHbaC8jlO9T6JIzAhDqw=="),
        "createdTime" : NumberLong("1397721311755"),
        "groupId" : "myMessageStore:entryChannel",
        "lastModifiedTime" : NumberLong("1397721311755"),
        "complete" : false,
        "lastReleasedSequence" : 0
}
```

Figure 6.6: screenshot

Once resumed, the message is polled from the collection:



```
> db.messageStoreCollection.find().pretty()
>
```

Figure 6.7: screenshot

The application that runs the example:

```
public class MsgStoreApp {
    public static void main(String[] args) throws InterruptedException {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("classpath: ←
            xpadro/spring/integration/msg/store/config/int-msg-store-config.xml");

        MsgStoreGateway gateway = context.getBean(MsgStoreGateway.class);

        gateway.send("myMessage");
        Thread.sleep(30000);
        context.close();
    }
}
```

6.8 Implementing idempotent components

If our application needs to avoid duplicate messages, Spring Integration provides this mechanism by implementing the idempotent receiver pattern. The responsible of detecting duplicate messages is the metadata store component. This component consists in storing key-value pairs. The framework provides two implementations of the interface `http://docs.spring.io/spring-integration/docs/3.0.2.RELEASE/api/org/springframework/integration/metadata/MetadataStore.html[MetadataStore]`:

- SimpleMetadataStore: Default implementation. It stores the information using an in-memory map.
- PropertiesPersistingMetadataStore: Useful if you need to persist the data. It uses a properties file. We are going to use this implementation in our example.

OK, let’s start with the configuration file:

```
<context:component-scan base-package="xpadro.spring.integration.msg.metadata"/>

<bean id="metadataStore"
class="org.springframework.integration.metadata.PropertiesPersistingMetadataStore"/>

<int:gateway id="metadataGateway" service-interface="xpadro.spring.integration.msg.metadata ↩
    .MetadataGateway"
    default-request-channel="entryChannel"/>

<int:channel id="entryChannel"/>

<int:filter input-channel="entryChannel" output-channel="processChannel"
    discard-channel="discardChannel" expression="@metadataStore.get(headers.messageId) ==  ↩
        null"/>


<!-- Process message -->
<int:publish-subscribe-channel id="processChannel"/>

<int:outbound-channel-adapter channel="processChannel" expression="@metadataStore.put( ↩
    headers.messageId, '')"/>

<int:service-activator input-channel="processChannel" ref="metadataActivator" method=" ↩
    process"/>


<!-- Duplicated message - discard it -->
<int:channel id="discardChannel"/>

<int:service-activator input-channel="discardChannel" ref="metadataActivator" method=" ↩
    discard"/>
```

We have defined a “metadataStore” in order to use our properties metadata store instead of the default in-memory implementation.

The flow is explained here:

- A message is sent to the gateway.
- The filter will send the message to the process channel since it is the first time it is sent.
- There are two subscribers to the process channel: the service activator that processes the message and an outbound channel adapter. The channel adapter will send the value of the message header `messagId` to the metadata store.

- The metadata store stores the value in the properties file.
- Next time the same message is sent; the filter will find the value and discard the message.

The metadata store creates a properties file in the file system. If you are using Windows, you will see a metadata-store.properties file in the 'C:UsersusernameAppDataLocalTempspring-integration' folder

The example uses a service activator to log if the message has been processed:

```
@Component("metadataActivator")
public class MetadataActivator {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    public void process(Message<String> msg) {
        logger.info("Message processed: {}", msg.getPayload());
    }

    public void discard(Message<String> msg) {
        logger.info("Message discarded: {}", msg.getPayload());
    }
}
```

The application will run the example:

```
public class MetadataApp {
    private static final String MESSAGE_STORE_HEADER = "messageId";

    public static void main(String[] args) throws InterruptedException {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("classpath: ↩
            xpadro/spring/integration/msg/metadata/config/int-msg-metadata-config.xml");

        MetadataGateway gateway = context.getBean(MetadataGateway.class);

        Map<String,String> headers = new HashMap<>();
        headers.put(MESSAGE_STORE_HEADER, "msg1");
        Message<String> msg1 = MessageBuilder.withPayload("msg1").copyHeaders(headers). ↩
            build();

        headers = new HashMap<>();
        headers.put(MESSAGE_STORE_HEADER, "msg2");
        Message<String> msg2 = MessageBuilder.withPayload("msg2").copyHeaders(headers). ↩
            build();

        gateway.sendMessage(msg1);
        Thread.sleep(500);
        gateway.sendMessage(msg1);
        Thread.sleep(500);
        gateway.sendMessage(msg2);

        Thread.sleep(3000);
        context.close();
    }
}
```

The first invocation will result in the following output on the console:

```
2014-04-17 13:00:08,223|MetadataActivator|Message processed: msg1
2014-04-17 13:00:08,726|MetadataActivator|Message discarded: msg1
2014-04-17 13:00:09,229|MetadataActivator|Message processed: msg2
```

Now remember that the PropertiesPersistingMetadataStore stores the data in a properties file. This means that the data will survive ApplicationContext restarts. So, if we don't delete the properties file and we run the example again, the result will be different:

```
2014-04-17 13:02:27,117|MetadataActivator|Message discarded: msg1
2014-04-17 13:02:27,620|MetadataActivator|Message discarded: msg1
2014-04-17 13:02:28,123|MetadataActivator|Message discarded: msg2
```

6.9 Sending operation invocation requests

The last mechanism discussed on this tutorial is the control bus. The control bus will let you manage the system the same way it is done by the application. The message will be executed as a Spring Expression Language. To be executable from the control bus, the method needs to use the @ManagedAttribute or @ManagedOperation annotation.

This section's example uses a control bus to invoke a method on a bean:

```
<context:component-scan base-package="xpadro.spring.integration.control.bus"/>

<int:channel id="entryChannel"/>

<int:control-bus input-channel="entryChannel" output-channel="resultChannel"/>

<int:channel id="resultChannel"/>

<int:service-activator input-channel="resultChannel" ref="controlbusActivator"/>
```

The operation that will be invoked is as follows:

```
@Component("controlbusBean")
public class ControlBusBean {

    @ManagedOperation
    public String greet(String name) {
        return "Hello " + name;
    }
}
```

The application that runs the example sends a message with the expression to be executed:

```
public class ControlBusApp {
    public static void main(String[] args) throws InterruptedException {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("classpath: ↩
            xpadro/spring/integration/control/bus/config/int-control-bus-config.xml");
        MessageChannel channel = context.getBean("entryChannel", MessageChannel.class);

        Message<String> msg = MessageBuilder.withPayload("@controlbusBean.greet('World!')") ↩
            .build();
        channel.send(msg);

        Thread.sleep(3000);
        context.close();
    }
}
```

The result is shown on the console:

```
2014-04-17 13:21:42,910|ControlBusActivator|Message received: Hello World!
```