Appendix E. Spring Integration Samples

E.1 Introduction

As of Spring Integration 2.0, the *samples* are no longer included with the Spring Integration distribution. Instead we have switched to a much simpler collaborative model that should promote better community participation and, ideally, more contributions. Samples now have a dedicated Git repository and a dedicated JIRA Issue Tracking system. Sample development will also have its own lifecycle which is not dependent on the lifecycle of the framework releases, although the repository will still be tagged with each major release for compatibility reasons.

The great benefit to the community is that we can now add more samples and make them available to you right away without waiting for the next release. Having its own JIRA that is not tied to the the actual framework is also a great benefit. You now have a dedicated place to suggest samples as well as report issues with existing samples. Or, _ you may want to submit a sample to us_ as an attachment through the JIRA or, better, through the collaborative model that Git promotes. If we believe your sample adds value, we would be more then glad to add it to the *samples* repository, properly crediting you as the author.

E.2 Where to get Samples

The Spring Integration Samples project is hosted on GitHub. You can find the repository at:

In order to check out or *clone* (Git parlance) the samples, please make sure you have a Git client installed on your system. There are several GUI-based products available for many platforms, e.g. EGit for the Eclipse IDE. A simple Google search will help you find them. Of course you can also just use the command line interface for <http://git-scm.com/,Git>.

If you need more information on how to install and/or use Git, please visit: http://git-scm.com/.

In order to checkout (clone in Git terms) the Spring Integration samples repository using the Git command line tool, issue the following commands:

```
$ git clone https://github.com/SpringSource/spring-integration-samples.git
```

That is all you need to do in order to clone the entire samples repository into a directory named *spring-integration-samples* within the working directory where you issued that *git* command. Since the samples repository is a live repository, you might want to perform periodic *pulls* (updates) to get new samples, as well as updates to the existing samples. In order to do so issue the following git *pull* command:

```
$ git pull
```

E.3 Submitting Samples or Sample Requests

How can I contribute my own Samples?
Github is for social coding: if you want to submit your own code examples to the Spring Integration Samples project, we encourage contributions through *pull requests* from *forks* of this repository. If you want to contribute code this way, please reference, if possible, ahttps://jira.springframework.org/browse/INTSAMPLES[*JIRA Ticket*] that provides some details regarding the provided sample.

Sign the contributor license agreement

Very important: before we can accept your Spring Integration sample, we will need you to sign the SpringSource contributor license agreement (CLA). Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. In order to read and sign the CLA, please go to:

https://support.springsource.com/spring_committer_signup

From the Project drop down, please select *Spring Integration*. The Project Lead is *Gary Russell*.

Code Contribution Process
For the actual code contribution process, please read the the *Contributor Guidelines* for Spring Integration, they apply for this project as well:

This process ensures that every commit gets peer-reviewed. As a matter of fact, the core committers follow the exact same rules. We are gratefully looking forward to your Spring Integration Samples!

Sample Requests
As mentioned earlier, the *Spring Integration Samples* project has a dedicated JIRA Issue tracking system. To submit new sample requests, please visit our JIRA Issue Tracking system:

E.4 Samples Structure

Starting with Spring Integration 2.0, the structure of the *samples* changed as well. With plans for more samples we realized that some samples have different goals than others. While they all share the common goal of showing you how to apply and work with the Spring Integration framework, they also differ in areas where some samples are meant to concentrate on a technical use case while others focus on a business use case, and some samples are all about showcasing various techniques that could be applied to address certain scenarios (both technical and business). The new categorization of samples will allow us to better organize them based on the problem each sample addresses while giving you a simpler way of finding the right sample for your needs.

Currently there are 4 categories. Within the samples repository each category has its own directory which is named after the category name:

BASIC (samples/basic)
This is a good place to get started. The samples here are technically motivated and demonstrate the bare minimum with regard to configuration and code. These should help you to get started quickly by introducing you to the basic concepts, API and configuration of Spring Integration as well as Enterprise Integration Patterns (EIP). For example, if you are looking for an answer on how to implement and wire a *Service Activator* to a *Message Channel* or how to use a *Messaging Gateway* as a facade to your message exchange, or how to get started with using MAIL or TCP/UDP modules etc., this would be the right place to find a good sample. The bottom line is this is a good place to get started.

INTERMEDIATE (samples/intermediate)
This category targets developers who are already familiar with the Spring Integration framework (past getting started), but need some more guidance while resolving the more advanced technical problems one might deal with after switching to a Messaging architecture. For example, if you are looking for an answer on how to handle errors in various message exchange scenarios or how to properly configure the *Aggregator* for the situations where some messages might not ever arrive for aggregation, or any other issue that goes beyond a basic implementation and configuration of a particular component and addresses *what else* types of problems, this would be the right place to find these type of samples.

ADVANCED (samples/advanced)
This category targets developers who are very familiar with the Spring Integration framework but are looking to extend it to address a specific custom need by using Spring Integration’s public API. For example, if you are looking for samples showing you how to implement a custom *Channel* or *Consumer* (event-based or polling-based), or you are trying to figure out what is the most appropriate way to implement a custom Bean parser on top of the Spring Integration Bean parser hierarchy when implementing your own namespace and schema for a custom component, this would be the right place to look. Here you can also find samples that will help you with *Adapter* development. Spring Integration comes with an extensive library of adapters to allow you to connect remote systems with the Spring Integration messaging framework. However you might have a need to integrate with a system for which the core framework does not provide an adapter. So, you may decide to implement your own (and potentially contribute it). This category would include samples showing you how.

APPLICATIONS (samples/applications)
This category targets developers and architects who have a good understanding of Message-driven architecture and EIP, and an above average understanding of Spring and Spring Integration who are looking for samples that address a particular *business problem*. In other words the emphasis of samples in this category is *business use cases* and how they can be solved with a Message-Driven Architecture and Spring Integration in particular. For example, if you are interested to see how a *Loan Broker* or *Travel Agent* process could be implemented and automated via Spring Integration, this would

be the right place to find these types of samples.

Important

Remember: Spring Integration is a community driven framework, therefore community participation is IMPORTANT. That includes Samples; so, if you can't find what you are looking for, let us know!

E.5 Samples

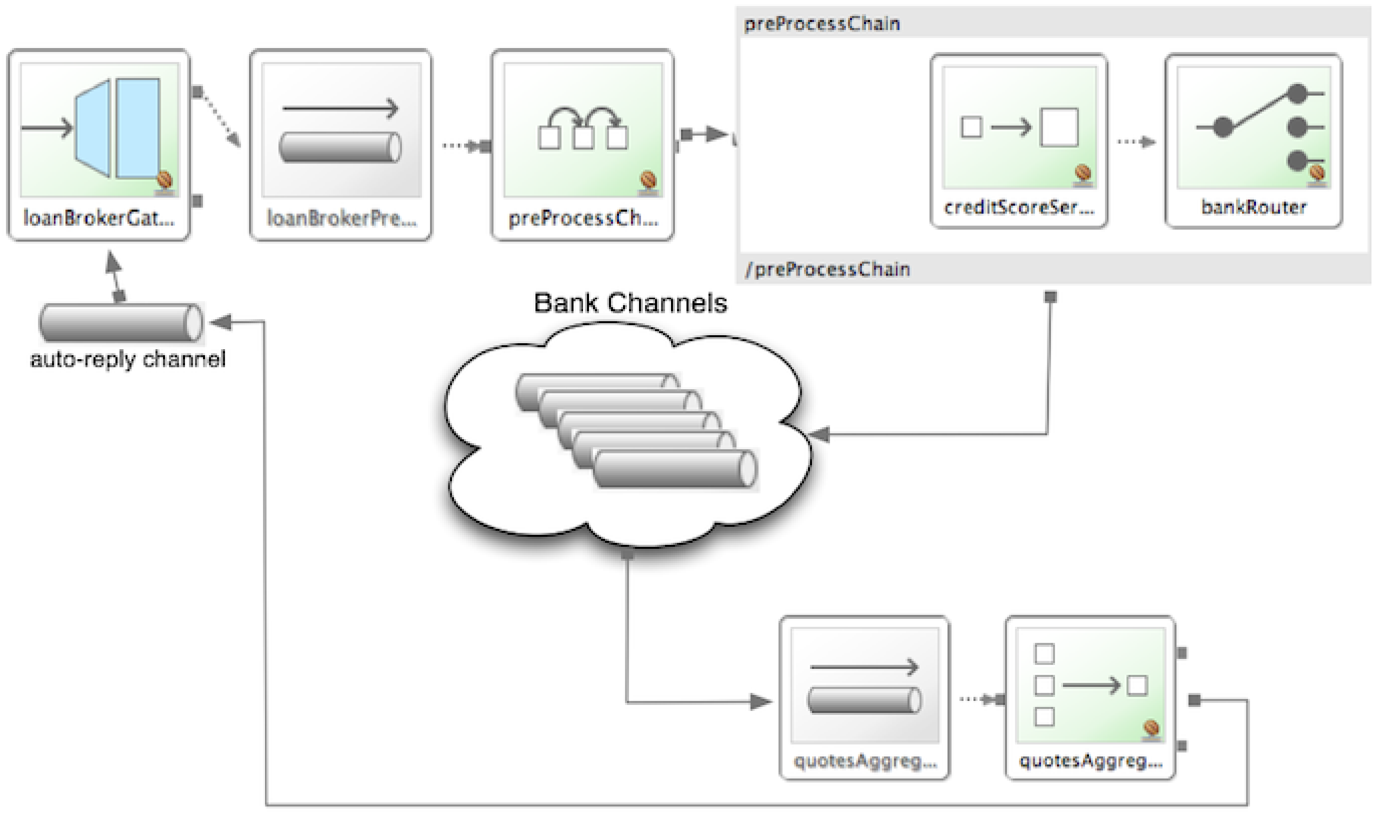
Currently Spring Integration comes with quite a few samples and you can only expect more. To help you better navigate through them, each sample comes with its own `readme.txt` file which covers several details about the sample (e.g., what EIP patterns it addresses, what problem it is trying to solve, how to run sample etc.). However, certain samples require a more detailed and sometimes graphical explanation. In this section you'll find details on samples that we believe require special attention.

E.5.1 Loan Broker

In this section, we will review the *Loan Broker* sample application that is included in the Spring Integration samples. This sample is inspired by one of the samples featured in Gregor Hohpe and Bobby Woolf's book, Enterprise Integration Patterns.

The diagram below represents the entire process

Figure E.1. Loan Broker Sample



Now lets look at this process in more detail

At the core of an EIP architecture are the very simple yet powerful concepts of Pipes and Filters, and of course: Messages. Endpoints (Filters) are connected with one another via Channels (Pipes). The producing endpoint sends Message to the Channel, and the Message is retrieved by the Consuming endpoint. This architecture is meant to define various mechanisms that describe HOW information is exchanged between the endpoints, without any awareness of WHAT those endpoints are or what information they are exchanging. Thus, it provides for a very loosely coupled and flexible collaboration model while also decoupling Integration concerns from Business concerns. EIP extends this architecture by further defining:

- The types of pipes (Point-to-Point Channel, Publish-Subscribe Channel, Channel Adapter, etc.)
- The core filters and patterns around how filters collaborate with pipes (Message Router, Splitters and Aggregators, various Message Transformation patterns, etc.)

The details and variations of this use case are very nicely described in Chapter 9 of the EIP Book, but here is the brief summary; A Consumer while shopping for the best Loan Quote(s) subscribes to the services of a Loan Broker, which handles details such as:

- Consumer pre-screening (e.g., obtain and review the consumer’s Credit history)
- Determine the most appropriate Banks (e.g., based on consumer’s credit history/score)
- Send a Loan quote request to each selected Bank
- Collect responses from each Bank
- Filter responses and determine the best quote(s), based on consumer’s requirements.
- Pass the Loan quote(s) back to the consumer.

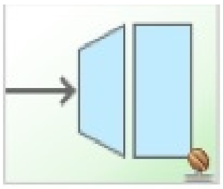
Obviously the real process of obtaining a loan quote is a bit more complex, but since our goal here is to demonstrate how Enterprise Integration Patterns are realized and implemented within SI, the use case has been simplified to concentrate only on the Integration aspects of the process. It is not an attempt to give you an advice in consumer finances.

As you can see, by hiring a Loan Broker, the consumer is isolated from the details of the Loan Broker’s operations, and each Loan Broker’s operations may defer from one another to maintain competitive advantage, so whatever we assemble/implement must be flexible so any changes could be introduced quickly and painlessly. Speaking of change, the Loan Broker sample does not actually talk to any *imaginary* Banks or Credit bureaus. Those services are stubbed out. Our goal here is to assemble, orchestrate and test the integration aspect of the process as a whole. Only then can we start thinking about wiring such process to the real services. At that time the assembled process and its configuration will not change regardless of the number of Banks a particular Loan Broker is dealing with, or the type of communication media (or protocols) used (JMS, WS, TCP, etc.) to communicate with these Banks.

DESIGN
As you analyze the 6 requirements above you’ll quickly see that they all fall into the category of Integration concerns. For example, in the consumer pre-screening step we need to gather additional

information about the consumer and the consumer’s desires and enrich the loan request with additional meta information. We then have to filter such information to select the most appropriate list of Banks, and so on. Enrich, filter, select – these are all integration concerns for which EIP defines a solution in the form of patterns. SI provides an implementation of these patterns.

Figure E.2. Messaging Gateway



The *Messaging Gateway* pattern provides a simple mechanism to access messaging systems, including our Loan Broker. In SI you define the *Gateway* as a Plain Old Java Interface (no need to provide an implementation), configure it via the XML <*gateway*> element or via annotation and use it as any other Spring bean. SI will take care of delegating and mapping method invocations to the Messaging infrastructure by generating a *Message* (payload is mapped to an input parameter of the method) and sending it to the designated channel.

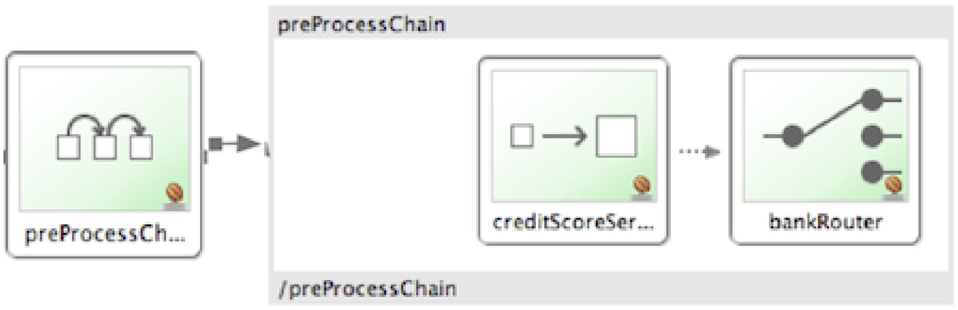
```
<int:gateway id="loanBrokerGateway"
  default-request-channel="loanBrokerPreProcessingChannel"
  service-
interface="org.springframework.integration.samples.loanbroker.LoanBrokerGa
teway">
  <int:method name="getBestLoanQuote">
    <int:header name="RESPONSE_TYPE" value="BEST"/>
  </int:method>
</int:gateway>
```

Our current *Gateway* provides two methods that could be invoked. One that will return the best single quote and another one that will return all quotes. Somehow downstream we need to know what type of reply the caller is looking for. The best way to achieve this in Messaging architecture is to enrich the content of the message with some meta-data describing your intentions. *Content Enricher* is one of the patterns that addresses this and although Spring Integration does provide a separate configuration element to enrich Message Headers with arbitrary data (we’ll see it later), as a convenience, since_Gateway_ element is responsible to construct the initial *Message* it provides embedded capability to enrich the newly created *Message* with arbitrary *Message Headers*. In our example we are adding header RESPONSE_TYPE with value *BEST*' whenever the getBestQuote() method is invoked. For other method we are not adding any header. Now we can check downstream for an existence of this header and based on its presence and its value we can determine what type of reply the caller is looking for.

Based on the use case we also know there are some pre-screening steps that needs to be performed such as getting and evaluating the consumer’s credit score, simply because some premiere Banks will only typically accept quote requests from consumers that meet a minimum credit score requirement. So it would be nice if the *Message* would be enriched with such information before it is forwarded to the Banks. It would also be nice if when several processes needs to be completed to provide such meta-information, those processes could be grouped in a single unit. In our use case we need to determine credit score and based on the credit score and some rule select a list of *Message Channels* (Bank Channels) we will sent quote request to.

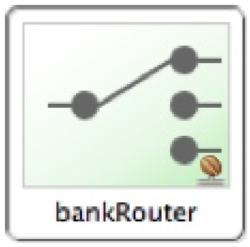
Composed Message Processor
The *Composed Message Processor* pattern describes rules around building endpoints that maintain control over message flow which consists of multiple message processors. In Spring Integration *Composed Message Processor* pattern is implemented via <*chain*> element.

Figure E.3. Chain



As you can see from the above configuration we have a chain with inner header-enricher element which will further enrich the content of the *Message* with the header CREDIT_SCORE and value that will be determined by the call to a credit service (simple POJO spring bean identified by *creditBureau* name) and then it will delegate to the *Message Router*

Figure E.4. Message Router



There are several implementations of the *Message Routing* pattern available in Spring Integration. Here we are using a router that will determine a list of channels based on evaluating an expression (Spring Expression Language) which will look at the credit score that was determined is the previous step and will select the list of channels from the Map bean with id *banks* whose values are *premier* or *secondary* based o the value of credit score. Once the list of *Channels* is selected, the *Message* will be routed to those *Channels*.

Now, one last thing the Loan Broker needs to to is to receive the loan quotes form the banks, aggregate them by consumer (we don’t want to show quotes from one consumer to another), assemble the response based on the consumer’s selection criteria (single best quote or all quotes) and reply back to the consumer.

Figure E.5. Message Aggregator

An *Aggregator* pattern describes an endpoint which groups related *Messages* into a single *Message*. Criteria and rules can be provided to determine an aggregation and correlation strategy. SI provides several implementations of the *Aggregator* pattern as well as a convenient name-space based configuration.

```
<int:aggregator id="quotesAggregator"
      input-channel="quotesAggregationChannel"
      method="aggregateQuotes">
  <beans:bean
class="org.springframework.integration.samples.loanbroker.LoanQuoteAggrega
tor"/>
</int:aggregator>
```

Our Loan Broker defines a *quotesAggregator* bean via the *<aggregator>* element which provides a default aggregation and correlation strategy. The default correlation strategy correlates messages based on the `correlationId` header (see *Correlation Identifier* pattern). What's interesting is that we never provided the value for this header. It was set earlier by the router automatically, when it generated a separate *Message* for each Bank channel.

Once the *Messages* are correlated they are released to the actual *Aggregator* implementation. Although default *Aggregator* is provided by SI, its strategy (gather the list of payloads from all *Messages* and construct a new *Message* with this List as payload) does not satisfy our requirement. The reason is that our consumer might require a single best quote or all quotes. To communicate the consumer's intention, earlier in the process we set the RESPONSE_TYPE header. Now we have to evaluate this header and return either all the quotes (the default aggregation strategy would work) or the best quote (the default aggregation strategy will not work because we have to determine which loan quote is the best).

Obviously selecting the best quote could be based on complex criteria and would influence the complexity of the aggregator implementation and configuration, but for now we are making it simple. If consumer wants the best quote we will select a quote with the lowest interest rate. To accomplish that the LoanQuoteAggregator.java will sort all the quotes and return the first one. The `LoanQuote.java` implements `Comparable` which compares quotes based on the rate attribute. Once the response *Message* is created it is sent to the default-reply-channel of the *Messaging Gateway* (thus the consumer) which started the process. Our consumer got the Loan Quote!

Conclusion

As you can see a rather complex process was assembled based on POJO (read existing, legacy), light weight, embeddable messaging framework (Spring Integration) with a loosely coupled programming model intended to simplify integration of heterogeneous systems without requiring a heavy-weight ESB-like engine or proprietary development and deployment environment, because as a developer you should not be porting your Swing or console-based application to an ESB-like server or implementing proprietary interfaces just because you have an integration concern.

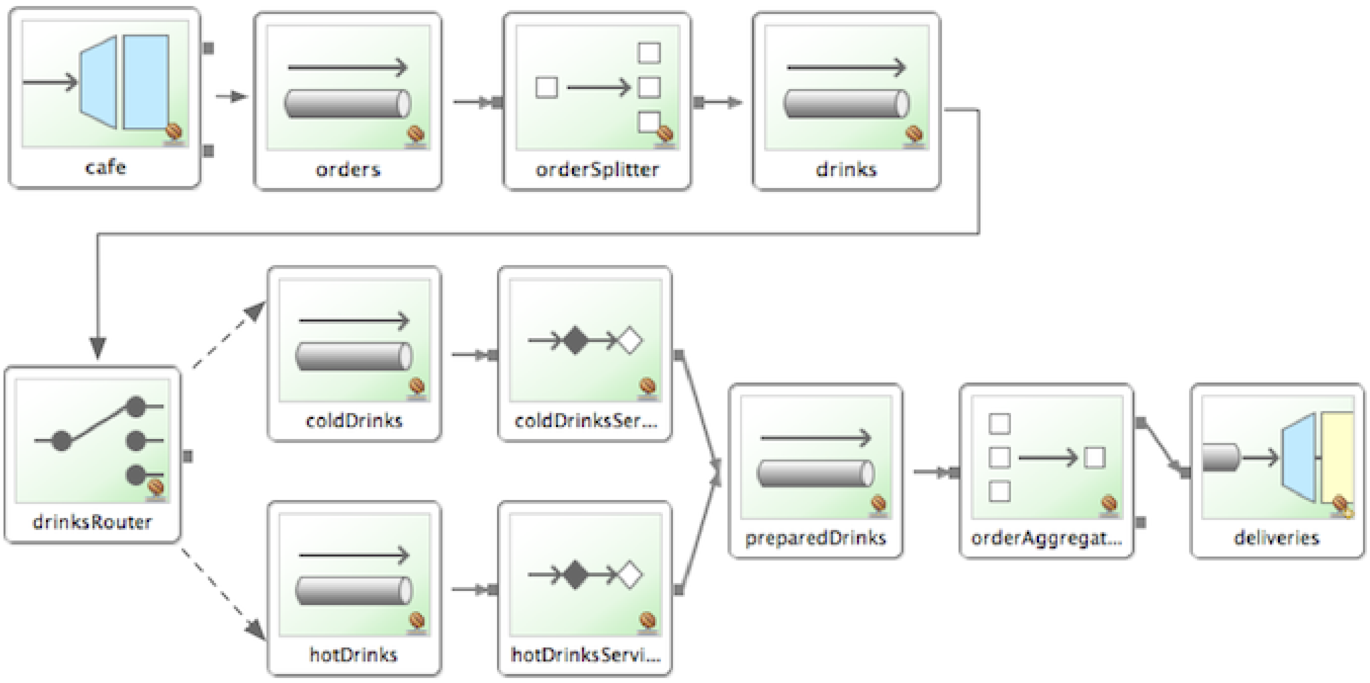
This and other samples in this section are built on top of Enterprise Integration Patterns and can be considered "building blocks" for YOUR solution; they are not intended to be complete solutions. Integration concerns exist in all types of application (whether server based or not). It should not require change in design, testing and deployment strategy if such applications need to be integrated.

E.5.2 The Cafe Sample

In this section, we will review a *Cafe* sample application that is included in the Spring Integration samples. This sample is inspired by another sample featured in Gregor Hohpe’s http://www.eaipatterns.com/ramblings.html[Ramblings].

The domain is that of a Cafe, and the basic flow is depicted in the following diagram:

Figure E.6. Cafe Sample



The `Order` object may contain multiple `OrderItems`. Once the order is placed, a *Splitter* will break the composite order message into a single message per drink. Each of these is then processed by a *Router* that determines whether the drink is hot or cold (checking the `OrderItem` object’s *isIced* property). The `Barista` prepares each drink, but hot and cold drink preparation are handled by two distinct methods: *prepareHotDrink* and *prepareColdDrink*. The prepared drinks are then sent to the Waiter where they are aggregated into a `Delivery` object.

Here is the XML configuration:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:int="http://www.springframework.org/schema/integration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:beans="http://www.springframework.org/schema/beans"
 xmlns:int-
stream="http://www.springframework.org/schema/integration/stream"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
  http://www.springframework.org/schema/integration/spring-integration.xsd
  http://www.springframework.org/schema/integration/stream
  http://www.springframework.org/schema/integration/stream/spring-
integration-stream.xsd">

    <int:gateway id="cafe" service-interface="o.s.i.samples.cafe.Cafe"/>

    <int:channel  id="orders"/>
    <int:splitter input-channel="orders" ref="orderSplitter"
                  method="split" output-channel="drinks"/>

    <int:channel id="drinks"/>
    <int:router  input-channel="drinks"
                 ref="drinkRouter" method="resolveOrderItemChannel"/>

    <int:channel id="coldDrinks"><int:queue capacity="10"/></int:channel>
    <int:service-activator input-channel="coldDrinks" ref="barista"
                           method="prepareColdDrink" output-
channel="preparedDrinks"/>

    <int:channel id="hotDrinks"><int:queue capacity="10"/></int:channel>
    <int:service-activator input-channel="hotDrinks" ref="barista"
                           method="prepareHotDrink" output-
channel="preparedDrinks"/>

    <int:channel id="preparedDrinks"/>
    <int:aggregator input-channel="preparedDrinks" ref="waiter"
                    method="prepareDelivery" output-channel="deliveries"/>

    <int-stream:stdout-channel-adapter id="deliveries"/>

    <beans:bean id="orderSplitter"

class="org.springframework.integration.samples.cafe.xml.OrderSplitter"/>
```

```
    <beans:bean id="drinkRouter"

class="org.springframework.integration.samples.cafe.xml.DrinkRouter"/>

    <beans:bean id="barista" class="o.s.i.samples.cafe.xml.Barista"/>
    <beans:bean id="waiter"  class="o.s.i.samples.cafe.xml.Waiter"/>

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

</beans:beans>
```

As you can see, each Message Endpoint is connected to input and/or output channels. Each endpoint will manage its own Lifecycle (by default endpoints start automatically upon initialization - to prevent that add the "auto-startup" attribute with a value of "false"). Most importantly, notice that the objects are simple POJOs with strongly typed method arguments. For example, here is the Splitter:

```
public class OrderSplitter {
    public List<OrderItem> split(Order order) {
        return order.getItems();
    }
}
```

In the case of the Router, the return value does not have to be a `MessageChannel` instance (although it can be). As you see in this example, a String-value representing the channel name is returned instead.

```
public class DrinkRouter {
    public String resolveOrderItemChannel(OrderItem orderItem) {
        return (orderItem.isIced()) ? "coldDrinks" : "hotDrinks";
    }
}
```

Now turning back to the XML, you see that there are two <service-activator> elements. Each of these is delegating to the same `Barista` instance but different methods: *prepareHotDrink* or *prepareColdDrink* corresponding to the two channels where order items have been routed.

```
public class Barista {

    private long hotDrinkDelay = 5000;
    private long coldDrinkDelay = 1000;

    private AtomicInteger hotDrinkCounter = new AtomicInteger();
    private AtomicInteger coldDrinkCounter = new AtomicInteger();
    public void setHotDrinkDelay(long hotDrinkDelay) {
        this.hotDrinkDelay = hotDrinkDelay;
    }

    public void setColdDrinkDelay(long coldDrinkDelay) {
        this.coldDrinkDelay = coldDrinkDelay;
    }

    public Drink prepareHotDrink(OrderItem orderItem) {
        try {
            Thread.sleep(this.hotDrinkDelay);
            System.out.println(Thread.currentThread().getName()
                    + " prepared hot drink #" +
hotDrinkCounter.incrementAndGet()
                    + " for order #" + orderItem.getOrder().getNumber()
                    + ": " + orderItem);
            return new Drink(orderItem.getOrder().getNumber(),
orderItem.getDrinkType(),
                    orderItem.isIced(), orderItem.getShots());
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return null;
        }
    }

    public Drink prepareColdDrink(OrderItem orderItem) {
        try {
            Thread.sleep(this.coldDrinkDelay);
            System.out.println(Thread.currentThread().getName()
                    + " prepared cold drink #" +
coldDrinkCounter.incrementAndGet()
                    + " for order #" + orderItem.getOrder().getNumber() +
": "
                    + orderItem);
            return new Drink(orderItem.getOrder().getNumber(),
orderItem.getDrinkType(),
                    orderItem.isIced(), orderItem.getShots());
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return null;
        }
    }
}
```

As you can see from the code excerpt above, the barista methods have different delays (the hot drinks take 5 times as long to prepare). This simulates work being completed at different rates. When the`CafeDemo` *main* method runs, it will loop 100 times sending a single hot drink and a single cold drink each time. It actually sends the messages by invoking the *placeOrder* method on the Cafe interface. Above, you will see that the <gateway> element is specified in the configuration file. This triggers the creation of a proxy that implements the given *service-interface* and connects it to a channel. The channel name is provided on the @Gateway annotation of the `Cafe` interface.

```
public interface Cafe {

    @Gateway(requestChannel="orders")
    void placeOrder(Order order);

}
```

Finally, have a look at the `main()` method of the `CafeDemo` itself.

```
public static void main(String[] args) {
    AbstractApplicationContext context = null;
    if (args.length > 0) {
        context = new FileSystemXmlApplicationContext(args);
    }
    else {
        context = new ClassPathXmlApplicationContext("cafeDemo.xml",
CafeDemo.class);
    }
    Cafe cafe = context.getBean("cafe", Cafe.class);
    for (int i = 1; i <= 100; i++) {
        Order order = new Order(i);
        order.addItem(DrinkType.LATTE, 2, false);
        order.addItem(DrinkType.MOCHA, 3, true);
        cafe.placeOrder(order);
    }
}
```

To run this sample as well as 8 others, refer to the `README.txt` within the "samples" directory of the main distribution as described at the beginning of this chapter.

When you run cafeDemo, you will see that the cold drinks are initially prepared more quickly than the hot drinks. Because there is an aggregator, the cold drinks are effectively limited by the rate of the hot drink preparation. This is to be expected based on their respective delays of 1000 and 5000 milliseconds. However, by configuring a poller with a concurrent task executor, you can dramatically change the results. For example, you could use a thread pool executor with 5 workers for the hot drink barista while keeping the cold drink barista as it is:

```
<int:service-activator input-channel="hotDrinks"
                     ref="barista"
                     method="prepareHotDrink"
                     output-channel="preparedDrinks"/>

  <int:service-activator input-channel="hotDrinks"
                     ref="barista"
                     method="prepareHotDrink"
                     output-channel="preparedDrinks">
      <int:poller task-executor="pool" fixed-rate="1000"/>
  </int:service-activator>

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

Also, notice that the worker thread name is displayed with each invocation. You will see that the hot drinks are prepared by the task-executor threads. If you provide a much shorter poller interval (such as 100 milliseconds), then you will notice that occasionally it throttles the input by forcing the task-scheduler (the caller) to invoke the operation.

In addition to experimenting with the poller’s concurrency settings, you can also add the *transactional* sub-element and then refer to any PlatformTransactionManager instance within the context.

E.5.3 The XML Messaging Sample

The xml messaging sample in `basic/xml` illustrates how to use some of the provided components which deal with xml payloads. The sample uses the idea of processing an order for books represented as xml.

NOTE:This sample shows that the namespace prefix can be whatever you want; while we usually use, `int-xml` for integration XML components, the sample uses `si-xml`.

First the order is split into a number of messages, each one representing a single order item using the XPath splitter component.

```
<si-xml:xpath-splitter id="orderItemSplitter" input-
channel="ordersChannel"
```

```
              output-channel="stockCheckerChannel" create-
documents="true">
      <si-xml:xpath-expression
expression="/orderNs:order/orderNs:orderItem"
                                namespace-map="orderNamespaceMap" />
  </si-xml:xpath-splitter>
```

A service activator is then used to pass the message into a stock checker POJO. The order item document is enriched with information from the stock checker about order item stock level. This enriched order item message is then used to route the message. In the case where the order item is in stock the message is routed to the warehouse.

```
<si-xml:xpath-router id="instockRouter" input-
channel="orderRoutingChannel" resolution-required="true">
    <si-xml:xpath-expression expression="/orderNs:orderItem/@in-stock"
namespace-map="orderNamespaceMap" />
    <si-xml:mapping value="true" channel="warehouseDispatchChannel"/>
    <si-xml:mapping value="false" channel="outOfStockChannel"/>
</si-xml:xpath-router>
```

Where the order item is not in stock the message is transformed using xslt into a format suitable for sending to the supplier.

```
<si-xml:xslt-transformer input-channel="outOfStockChannel"
  output-channel="resupplyOrderChannel"
  xsl-
resource="classpath:org/springframework/integration/samples/xml/bigBooksSu
pplierTransformer.xsl"/>
```