GlobalLogic®

Enterprise Application Integration Using Spring Integration Framework

Sourabh Goel, Engineering Manager
Anshul Rohilla, Engineering Developer

Noida, India

This white paper explores a use case in which GlobalLogic implemented an enterprise application integration (EAI) solution for a customer to manage multiple challenges such as integrating heterogeneous information systems, managing workflows, applying queuing theory for asynchronous communication, managing messaging and security, and leveraging services-oriented architecture. We will provide a basic introduction of EAI, discuss why we chose to use a Spring Integration Framework for our use case, and demonstrate how we implemented the framework to build an extensive, scalable, enterprise-wide middleware solution.

www.globallogic.com

Sourabh Goel, Engineering Manager
Anshul Rohilla, Engineering Developer

Table of Contents

The Initial Challenge **3**

What is Enterprise Application Integration? **3**

Spring Integration Framework **4**
Spring Integration for FTP 5
Extending Spring Integration for SOAP Services 7

Middleware Solution Using Spring Integration **9**

Conclusion **9**

The Initial Challenge

The GlobalLogic customer in this use case is a U.S. telecom services provider that partners with multiple vendors all over the country to sell its products and services. Some of these vendors are themselves major corporations with proprietary data structures for managing order information. This order information is compiled together based on a package or standalone products that the vendors sell to end users.

The customer's existing solution for vendor integration was scattered across multiple applications, each dedicated to managing both upstream and downstream communications with a specific vendor's information system. It was quite challenging for the customer to keep both the vendors' and its own systems in sync. Furthermore, the customer had recently acquired another company, which meant integrating and managing an entirely new set of vendor systems.

Rather than continue to support a patchwork solution, the customer decided to invest in a long-term, scalable solution. The goal was to build a middleware integration platform that could support existing vendors from the merged entities, as well as be flexible enough to seamlessly integrate future requirements. The customer partnered with GlobalLogic to explore this new solution through a pilot project.

The initial use case was limited to supporting integration for a vendor subset and supporting the information systems that use FTP for message sharing. This use case was later extended to implement support for vendors with Asynchronous SOAP web services, and it was backed by a queue for asynchronicity.

What is Enterprise Application Integration?

For an organization like that of our customer, which has a multitude of applications for implementing operations and billing support solutions, its applications must talk to each other. But because these apps may have been developed over a period of time using a disparate set of technologies, integration may not be exactly straightforward. Also, the data structures used to represent a piece of information might be different from one application to the next, which would make it very difficult for them to communicate. Any communication processes would potentially need specific transformation rules.

Furthermore, it's possible that the transformation rules may not be managed in a global manner. Two applications may implement some of these rules to talk to each other without being aware of similar intelligence available elsewhere in the system. For this reason, such applications are sometimes referred to as "islands of automation" or "information silos."

This lack of communication leads to inefficiencies since identical data is stored in multiple locations. It may also lead to a situation in which straightforward processes cannot be automated. If integration is applied without following a structured approach, then point-to-point connections may grow across an organization. Dependencies would be added on an impromptu basis, which could result in a complex structure. This is where enterprise application integration (EAI) comes into play.

EAI is a set of architectural principles related to system integration. It lays out the best practices, technologies, and service solutions for common application and system integration challenges. EAI is the process of linking such applications within the context of an organization, which may include both inbound and outbound calls that transcend the boundary of the organization's information systems. EAI attempts to simplify and automate business processes to the greatest extent possible, while simultaneously trying to avoid making sweeping changes to the existing applications or data structures.

In his seminal work *Enterprise Integration Patterns*, Gregor Hohpe outlines the most commonly used integration patterns within four broad categories:

- File transfer (systems communicate by placing the messages they want to exchange under a directory accessible via FTP)
- Shared database(s)
- Remote procedure call (most commonly implemented by SOAP web services)
- Messaging (implemented in JMS; systems connect to a messaging channel and exchange data)

Most modern middleware solutions implement these patterns to integrate multiple heterogeneous systems in which each integration endpoint has its own communication protocol requirements. Some of the more popular solutions that are bundled together as an Enterprise Service Bus (ESB) for this purpose include Apache ServiceMix and Mule ESB.

Spring Integration Framework

Spring Integration is an API-centric framework that leverages Spring's core POJO-centric model for creating intuitive integration workflows. Integration workflows are defined via standard Spring configuration files. Unlike Apache ServiceMix or Mule ESB, it is not tied to a server and runs within the context of either a standalone or web-based Spring application. The API-centric nature of the Spring Integration Framework borrows heavily from the approach recommended in *Enterprise Integration Patterns*.

As a standard integration practice, Spring Integration is driven by a message-based approach to programming that provides greater scalability and flexibility. It implements messaging using the ubiquitous principles of inversion of control (IOC).

Traditionally, application developers have been responsible for invoking the appropriate methods on objects as relevant to specific message types. Message-driven applications in Spring invoke the methods of their objects using IOC. It is the framework's responsibility to operate the integration machinery of the application. Instead of calling methods on objects, we put messages on channels. The framework then routes the messages to the appropriate components.

Some of the out-of-box adapters and channels that the Spring Integration Framework provides include:

- Feed inbound channel adapter for feed format such as RSS or ATOM
- AMQP-backed message channels
- FTP/FTPS adapter
- Http inbound/outbound gateway
- TCP and UDP support
- JDBC support
- Message-driven channel adapter
- Mail sending/receiving channel adapter
- MongoDB inbound/outbound channel adapter
- Redis inbound/outbound channel adapter
- RMI support
- Twitter adapter
- Inbound/outbound web service gateways

An introduction to some of the core messaging components of the Spring Integration Framework is warranted at this point:

A message consists of headers and a payload, and it is directed along channels for processing. The payload of a message can be any type and can even be routed based on this type.

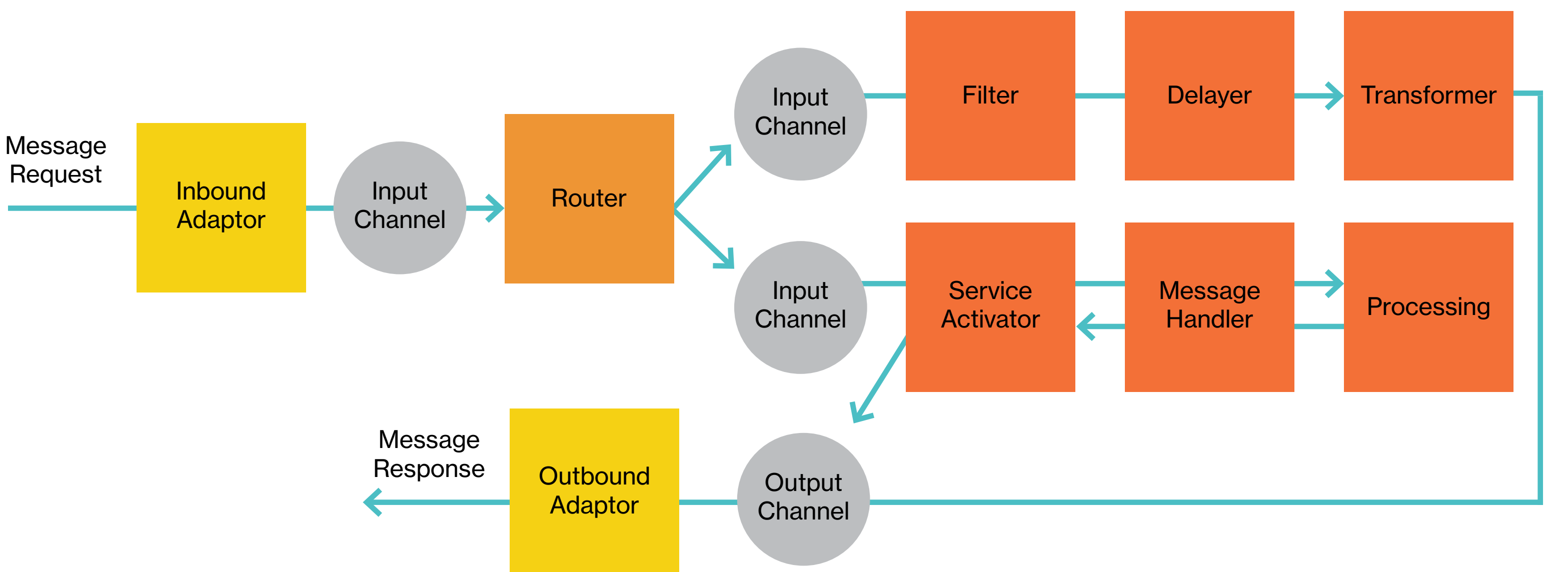

Figure 1: *An example of a Spring Integration workflow involving various components*

A channel is used for directing messages to the appropriate endpoints for processing. A channel may either be point-to-point or publish/subscribe and can support buffering and polling if required.

A message endpoint is a POJO that allows you to apply domain-specific logic on messages. This means you can translate inbound requests into service layer invocations, as well as service layer returns into outbound replies.

Transformers are used to convert the content or structure of a message, returning the modified message. Content transformers allow you to enhance messages with additional information (e.g., adding an additional header).

Filters determine whether messages should be passed on or not. The message is either passed on to the output channel, dropped, or an exception is thrown depending on the implementation.

Splitters allow multiple messages to be created by dividing the content of one message into parts, which is useful for composite payloads.

A service activator is a generic message endpoint that enables you to connect a target object method or service to the messaging system for performing domain-specific logic upon the receipt of a message.

Channel adapters may be either inbound or outbound and will typically perform any conversion required on the message to translate it to the format required by the external system.

Spring Integration for FTP

Going back to our use case, we found that one of our customer's vendors shares its order information in the format of an XML file. This file is generally sent using an FTP and is stored in a specific directory on the FTP server. We decided to leverage Spring Integration to create a more streamlined (and asynchronous) process, as demonstrated below:

- Solution reads the files in a batch from the FTP directory.
- Solution pushes the files into a queue that is read by a processor.
- Processor uses a vendor-specific transformer.
- Processor sends the processed file to the order management system.
- Processor sends out an email notification to the group responsible for managing the vendor order.
- A response is written back in a vendor-specific location on the FTP server.
- Another batch of files is read from the FTP server once the entire queue has been exhausted.

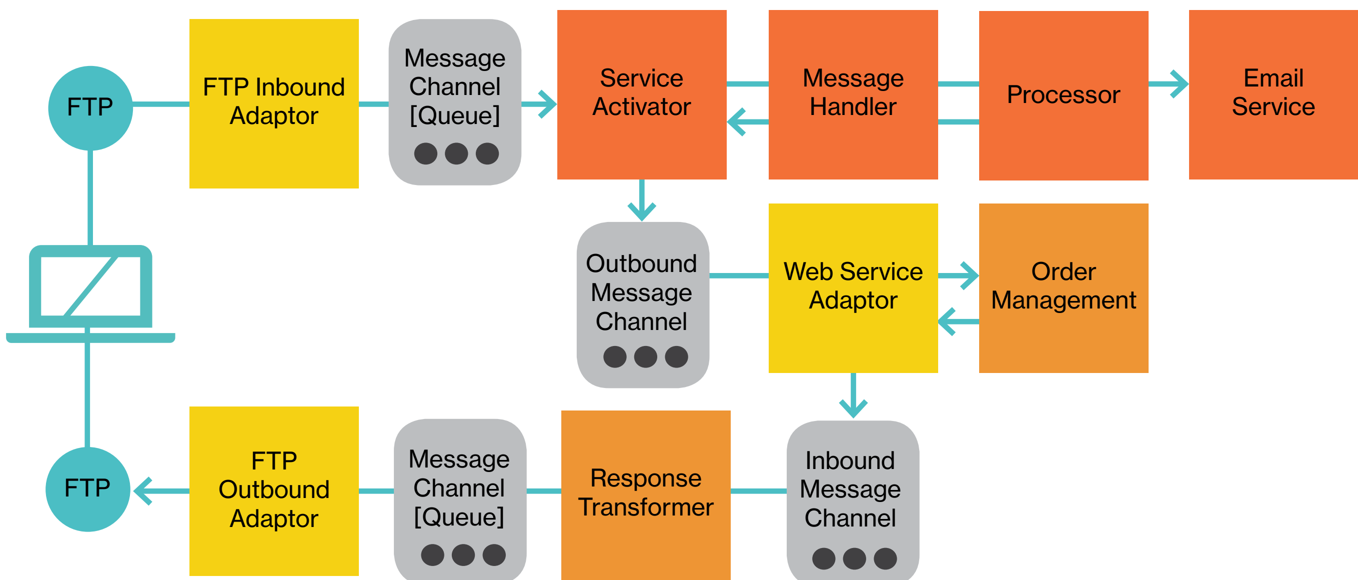

Figure 2: *Components of Spring Integration Framework for FTP integration*

The following code demonstrates how we configured the Spring Integration components under Spring Context to create the FTP solution.

```
<int:channel id="filesIn">
                    <int:queue capacity="40" />
</int:channel>

<int:channel id="baseFileChannel">
          <int:queue capacity="40" />
</int:channel>

<int:channel id="filesOut">
          <int:queue capacity="40" />
</int:channel>

<file:inbound-channel-adapter channel="filesIn"
directory="#{applicationProperties.orderBaseDirectory}" prevent-duplicates="true"
scanner="recursiveDirectoryScanner" filename-pattern="*.xml">
                    <int:poller fixed-rate="5000" />
</file:inbound-channel-adapter>

<task:executor id="baseFileMessageProcessingThreadPool"   pool-size="5"
queue-capacity="20"
keep-alive="120" />

<int:service-activator id="baseFileMessageProcessor" ref="baseFileMessageProcessingManager"
input-channel="filesIn"
output-channel="filesOut"
method="process" >
<int:poller task-executor="baseFileMessageProcessingThreadPool" fixed-rate="6000" />
</int:service-activator>

<bean id="baseFileMessageProcessingManager" class=" BaseFileProcessingManagerImpl" scope="thread"/>

<file:outbound-channel-adapter directory="#{applicationProperties.responseBaseDirectory}"
 channel="filesOut ">
                    <int:poller fixed-rate="5000" />
</file:outbound-channel-adapter>
```

Extending Spring Integration for SOAP Services

Once our team successfully demonstrated Spring Integration Framework's ease of configuration and flexibility, we decided to focus on extending our use case to support vendors that use SOAP web services for sharing order information.

The process of receiving order information in an XML payload, processing it, sending notifications out, and updating the order management system needed to be decoupled from sending a response back to the vendor. To address this issue, we created an inbound web service channel to host a service for receiving incoming orders. A separate outbound channel for sending a response back to the vendor-specific web service was wrapped in an inbound channel, which was integrated with the local order management system (as demonstrated in the below diagram).

If you compare this workflow to the one we created for the FTP solution, you can see that we reused many of the components (e.g., integration via an outbound channel adapter with the order management system and email service). Only the transformation logic specific to a particular vendor, both for requests and responses, changed in a seamless manner.

This workflow also demonstrates the component-oriented and loosely coupled nature of an integration solution that is built using the Spring Integration Framework, wherein the probability of reuse and support for plug-and-play is extremely high.

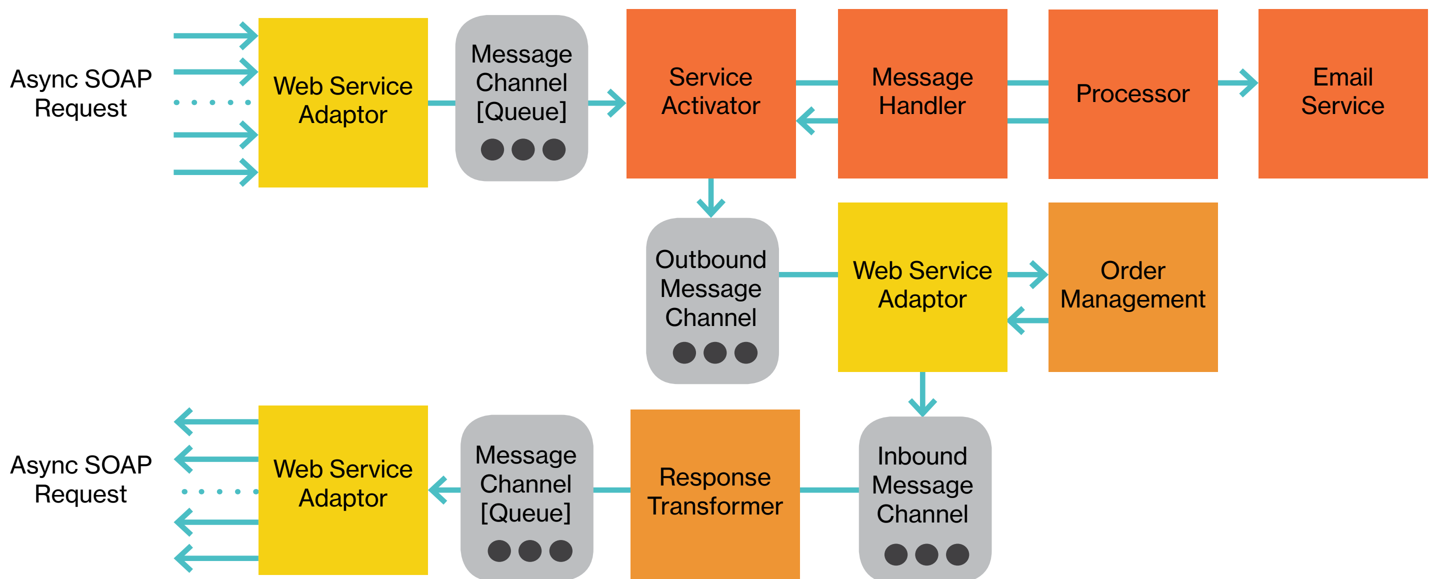

Figure 3: *Components of Spring Integration Framework for web service integration*

Below is a demonstration of how you can configure a SOAP web service endpoint that is backed by a pollable queue.

```
<int:channel id=”input”>
                    <int:queue capacity=”40” />
</int:channel>

<int:channel id=”output”>
          <int:queue capacity=”40” />
</int:channel>

<int-ws:inbound-gateway id=”ws-inbound-gateway” request-channel=”input”/>

<task:executor id=”baseMessageProcessingThreadPool”                 pool-size=”5”
queue-capacity=”20”
keep-alive=”120” />

<int:service-activator id=”baseMessageProcessor” ref=”baseMessageProcessingManager”
  input-channel=”input”
 method=”process” >
<int:poller task-executor=”baseMessageProcessingThreadPool” fixed-rate=”6000” />
</int:service-activator>

<bean id=”baseMessageProcessingManager” class=” BaseProcessingManagerImpl” scope=”thread”/>
```

Middleware Solution Using Spring Integration

A potential drawback of trying to create multiple integration endpoints managed under a Spring context is overly complex context configuration file(s). To avoid this issue, we chose to organize the context configuration file(s) in a modular manner that would seamlessly support the plug-and-play of multiple endpoints. Our intent was to protect existing endpoints from the impact of changes that resulted from adding new endpoints. At the same time, it was important to ensure that component reuse could be supported in an unobtrusive manner.

In our use case implementation, we began by creating a Spring context for an FTP endpoint. After successfully completing this integration, we attempted to integrate a SOAP endpoint. The guiding principles of plug-and-play leveraging OOPS directed our hand in refactoring the existing Spring context into:

- A parent context with configuration support for common channels, services, routers, and adapters
- A child context for an FTP endpoint that was declaratively included in the parent context
- A child context for a SOAP endpoint that was declaratively included in the parent context

Having separate context for each endpoint enabled us to ensure that issues like the unavailability of any individual endpoint did not bring the entire system down. We used the same strategy when we needed to integrate an ApacheMQ-based endpoint to support another vendor.

Conclusion

Based on our experience implementing this use case, we can claim with a reasonable degree of confidence that it is possible to orchestrate a scalable middleware solution by leveraging Spring's built-in support for various integration endpoints and modularizing their weaving together in a project.

About GlobalLogic Inc.

GlobalLogic is a full-lifecycle product development services leader that combines deep domain expertise and cross-industry experience to connect makers with markets worldwide.Using insight gained from working on innovative products and disruptive technologies, we collaborate with customers to show them how strategic research and development can become a tool for managing their future. We build partnerships with market-defining business and technology leaders who want to make amazing products, discover new revenue opportunities, and accelerate time to market.

For more information, visit www.globallogic.com

GlobalLogic®

Contact

Emily Younger
+1.512.394.7745
emily.younger@globallogic.com