Krzysztof Grodzicki

Spring Integration as building block for SOA

« index

This is a short review of the Spring Integration Framework(SI)[1]. To understand this article you should be familiar with Enterprise Application Integration(EAI) and have basic understanding of Enterprise Integration Patterns(EIP)[2]. For those new to EAI or EIP:

- *EAI* is an architectural approach for integrate data, services across the enterprise. EAI simplify and automate business process without requiring comprehensive changes in existing applications and data structures[3].
- *EIP* is set of integration patterns. EIP helps developers and architects describe and develop robust integration solutions[2].

For those new to SI, definition from project home page[1]:

> *"Spring Integration provides an extension of the Spring programming model to support the well-known Enterprise Integration Patterns. It 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. Spring Integration's primary goal is to provide a simple model for building enterprise integration solutions while maintaining the separation of concerns that is essential for producing maintainable, testable code."*

What is the place of SI in Service Oriented Architecture(SOA)? Tom McCuch explains this in one of his comment[4]:

> *"SOA is an architectural pattern. A pattern, by definition, is the encapsulation of a complex system into a reusable component. Patterns are meant to describe "building blocks" for YOUR solution - they are not meant to be solutions in of themselves. Patterns therefore lend themselves to be best implemented by lightweight, embeddable frameworks that serve to support your solution, not heavy, commercial-off-the shelf products that aim to control it. This is where the big vendors have all gone wrong with SOA - and where SpringSource, yet again, has gotten it right with Spring Integration."*

For more information refer to wikipedia[5] or SI home page[1]. You will find there documentation, reference manual, articles, podcasts and much more.

Nowadays there are many open source integration platforms on the market e.g., Mule ESB[6], Camel[7], ServiceMIX[8], OpenESB[9], JbossESB[10]. You can find good

comparison of the SI alternatives based on the three criteria like “Easy of Use”, “Maintainability” and “Extensibility” in *Pro Spring Inegration*[3]. SI has great potential and lot to offer. Lets review some advantages and disadvantages.

Advantages:

- **Implements proven patterns** - The SI team didn’t invent the wheel again. The whole architecture and solutions in SI are based on Enterprise Integration Patterns[2]. For example Service Activator pattern:

```
<beans:beans xmlns="...">

    <channel id="inputChannel" />

    <channel id="outputChannel">
        <queue capacity="10" />
    </channel>

    <service-activator input-channel="inputChannel" outpu

    <beans:bean id="sampleService" class="com.kgrodzicki.
</beans:beans>
```

This Gist brought to you by GitHub. ServiceActivator.xml view raw

- **Lightweight, maintainable, easy debugging and testable code** - SI is based on the Spring programming model which gives you possibility to:
  - develop code using POJO approach - There is rarely any need to couple your code with SI API.
  - unit testability and system testing support is provided by framework, you can do it without requiring deployment to your application server or connecting to other enterprise infrastructure.

  Sample test configuration:

```
<beans:beans xmlns="...">
        <!-- Import Flow configuration -->
        <beans:import resource="classpath:META-INF/Servic

        <!-- Create the bridge for output channel -->
        <bridge input-channel="outputChannel" output-char
                <poller max-messages-per-poll="10" fixed-
        </bridge>

        <channel id="testChannel">
                <queue />
        </channel>
</beans:beans>
```

This Gist brought to you by GitHub. ServiceActivatorTest-context.xml view raw

And test suite:

```
@RunWith(classOf[SpringJUnit4ClassRunner])
@ContextConfiguration
class ServiceActivatorTest {

  var inputChannel:MessageChannel = null
  var testChannel:QueueChannel  = null

  @Autowired
  @Qualifier("inputChannel")
  def setInputChannel(ic: MessageChannel) = inputChannel

  @Autowired
  @Qualifier("testChannel")
  def setTestChannel(tc: QueueChannel) = testChannel = tc

  @Test
  def testEntireFlow {
    // given
    val message = new GenericMessage[String]("TestMessage

    // when
    inputChannel.send(message)

    // then
    assertEquals("Activated: TestMessage", testChannel.re
  }
}
```

This Gist brought to you by ServiceActivatorTest.scala view raw GitHub.

- **Documentation** - SI has good documentation with practical examples that will help you understand channels, adapters, routers etc. You can find also very good books about SI e.g., Pro Spring Integration[3] from Apress. It covers contemporary technologies and real-world examples, facing the real problems. Another important thing is community which is really helpful and eager to solve your problems fast.

- **Low Coupling - channel as a first class citizen** - One of the biggest SOA principles is loose-coupling. With SI you can build flexible, configurable architecture. By Oleg Zhurakousky:

    "In SI we have Channel as a first class citizen. In all other frameworks it is mentioned only as an internal implementation not exposed to the end-user.

    - *Channels is what enforces the decoupling between producers and consumers.*

    - *Channels is what's allowing us to manage message-exchange protocols. From throttling to parallelism all is accomplished via*

channels.

- *Channels gives end-user transparency with regard to how these message-exchange protocols are applied. Change the type of a channel without any producer/consumer awareness."*

- **Model complex solutions** - using SI you are able to model complex solutions using various technologies like EJB, RMI, AMQP etc. in one place[11].

- **Adapters for legacy system** - possible adapters for legacy systems like an old mainframe application where input/output are batch files deposited on some FTP endpoint. Writing your own adapters in most cases are easy and straightforward. For more details check out "Pro Spring Integration" chapter 15[3].

- **Easy integration with other spring projects** like Spring Batch, Spring AMQP(RabbitMQ). SI has the ability to launch Spring Batch jobs via messaging, allowing for event-driven batch process. In addition, SI can be used to scale out Spring Batch using partitioning. This provides the ability to partition big batch jobs over many nodes using message channels as the coordination fabric[3].

- **GUI designer in SpringSource Tool Suite** - Simple designer will help you with fast modelling of flows and let you visualize your configuration file adhoc. Mark Fisher about GUI designer:

  > *"One of the best things about it is that it was designed from day one to support "round trip" editing. In other words, changes in the graph immediately show up in XML and vice-versa. They are just 2 tabs on the same view - sitting on top of the exact same metadata model. Unlike the big commercial products, we know it's very important that a GUI editor does NOT hide the details from a developer's perspective. XML, if kept as concise as possible (something we have worked very hard at doing), is actually pretty good for configuration (although IMHO it's NOT good at all for coding imperative logic like when/otherwise or try/catch!), and we made sure that the editor does not interfere in any way with the developer's ability to read/understand/modify that XML."*

  For quick introduction refer to Refcards: Eclipse Tools for Spring: The SpringSource Tool Suite

- **Spring Integration Scala DSL** - domain-specific language for Scala[12]. Example:

```scala
/**
 * @author Oleg Zhurakousky
 *
 */
// snippet http://bit.ly/pPU2ee
val integrationContext = IntegrationContext(
      inputChannel >=> (
        // subscriber 1
        {
                transform.withName("xfmrA").using { "'Fro
                middleChannel >=>
                transform.withName("xfmrB").using { m: Me
                resultChannel
        },
        // subscriber 2
        {
          service.using { m: Message[String] => println('
        })
    )

    inputChannel.send(new GenericMessage("==> Hello from
    val outputMessage = resultChannel.receive
    println("Output Message: " + outputMessage)
  }
```


Disadvantages:

- **New technology** - tree years on the market might not be enough for the bigger companies.
- **Missing adapters** like ssh, shttp. Lack of the many adapters comparing to e.g., MuleESB. Always you can fill JIRA for SI if you see some functionality which is crucial for you and can be used by community.

Is the SI framework for you? If you are familiar with Spring idioms and looking for more natural way to build event-driven applications SI is lightweight, powerful technology build on the proven platform which you can use to model ESB which fits your needs.

Gists are from https://github.com/kgrodzicki/spring-integration-samples project.

Any comments? Have more advantages or disadvatages? Let me know on Twitter @kgrodzicki.

---

1. http://www.springsource.org/spring-integration

   ↩

2. http://www.eaipatterns.com/

   ↩

3. Pro Spring Integration Josh Long , Dr. Mark Lui , Mario Gray , Andy Chan

3. Pro Spring Integration Josh Long , Dr. Mark Fisher , Mario Gray , Andy Chan

↩

4. http://www.infoq.com/articles/Spring-Integration-Joshua-Long#view_38115

↩

5. http://en.wikipedia.org/wiki/Enterprise_application_integration

↩

6. http://www.mulesoft.org/

↩

7. http://camel.apache.org/

↩

8. http://servicemix.apache.org/

↩

9. http://openesb-dev.org/

↩

10. http://www.jboss.org/jbossesb/

↩

11. http://www.digitalsanctum.com/2010/08/31/using-rabbitmq,-spring-amqp-and-spring-integration/

↩

12. https://github.com/SpringSource/spring-integration-scala

↩