Spring in the ESB: A Foundation for SOA

Aaron Mulder
CTO
Chariot Solutions

CHARIOT SOLUTIONS

Speaker

- Aaron Mulder
- Mule contributor
- ServiceMix committer
- CTO of Chariot Solutions
  - Java/Open Source consulting firm
  - Have used Mule and ServiceMix with Spring on consulting engagements

Agenda

- Part 1: Overview
  - Why Spring?
  - Why ESB?
- Part 2: Using Spring components in:
  - ServiceMix
  - Mule
- Part 3: More on Spring in the ESB

Overview

Why Spring?

- If you don't know by now...

...you should ask for your money back

Background on ESB

- A bus architecture with messages, components listening for messages, and routing of messages between components
- Separation of component logic from network transports in/out of the bus
- Optional orchestration for advanced workflow, business process, etc.
- Consistent platform for integration

ESB Concepts

- A component handles incoming and/or outgoing messages
  - Product-specific components
  - Spring components
- An endpoint handles connectivity to the bus
  - JMS
  - HTTP (REST or SOAP, etc.)
- Components on the bus may handle logic, routing, legacy connectivity, etc.

ESB for Integration

- Before ESB:
  - Spaghetti integration “architecture”
  - Many systems, many data sources, many applications, many technologies, many paths
- After ESB:
  - Single point of contact for integration
  - Standardized messages, routing, etc.
  - Fewer pathways, more consistent, easier to manage and scale, etc.

Open Source ESB Options

- Mule (mule.codehaus.org)
  - Mature, but still actively developed
  - POJO-based messages
  - Can directly load an external Spring config file
- ServiceMix (servicemix.org)
  - Coming up on 1 year old
  - JBI-based messages (JSR-208)
  - More robust container model
  - Spring via XBean and JSR-181

Java Business Integration (JBI)

- A JCP standard for integration & ESBs
- Not yet fully supported by all vendors, though several on board
- Supports containers, components, XML-based messages, endpoints, etc.
- Includes deployment model with hot deploy
- Should make it easier to share components across ESB products...

Spring Runtime Options

Spring can be run…

- Standalone (with a simple main method)
- In a minimal runtime (like XBean)
- In a Java EE application / app server
- In an ESB runtime

But which is best?

Spring Standalone

- You have full control in the Spring config(s)
- A lot to configure for an enterprise application (database, transactions, management, messaging, ...)
- You're responsible for any runtime configuration, management, monitoring...
- No deployment model
- Easy to test

Spring in XBean

- Adds modules/plugins, component lifecycle, management, etc.
- XBean configuration files accept Spring syntax for instant integration
- Easy integration with XBean-based products like ActiveMQ, ServiceMix, ...
- Still no configuration GUI, limited canned components... a lightweight framework

Spring in Java EE Applications

- No more manual configuration of database, transactions, messaging, other resources
- Java EE focused more on business logic and presentation
- RPC-heavy networking
- Painful integration, web services – code tied to network transports
- Hard to be real testable

Spring in the ESB

- ESB focused on routing and integration
  - Standardizing messages and communication
  - Code separate from transports
- Orchestration options
- Deployment and management
- No particular persistence, resource handling, or presentation options
- Test in or out of the ESB

Spring, App Server, & ESB?

- Spring has the simple, configurable POJOs
- An application server can handle resource configuration, management, deployment,and presentation
- The ESB can handle orchestration, integration, standardized messaging
- Powerful combination, if a bit heavy
- No integrated application package... yet

Spring in Mule

Introduction to Mule

- Driven by an XML configuration file
  - Includes network connectors, message transformers, management agents
  - Includes a model defining service components with endpoints and routing
- The Mule config file can import a separate Spring config file
  - Then the Mule config can directly interact with Spring beans

The Mule Config File

```
<mule-configuration id="prototype" version="1.0">
    <container-context ...>
    <security-manager ...>
    <agents ...>
    <connector ...>
    <transformers ...>
    <model name="PrototypeModel">
        <mule-descriptor name="MyService"
                         implementation="test.MyClass">
            <inbound-router ...>
            <outbound-router ...>
        </mule-descriptor>
    </model>
</mule-configuration>
```

Routing by POJO type

```
<mule-configuration id="prototype" version="1.0">
  <model name="PrototypeModel">
    <mule-descriptor name="MyService"
                     implementation="test.MyClass">
      <outbound-router>
        <router
    className="org.mule.routing.outbound.FilteringOutboundRouter">
          <endpoint address="vm://ServiceOne"/>
          <filter expectedType="com.example.SrvOneRequest"
   className="org.mule.routing.filters.PayloadTypeFilter"/>
        </router>
        <router
     className="org.mule.routing.outbound.FilteringOutboundRouter">
          <endpoint address="vm://SNMP"/>
          <filter expectedType="com.example.MuleSNMPMessage"
 className="org.mule.routing.filters.PayloadTypeFilter"/>
        </router>
      </outbound-router>
```

Spring in Mule

```
<mule-configuration id="prototype" version="1.0">
    <container-context name=”Spring”
className="org.mule.extras.spring.SpringContainerContext">
        <properties>
            <property name="configFile"
                  value="spring-config.xml"/>
        </properties>
    </container-context>
    <security-manager ...>
    <agents ...>
    <connector ...>
    <transformers ...>
    <model name="PrototypeModel">
        <mule-descriptor name=”MyService”
                         implementation=”MySpringBean”>
...
```

Advantages & Disadvantages

- Currently can only reference one Spring config file
- No runtime deployment – components all fixed in Mule config file
- Messaging and routing based on POJOs
  - If return type is Foo, looks for another service with a method with a single argument of type Foo to call as the next in the chain
- Extensive Acegi security support

More Spring & Mule

- Can use a combined syntax to configure beans/properties in a Spring-like way directly in the Mule config file
  - But, some of the names change to distinguish between Spring “property” and Mule “property”
- Can actually configure Mule itself inside a Spring config file, using beans for all the Mule components
  - More verbose, but more flexible Spring options

Is Spring Needed?

- Mule services are POJO-like by default
- But Mule configuration is not as robust as Spring configuration
- Integrating Spring helps for complex components with interdependencies
- Can also take advantage of loads of canned Spring components
  - Database pools & persistence libraries, AOP/interceptors, security, etc.

Spring in ServiceMix

Introduction to ServiceMix

- A JBI server based on XBean
- May be configured statically (monolithic config file) or dynamically (static core plus hot deployed services, etc.)
- Main config file and module config files use XBean/Spring syntax (very similar to Spring 2 custom schemas)
- Some constraints due to JBI

A bit of JBI

- JBI defines:
  - JBI Components (really, service containers)
  - Service Units (a group of services, deployed to JBI Component / service container)
  - Service Assemblies (deployable group of service units)
  - Normalized Messages (a message with headers, an XML body, and attachments)

JBI Advantages / Disadvantages

- A component that wants to send or receive messages must deal with JBI artifacts (normalized messages, etc.)
- Solid component model, with lifecycle, deployment, etc.
- Still, the JBI components are not real POJO-like, and the spec doesn't deal much with application-level issues

Spring in ServiceMix

- Bottom line, you need a JBI Component (a service container) that you can deploy Spring or POJO components to
- ServiceMix has multiple Spring options:
  - Can use Spring syntax inside most ServiceMix config files, e.g. the main static config file
  - Can use the Lightweight Container & “POJOs”
  - Can use the JSR-181 Container to connect Spring POJOs to the bus

The Lightweight Container

- Supports many service types (files, JMS, e-mail, rules, scheduler, ...)
- Supports “POJOs”, so long as they implement various JBI interfaces and use JBI methods to send and receive NormalizedMessages
- Spring syntax used to configure each “POJO”

Lightest-Weight Component

```
public class PojoSender {
    private ServiceMixClient client;

    public void sendMessage() throws MessagingException {
        InOnly exchange = client.createInOnlyExchange();
        NormalizedMessage message = exchange.getInMessage();
        message.setProperty("id", new Integer(i));
        message.setContent(new StringSource("some XML"));
        client.send(exchange);
    }

    public void setClient(ServiceMixClient client) {
        this.client = client;
    }
}
```

Lightweight Config File

```
<beans xmlns:sm="http://org.apache.servicemix/config/1.0">
  <sm:serviceunit id="jbi">
    <sm:activationSpecs>
      <sm:activationSpec componentName="receiver"
                         service="foo:receiver">
        <sm:component>
          <bean class="com.example.Receiver"/>
        </sm:component>
      </sm:activationSpec>
      ...
    </sm:activationSpecs>
  </sm:serviceunit>

  <bean id="sender" class="com.example.PojoSender">
    <property name="client" ref="..." />
  </bean>
</beans>
```

The JSR-181 Container

- Exposes any POJO to the JBI bus using JSR-181 to generate WSDL for the POJO
- NormalizedMessages whose XML complies with the WSDL will be decoded to calls against the POJO (with argument objects, return objects, etc.)
- POJOs can be pure POJOs, defined in a Spring config file

JSR-181 Component

```
public class SpringPojo {
    public SomeObject doSomething(String foo,
                                  OtherObject bar) {
        ...
        return ...;
    }
}
```

JSR-181 Config File

```
<beans
    xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0">

    <jsr181:endpoint pojo="#myPojo" />

    <bean id="myPojo" class="com.example.SpringPojo">
        ...
    </bean>
</beans>
```

Spring in ServiceMix Conclusions

- The lightweight container is best if you're using some of the featured types other than POJOs (file monitoring, scheduling, etc.)
  - Or if you want to do detailed orchestration with JBI interactions
- The JSR-181 container is best for a pure Spring approach
- Either one can be statically configured or hot deployed

Is Spring Needed?

- Some kind of component model must be used...
- JBI-style components are not real clean, not so easy to test
- Perhaps best to build Spring components and then either expose via JSR-181 or adapters to fit into the Lightweight Container

More on Spring in the ESB

Spring in ESB vs. Spring alone

- In Spring standalone
  - You must configure in/out messaging (JMS, Web Services, network bindings, etc.)
  - You must decode messages to calls on Spring object, and encode responses to messages
  - You must define data / error formats
  - You must run Web/JMS server
- In the ESB
  - All this is configurable and taken care of for you

But do I care?

- Most of that isn't a big deal for a traditional web application
- It's best for:
  - Messaging-oriented systems
  - An integration platform connecting many disparate systems
  - A service-oriented architecture connecting many services with routing, orchestration, etc.

Spring / ESB / Java EE

- The ESB doesn't provide a typical web application container
- Not as elaborate a component model as EJBs
- The ESB can be run either inside or in conjunction with a Java EE application server to get the best of both worlds
  - Open source ESB/app server integration is still evolving (exposing EJBs as endpoints, etc.)

Advanced Services

- What if a service needs to be initialized, managed, or orchestrated?
- Some of this can be done in Spring
- All of this can be done by a component in the ESB
- Normally there are product-specific interfaces a component can implement
  - Initialisable (Mule), InitializingBean (Spring), ComponentLifeCycle (JBI)

Advanced Services, cont.

- Nicest to keep logic in Spring components where possible
  - May need product-specific adapter layer
  - Easier to test, easier to make portable
- Not necessarily feasible for orchestration
  - For detailed interaction with messaging and routing, will need to be product-specific
  - JBI has the “standards-based” advantage
  - Still can use Spring persistence
  - Or just integrate a BPEL engine

Build Environment

- Message-oriented or service-oriented builds tend to be challenging
- Tools like Maven make it easier to use binaries of every service except the one you're working on
- Spring, Mule, and ServiceMix can all be run with JUnit to run tests as part of a build
  - But often relies on static configuration instead of runtime deployment

Testing

- Unit testing handled by JUnit
  - With or without the ESB
  - Maven can coordinate running tests
- Continuous Integration can be handled by Continuum
  - Some minor tweaking needed to run multi-module builds on a test server
- Load testing can be done with JMeter
  - Capture and re-emit messages

Security

- ESB Security typically applied at endpoints (normally for incoming messages)
- May also be able to apply security on component invocation
- Often, a message originates from another component (original user identity lost)
- Can use JAAS for authentication and perhaps Acegi
  - At Spring component or endpoint level

Deployment

- Normally Spring is only started/stopped once
  - ESBs with static configuration work like this
- Can run the ESB standalone
- Can deploy Spring and the ESB as part of a web application for app-level hot/redeploy
- ServiceMix can be integrated with app servers to leverage the server's deployment tools for JBI hot/redeploy

Deployment, cont.

- Static config in application module best for apps with async/messaging requirements
  - Or to get redeploy for Mule
- ServiceMix app server integration with hot deploy best when ESB services may use application logic and vice versa
  - Or to use app server config, management, etc.
- Standalone deployment best for ESB as standalone middleware accessed by many applications

Spring-Only Deployment

- Both Mule and ServiceMix can be configured and run solely via a Spring bean context
  - Mule/ServiceMix implementation objects declared and configured as Spring beans
- Makes for a pretty unpleasant Spring config file...
- But the ESB can be run and managed anywhere that Spring can be run

Versioning

- If Spring components change, will endpoints (WSDL/Schema/etc.) change?
  - May be best to change e.g. listen URL for each version
- Can use request transformers to accept "old" messages – upconvert and dispatch
- Or can run multiple versions of the services simultaneously
  - Easier in ServiceMix (per-deploy class loader)

Summary

An ESB enables SOA-based Spring apps

- Can take advantage of the strengths of:
  - **Spring**: simple code, tests, dep. injection
  - **ESB**: simple messaging, integration, transports
  - **Java EE**: presentation, session bean logic
- Can leverage features in either Spring or an app server for management, persistence, configuration
- Mule and ServiceMix have very solid Spring integration options
- Still some architecture decisions, like anything else

Q&A

Aaron Mulder: ammulder@chariotsolutions.com

Download slides at: http://www.chariotsolutions.com/

CHARIOT SOLUTIONS