Welcome to SpringOne

SpringONE

INTERFACE21
Spring from the source

The Role of Spring in an ESB

Mark Fisher
Interface21
markf@interface21.com

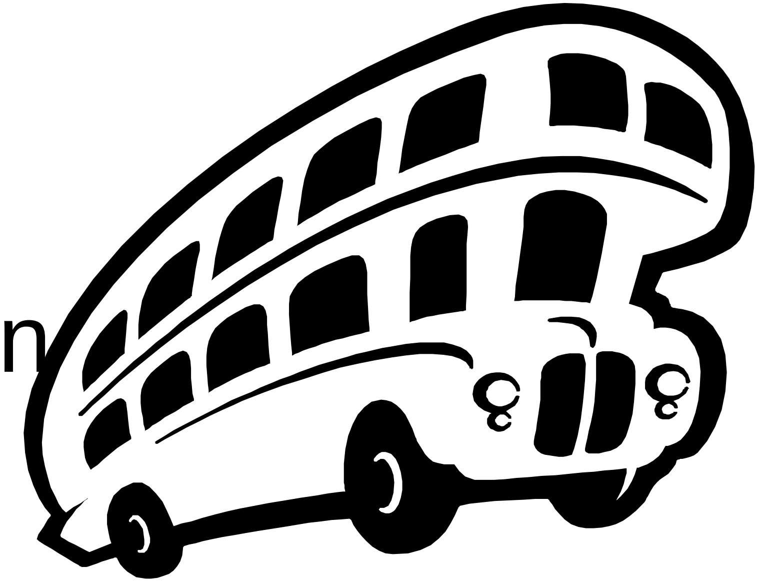
WWW.SPRINGONE.COM
JUNE 15-16, 2006 METROPOLIS ANTWERP-BELGIUM

BeJUG

Enterprise Service Bus (ESB)

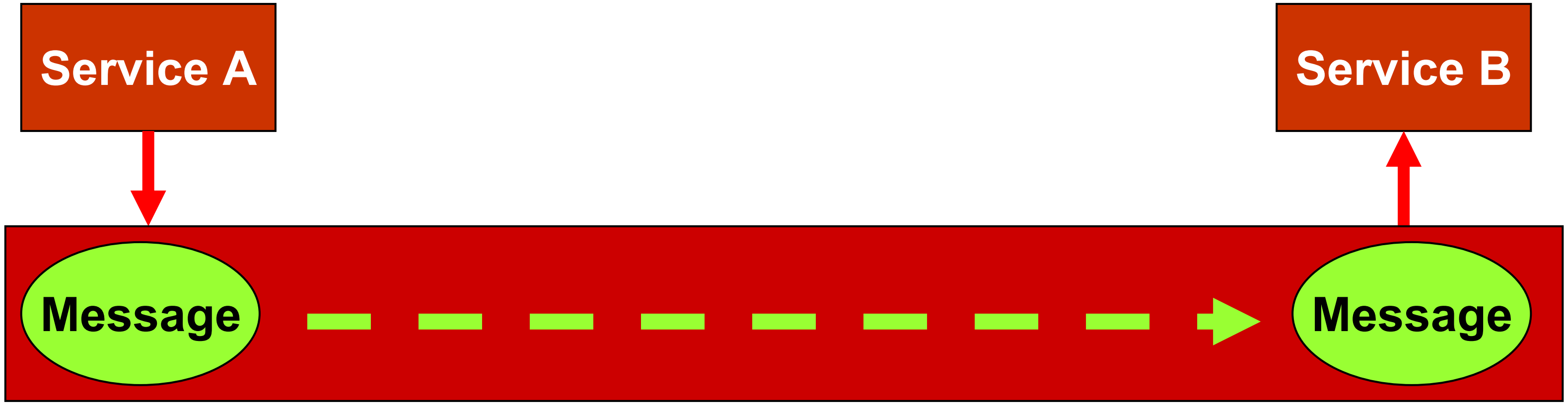
Key Topics:

- Messaging
- Transformation
- Routing



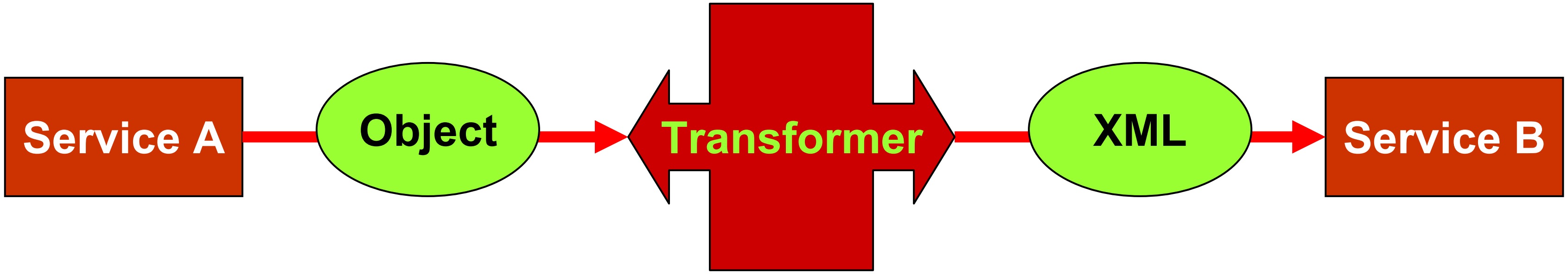
Introduction: *Messaging*

- low-impact extensibility
- promotes encapsulation (vis a vis RPC)
- normalized / canonical format
- scalable (avoids n(n-1)/2 endpoints)



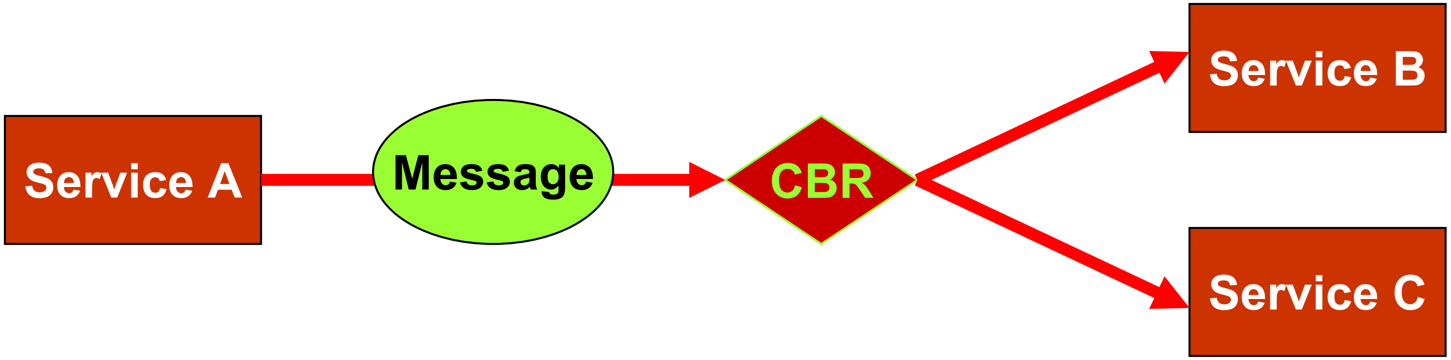
Introduction: *Transformation*

- data format change
- data enhancement / modification
- message normalization
- if XML-based, may use XSLT



Introduction: *Routing*

- content-based (CBR) by payload type
- content-based by property’s value
- rule-based (JSR-94)
- if XML-based, may use XPath



An ESB has…MANY BUZZWORDS

According to Wikipedia:

“In computing, an enterprise service bus refers to a software architecture construct, implemented by technologies found in a category of middleware infrastructure products usually based on Web services standards, that provides foundational services for more complex service-oriented architectures via an event-driven and XML-based[1] messaging engine (the bus). An enterprise service bus generally provides an abstraction layer on top of an Enterprise Messaging System which allows integration architects to exploit the value of messaging without writing code. Contrary to the more classical EAI approach of a monolithic stack in a hub and spoke architecture, the foundation of an enterprise service bus is built of base functions broken up into their constituent parts, with distributed deployment where needed, working in harmony as necessary.”

Messages = XML ?

Wikipedia recap:

"…usually based on Web services standards, that provides foundational services for more complex service-oriented architectures via an event

1. ↑ **An alternative view, particularly for high performance enterprise service buses, is that "standard" message formats should flow across the bus, not just XML. Generating XML and parsing it can be costly in terms of processing and memory, and high volume scenarios may not be viable.**

Related Specifications and Models

Briefly Surveying The Landscape…

- Service Oriented Architecture (SOA)
- Event Driven Architecture (EDA)
- JSR-208: Java Business Integration (JBI)
- Service Component Architecture (SCA)

Service Oriented Architecture

Relationship between ESB and SOA:

- ESB brings *Message Exchange* to SOA.
- SOA emphasizes *loosely coupled* services exposing functionality via well-defined, *standards-based* interfaces.
- ESB provides a *backbone* for integrating those services while maintaining the goal of loose coupling.

Event Driven Architecture

- EDA enables *asynchronous* calls or triggers within an SOA promoting increasingly dynamic responsiveness.
- Client/Server dichotomy breaks down in favor of *Producers* and *Consumers.*
- Staged Event Driven Architecture (SEDA): Services are connected by *queues*, enabling better performance monitoring and load balancing.

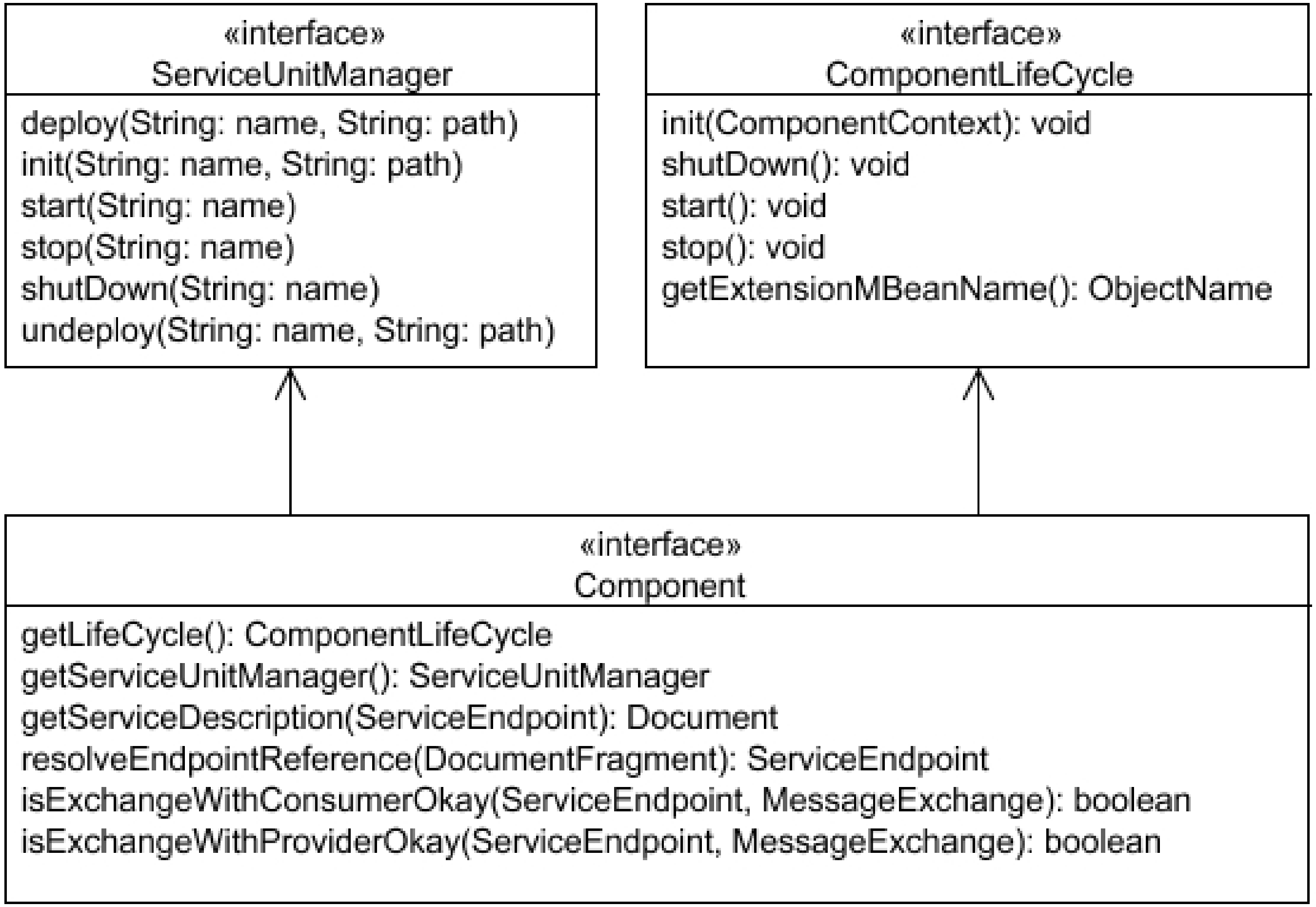
Java Business Integration (1)

JBI Component: packaging and deployment unit based on *abstract business process* metadata that describes the component's role within a service composition (interaction with other processes).

Core Interfaces:

- `javax.jbi.component.Component`
- `javax.jbi.component.ComponentLifeCycle`
- `javax.jbi.component.ServiceUnitManager`

Java Business Integration (2)



«interface»
ServiceUnitManager

deploy(String: name, String: path)
init(String: name, String: path)
start(String: name)
stop(String: name)
shutDown(String: name)
undeploy(String: name, String: path)

«interface»
ComponentLifeCycle

init(ComponentContext): void
shutDown(): void
start(): void
stop(): void
getExtensionMBeanName(): ObjectName

«interface»
Component

getLifeCycle(): ComponentLifeCycle
getServiceUnitManager(): ServiceUnitManager
getServiceDescription(ServiceEndpoint): Document
resolveEndpointReference(DocumentFragment): ServiceEndpoint
isExchangeWithConsumerOkay(ServiceEndpoint, MessageExchange): boolean
isExchangeWithProviderOkay(ServiceEndpoint, MessageExchange): boolean

Java Business Integration (3)

The JBI Environment manages its components: lifecycle, deployment, and monitoring

Two types of Components:

- *Service Engines*
  - business logic and transformation
- *Binding Components*
  - connectivity to external services

Services are defined by ***WSDL***.

Normalized Message Router

- decouples producers from consumers
- decouples message format from protocol

Java Business Integration (4)

Development Cycle:

1. Develop the service and package as a ServiceUnit.
2. Optionally compose services with an orchestration engine (a SequencingEngine is built into the SDK) and package WSDL + orchestration process as a ServiceUnit.
3. Configure BindingComponents for inbound and outbound endpoints and package each as a ServiceUnit.
4. Package the ServiceUnits for all ServiceEngines and BindingComponents into a ServiceAssembly.
5. Deploy the ServiceAssembly

Java Business Integration (5)

QuickStart Guide:

http://java.sun.com/integration/1.0/docs/sdk/quick_start/index.html

The OpenESB Project:

- **https://open-esb.dev.java.net/**
- Built upon the JBI specification
- Starter Kit available with Java EE 5 SDK

An Application Server extension that includes:

- Runtime JBI Framework
- BPEL service choreographer
- HTTP SOAP BindingComponent
- Java EE Service Engine (connect to EJBs)
- Ant-based administration

SCA (1)

Service Component Architecture

Contributors:

- IBM
- BEA Systems
- IONA
- Oracle
- SAP AG
- Siebel Systems
- Sybase

SCA (2)

- Components have Service-Oriented interfaces
- Consuming components access producing components' *service references.*
- Metadata provided via Annotations
- 2 steps:
  1. **Implement the components**
  2. **Assemble the components by wiring service references into their consumers**

SCA (3)

Infrastructure is decoupled from Services:

- Declarative Bindings

  (including Web Services and Messaging)
- Declarative Security and Transactions

Service Data Objects:

representation of business data for parameters and return values of service invocations.

SCA (4)

- SCA API is currently at version 0.9
- Tuscany SCA Runtime implementation
  - **http://incubator.apache.org/tuscany/**
- Eclipse SOA Tools Platform (STP)
  - **http://www.eclipse.org/stp/**
- SCA support in Lingo (POJO remoting/messaging):
  - **http://lingo.codehaus.org/**

Landscape Survey: Summary

- Many interesting developments in the realm of SOA, EDA, and ESB.
- However, the environment seems relatively instable with competing forces and evolving “standards”.
- Some of the newer products seem promising but are not yet proven.
- When possible stick with POJOs

but

An ESB has… SPRING SYNERGY

allows integration architects to exploit the value of messaging without writing code

- “In computing, an **enterprise service bus** refers to a software architecture construct, implemented by technologies found in a category of middleware infrastructure products usually based on Web services standards, that provides foundational services for more complex service-oriented architectures via an event-driven and XML-based[1] messaging engine (the bus). An **enterprise service bus** generally provides an abstraction layer on top of an Enterprise Messaging System which allows integration architects to exploit the value of messaging without writing code. Contrary to the more classical EAI approach of a monolithic stack in a hub and spoke architecture, the foundation of an **enterprise service bus** is built of base functions broken up into their constituent parts, with distributed deployment where needed, working in harmony as necessary.”

provides an abstraction layer

Spring/ESB philosophy parallels (1)

- Loose Coupling
  - A layer of abstraction
  - Avoid vendor or technology lock-in
  - Accommodate changing requirements

- Separation of Concerns
  - Decouple orthogonal logic
  - Business logic vs. Routing/Transport logic
  - Similar to MVC and presentation logic

Spring/ESB philosophy parallels (2)

- Reusable Components
  - Across physical environments
  - Across development lifecycle
  - Across transport protocols

- Stateless Services
  - Simplifies distribution
  - Coarse-grained interfaces
  - Represent Use-Cases

Spring/ESB philosophy parallels (3)

- Deployment Flexibility
  - Scale up OR down as needed
  - Start simple, but keep options open
  - Similar to Spring with local vs. global TX
  - Scale down for development or debugging

As with any Spring-based application, a well-designed ESB should promote *consistency* and *simplicity*.

Spring/ESB philosophy parallels (4)

ESB and Inversion of Control

IoC and The Hollywood Principle:

“Don’t call us, we’ll call you”

Enterprise Service Bus adoption motto:*

“If you can’t bring the application to the bus,

bring the bus to the application.”

*David Chappell’s *Enterprise Service Bus*

Introducing Mule (1)

Universal Message Objects™ (UMO)

- Mule manages UMO Components.
- Any object can be managed as a component, and any object can be passed as a message payload.
- Mule is non-invasive: components can *optionally* participate in lifecycle events.

Introducing Mule (2)

- Messages are transported between ***Endpoints***, but details of the transport protocol are hidden by an abstraction layer.
- ***Transformers*** can modify message content between Endpoints.
- ***Routers*** can control a message’s path.
- Components, Routers, and Transformers are configured *declaratively* – and can take advantage of ***Inversion of Control*** with a Spring container (more soon).

Introducing Mule (3)

Transport Providers:

- Email (SMTP, IMAP)
- File (file:, FTP, SSL)
- JMS
- Quartz
- Stream (System.in)
- VM (local method calls)
- Web Services (Axis, Glue, XFire)
- XMPP (Jabber)
- ***many more provided… also extensible***

Mule and JMS

Endpoint URIs:

```
jms://someQueue
Jms://topic:someTopic
```

If the jndiDestinations property is set to true, the names will resolve by JNDI.

The JMS ConnectionFactory can be configured directly in the Mule configuration or within Spring:

```
<connector name="jmsConnector" className="org.mule.providers.jms.JmsConnector">
   <properties>
      <container-property name="connectionFactory"
            reference="jmsConnectionFactory"
            container="spring" />
```

refers to a Spring bean

Mule and WebServices: Axis

```
<transformers>
 <transformer name="httpToSoap"
   className="org.mule.providers.soap.transformers.HttpRequestToSoapRequest"/>
</transformers>

<mule-descriptor name= "axisJumbler"
                implementation="com.springone.esb.webservice.Jumbler">
  <inbound-router>
    <endpoint address= "axis:http://localhost:8081/ws"
              transformers="httpToSoap"/>
  </inbound-router>
</mule-descriptor>
```

Client Code:

```
MuleClient client = new MuleClient();
UMOMessage result = client.send(
        "axis:http://localhost:8081/ws/axisJumbler?method=jumble", "testing", null);
String s = (String) result.getPayload();
```

Mule and WebServices: XFire

```
<transformers>
  <transformer name="httpToSoap"
      className="org.mule.providers.soap.transformers.HttpRequestToSoapRequest"/>
</transformers>

<mule-descriptor name= "xfireJumbler"
                                implementation="com.springone.esb.webservice.Jumbler">
    <inbound-router>
        <endpoint address="xfire:http://localhost:8081/ws"
                            transformers="httpToSoap"/>
    </inbound-router>
</mule-descriptor>
```

Client Code:

```
MuleClient client = new MuleClient();
UMOMessage result = client.send(
      "xfire:http://localhost:8081/ws/xfireJumbler?method=jumble", "testing", null);
String s = (String) result.getPayload();
```

Mule and Spring: Integration Options

1. Use Spring as a bean factory to manage the Mule components (including transformers, connectors, and implementations for MuleDescriptors).
2. Use a Spring ApplicationContext as the publisher for Mule events.
3. Configure the MuleServer itself in Spring.
4. Configure with Mule’s XML and add Spring beans as needed.
5. Create a custom Spring 2.0 XSD for a much more concise syntax for option #3 above.

Code Example:

Mule and Spring

Introducing ServiceMix (1)

- An open-source JBI-based ESB
- Hosted in Apache incuabator
- Fully-integrated with Geronimo

  (also integrates with other app servers, as well as Tomcat)
- Motivation: to provide an ***Agile ESB***
  - standards-based but flexible
    - deployment environment portability
    - multiple protocol support
  - enable deployment of POJO components

Introducing ServiceMix (2)

Transformation and Routing

- BPEL
- XSLT
- XPath
- Rules Engines (JSR-94)
- Scripting (JSR-223)

Introducing ServiceMix (3)

Binding

- JMS
- HTTP
- Files
- Email
- Quartz
- XMPP
- WebServices (SOAP, SAAJ, XFire)
- ***and more…***

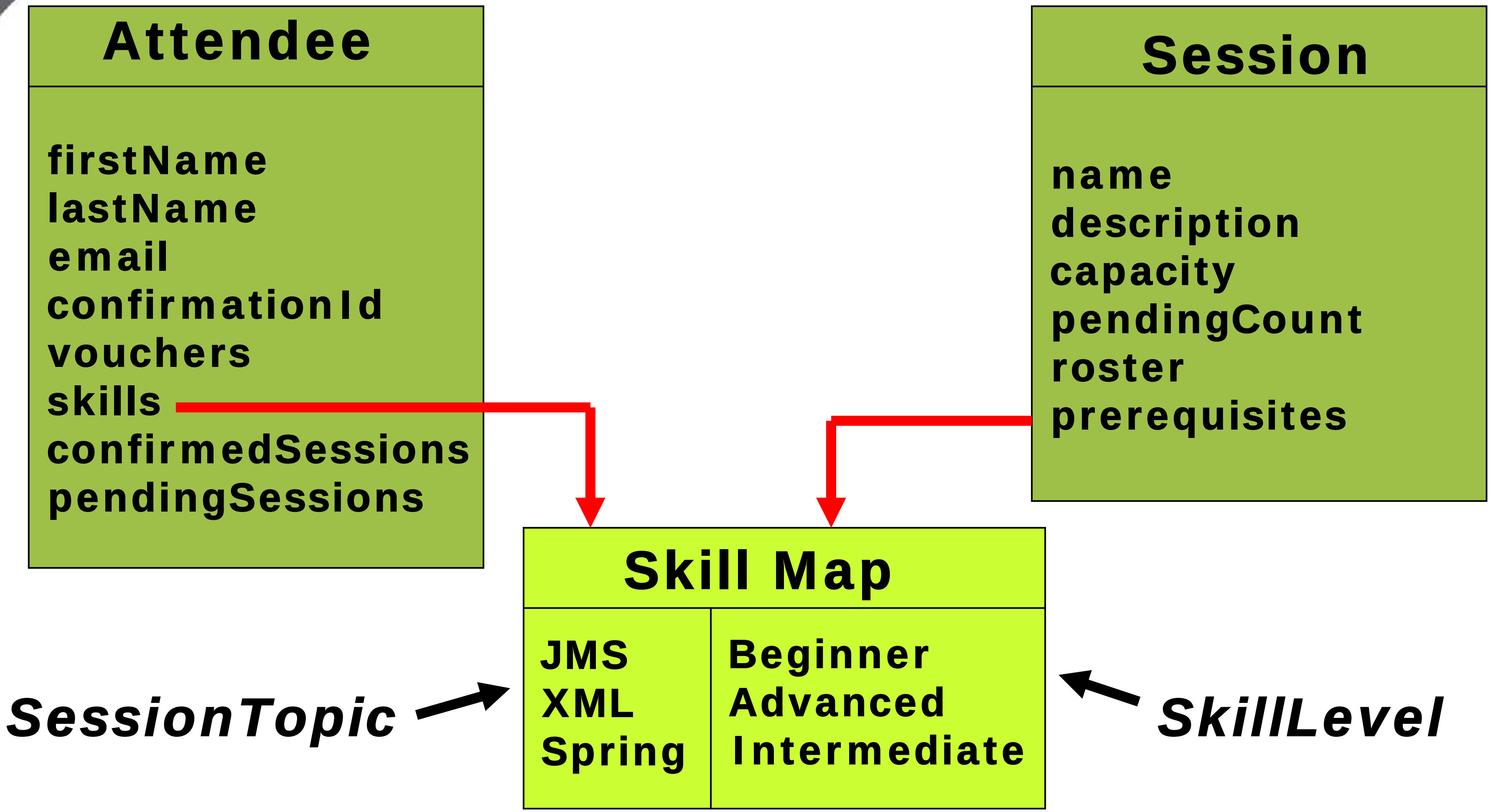
ServiceMix and Spring

- Spring simplifies the deployment units: simply add a bean to the Spring configuration - as opposed to separate XML for the component.
- ServiceMix provides an implementation of Spring’s *ApplicationContext* that enables the developer to take full advantage of Spring’s IoC while also providing concise, JBI-specific syntax.

Code Example:

ServiceMix and Spring

Case Study: The ESB Conference (1)



Attendee

firstName
lastName
email
confirmationId
vouchers
skills
confirmedSessions
pendingSessions

Session

name
description
capacity
pendingCount
roster
prerequisites

Skill Map

JMS XML Spring	Beginner Advanced Intermediate

SessionTopic

SkillLevel

Case Study: The ESB Conference (2)

The Use Cases: ***Oh, and by the way…***

1. add a new Session to the Conference ***…in a Directory***
2. an Attendee registers for the Conference
3. notify Attendee of Conference confirmation ***… via*** ID ***email and I.M.***
4. Attendee registers for a Session
   - If a voucher is available, register Attendee with Session and add to Attendees' confirmed list.
   - If no voucher, add to Attendee's pending list and increment pending count for the Session.
5. clear pending lists at a regular interval ***… and it needs to be modifiable without changing any code***

Case Study: Messaging (1)

```
public interface MessageBus {

  public void sendAndWait (String endpoint,
                           ConferenceMessage message,
                           Map properties);
 public void sendAndNoWait (String endpoint,
                            ConferenceMessage message,
                            Map properties);
public ConferenceMessage sendAndReceive (
                            String endpoint,
                            ConferenceMessage message,
                            Map properties);
}
```

Case Study: Messaging (2)

```
public class ConferenceMessage {

    private Attendee attendee;
    private Session session;

  // getters and setters

}
```

Case Study: Messaging (3)

Implementing the MessageBus in Mule

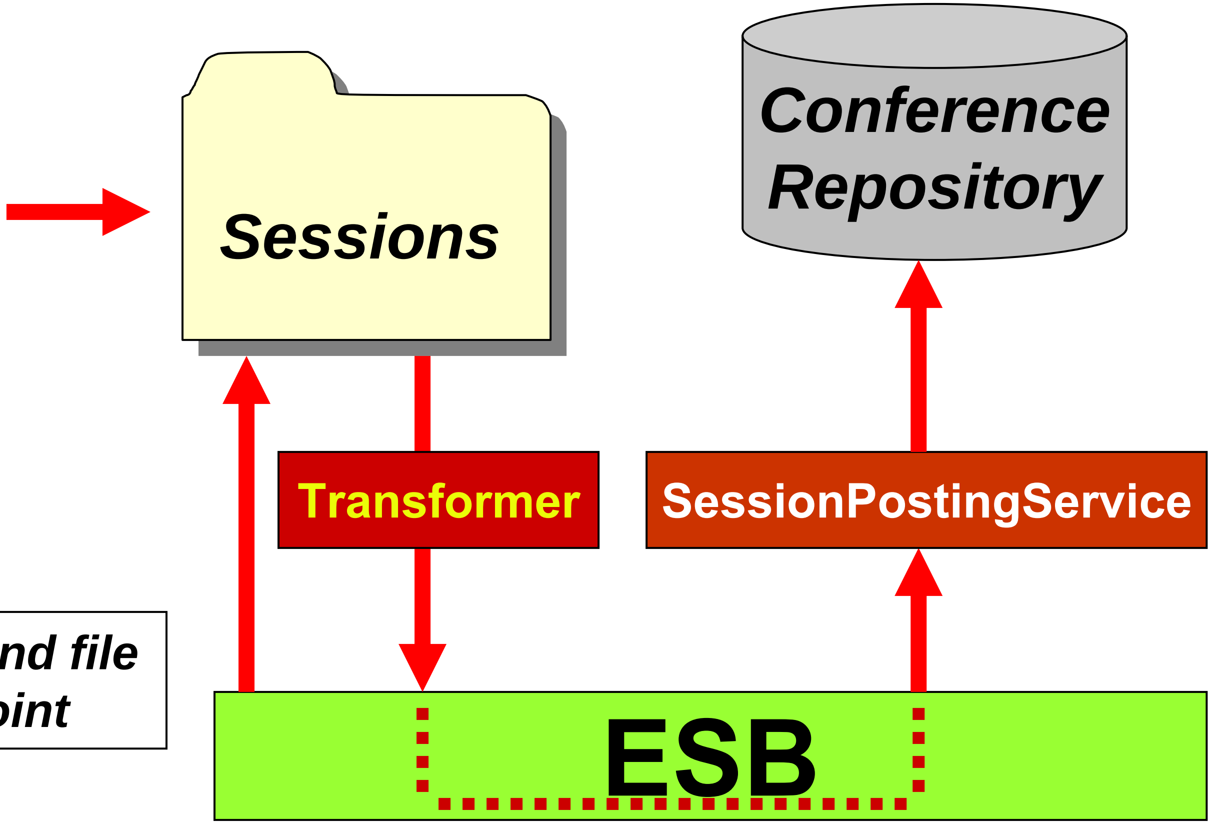
```
MuleClient client = new MuleClient();

// for sendAndWait:
client.send(endpoint, msg, props);

// for sendAndNoWait:
client.sendAsync(endpoint, msg, props);

// for sendAndReceive:
UMOMessage result = client.send(endpoint, msg, props);
return (ConferenceMessage) result.getPayload();
```

Case Study: Use Case #1 (1)

Add a new Session to the Conference

Session Information in a properties file:

name=JMS Intro
description=…
capacity=50
topic.JMS=Beginner



inbound file endpoint

Case Study: Use Case #1 (2)

The code:

```
public class SessionPostingService {

      private ConferenceRepository conferenceRepository;

      public void setConferenceRepository(ConferenceRepository conferenceRepository) {
            this.conferenceRepository = conferenceRepository;
      }

      public void postSession(Session session) {
            conferenceRepository.addSession(session);
      }

}
```

Case Study: Use Case #1 (3)

The configuration (service):

```
<bean id="sessionPostingService“
      class="org.mule.impl.MuleDescriptor">
 <property name="implementation">
   <bean class=“..service.SessionPostingService">
      <property name="conferenceRepository“
                ref="conferenceRepository"/>
    </bean>
  </property>
...
</bean>
```

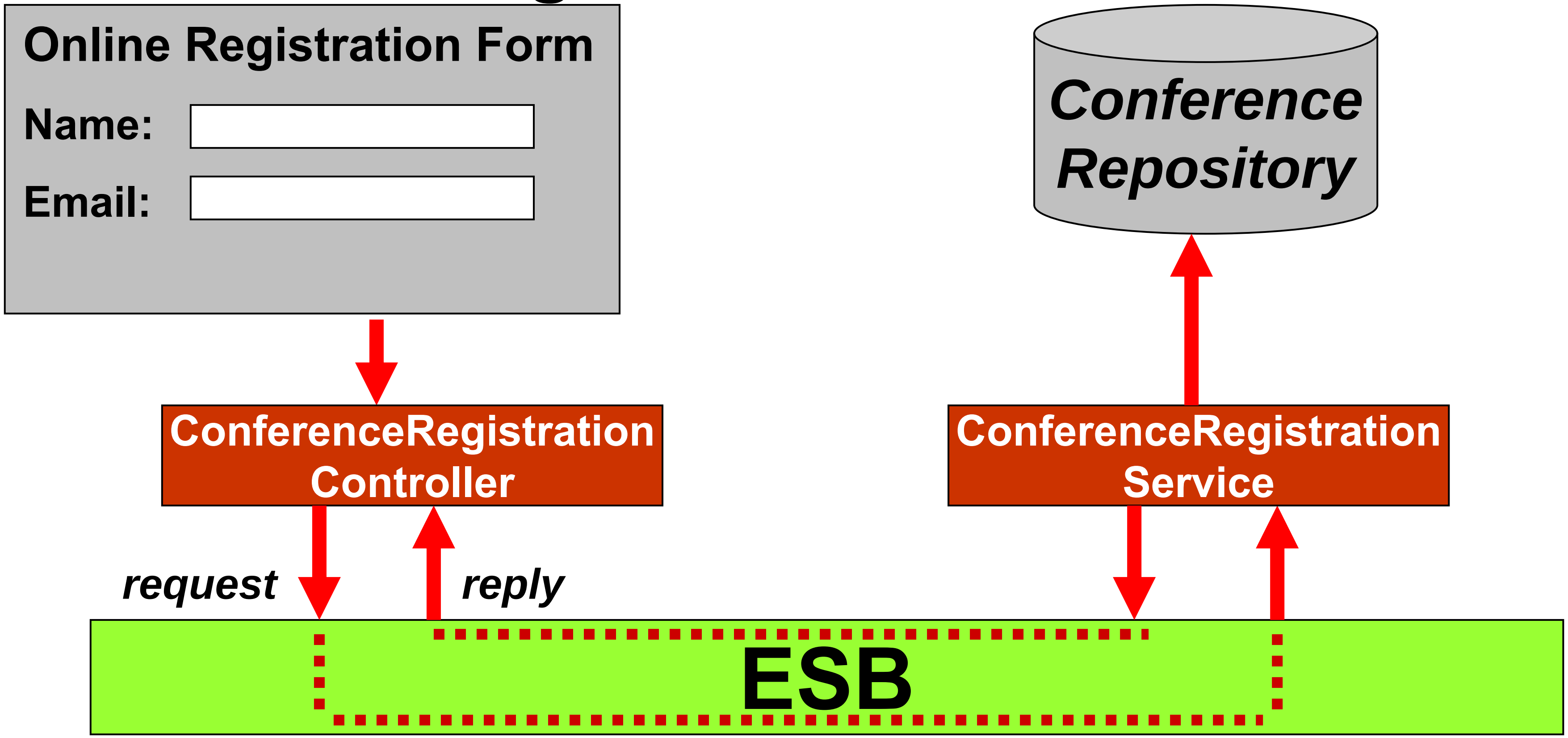
Case Study: Use Case #1 (4)

The configuration (endpoint):

```
<bean id="sessionPostingService"
      class="org.mule.impl.MuleDescriptor">
...
<property name="inboundEndpoint">
  <bean  class="org.mule.impl.endpoint.MuleEndpoint">
    <property name="endpointURI">
      <bean class="org.mule.impl.endpoint.MuleEndpointURI">
        <constructor-arg>
          <value>file://path/to/sessions?transformers=fileToString,
                                        stringToProperties,propertiesToSession
          </value>
        </constructor-arg>
      </bean>
    </property>
  </bean>
</property>
</bean>
```

Case Study: Use Case #2 (1)

Attendee registers for the Conference



Case Study: Use Case #2 (2)

The code:

```
protected void doSubmitAction(Object command) throws Exception {
   Attendee attendee = (Attendee) command;
   ConferenceMessage message = new ConferenceMessage();
   message.setAttendee(attendee);
   ConferenceMessage result =
     messageBus.sendAndReceive(registrationEndpoint, message, null);
   attendee = result.getAttendee();
    …
}
```

```
        <property name="registrationEndpoint"
                              value="vm://conferenceRegistration"/>
```

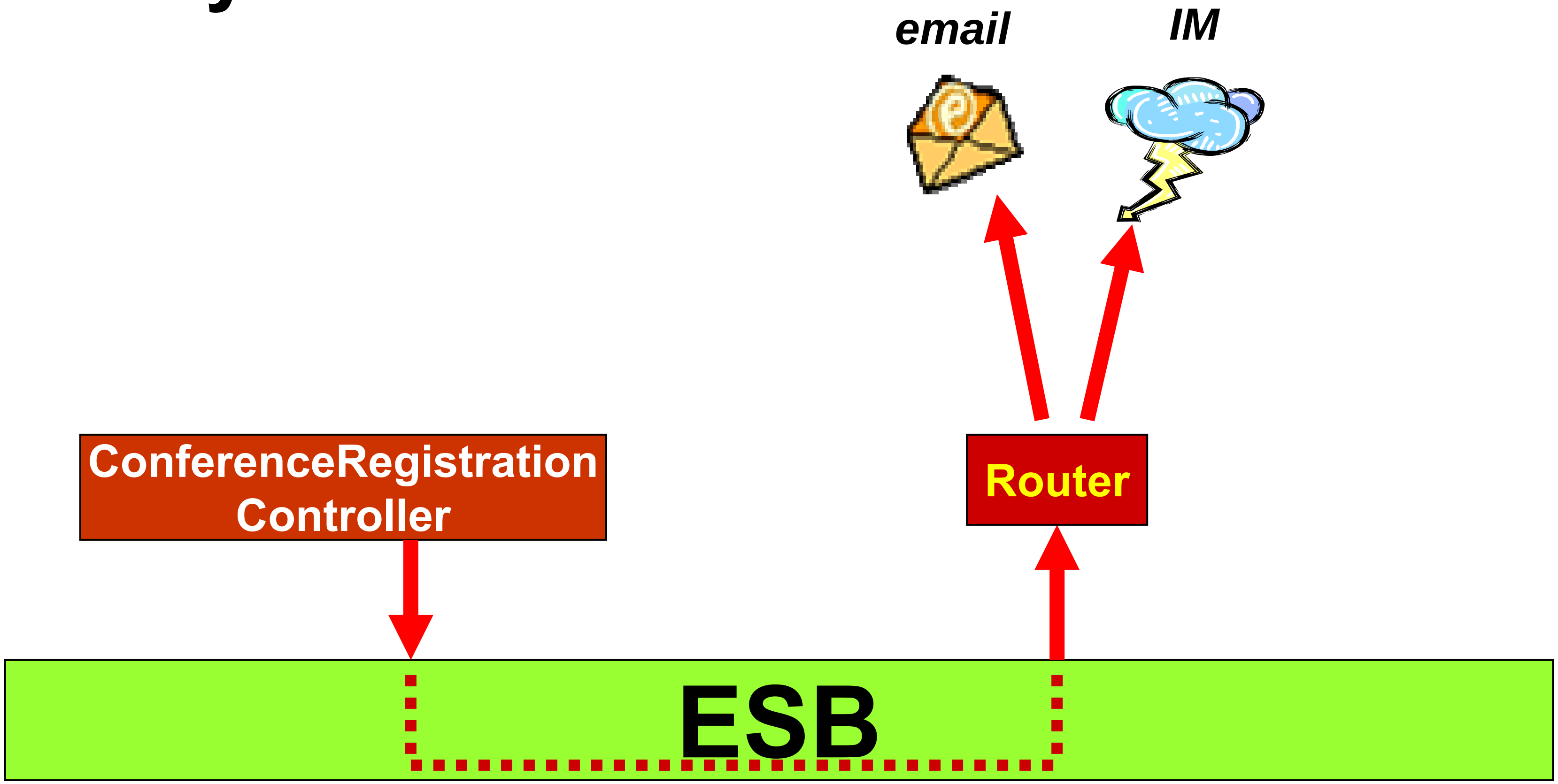
Case Study: Use Case #2 (3)

The configuration:

```
<bean id="conferenceRegistrationDescriptor"
            class="org.mule.impl.MuleDescriptor">
 <property name="implementation"
                    ref="conferenceRegistrationService"/>
 <property name="inboundEndpoint">
  <bean  class="org.mule.impl.endpoint.MuleEndpoint">
    <property name="endpointURI">
      <bean class="org.mule.impl.endpoint.MuleEndpointURI">
        <constructor-arg value="vm://conferenceRegistration"/>
      </bean>
    </property>
  </bean>
 </property>
</bean>
```

Case Study: Use Case #3 (1)

Notify Attendee of the Confirmation ID



Case Study: Use Case #3 (2)

The code:

```
protected void doSubmitAction(Object command) throws Exception {
…
  Properties props = new Properties();
  props.setProperty("toAddress", attendee.getEmail());
  props.setProperty("fromAddress", "admin@esbconference.com");
  props.setProperty("subject", "Conference Registration Status");
  messageBus.sendAndNoWait(notificationEndpoint, message, props);
}
```

```
    <property name="notificationEndpoint"
                  value="vm://conferenceNotification"/>
```

Case Study: Use Case #3 (3)

The configuration:

```
 <bean id="conferenceRegistrationController"
        class=“..web.controllers.ConferenceRegistrationController">
   <property name="commandClass"
                    value="com.springone.conference.domain.Attendee"/>
    <property name="commandName"
                    value="attendee"/>
    <property name="formView"
                    value="conferenceRegistration"/>
    <property name="successView"
                    value="conferenceRegistrationSuccess"/>
    <property name="messageBus"
                    ref="messageBus"/>
    <property name="registrationEndpoint"
                    value="vm://conferenceRegistration"/>
    <property name="notificationEndpoint"
                    value="vm://conferenceNotification"/>
</bean>
```

Case Study: Use Case #3 (4)

The configuration (inbound):

```
<bean id="conferenceNotificationDescriptor"
      class="org.mule.impl.MuleDescriptor">
 <property name="implementationInstance">
  <bean class="org.mule.components.simple.PassThroughComponent"/>
 </property>
 <property name="inboundEndpoint">
  <bean  class="org.mule.impl.endpoint.MuleEndpoint">
   <property name="endpointURI">
    <bean class="org.mule.impl.endpoint.MuleEndpointURI">
     <constructor-arg value="vm://conferenceNotification"/>
    </bean>
   </property>
  </bean>
 </property>
…
</bean>
```

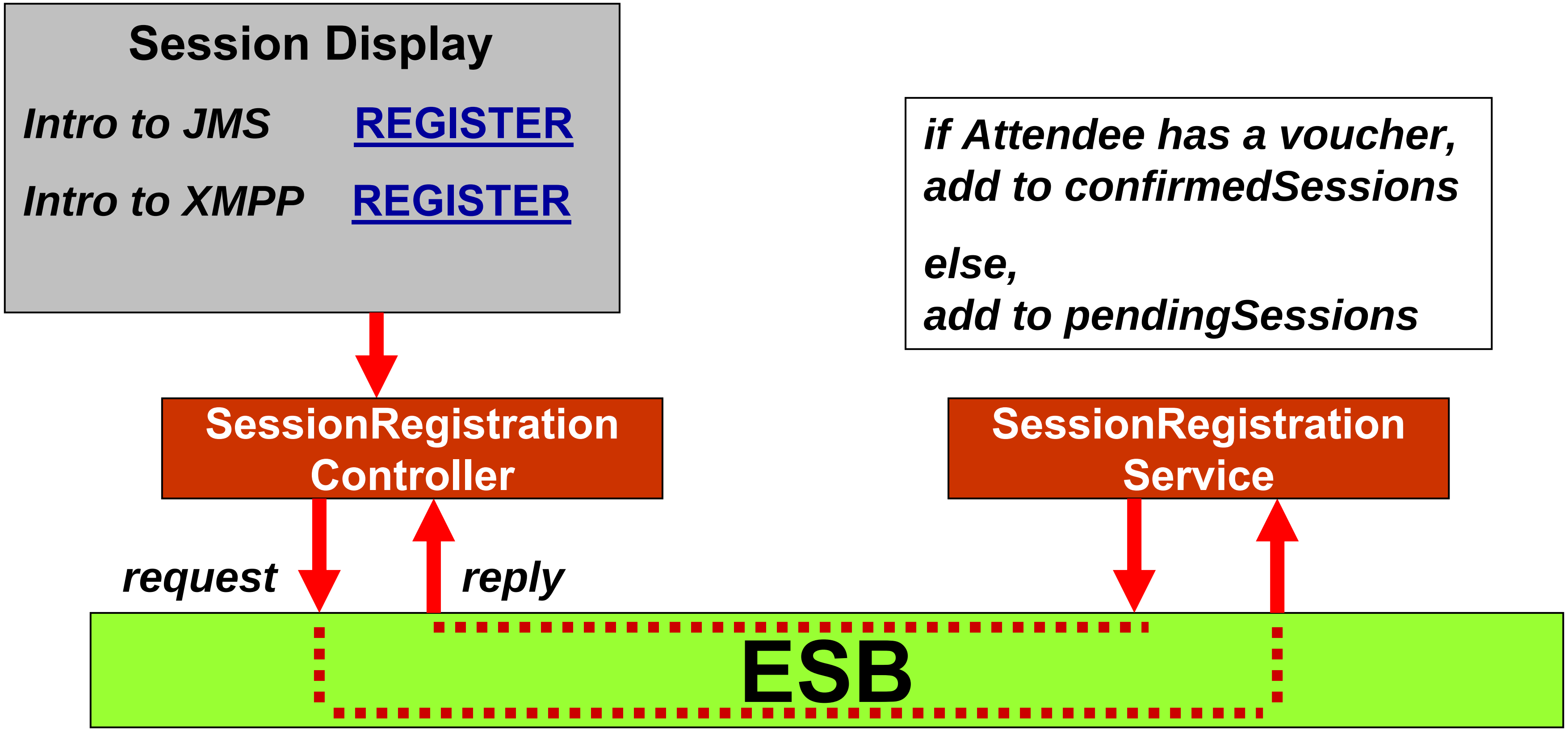
Case Study: Use Case #3 (5)

The configuration (outbound):

```
<property name="outboundRouter">
 <bean class="org.mule.routing.outbound.OutboundMessageRouter">
  <property name="routers">
   <bean class="org.mule.routing.outbound.StaticRecipientList">
    <property name="recipients">
     <list>
      <value>smtp://usr:pwd@host?transformers=confMsgToString</value>
      <value>xmpp://usr:pwd@host/?transformers=confMsgToString</value>
     </list>
    </property>
   </bean>
  </property>
 </bean>
</property>
```

Case Study: Use Case #4 (1)

Attendee registers for a Session



Session Display

Intro to JMS REGISTER

Intro to XMPP REGISTER

if Attendee has a voucher,
add to confirmedSessions

else,
add to pendingSessions

Case Study: Use Case #4 (2)

```
protected ModelAndView handleRequestInternal(
        HttpServletRequest request, HttpServletResponse response)
throws Exception {
   String sessionName =
      ServletRequestUtils.getRequiredStringParameter(request,
                                                    "session");
   Session session =
      conferenceInfoService.findSessionByName(sessionName);
   Attendee attendee = (Attendee)
      request.getSession().getAttribute("attendee");

   ConferenceMessage message = new ConferenceMessage();
   message.setSession(session);
   message.setAttendee(attendee);
   messageBus.sendAndWait(messageEndpoint, message, null);

   return new ModelAndView(viewName);
}
```

Case Study: Use Case #4 (3)

The configuration:

```
<bean id="sessionRegistrationDescriptor"
          class="org.mule.impl.MuleDescriptor">
  <property name="implementation“
                   ref="sessionRegistrationService"/>
  <property name="inboundEndpoint">
    <bean  class="org.mule.impl.endpoint.MuleEndpoint">
      <property name="endpointURI">
       <bean class="org.mule.impl.endpoint.MuleEndpointURI">
         <constructor-arg value="vm://sessionRegistration"/>
       </bean>
      </property>
    </bean>
  </property>
</bean>
```

Case Study: Use Case #4 (4)

The SessionRegistrationService:

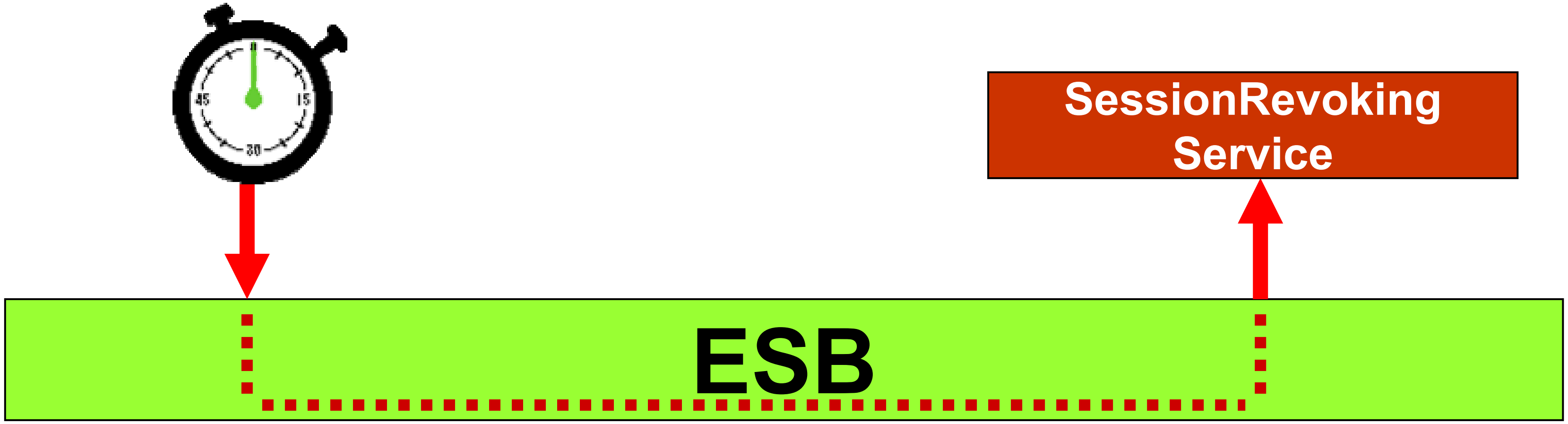
```
// if session.hasAttendee(attendee))
// if (session.atCapacity())
// if (!session.satisfiesPrerequisites(attendee.getSkills()))
// throw new RegistrationException…

if (attendee.getVouchers() > 0) {
    session.addAttendee(confirmationId, attendee);
    attendee.addConfirmedSession(session);
    attendee.decrementVouchers();
}
else {
    session.incrementPendingCount();
    attendee.addPendingSession(session);
}
```

Case Study: Use Case #5 (1)

Clear Pending Session Lists

Configuration specifies either an interval or a cron expression for triggering an inbound endpoint.

Clear pendingSessions from Attendees and decrement pendingCount on Sessions accordingly.



SessionRevoking Service

ESB

Case Study: Use Case #5 (2)

The code:

```
public class SessionRevokingService {
    private ConferenceRepository conferenceRepository;
    public void setConferenceRepository(ConferenceRepository repo) {
        this.conferenceRepository = repo;
    }
    public void revokeSessions(Object source) {
        List<Attendee> attendees = conferenceRepository.getAttendees();
        for (Attendee attendee : attendees) {
             attendee.setPendingSessions(new ArrayList<Session>());
        }
        List<Session> sessions = conferenceRepository.getSessions();
        for (Session session : sessions) {
            session.setPendingCount(0);
        }
    }
}
```

Case Study: Use Case #5 (3)

The configuration (service):

```
<bean id="sessionRevokingService"
          class="org.mule.impl.MuleDescriptor">

   <property name="implementation">
      <bean class="..SessionRevokingService">
         <property name="conferenceRepository"
                           ref="conferenceRepository"/>
      </bean>
   </property>

   . . .

</bean>
```

Case Study: Use Case #5 (4)

The configuration (endpoint):

```
<property name="inboundEndpoint">
  <bean class="org.mule.impl.endpoint.MuleEndpoint">
    <property name="properties">
      <map>
        <entry key="cronExpression" value="0 0/5 * * * ?"/>
        <entry key="payload" value="none"/>
      </map>
    </property>
    <property name="endpointURI">
      <bean class="org.mule.impl.endpoint.MuleEndpointURI">
        <constructor-arg value="quartz:/sessionRevokingService"/>
      </bean>
    </property>
  </bean>
</property>
```

Live Demo:

The ESB Conference

Summary

- An ESB bring *Messaging*, *Transformation*, and *Routing* to a Service-Oriented Architecture.
- *Endpoints* abstract the underlying protocols – allowing flexibility and avoiding *n(n-1)/2.*
- ESB specifications are emerging (JBI and SCA).
- Active open source ESB frameworks exist - such as ServiceMix and Mule.
- Many of the ESB motivations are close parallels to those of Spring: loose coupling, declarative configuration, portability, and consistency.
- Spring can play a significant role in achieving the goals of a well-designed ESB solution.

Further Reading

- Enterprise Service Bus,

  David Chappell (O’Reilly 2004)
- Enterprise Integration Patterns,

  Gregor Hohpe and Bobby Woolf (Addison Wesley 2004)
- The JBI Specification:

  http://www.jcp.org/en/jsr/detail?id=208
- The SCA Homepage:

  http://www-128.ibm.com/developerworks/library/specification/ws-sca/
- Mule: http://mule.codehaus.org
- ServiceMix: http://www.servicemix.org/