Contemporary Application Integration: ESBs and SCA

Dr Gerald Loeffler, MBA

JavaDeus 08
Sun Microsystems Austria
Java Developer Conference

this presentation

- agenda
  - recap/introduce **Mule 2** as a representative and popular example of an **ESB**
  - introduce **SCA**, the Service Component Architecture
    - focus on Java-related implementation models
  - briefly **discuss SCA** in the context of Mule 2 and Java EE
- this is a developer conference, so we will
  - take an **example-first approach**
    - developed against Mule 2.0.1 and Tuscany 1.2.1
    - focus on the exemplary and leave most generalisations to you
  - omit SOA governance issues and non-development concerns
  - skip the philosophical and non-technical aspects of SOA

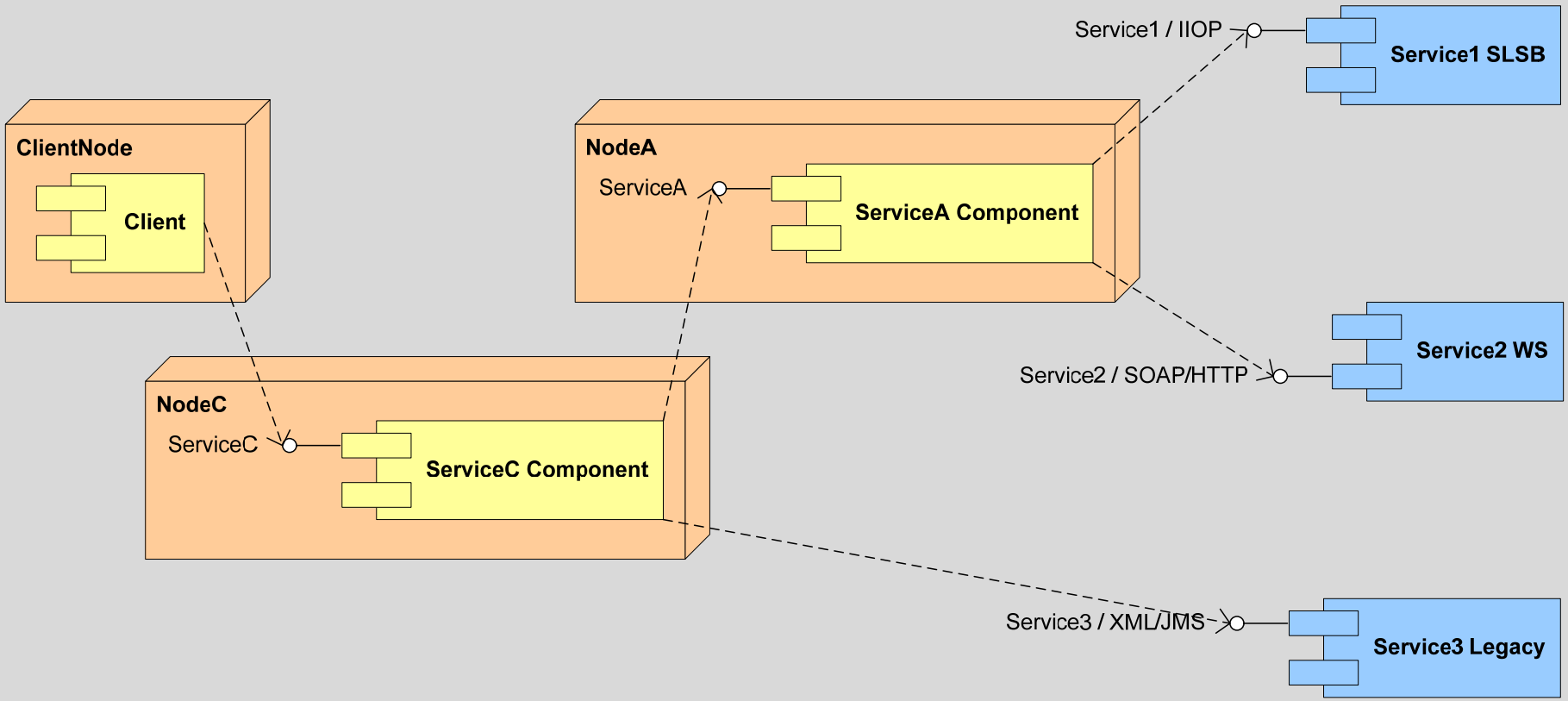
references and bibliography

[1] Mule website, www.mulesource.org/
[2] OSOA website, www.osoa.org/
[3] Open CSA website, www.oasis-opencsa.org/
[4] Apache Tuscany website, tuscany.apache.org/
[5] SCA Assembly Model Specification 1.0
[6] SCA Java Common Annotations and APIs 1.0
[7] SCA Java Component Implementation Specification 1.0
[8] SCA EJB Session Bean Binding 1.0
[9] SCA Web Service Binding Specification 1.0
[10] SCA JMS Binding Specification 1.0

references and bibliography

[11] SCA Java EE Integration Specification 0.9
[12] “SCA: Support for Composing Existing Applications in an SOA Solution”, OSOA
[13] Mule SCA website, www.mulesource.org/display/SCA
[14] Newton website, newton.codecauldron.org/
[15] Fabric3 website, fabric3.codehaus.org/

integration scenario for this presentation

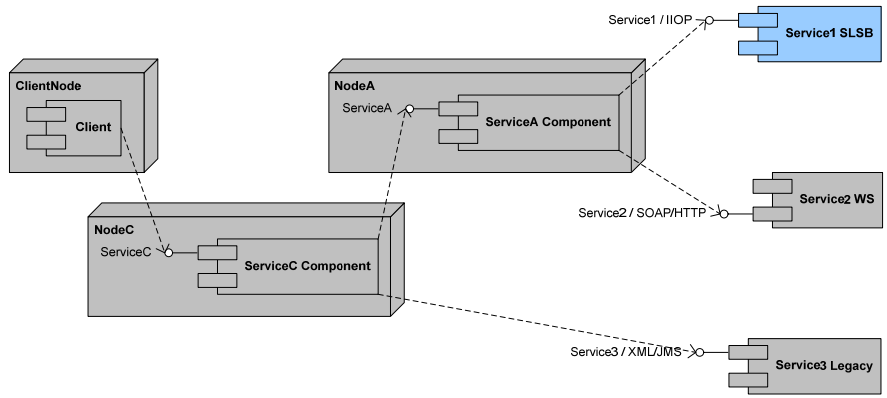


integration scenario for this presentation

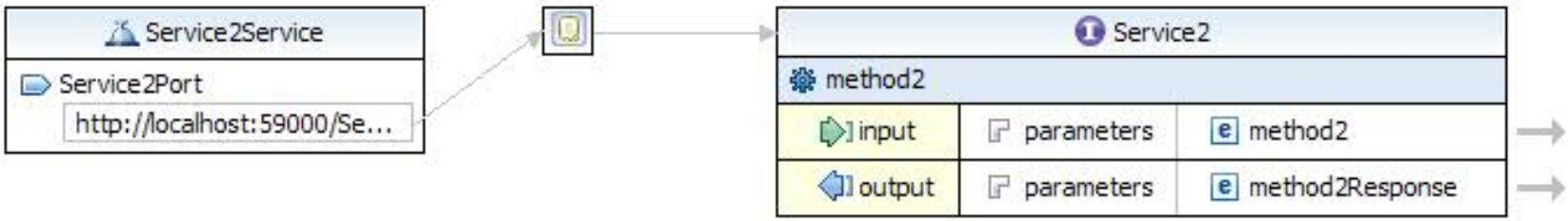
- build a composite application that
  - exposes ServiceC to a user interface component
  - and makes use of the existing services Service1, Service2 and Service3, which are exposed over IIOP, SOAP/HTTP and XML/JMS, respectively
- in addition, analysis has shown that a sub-functionality of the composite application – ServiceA – is a meaningful aggregate service over Service1 and Service2 and should thus be exposed as such
  - shall be hosted on a node "close" to Service1 and Service2 (and separate from the node hosting ServiceC)

Service1: SLSB exposed via IIOP

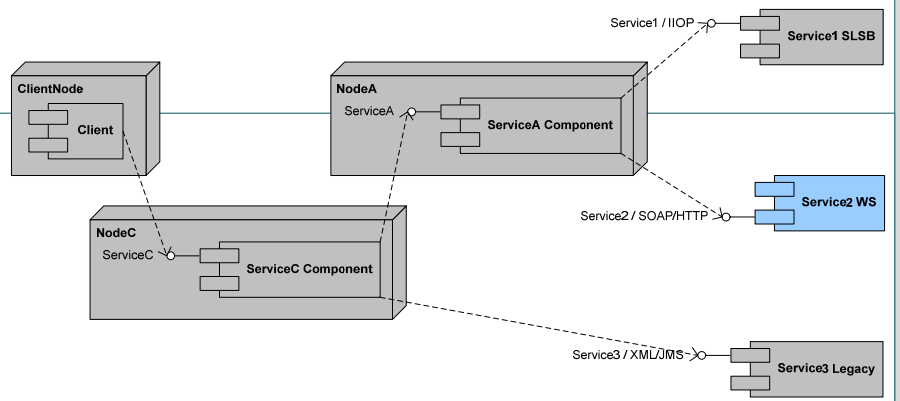
```
package net.gl.service1;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Service1 extends EJBObject {

    String method1(ComplexType1 param1) throws RemoteException;
}
```



Service2: doc/lit/wrapped SOAP 1.1 websvc

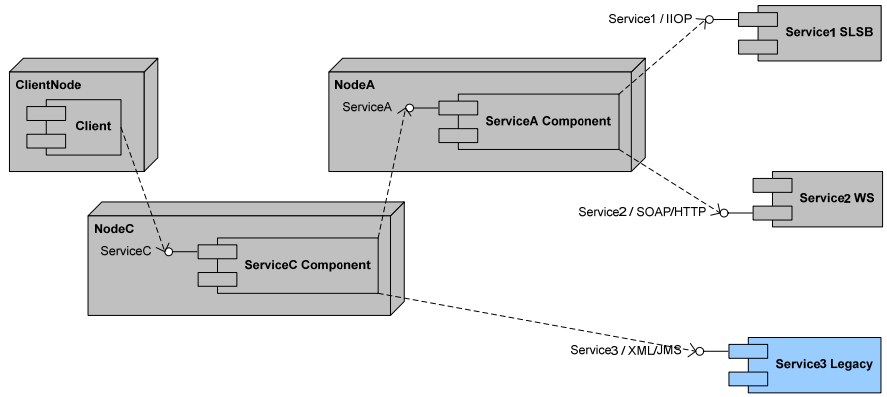


- input parameter to method2 is a complex type



Service3: consumes XML message via JMS

```
<net.gl.service3.ComplexType3>
  <prop31>UNBELIEVABLE!</prop31>
  <prop32>14</prop32>
</net.gl.service3.ComplexType3>
```



- sends response as string in JMS TextMessage to reply-destination given in reply-to header of request message
  - stating message-id of request message in correlation-id header of response message

Mule 2 basics

- re-design of the well-established **open-source ESB**
  - now tightly integrated with **Spring**
- Java **messaging framework** focused on integration
  - **read, transform, send data** as messages
  - no restriction on message format (payload type)
  - different component implementation technologies
    - POJO, EJB SLSB, Spring, Groovy, ...
  - various **transports** supported
    - SOAP/HTTP, IIOP, JMS, RMI, SMTP/IMAP/POP3, file, FTP, HTTP, JDBC, TCP/IP, UDP, XMPP, VM, ...
  - implementations of many integration patterns
    - routers, filters, transformers, ...

reference: [1]

Mule 2 uses Spring's schema-based config

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.0"
  xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.0"
  xsi:schemaLocation="
    http://www.mulesource.org/schema/mule/core/2.0
  http://www.mulesource.org/schema/mule/core/2.0/mule.xsd
    http://www.mulesource.org/schema/mule/stdio/2.0
  http://www.mulesource.org/schema/mule/stdio/2.0/mule-stdio.xsd
    http://www.mulesource.org/schema/mule/vm/2.0
  http://www.mulesource.org/schema/mule/vm/2.0/mule-vm.xsd">
...
</mule>
```

EJB connector to access SLSB

```
<ejb:connector
  name="ejb_glassfish"
  jndiInitialFactory=
  "com.sun.enterprise.naming.SerialInitContextFactory"
  jndiUrlPkgPrefixes="com.sun.enterprise.naming"
  securityPolicy="security.policy">
  <jndi-provider-property
    key="java.naming.factory.state"
    value=
    "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl" />
```

- libraries needed on Mule classpath
  - application server client library (here for Glassfish)
  - EJB client jar for invoked SLSB (here: Service1)

CXF connector to invoke/expose SOAP websvc

```
<cxf:connector name="soap_cxf" />
```

- CXF may use JAX-WS annotated Java classes
- Mule also supports Axis1 (not 2!) as a SOAP stack

JMS transport with special ActiveMQ support

```
<jms:activemq-connector name="jms_activemq"
  persistentDelivery="true" specification="1.1" />
```

- instantiates an embedded ActiveMQ broker
- often ActiveMQ is used in this way with Mule
  - ActiveMQ network of brokers (one in each Mule instance) may be used to route messages between Mule instances

VM connector for intra-VM communication

```
<vm:connector name="vm" />
```

- could also be used in an asynchronous mode

custom transformers as Mule services

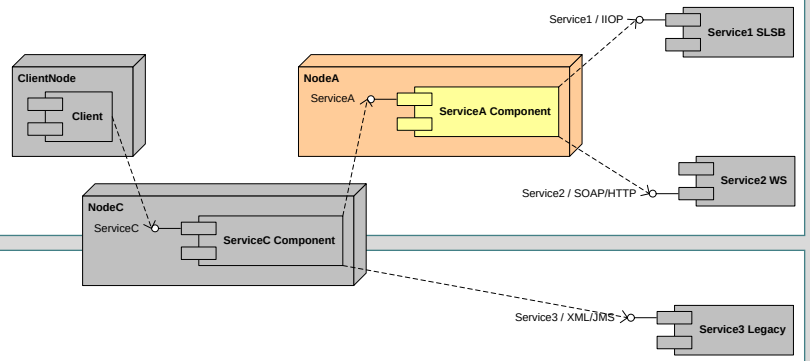
```
<service name="ComplexTypeATo1Transformer">
  <inbound>
    <vm:inbound-endpoint path="ComplexTypeATo1Transformer"
      synchronous="true" connector-ref="vm" /></inbound>
  <component>
    <singleton-object
      class="net.gl.serviceA.ComplexTypeATo1TransformerImpl" />
  </component>
</service>
```

- transformation logic implemented in Java
- used as a singleton component in Mule
- made available as a synchronous local Mule service
- alternatively: Mule also has a special framework for transformers, which the Mule-supplied transformers use

transformer code has no Mule dependencies

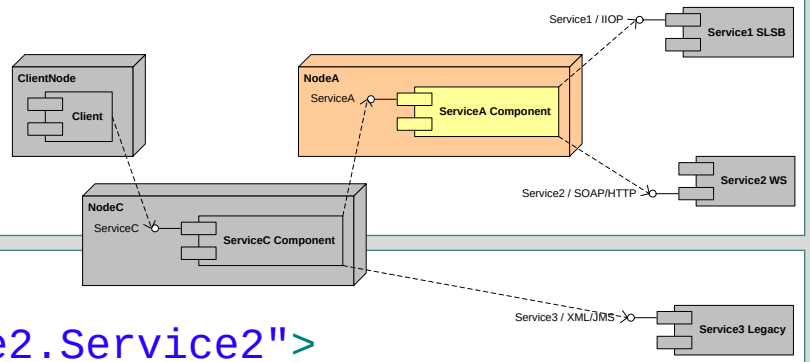
```
public class ComplexTypeATo1TransformerImpl implements
  ComplexTypeATo1Transformer {

  public ComplexType1 transform(ComplexTypeA input) {
    ComplexType1 result;
    if (input == null) {
      result = null;
    } else {
      result = new ComplexType1(input.getPropA1(),
                                input.getPropA2());
    }
    return result;
  }
}
```

ServiceA



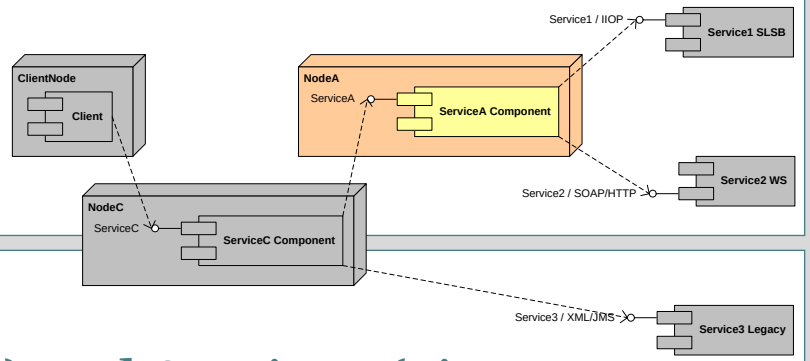

```
<service name="ServiceA">
  <inbound>
    <cxf:inbound-endpoint serviceClass="net.gl.serviceA.ServiceA"
      address="http://localhost:59001/ServiceA"
      synchronous="true" connector-ref="soap_cxf" />
   </inbound>
   <component>
     <singleton-object class="net.gl.serviceA.ServiceAImpl">
       <property key="serviceDivider" value="4" />
     </singleton-object>
     <binding interface="net.gl.service1.Service1">
       <ejb:outbound-endpoint object="ejb/Service1Bean"
         method="method1" remoteSync="true" host="localhost"
         port="3700" connector-ref="ejb_glassfish" />
     </binding>
(contd.)
```

ServiceA




```
(contd.)
    <binding interface="net.gl.service2.Service2">
      <cxf:outbound-endpoint
        address="http://localhost:59000/Service2"
        operation="method2" remoteSync="true"
        connector-ref="soap_cxf" />
    </binding>
    <binding interface="net.gl.serviceA.ComplexTypeATo1Transformer">
      <vm:outbound-endpoint path="ComplexTypeATo1Transformer"
        remoteSync="true" connector-ref="vm" />
    </binding>
    <binding interface="net.gl.serviceA.ComplexTypeATo2Transformer">
      <vm:outbound-endpoint path="ComplexTypeATo2Transformer"
        remoteSync="true" connector-ref="vm" />
    </binding>
  </component>
</service>
```

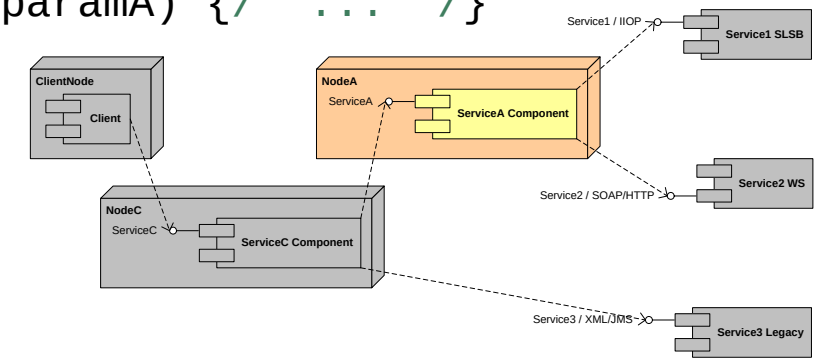
ServiceA



- is an aggregate service
  - invoking Service1 (via EJB/IIOP) and Service2 (via SOAP/HTTP)
  - also invokes 2 transformation services via the Mule VM protocol
- is itself exposed via SOAP/HTTP
- ServiceAImpl has no Mule dependencies (see next)
- the Java interface ServiceA will be mapped to WSDL
  - mapping should be customized, e.g. with JAX-WS annotations
- ServiceA is a content-based router
  - routes messages based on input message content
  - but does not use Mule's router framework

ServiceAImpl has no Mule dependencies

```
public class ServiceAImpl implements ServiceA {

  public void setService1(Service1 service1) {/* ... */}
  public void setService2(Service2 service2) {/* ... */}
  public void setComplexTypeATo1Transformer(
    ComplexTypeATo1Transformer xf) {/* ... */}
  public void setComplexTypeATo2Transformer(
    ComplexTypeATo2Transformer xf) {/* ... */}
  public void setServiceDivider(int serviceDivider) {/* ... */}

  public String methodA(ComplexTypeA paramA) {/* ... */}
}
```



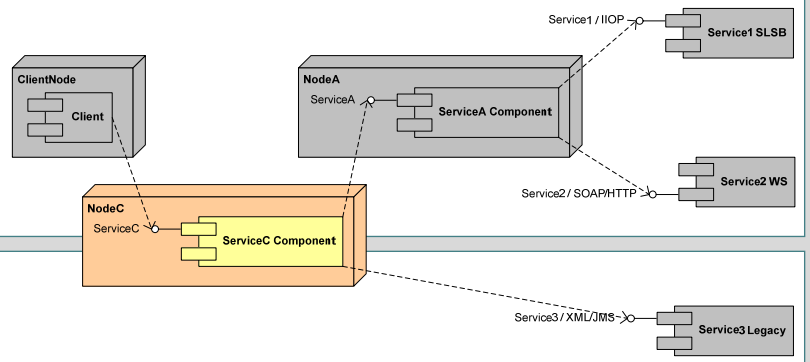
Mule message transformers for XML/JMS

```
<jms:object-to-jmsmessage-transformer name="tojms" />
<jms:jmsmessage-to-object-transformer name="fromjms" />
<mulexml:xml-to-object-transformer name="xml2o" />
<mulexml:object-to-xml-transformer name="o2xml" />
```

- Mule ships with several “technical” transformers like these
  - en/decryption, de/compression, en/decoding, several XML-related transformers, ...

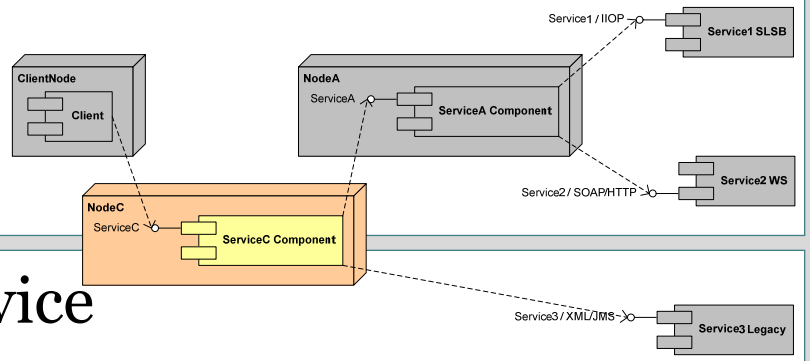
reference: [1]

ServiceC




```
<service name="ServiceC">
  <inbound>
    <vm:inbound-endpoint path="ServiceC" synchronous="true"
      connector-ref="vm" /></inbound>
  <component class="net.gl.serviceC.ServiceCImpl">
    <binding interface="net.gl.serviceA.ServiceA">
      <cxf:outbound-endpoint
        address=http://localhost:59001/ServiceA" operation="methodA"
        remoteSync="true" connector-ref="soap_cxf" />
    </binding>
    <binding interface="net.gl.service3.Service3">
      <jms:outbound-endpoint queue="Service3.requests"
        remoteSync="true" transformer-refs="o2xml tojms"
        responseTransformer-refs="fromjms xml2o"
        connector-ref="jms_activemq" /></binding>
  </component>
</service>
```

ServiceC



- is our composite application service
  - exposed via Mule VM protocol
    - should use remoteable protocol for remote clients
  - invoking ServiceA (via SOAP/HTTP) and Service3 (via XML/JMS)
- ServiceCImpl has no Mule dependencies
- because the JMS endpoint is “remote-sync” the JMS transport makes use of a temporary queue for response messages
  - that queue is identified in the reply-to header of the request message
  - does not appear in the Mule configuration

reference: [1]

we have seen Mule 2 as a prototypical ESB

- multiple transport protocols
- synchronous and asynchronous interactions
- message routing
- message transformation: technical and “business”
- distributing the ESB itself
- how business logic stays clean of infrastructure concerns
- core Mule concepts:
  - message abstraction
  - component, service
  - transport, endpoint
  - router

Mule 2: what we did not talk about

- **deployment** (stand-alone, web container, app server)
- most of the **transports** supported by Mule (other than CXF, EJB, JMS and VM)
- most of the standard Mule **transformers** (other than JMS and simple XML-related)
- implementing transformers following the Mule framework
- **router** implementations that come with Mule
- using non-trivial **Spring** features for configuring Mule
- any **quality-of-service** concerns (guaranteed delivery over JMS, webservice reliable messaging and security, etc.)
- Mule thread management and **messaging styles**

reference: [1]

Mule 2: what we did not talk about

- **tooling** (development, management, monitoring; SOA governance)
- Mule SCA project (currently no code available)

reference: [1, 13]

Service Component Architecture (SCA) basics

- a **programming model** to implement software using Service-Oriented Architecture (**SOA**) principles
  - concepts: component, composite, service, reference, binding
- developed by Open Service Oriented Architecture collaboration (**OSOA**) in collaboration with **OASIS** Open Composite Services Architecture (**Open CSA**) Member Section
- significant backing: IBM, Oracle/BEA, TIBCO, Sun, ...
- **new**/work in progress
  - 1.0 specs from mid 2007
  - Java EE integration currently 0.9

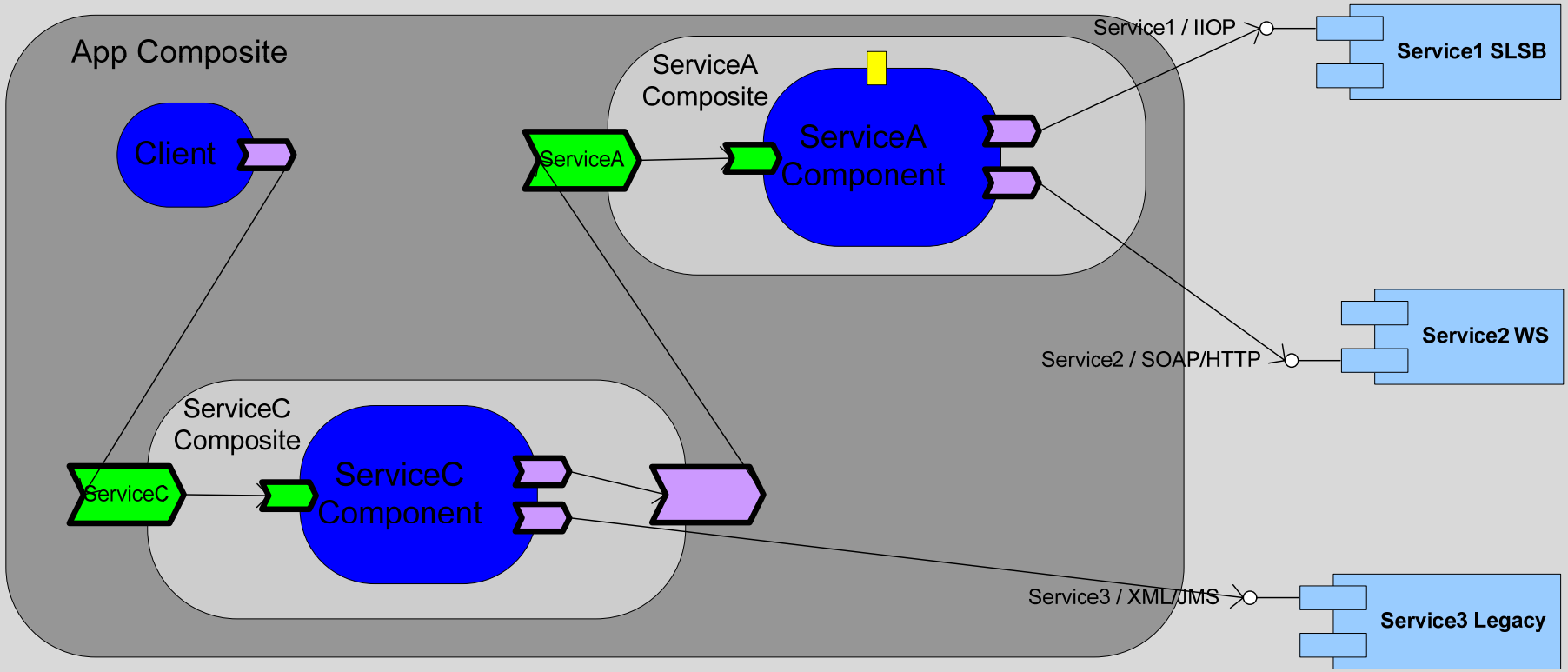
reference: [2, 3]

SCA implementations

- commercial implementations:
  - Rogue Wave, IBM, Oracle/BEA, Paremus, Covansys, TIBCO
- main open-source implementations:
  - Apache Tuscany
    - backed by IBM
  - Newton
    - backed by Paremus
    - OSGi-based and supports Spring Dynamic Modules
  - Fabric3
    - alpha-level

reference: [2, 4, 14, 15]

our integration scenario in SCA notation



core SCA concepts

- **component**: configured instance of an implementation; typically provides and references (consumes) services and has properties
- **implementation**: business logic for the component, implementing its services; in Java/POJO, Spring, EJB SB, BPEL, C++, scripts, composite, ...
- **composite**: **assembly** of components, services, references and wires; assembly is **recursive** (composites are valid component implementations)
- **service**: set of **operations** provided by an implementation; typed by an **interface**; local or remote; exposed via bindings

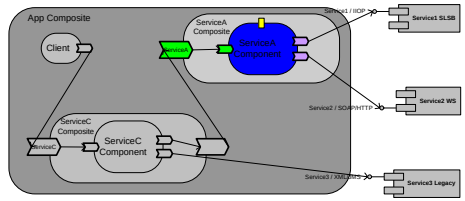
reference: [5]

core SCA concepts

- **reference**: a service that an implementation consumes and thus depends on; typed by an **interface**; accessed via binding
- **binding**: the transport protocol used to expose services and consume references; SCA binding, webservices, EJB (IIOP), JMS, JCA, ...
- **wire**: connects a reference to a service and thereby selects the binding to use for that communication
- **component type**: the services, references and properties that can be configured on an implementation
- **domain**: set of SCA **nodes** from the same vendor; basis for distributing the SCA runtime

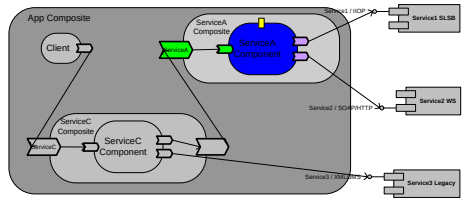
reference: [5]

ServiceA.composite




```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  targetNamespace="urn:example" name="ServiceAComposite">

  <service name="ServiceA" promote="ServiceAComponent" />

  <component name="ServiceAComponent">...</component>
  <component name="ComplexTypeATo1Transformer">...</component>
  <component name="ComplexTypeATo2Transformer">...</component>

</composite>
```

ServiceA.composite



- defines 3 components
  - of as yet unknown implementation type
- composite itself
  - exposes a service, ServiceA, that is originally exposed by the ServiceAComponent within the composite (= propagation)
    - ServiceA states no binding: SCA binding is implied
  - has no references, i.e. no declared (and thus externally visible) dependencies on other services
  - has no properties

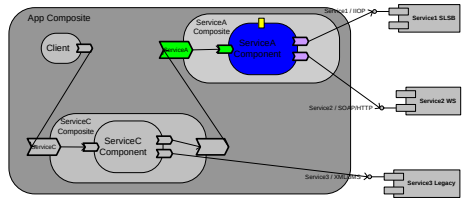
reference:

custom transformers as SCA components

```
<component name="ComplexTypeATo1Transformer">
  <implementation.java
    class="net.gl.serviceA.ComplexTypeATo1TransformerImpl" />
</component>

public interface ComplexTypeATo1Transformer {/* ... */}
public class ComplexTypeATo1TransformerImpl implements
  ComplexTypeATo1Transformer {/* ... */}
```

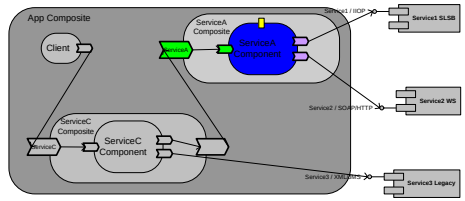
- same Java interface and impl class as before
  - no SCA dependencies
- no scope specified: defaults to STATELESS
  - unknown number of instances in use
- service is detected by reflection
  - determined as local service defined by Java interface

ServiceAComponent definition




```
<component name="ServiceAComponent">
  <implementation.java class="net.gl.serviceA.ServiceAImpl" />
  <property name="serviceDivider">4</property>
  <reference name="service1">
    <binding.ejb
      uri="corbaname:iiop:1.2@localhost:3700#ejb/Service1Bean" />
  </reference>
  <reference name="service2">
    <binding.ws uri="http://localhost:59000/Service2" />
  </reference>
  <reference name="complexTypeATo1Transformer"
    target="ComplexTypeATo1Transformer" />
  <reference name="complexTypeATo2Transformer"
    target="ComplexTypeATo2Transformer" />
</component>
```

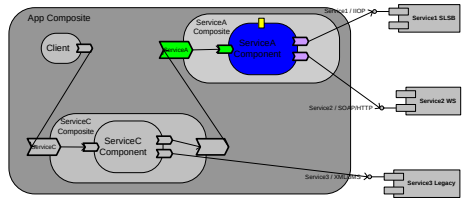
ServiceAComponent definition



- transformer references wired to local components
  - use local variant of SCA binding because service interfaces are local
- service1 reference wired via EJB binding
- service2 reference wired via webservice binding
  - uses default values for WSDL mapping
- all references of ServiceAComponent wired within the composite and hence not changeable from the outside
- service provided by component (ServiceA) not specified but discovered by reflection of the implementation (see next)

ServiceA Java interface uses SCA annotations

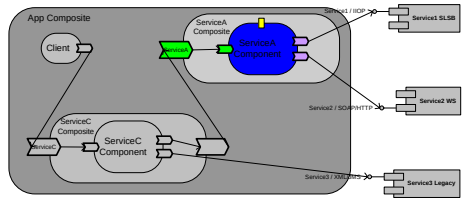
```
package net.gl.serviceA;

import org.osoa.sca.annotations.Remotable;

@Remotable
public interface ServiceA {
  String methodA(ComplexTypeA paramA);
}
```



- remotable services imply pass-by-value semantics
- without this annotation it would be a local service and not usable with any remote binding (remote SCA binding, webservices, JMS, ...)

ServiceAImpl may use SCA annotations

```
import org.osoa.sca.annotations.Property;
import org.osoa.sca.annotations.Reference;

public class ServiceAImpl implements ServiceA {

  @Reference protected Service1 service1;
  @Reference protected Service2 service2;

  @Reference protected ComplexTypeATo1Transformer a21;
  @Reference protected ComplexTypeATo2Transformer a22;

  @Property protected int serviceDivider;

  public String methodA(ComplexTypeA paramA) {...}
}
```

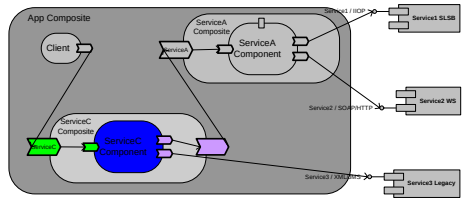


relaxed interface compatibility for wires

- to wire a reference to a service, the interface used in the reference need not be the same as the interface exposed by the service
  - both local or remote
  - service interface must provide all operations defined by reference interface
    - SCA uses “signature compatibility”: operation name, input and output types; order of inputs; exceptions
  - applies to all language combinations for interfaces (Java, WSDL)
  - the actual name, namespace and/or package of the interfaces are irrelevant!

reference: [5]

ServiceC.composite

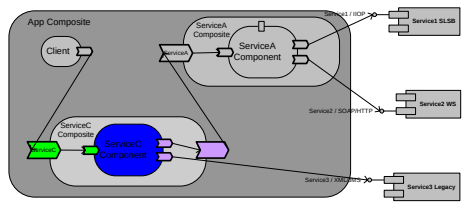



```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  targetNamespace="urn:example" name="ServiceCComposite">
  <service name="ServiceC" promote="ServiceCComponent" />
  <component name="ServiceCComponent">...</component>
  <reference name="serviceA" promote="ServiceCComponent/serviceA" />
</composite>
```

- defines one component ServiceCComponent which
  - implements a service (ServiceC) that is propagated to be a service exposed by the composite
    - ServiceC Java interface is annotated with @Remotable so it is exposed via the remote variant of the SCA binding
  - has an unwired reference (serviceA) that is propagated to be a reference of the composite itself

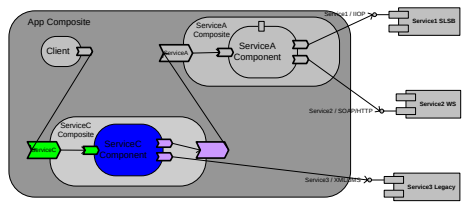
ServiceCComponent wired via SOAP/JMS

```
<component name="ServiceCComponent">
  <implementation.java class="net.gl.serviceC.ServiceCImpl" />
  <reference name="service3">
    <binding.ws
      uri="jms:/Service3_requests?
      transport.jms.ConnectionFactoryJNDIName=
      ConnectionFactory&amp;
      java.naming.factory.initial=
      org.apache.activemq.jndi.ActiveMQInitialContextFactory&amp;
      java.naming.provider.url=tcp://localhost:50061" />
  </reference>
</component>
```

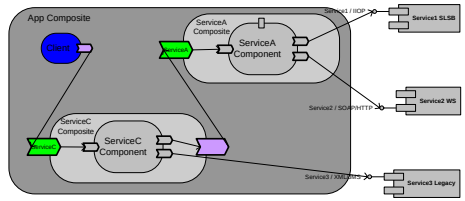


ServiceCComponent wired via SOAP/JMS

- reflection on the implementation class discovers component type consisting of
  - service with interface defined by remotable Java interface ServiceC (annotated)
  - two references: serviceA and service3 (no annotation necessary as interfaces are annotated with @Remotable)
- uses Axis2 SOAP/JMS capabilities under the hood
  - SOAP messages (including SOAP header) sent in JMS TextMessage
  - uses temporary reply queue

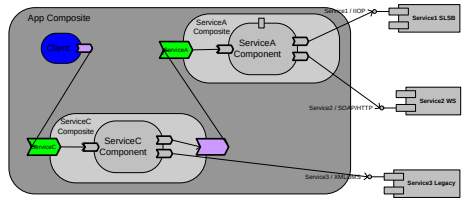


App.composite




```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  targetNamespace="urn:example" name="AppComposite">

  <component name="ServiceCComponent">
    <implementation.composite name="example:ServiceCComposite" />
    <service name="ServiceC">
      <binding.sca />
      <binding.ws uri="http://localhost:59002/ServiceC" />
    </service>
    <reference name="serviceA" target="ServiceAComponent" />
  </component>
  <component name="ServiceAComponent">
    <implementation.composite name="example:ServiceAComposite" />
  </component>

</composite>
```

App.composite



- represents the composite application
- defines 2 components as implemented by the 2 composites
  - an example of recursive assembly
  - again, reflection on these composites discovers the component type of the components
- exposes ServiceC over SCA and webservice bindings
- wires serviceA reference on ServiceCComponent to ServiceA exposed by ServiceAComponent
  - assuming that these components run on separate nodes (as required by this integration scenario) then this is a remote wire, transparently using the remote variant of the SCA binding

SCA: what we did not talk about

- the value that the notion of **component type** (and the possibility to define it external to the implementation) brings
- almost all component **implementation models** apart from Java/POJO and composite (Spring, EJB, BPEL, JAX-WS, C++, scripts, ...)
- **scopes** (STATELESS, REQUEST, CONVERSATION, COMPOSITE)
- constraining types
- **bidirectional** (callbacks) and **conversational** interfaces
- **policy** framework! (security, transactions, ...)

reference: [5, 6, 7]

SCA: what we did not talk about

- **how to distribute** a composite application over several SCA nodes in a SCA domain
  - implementation of the example integration scenario that runs on a distributed Tuscany domain is available and differs slightly from the code shown here
- Java **client and component APIs**
- **packaging** and contributions
- the relationship of SCA to **SDO** (a generic DTO impl.)
- **tooling** (development, management, monitoring; SOA governance)
  - e.g. the Eclipse SOA Web Tools project

reference: [5, 6, 7]

SCA question marks

- maturity of SCA implementations
  - Tuscany JMS binding
- compatibility of SCA implementations
  - TCK / compatibility test suite?
  - reference implementation?
  - common set of bindings and implementation models supported on all SCA implementations?
- how is interface compatibility for wires defined in the presence of complex types as inputs/outputs in operations?
  - does this imply something like “structure compatibility” for complex types in order to check “signature compatibility” of the operations in which these complex types are used

reference: [6]

SCA question marks

- ability to construct arbitrary JMS messages to interface with a legacy system
  - binary messages? non-XML text messages?
  - without requiring components to be coded against JMS APIs (i.e., handle JMSMessage as operation arguments or return types)

reference: [5, 10]

comparing SCA, Mule 2 and Java EE

- the following table compares SCA, Mule 2 (as a representative ESB) and Java EE (in particular EJB Session Beans and Message Driven Beans)
- this does *not* imply that these technologies are alternatives or even cover the same technology space:
  - SCA could serve as a programming model for ESBs and Mule 2 (assuming that the notion of well-defined service interfaces is introduced into Mule 2) – see the Mule SCA project
  - SCA integrates with Session Beans on several levels
    - Session Beans as a component implementation technology
    - EJB binding to invoke Session Beans or expose services via IIOP
- rather this comparison focuses on the programming models

reference: [8, 11, 13]

comparing SCA, Mule 2 and Java EE

	SCA	Mule 2	Java EE / EJB
main abstraction	service, methods	message, payload	depends
status	emerging (standard)	mature but redesigned (implementation)	mature (standard)
communication protocols supported	many, extensible	very many, extensible	many, fixed (w/o 3rd party libraries)
framework and implementations of advanced EIPs (router, splitter, aggregator, ...)	minimal (mainly technical message transformers)	yes	no
component implementation options	flexible, extensible	flexible, extensible, must run on JVM	limited, fixed, must run on JVM
business logic kept clean of communication code	yes	yes	no
business logic kept clean of middleware dependencies	almost (e.g. advanced uses require SCA APIs)	yes (unless using router, transformer frameworks, etc.)	no

comparing SCA, Mule 2 and Java EE

	SCA	Mule 2	Java EE / EJB
change communication protocol through reconfiguration	yes	yes	very limited (IIOP to appserver specific)
components publish well-defined service contract	yes	no	depends (SLSB yes, MDB no)
assembly of components independent of component implementation	yes	yes	no
dependency injection	yes (service references and properties)	yes (anything via Spring)	yes (fixed and limited)
AOP-style interception of component invocations	no (but has powerful policy framework)	yes (via Spring)	very limited
distributed runtime	yes	no (but can be added)	yes (clusters)

Contemporary Application Integration: ESBs and SCA

Dr Gerald Loeffler, MBA

JavaDeus 08
Sun Microsystems Austria
Java Developer Conference