SCA4J User Guide

Introduction 4

An Overview of SCA and SCA4J 5

What is SCA? 5

SCA Key Concepts 5

Summary 8

First Taste of SCA 9

Overview of the Application 9

Order Component 9

Shipping Component 10

Billing Component 10

Test Class 11

Composite SCDL 11

Maven POM Descriptor 12

Looking Forward 13

Components 15

Implementation Type 15

Services 16

References 17

Properties 21

Looking Forward 25

Component Scopes 26

Stateless Components 26

Composite Components 27

Request Components 29

Conversational Components 30

Component Lifecycle 34

Looking Forward 35

Asynchronous Services and Callbacks 36

Asynchronous Services 36

Callbacks and Bidirectional Interfaces 36

Looking Forward 36

Principles of Composition 37

Composite Implementation Type 37

Promoting Services 37

Promoting References 37

Composite Level Properties 37

Composition through Inclusion 37

SCA4J Runtime Environments 38

Configuring the Runtime 39

Integration Test Runtime 40

JEE Web Application Runtime 41

Generic Runtime API 42

Policies 43

Transactions 44

Web Services Binding 45

JMS Binding 46

FTP Binding 47

TCP Binding 48

Hessian Binding 49

Burlap Binding 50

REST 51

JDBC Datasources 52

Java Persistence API 53

Custom Policies 54

Timed and Scheduled Events 55

Extending the Runtime 56

Introduction

SCA4J is one of the most complete and leading open source SCA implementation available under the ASL 2.0 license. SCA4J extends it capabilities beyond the specifications covered under SCA and offers additional capabilities that are key to build complex enterprise-class business applications. This document provides a comprehensive coverage of all aspects of SCA4J from an end user perspective.

The source code used by all the examples used in this guide maybe downloaded from the URL http://www.sca4j.org/svn/sca4j/tutorials. The examples within this guide will require to have at least version 2.0.9 of Apache Maven installed. Any additional requirements that are specific to individual examples will be detailed with the examples.

An Overview of SCA and SCA4J

In the past few years SOA has evolved into one of the most prevalent architectural paradigms in building enterprise class middleware applications. Despite its wide spread adoption, SOA has pretty much stayed a vague set of principles, interpreted differently based on the perspective of the interpreter. More than often practitioners use the terms SOA and Web Services invariably, even though, web services based technologies are only one of the means of realizing service-oriented architecture.

There has been a significant lack of standards and specifications that provide prescriptive and definitive guidelines into how service-oriented business applications are to be developed. However, with the inception of Service Component Architecture (SCA) by OSOA, SOA has been made a realistic proposition for enterprise developers and architects. SCA has since been adopted by OASIS and is now being ratified under the Open Composite Services Architecture (OpenCSA) initiative.

What is SCA?

SCA is a set of specifications, addressing both runtime vendors and enterprise application developers and architects, for developing applications based on service oriented architectural principles and paradigms. SCA advocates a compositional architectural pattern for building enterprise class service oriented applications.

It addresses the assembly of both fine-grained tightly coupled and coarse-grained loosely coupled components. SCA composition is based on recursive assembly where fine-grained tightly coupled components are composed together to build coarse-grained components. These coarse grained components can be used in a higher-level contexts to build even coarser grained components and enterprise applications.

SCA Key Concepts

The key SCA concepts are,

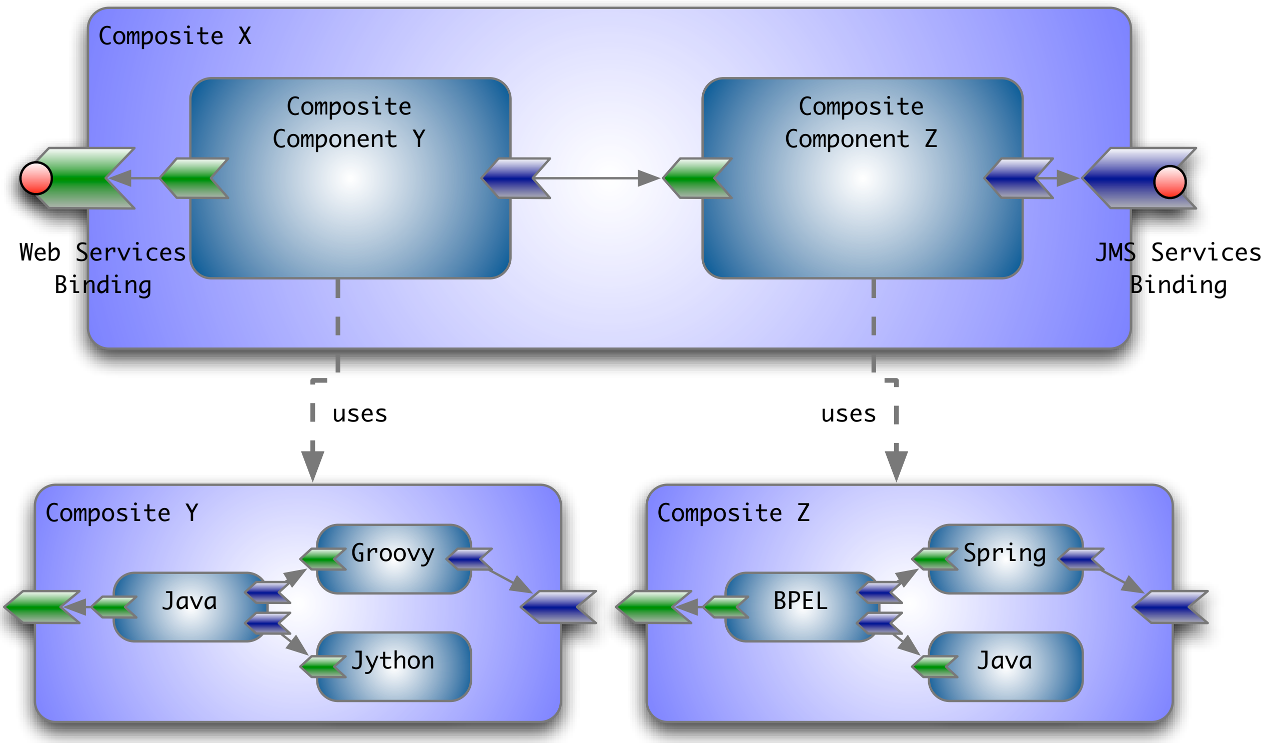
- Assembly model
- Component implementation model
- Bindings
- Intents and policies
- Domain

Assembly Model

SCA assembly model primarily defines the recursive assembly of components. The recursive assembly allows fine-grained components to be wired into what in SCA terms is called composites. These composites than can be used as components in higher-level composites and applications.

The type of a component generally defines the services exposed by the component, properties that can be configured on the component and dependencies the component has on services offered by other components.

The diagram below shows an example of an SCA assembly model,



In the diagram above Composite X is composed of two components Composite Component Y and Composite Component Z, which themselves are composites.

Composite Y is composed of a Java component, a Groovy component and a Jython component. The service offered by Java component is promoted outside the composite. This service is exposed using a web services binding when the Composite Y is used as a component (Composite Component Y) within Composite X. The Java component depends on two external services. In SCA terminology these dependencies are called references. All the references for the Java component are satisfied by services offered by other components within the same composite, the Groovy and the Jython components respectively. However, the reference on the Groovy component has been promoted outside the composite and will have to be satisfied in a context within which Composite Y is used as a component. In the diagram above this is achieved using the service promoted by Composite Z.

Composite Z is composed of a Java component, a BPEL component and a Spring component, the service offered by the BPEL component is promoted to be used when the composite is used as a component in a higher level composite. In the above example this service caters for the reference from Composite Component Y, which in turn was the promoted reference from the Groovy component within Composite Y. The references on the BPEL component within Composite Z are catered to by the services offered by the Java and Spring components within Composite Z. The reference on the Spring component is promoted and when the Composite Z is used as a component (Composite Component Z) within the Composite X, the reference is bound to a JMS binding to an external system.

As you can see SCA allows building of compositional service-oriented applications using components built on heterogeneous technologies and platforms. SCA assembly model is generally expressed in an XML based language called Service Component Description Language (SCDL). The sections below lists the SCDL snippets for Composite X, Composite Y and Composite Z. Please don't worry about the details of the code snippets, we will cover them in detail in the coming chapters.

Component & Implementation Model

SCA component and implementation (C & I) model describes how components, services, references etc are represented, defined and packaged in multiple languages and technology platforms. The current SCA suite of specifications cover the client and implementation model for Java, Spring, BPEL, C++ etc. SCA4J provides an extension model that allows adding new component types like Groovy, JRuby and other scripting languages.

Bindings

SCA bindings enable services and references to be accessed over multiple transport and data binding protocols. The current SCA suite of specifications cover bindings for Web Services, JMS, EJB etc. SCA4J provides an extension model that allows adding new binding types like Hessian, Burlap, FTP etc.

Intents and Policies

The policy framework is one of the key aspects of SCA. Policy framework allows Quality of Service (QoS) aspects to be defined, managed, enforced and governed, orthogonally to component implementations and assembly using declarative intents and corresponding policy sets. Like all the other SCA aspects, policy framework is highly extensible as well. Policy framework allows policies to be defined using standard mechanisms like WS-Policy or using your own policy language.

Summary

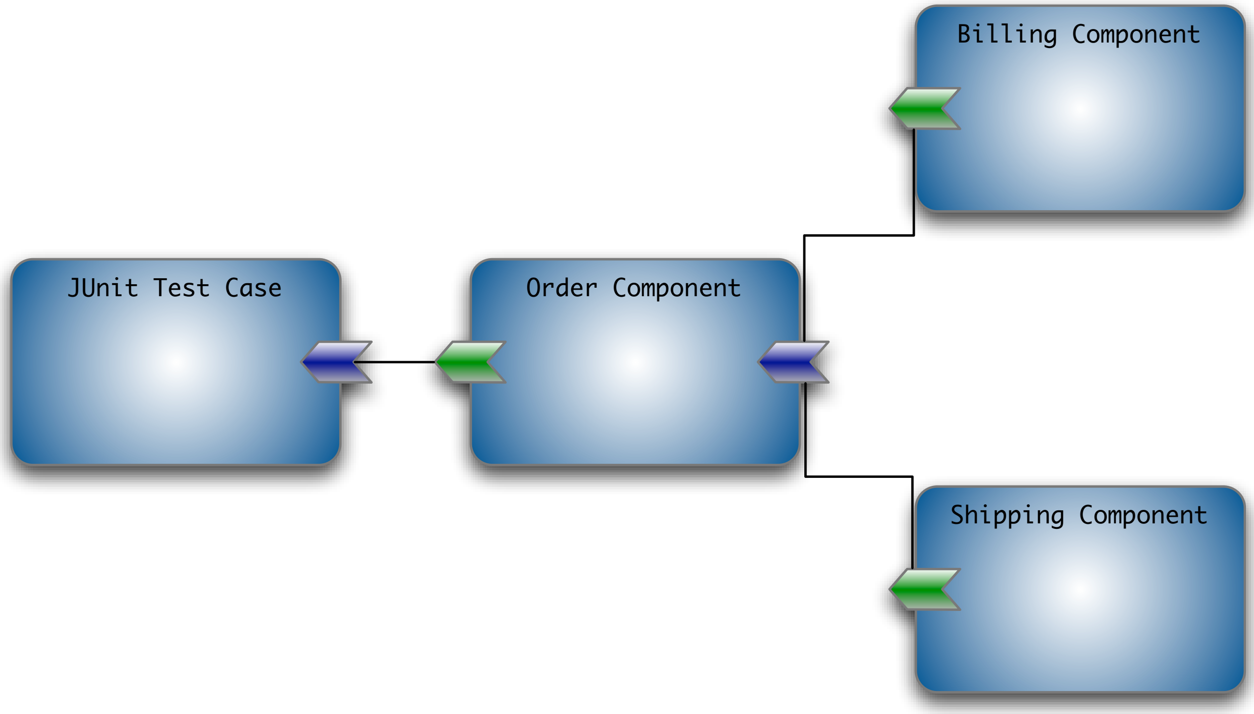
SCA provides a set of specifications, addressing both runtime vendors and enterprise developers/architects, for building service-oriented enterprise applications. For the first time we have a set of specifications that address how SOA can be built rather than what SOA is about.

First Taste of SCA

In this chapter we will develop a small SCA based application and test it, so that we can get a first taste of SCA. The example will demonstrate some of the basic aspects of SCA. We will build on this example in forthcoming chapters to demonstrate the more advanced tenets of SCA specifications the SCA4J runtime implementation. The source code for the example used in this chapter can be found at http://www.sca4j.org/svn/sca4j/tutorials/tutorial-getting-started.

Overview of the Application

The diagram below depicts the components involved in the example application,



All our components in this example are written in Java. We have an order component, that utilizes a billing and shipping component for fulfilling the order. We will use JUnit test cases to test the application. SCA4J offers a Maven based SCA runtime that will allow you to treat JUnit test cases as first class service component and allows you to test your service oriented applications as part of your continuous build and integration process.

Order Component

The order component is implemented in Java as shown below,

```
package org.sca4j.tutorial.order;

import org.osoa.sca.annotations.Reference;
import org.sca4j.tutorial.billing.BillingComponent;
import org.sca4j.tutorial.shipping.ShippingComponent;

public class OrderComponent {

    @Reference protected BillingComponent billingComponent;
    @Reference protected ShippingComponent shippingComponent;

    public boolean placeOrder(String productName, String address, String creditCard) {
        if (billingComponent.bill(address, creditCard, 2.0)) {
            shippingComponent.ship(productName, address);
            return true;
        }
        return false;
    }

}
```

The order component POJO class, with two instance variables. The @Reference annotation indicates at runtime the underlying container is required to provide object instances for these variables. The logic itself is pretty straightforward, it calls the billing component first and if successful calls the shipping component.

Shipping Component

The shipping component is also a POJO class that is listed below,

```
package org.sca4j.tutorial.shipping;

public class ShippingComponent {

    public void ship(String productName, String address) {
    }

}
```

Billing Component

The listing for the billing component is shown below,

```
package org.sca4j.tutorial.billing;

public class BillingComponent {

    public boolean bill(String address, String creditCard, double amount) {
        return true;
    }

}
```

Test Class

The test class is a plain JUnit test case that can be run within the embedded SCA4J Maven runtime. The runtime uses Surefire for running the test cases and reporting the results. JUnit test cases are service components within SCA4J embedded maven runtime and recognizes all the SCA vocabulary. In the snippet below the test class has a reference to the order component, which would be made available by the runtime when you run the test using Maven.

```
package org.sca4j.tutorial.order;

import org.osoa.sca.annotations.Reference;

import junit.framework.TestCase;

public class OrderITest extends TestCase {

    @Reference protected OrderComponent orderComponent;

    public void testPlaceOrder() {
        assertTrue(orderComponent.placeOrder("Pizza", "70 Byron Road", "12345"));
    }

}
```

When the test is run, the test case calls the order component instance made available by the container and the asserts the return value of invoking the method on the order component.

Composite SCDL

Now we need to wire the billing and shipping components to the order component and the order component to the test case. In SCA this is done using the SCDL, which is an XML based language for defining components and wiring them together. The embedded Maven runtime looks for a file called itest.composite to find the tests it needs to run. For the purpose of simplicity we will define our application classes and test classes in that same file. In the forthcoming chapters we will have a look at splitting the composite files, when we delve into the details of the principles of composition.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:sca4j="urn:sca4j.org"
           name="OrderTestComposite">

    <component name="OrderITest">
        <sca4j:junit class="org.sca4j.tutorial.order.OrderITest"/>
        <reference name="orderComponent" target="OrderComponent" />
    </component>

    <component name="OrderComponent">
        <implementation.java class="org.sca4j.tutorial.order.OrderComponent"/>
        <reference name="billingComponent" target="BillingComponent" />
        <reference name="shippingComponent" target="ShippingComponent" />
    </component>

    <component name="BillingComponent">
        <implementation.java class="org.sca4j.tutorial.billing.BillingComponent"/>
    </component>

    <component name="ShippingComponent">
        <implementation.java class="org.sca4j.tutorial.shipping.ShippingComponent"/>
    </component>

</composite>
```

In the snippet above, we have four components in the composite file. The test case is defined using the sca4j:junit element and the application classes are defined using the implementation.java element. The junit element is in the sca4j namespace as it is an extension offered by the the SCA4J implementation, where the application components are defined using the standard implementation.java element that comes from the OSOA namespace. The references are bound to target components explicitly using the target element.

Don’t worry too much on the details of the composite. We will delve into the details of this in subsequent chapters. The main purpose of this chapter is to give you a feel of SCA and SCA4J runtime.

Maven POM Descriptor

Finally we need a Maven POM descriptor that will define the necessary dependencies and the plugin for running the embedded SCA4J Maven runtime. Please note that SCA4J also offers a programmatic API for testing SCA code in your favorite IDE, we will cover that later.

```
<project>

    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.sca4j.tutorial</groupId>
        <artifactId>parent</artifactId>
        <version>1.0</version>
    </parent>

    <artifactId>tutorial-getting-started</artifactId>
    <packaging>jar</packaging>
    <name>Getting Started</name>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </dependency>
        <dependency>
            <groupId>org.sca4j</groupId>
            <artifactId>sca4j-maven-host</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.sca4j</groupId>
                <artifactId>sca4j-itest-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

The POM defines the dependency on junit and sca4j-maven-host and also defines the sca4j-itest-plugin in the build lifecycle. The plugin is bound to the integration-test phase of the build lifecycle. You can run the test by invoking the command ‘mvn verify’ from your favorite shell.

Looking Forward

In the next chapter we will be looking at the basics of authoring atomic components within SCA emphasizing on components written in Java. In the chapter you will learn about

- Writing components
- Defining services on components
- Component references
- Properties on components
- Dependency injection
- Wiring components

Components

Components form the basic building blocks of SCA based applications. In SCA components can be either atomic or composite. Atomic components are the ones from an SCA perspective that can't be decomposed any further. Atomic components may be implemented in a variety of technologies like Java, C++, BPEL, PHP, scripting languages like Groovy and JRuby, XQuery etc. Composite components are virtual and is one of the key concepts behind SCA. Composite components are built by composing both atomic and other composite components. Recursive hierarchical composition is a guiding principle within SCA where fine-grained tightly coupled atomic components are composed together to form course-grained composites and these coarse-grained composites are themselves used as first class components in higher level composites and applications.

In this chapter we will look at implementing atomic components in Java.

Four key aspects of a component, whether it is atomic or composite, that defines the finger print or type of the component are the following,

- The implementation type of the component
- Services that are exposed by the component
- References that are required by the component
- Properties available on the component

Implementation Type

For a composite component the implementation type of the component is introspected from the composite file that contains the component definition. The implementation type of atomic components are dependent on the type of technology used to implement the component. For e.g, an atomic component implemented in Java, the implementation type is a Java class. Similarly, for a BPEL atomic component the implementation type is WS-BPEL process.

A component that is implemented as a Java class should provide a non-ambiguous public or package access constructor that can be used by the container to instantiate an instance. The constructor doesn't need to be no-argument. For example, the constructor can be used for injecting references and properties into the component using CDI (Constructor Dependency Injection). In the absence of a non-ambiguous constructor, the constructor that is expected to be used by the container for instantiating an instance of the component will have to be annotated with the @Constructor annotation.

```java
package org.sca4j.tutorial.order;

import org.osoa.sca.annotations.Constructor;
import org.osoa.sca.annotations.Reference;
import org.sca4j.tutorial.billing.BillingComponent;
import org.sca4j.tutorial.shipping.ShippingComponent;

public class OrderComponent {

    private BillingComponent billingComponent;
    private ShippingComponent shippingComponent;

    @Constructor
    public OrderComponent(@Reference BillingComponent billingComponent,
                          @Reference ShippingComponent shippingComponent) {
        this.billingComponent = billingComponent;
        this.shippingComponent = shippingComponent;
    }

    public OrderComponent() {
    }

}
```

A Java component implementation is defined a composite assembly using SCDL with the implementation.java element as shown below,

```xml
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:sca4j="urn:sca4j.org"
           name="OrderTestComposite">

    <component name="ShippingComponent">
        <implementation.java class="org.sca4j.tutorial.shipping.ShippingComponent"/>
    </component>

</composite>
```

The class attribute is used to specify the fully qualified name of the Java class that implements the component.

Services

Services constitute business operations that are offered by a component to its consumers. Services have defined contracts, and the contracts can be expressed in a variety of mechanisms including,

- Java interfaces

- Java classes
- WSDL 1.1 port types
- WSDL 2.0 interfaces

A Java implementation class may implement the interface that defines the service contract for the service it offers. If the implementation class doesn't implement any interface, all public methods within the class form the service that is offered by the component. A class may also implement more than one interface. In such scenarios, the @Service interface is required to be used to specify the interfaces which are part of the services offered by the component as below.

```
package org.sca4j.tutorial.order;

import org.osoa.sca.annotations.Service;

@Service(interfaces = {OrderService.class})
public class OrderComponent implements OrderService, SomeOtherInterface {
}
```

References

References are dependencies a component has on the services offered by other internal or external components. In the Java programming model a reference is indicated using the @Reference annotation as shown below. The annotation can appear on an instance variable, a setter method or a constructor argument. The access modifier for any of these needs to be either public or protected.

```
package org.sca4j.tutorial.order;

import org.osoa.sca.annotations.Reference;
import org.sca4j.tutorial.billing.BillingComponent;
import org.sca4j.tutorial.shipping.ShippingComponent;

public class OrderComponent {

    @Reference protected BillingComponent billingComponent;
    @Reference protected ShippingComponent shippingComponent;

}
```

A component may have one or more references. References by default in SCA are mandatory. This means if a reference is not provided by the container at runtime, the deployment of the composite will fail. You can make a reference optional by setting the required attribute on the annotation to false.

References have names, which are important for wiring the references to target components in the assembly. The name can be either inferred or explicitly specified using the name attribute on the @Reference annotation. The name is inferred as the name of the field or the setter property based on the Java bean naming convention if the annotation appear on the field or setter. If they appear on constructor arguments they are not inferable, as parameter names are not available via reflection at runtime. You can only wire targets explicitly to a reference if they have names. This means references injected through constructor arguments without an explicit name attribute can only be auto-wired.

Wiring References

A reference can be wired explicitly to a target using the target attribute on the reference as shown below,

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="OrderTestComposite">

    <component name="OrderComponent">
        <implementation.java class="org.sca4j.tutorial.order.OrderComponent"/>
        <reference name="billingComponent" target="BillingComponent" />
    </component>

    <component name="BillingComponent">
        <implementation.java class="org.sca4j.tutorial.billing.BillingComponent"/>
    </component>

</composite>
```

Alternatively, you can resort to auto-wiring, by switching it on explicitly using the autowire attribute on the composite element as shown below,

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           name="OrderTestComposite" autowire="true" >

    <component name="OrderComponent">
        <implementation.java class="org.sca4j.tutorial.order.OrderComponent"/>
    </component>

    <component name="BillingComponent">
        <implementation.java class="org.sca4j.tutorial.billing.BillingComponent"/>
    </component>

</composite>
```

By default autowiring is switched off. If you explicitly switch it on, it is only enabled within the context of the composite on which it is turned on.

Cardinality of References

References can have a multiplicity of more than one. SCA4J supports multiple references of types Collection, Map, Set and List. The example below shows an example of list of references,

```
package org.sca4j.tutorial.order;

import org.osoa.sca.annotations.Reference;
import org.sca4j.tutorial.billing.BillingComponent;
import java.util.List;

public class OrderComponent {

    @Reference protected List<BillingComponent> billingComponents;

}
```

A list of references can be explicitly wired as shown below,

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="OrderTestComposite">

    <component name="OrderComponent">
        <implementation.java class="org.sca4j.tutorial.order.OrderComponent"/>
        <reference name="billingComponents"
                   target="BillingComponent1 BillingComponent2" />
    </component>

    <component name="BillingComponent1">
        <implementation.java class="org.sca4j.tutorial.billing.BillingComponent"/>
    </component>

    <component name="BillingComponent2">
        <implementation.java class="org.sca4j.tutorial.billing.BillingComponent"/>
    </component>

</composite>
```

The target attribute on the source component uses a token separated list to specify the the names of the target components to which the source component is wired.

Some times you may want to have a map of references that are keyed against certain values. For example, the order component may receive a delivery method and based on the requested delivery method, it may want to select a different shipping service. You can download an example of using a map of references from http://www.sca4j.org/svn/sca4j/tutorials/tutorial-map-reference.

The first thing you need to do is defined the map of references in your order component as shown below,

```
package org.sca4j.tutorial.order;

import java.util.Map;

import org.osoa.sca.annotations.Reference;
import org.sca4j.tutorial.shipping.ShippingComponent;

public class OrderComponent {

    @Reference protected Map<String, ShippingComponent> shippingComponents;

    public boolean placeOrder(String productName, String address, String delivery) {
        shippingComponents.get(delivery).ship(productName, address);
        return false;
    }

}
```

Next your wire all the instances of the shipping components to the order component in the assembly as shown below. You can either explicitly wire them or autowire them. The example below uses autowire for brevity.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" xmlns:sca4j="urn:sca4j.org"
           name="OrderTestComposite" autowire="true" >

    <component name="OrderComponent">
        <implementation.java class="org.sca4j.tutorial.order.OrderComponent"/>
    </component>

    <component name="ExpressShippingComponent" sca4j:key="express">
        <implementation.java
            class="org.sca4j.tutorial.billing.ExpressShippingComponent"/>
    </component>

    <component name="StandardShippingComponent" sca4j:key="standard" >
        <implementation.java
            class="org.sca4j.tutorial.shipping.StandardShippingComponent"/>
    </component>

</composite>
```

The important aspect above is the key attribute that belongs to the sca4j namespace on the component element of the target components. The value of this attribute is used to key the targeted references.

Properties

The examples used in this section may be downloaded from the tutorials website at, http://www.sca4j.org/svn/sca4j/tutorials/tutorial-component-property.

Properties in SCA are used to configure the behavior of a component implementation through externally set data values. Properties can be of simple type or complex type. Properties are by default optional. In the Java programming model a property is indicated using the @Property annotation. You can use the required attribute on the property to indicate a property is mandatory. Properties can be injected through protected/public fields, setter methods and constructor arguments.

```
public class OrderComponent {
    @Property protected double deliveryCharge;
}
```

Property Names

The names of properties can either be specified explicitly using the name attribute or inferred. They are inferred when the @Property annotation appears on field or a setter method. If it is on a constructor argument and it is a required property, you always will have to specify an explicit name as parameters names are not available at runtime in the JVM and you will have to refer to the name of the property in the assembly, to specify a value for a required property.

Property Values

Property values are specified in the assembly as shown below,

```
<component name="OrderComponent">
    <implementation.java class="org.sca4j.tutorial.order.OrderComponent"/>
    <property name="deliveryCharge">100</property>
</component>
```

The value of the property is specified as the contents of the property element within the component on which the property is defined. Property values can also be sources from the domain and composite level properties using the source attribute on the property element, we will have a look at that in detail in forthcoming chapters.

Property Types

With the Java programming model, SCA4J supports a number of simple and complex property types,

Simple Property Types

- boolean/java.lang.Boolean
- byte/java.lang.Byte
- java.util.Calendar
- java.lang.Class
- java.util.Date
- double/java.lang.Double
- float/java.lang.Float
- int/java.lang.Integer
- long/java.lang.Long
- short/java.lang.Short
- java.lang.String
- java.net.URI
- java.net.URL
- javax.xml.namespace.QName

List Property Types

SCA4J supports properties of type java.util.List which can be parameterized with one of the following,

- byte/java.lang.Byte
- java.lang.Class
- double/java.lang.Double
- float/java.lang.Float
- int/java.lang.Integer
- long/java.lang.Long
- short/java.lang.Short
- java.lang.String
- javax.xml.namespace.QName

List property values are defined as a token separated list in the assembly as shown below,

```
<component name="OrderComponent">
    <implementation.java class="org.sca4j.tutorial.order.OrderComponent"/>
    <property name="deliveryCharges">
        100
        200
        300
    </property>
</component>
```

In the implementation class the above will be mapped as shown below,

```
public class OrderComponent {
    @Property protected List<Double> deliveryCharges;
}
```

Map Property Types

Properties can be of type java.utl.Map and the key and value types can be parameterized by,

- javax.xml.namespace.QName -> java.lang.Class
- java.lang.String -> byte/java.lang.Byte
- java.lang.String -> java.lang.Class
- java.lang.String -> double/java.lang.Double
- java.lang.String -> float/java.lang.Float
- java.lang.String -> int/java.lang.Integer
- java.lang.String -> long/java.lang.Long
- java.lang.String -> short/java.lang.Short
- java.lang.String -> java.lang.String

The snippet below shows the order component implementation where the prices for product codes are defined as a map,

```
public class OrderComponent {
    @Property protected Map<String, Integer> prices;
    public int calculatePrice(String productName) {
        return prices.get(productName);
    }
}
```

The snippet below shows how the property values are defined as a map in the composite. We use the element tag as the key and the element content as the value of the map (in the example below the key will be pizza and the value will be 100).

```
<component name="OrderComponent">
    <implementation.java class="org.sca4j.tutorial.order.OrderComponent"/>
    <property name="prices">
        <pizza>100</pizza>
    </property>
</component>
```

Complex Property Types

Complex properties are also supported using JAXB. The code below shows a complex type in Java annotated using the JAXB annotation,

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class ChargeDetail {
    public int valueAddedTax;
    public int deliveryCharge;
}
```

The code below shows how the above type is used as a property type in an implementation component,

```
public class OrderComponent {

    @Property protected Map<String, Integer> prices;
    @Property protected ChargeDetail chargeDetail;

    public int calculatePrice(String productName) {
        int price = prices.get(productName);
        return price +
               price * chargeDetail.valueAddedTax/100 +
               chargeDetail.deliveryCharge;
    }

}
```

The value for the property is defined in the composite using a JAXB serialized XML embedded within the property element as shown below,

```xml
<component name="OrderComponent">
    <implementation.java class="org.sca4j.tutorial.order.OrderComponent"/>
    <property name="chargeDetail">
        <chargeDetail xmlns="">
            <valueAddedTax>15</valueAddedTax>
            <deliveryCharge>2</deliveryCharge>
        </chargeDetail>
    </property>
</component>
```

Looking Forward

In this chapter we have looked at the basics of authoring Java components using implementation classes, services, references and properties. In the next chapter we will look at how to manage component lifecycles using scopes. We learn how to implement,

1. Composite scope components
2. Stateless components
3. Request scope components
4. Conversational components
5. Component lifecycle callbacks

Component Scopes

The lifecycles of the components you deploy in the SCA4J runtime is managed by the runtime. The lifecycle determine the instance of the component a client interact with when it makes an invocation to the component. The SCA specification also provides lifecycle callbacks that are invoked before the first time a component instance is made available in an invocation callstack and also before it gets destroyed. The lifecycle and instance management of components are determined by the declared scopes on the component.

There are four different component scopes supported by SCA,

1. Composite scope: A composite scope component is like a singleton and there is only one instance available within a deployed SCA application.
2. Stateless scope: A stateless scope component is created each time the component is invoked by a client, and destroyed straight after the invocation.
3. Conversation scope: An instance of a conversation scope component is maintained by the client. The instance is created first time a client invokes the component and destroyed when the client explicitly ends the conversation. Please note that the use of conversation scope components has been removed from the SCA specifications since version 1.1, though SCA4J continues to support it for backward compatibility.
4. Request scope: Request scope components are associated with the thread that serves a remote request coming into the SCA runtime. They are created the first time they are accessed within the invocation callstack and destroyed when the thread finish serving the remote request.

All examples used in this chapter are available for download from the tutorial website at http://www.sca4j.org/svn/sca4j/tutorials/tutorial-component-scope.

The scope a components is indicate at the class-level annotation @Scope with the name attribute in the Java programming model. The possible values for the name attribute are REQUEST, COMPOSITE, STATELESS and CONVERSATIONAL. You can alternatively use the meta annotations described in the following sections. The scope annotation is always specified on the implementation class.

Stateless Components

The example below shows the use of @Stateless annotation to indicate that the component is stateless.

```
import java.util.LinkedList;
import java.util.List;
import org.sca4j.api.annotation.scope.Stateless;

@Stateless
public class StatelessOrderComponent implements OrderService {

    private List<String> orders = new LinkedList<String>();

    public void addOrder(String product) {
        orders.add(product);
    }

    public List<String> getOrders() {
        return orders;
    }

}
```

The snippet below shows the test for the component. As indicated, each time an invocation is made on the instance variable statelessOrderService, the container creates an instance of the order component, serves the request and destroys it. This is why no state is maintained between invocations on a stateless component.

```
import org.osoa.sca.annotations.Reference;
import junit.framework.TestCase;

public class ComponentScopeITest extends TestCase {

    @Reference protected OrderService statelessOrderService;

    public void testStateless() {
        statelessOrderService.addOrder("pizza");
        assertTrue(statelessOrderService.getOrders().isEmpty());
    }

}
```

Composite Components

Composite components like singletons and live for the life of the application within the runtime. They are stateful, however any mutable state within the component is not thread-safe. In the Java programming model the composite scope can be defined using one of the class-level annotations @Scope("COMPOSITE") or @Composite. Composite components are created on demand, the first time they get used. If you want them eagerly created, you can indicate so using the class-level annotation @EagerInit.

The snippet below shows the order component implemented using composite scope. As you can see the only difference is we use the @Composite scope annotation instead of @Stateless.

```
import java.util.LinkedList;
import java.util.List;
import org.sca4j.api.annotation.scope.Composite;

@Composite
public class CompositeOrderComponent implements OrderService {

    private List<String> orders = new LinkedList<String>();

    public void addOrder(String product) {
        orders.add(product);
    }

    public List<String> getOrders() {
        return orders;
    }

}
```

The test class shows two test methods testComposite1 and testComposite2. The SCA4J runtime invokes the test method in alphabetical order. As you can see the state information stored in the component is persisted across two test methods. Everytime, the test methods access the instance variable compositeService, they get the same instance of the order component.

Even if the traget order component is wired to one or more source components they will all share the same instance of the composite component across multiple threads. So the component implementation should make sure it provides thread-safe access to mutable state. Composite components are useful for implementing shared read-only state that needs to be accessed from different parts of the application.

```java
import org.osoa.sca.annotations.Reference;
import junit.framework.TestCase;

public class ComponentScopeITest extends TestCase {

    @Reference protected OrderService compositeOrderService;

    public void testComposite1() {
        compositeOrderService.addOrder("pizza");
        compositeOrderService.addOrder("coke");
        assertEquals(2, compositeOrderService.getOrders().size());
    }

    public void testComposite2() {
        compositeOrderService.addOrder("pizza");
        compositeOrderService.addOrder("coke");
        assertEquals(4, compositeOrderService.getOrders().size());
    }

}
```

Request Components

Lifecycle of components that are request scope are maintained from the point from which the component is accessed first time within a thread that is serving a remote request to the end of service the request. A remote request can either be an invocation coming over a remote transport binding like web services or JMS as well as individual test execution, within a JUnit test.

In the Java programming model the composite scope can be defined using one of the class-level annotations @Scope("COMPOSITE") or @Composite. Composite components are created on demand, the first time they get used. The snippet below shows the order component implemented using composite scope. As you can see the only difference is we use the @Composite scope annotation instead of @Stateless.

```
import java.util.LinkedList;
import java.util.List;
import org.sca4j.api.annotation.scope.Request;

@Request
public class RequestOrderComponent implements OrderService {

    private List<String> orders = new LinkedList<String>();

    public void addOrder(String product) {
        orders.add(product);
    }

    public List<String> getOrders() {
        return orders;
    }

}
```

The test class shows two test methods testRequest and testRequest2. As you can see the state information stored in the component is persisted within a given test method, but not across two test methods as we saw with the composite order component. This is because the SCA4J runtime treats each test method execution as a new request coming into the domain.

```
import org.osoa.sca.annotations.Reference;
import junit.framework.TestCase;

public class ComponentScopeITest extends TestCase {

    @Reference protected OrderService compositeOrderService;

    public void testRequest1() {
        requestOrderService.addOrder("pizza");
        requestOrderService.addOrder("coke");
        assertEquals(2, requestOrderService.getOrders().size());
    }

    public void testRequest2() {
        requestOrderService.addOrder("pizza");
        requestOrderService.addOrder("coke");
        assertEquals(2, requestOrderService.getOrders().size());
    }

}
```

Conversational Components

With components that are of conversational scope, the lifecycle is controlled by the client components. Because of this the semantics of conversationality needs to be visible to the clients. In addition to indicating the component scope on the implementation class with @Scope("CONVERSATION") or its meta-annotation @Conversation, the client facing interface for the component needs to be markes as @Conversational and the interface should also have a method annotated with @EndsConversation. When the client calls the method annotated with @EndsConversation, the component interface is destroyed. The snippet below shows an instance of a conversational interface,

```
import java.util.List;
import org.osoa.sca.annotations.Conversational;
import org.osoa.sca.annotations.EndsConversation;

@Conversational
public interface ConversationalOrderService extends OrderService {
    void addOrder(String product);
    List<String> getOrders();
    @EndsConversation void close();
}
```

The snippet below shows a conversational component that implements the above interface,

```
import java.util.LinkedList;
import java.util.List;
import org.sca4j.api.annotation.scope.Conversation;

@Conversation
public class ConversationalOrderComponent implements ConversationalOrderService {

    private List<String> orders = new LinkedList<String>();

    public void addOrder(String product) {
        orders.add(product);
    }

    public List<String> getOrders() {
        return orders;
    }

    public void close() {
    }

}
```

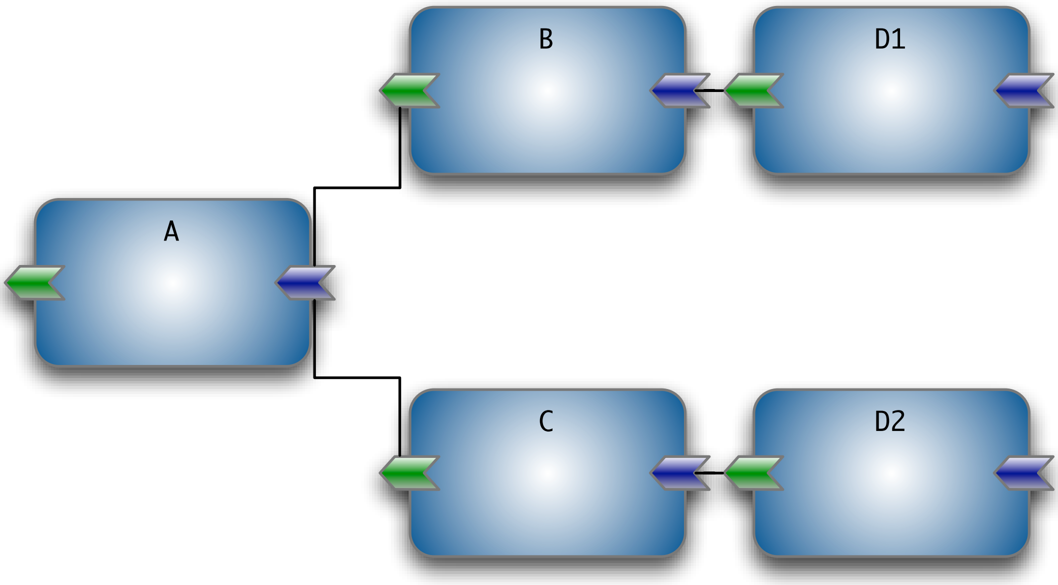
The lifecycle of a conversational component starts the first time it is accessed by its clients and ends when the client calls the method marked with the annotation @EndsConversation on the conversational interface.

In the test below, when the conversationalOrderService instance variable is accessed the first time in the test method an instance is created and the state is maintained till the close method is called. Next time the the variable is accessed in the test method a new conversational instance of the component is created.

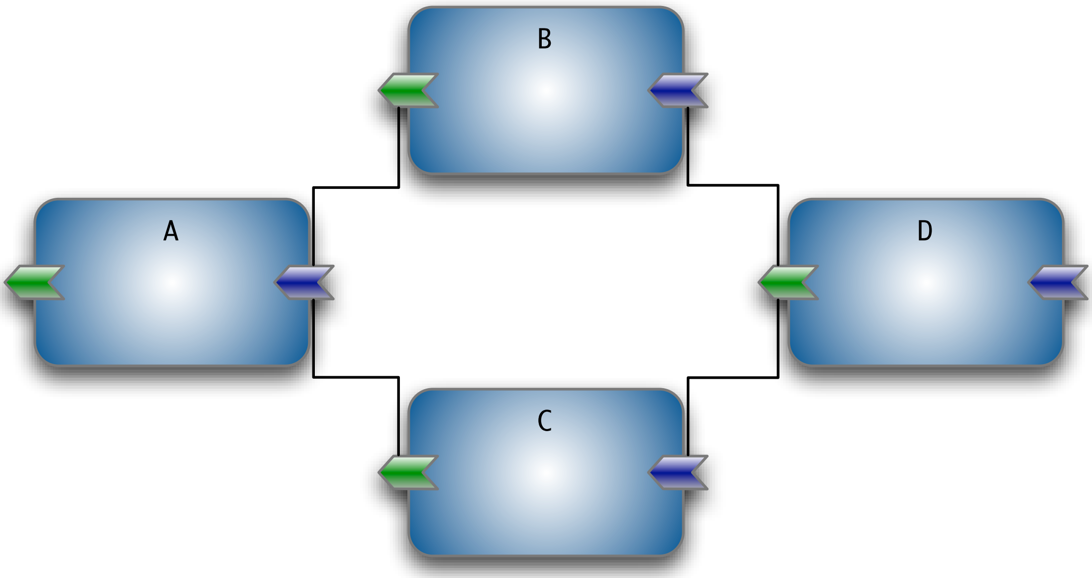
```
import java.util.LinkedList;
import java.util.List;
import org.sca4j.api.annotation.scope.Conversation;

@Conversation
public class ConversationalOrderComponent implements ConversationalOrderService {

    private List<String> orders = new LinkedList<String>();

    public void addOrder(String product) {
        orders.add(product);
    }

    public List<String> getOrders() {
        return orders;
    }

    public void close() {
    }

}
```

Points to Note

In SCA the conversation semantics are between the consumer and the producer components. This means SCA doesn’t support conversation diamonds. This means if you have four conversational components A, B, C and D, such that A calls B and C, B calls D and C calls D, B and C will get a different instance of D.



If you want B and C to share the same instance of D, you may want to use request scope instead of conversation scope as shown below,



It is also worth noting that conversation scope has been removed from latest SCA specifications because of some of the nuances around conversational callbacks and a portable mechanism of maintaining conversationality on components provisioned over remote transport bindings. Most of the use cases addressed by conversational scoped components on fine-grained thread-safe state management can be achieved using request scope components.

Component Lifecycle

Since the lifecycle of the components are maintained by the container based on their declared scope, it is important to understand their lifecycle in terms of when they get created and destroyed.

1. Composite components are created the first time they are accessed and destroyed when the application is stopped or the runtime is shutdown. You can also eagerly instantiate a composite scope component by annotating it with the class-level annotation @EagerInit
2. Stateless components are instantiated everytime they are accessed and removed straight after each invocation
3. Request scope components are instantiated the first time they are accessed and destroyed when the runtime completes serving the remote request associated with the thread
4. Conversational scope components are instantiated the first time they are accessed and destroyed when the client calls the @EndsConversation method on the conversational interface

The components can receive lifecycle callback methods, just after they are created by having methods annotated with @Init and @Destroy respectively as shown below,

```
import java.util.LinkedList;
import java.util.List;
import org.sca4j.api.annotation.scope.Request;
import org.osoa.sca.annotations.Init;
import org.osoa.sca.annotations.Destroy;

@Request
public class RequestOrderComponent implements OrderService {

    private List<String> orders = new LinkedList<String>();

    public void addOrder(String product) { orders.add(product); }

    public List<String> getOrders() { return orders; }

    @Init public void start() { }

    @Destroy public void stop() { }

}
```

Looking Forward

In the next chapter we will be looking at two key aspects of SCA,

1. Callbacks and bi-directional interfaces
2. Asynchrnous invocation

Asynchronous Services and Callbacks

TODO

Asynchronous Services

TODO

Callbacks and Bidirectional Interfaces

TODO

Looking Forward

In this chapter we have looked at invoking services asynchronously and implementing callbacks using conversational interfaces. In the next chapter we will look at one of the key value propositions of SCA, recursive and hierarchical composition. This will cover how we can recursively compose atomic components into virtual coarse-grained components to build large-scale applications. In the next chapter will cover,

1. Composite implementation types
2. Composition using inclusion
3. Promoting services
4. Promoting references
5. Composite level properties

Principles of Composition

TODO

Composite Implementation Type

TODO

Promoting Services

TODO

Promoting References

TODO

Composite Level Properties

TODO

Composition through Inclusion

TODO

SCA4J Runtime Environments

Configuring the Runtime

Integration Test Runtime

JEE Web Application Runtime

Generic Runtime API

Policies

Transactions

Web Services Binding

JMS Binding

FTP Binding

TCP Binding

Hessian Binding

Burlap Binding

REST

JDBC Datasources

Java Persistence API

Custom Policies

Timed and Scheduled Events

Extending the Runtime