SCA Service Component Architecture

Java Common Annotations and APIs

SCA Version 1.00. March 21 2007

Technical Contacts:

	Ron Barack	SAP AG
	Michael Beisiegel	IBM Corporation
	Henning Blohm	SAP AG
	Dave Booz	IBM Corporation
	Jeremy Boynes	Independent
	Ching-Yun Chao	IBM Corporation
	Adrian Colyer	Interface21
	Mike Edwards	IBM Corporation
	Hal Hildebrand	Oracle
	Sabin Ielceanu	TIBCO Software, Inc
	Anish Karmarkar	Oracle
	Daniel Kulp	IONA Technologies plc.
	Ashok Malhotra	Oracle
	Jim Marino	BEA Systems, Inc.
	Michael Rowley	BEA Systems, Inc.
	Ken Tam	BEA Systems, Inc
	Scott Vorthmann	TIBCO Software, Inc
	Lance Waterman	Sybase, Inc.

Copyright Notice

License

The Service Component Architecture Specification is being provided by the copyright holders under the following license. By using and/or copying this work, you agree that you have read, understood and will comply with the following terms and conditions:

Permission to copy and display the Service Component Architecture Specification and/or portions thereof, without modification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Service Component Architecture Specification, or portions thereof, that you make:

1. A link or URL to the Service Component Architecture Specification at this location:
   - http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

2. The full text of this copyright notice as shown in the Service Component Architecture Specification.

BEA, Cape Clear, IBM, Interface21, IONA, Oracle, Primeton, Progress Software, Red Hat, Rogue

Wave, SAP, Siemens, Software AG., Sun, Sybase, TIBCO (collectively, the “Authors”) agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to patents that they deem necessary to implement the Service Component Architecture Specification.

THE Service Component Architecture SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SPECIFICATION AND THE IMPLEMENTATION OF ITS CONTENTS, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OR TITLE.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE Service Components Architecture SPECIFICATION.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Service Component Architecture Specification or its contents without specific, written prior permission. Title to copyright in the Service Component Architecture Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

Table of Contents

1. Common Annotations, APIs, Client and Implementation Model

1.1. Introduction

The SCA Common Annotation, APIs, Client and Implementation Model specifies a Java syntax for programming concepts defined in the SCA Assembly Model Specification [1]. It specifies a set of APIs and annotations that may be used by Java-based SCA specifications.

Specifically, this specification covers:

1 Implementation metadata for specifying component services, references, and properties

2. A client and component API J3. Metadata for asynchronous and conversational services

3. Metadata for callbacks4. Definitions of standard component implementation scopes

5. Java to WSDL and WSDL to Java mappings

6. Security policy annotations

Note that individual programming models may chose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate .

The goal of specifying the annotations, APIs, client and implementation model in this specification is to promote consistency and reduce duplication across various Java-related component implementation type specifications. The annotations, APIs, client and implementation model defined in this specification are designed to be used by other SCA Java-related specifications in either a partial or complete fashion.

This document defines implementation metadata using the annotation capability from Java™ 2 Standard Edition (J2SE) 5. However, SCA also allows service clients and implementations to be written using J2SE 1.4. All metadata that is represented by annotations can also be expressed using a component type side file, as defined in the SCA Assembly Specification [1].

1.2. Implementation Metadata

This section describes how SCA Java-based metadata pertaining to Java-based implementation types, .

1.2.1. Service Metadata

1.2.1.1. @Service

The ***@Service annotation*** is used on a Java class to specify the interfaces of the services implemented by the implementation. Service interfaces are typically defined in one of the following ways:

- As a Java interface
- As a Java class
- As a Java interface generated from a Web Services Description Language [4] (WSDL) portType (Java interfaces generated from a WSDL portType are always remotable.

1.2.1.2. Java Semantics of a Remote Service

A remotable service is defined using the @Remotable annotation on the Java interface that defines the service. Remotable services are intended to be used for **coarse grained** services, and the parameters are passed **by-value**.

1.2.1.3. Java Semantics of a Local Service

A local service can only be called by clients that are deployed within the same address space as the component implementing the local service.

A local interface is defined by a Java interface with no @Remotable annotation or is defined by a Java class.

The following snippet shows the Java interface for a local service.

```
package services.hello;

public interface HelloService {

String hello(String message);
}
```

The style of local interfaces is typically ***fine grained*** and intended for ***tightly coupled*** interactions.

The data exchange semantic for calls to local services is ***by-reference***. This means that code must be written with the knowledge that changes made to parameters (other than simple types) by either the client or the provider of the service are visible to the other.

1.2.2. @Reference

Accessing a service using reference injection is done by defining a field, a setter method parameter, or a constructor parameter typed by the service interface and annotated with an ***@Reference*** annotation.

1.2.3. @Property

Implementations can be configured through properties, as defined in the SCA Assembly specification [1]. The ***@Property*** annotation is used to define an SCA property .

1.2.4. Implementation Scopes: @Scope, @Init, @Destroy

Component implementations can either manage their own state or allow the SCA runtime to do so. In the latter case, SCA defines the concept of ***implementation scope,*** which specifies a visibility and lifecycle contract an implementation has with the SCA runtime. Invocations on a service offered by a component will be dispatched by the SCA runtime to an implementation instance according to the semantics of its implementation scope.

Scopes are specified using the @Scope annotation on the implementation class.

This document defines four basic scopes:

- STATELESS
- REQUEST
- CONVERSATION
- COMPOSITE

Java-based implementation types can choose to support any of these scopes, and they may define new scopes specific to their type.

An implementation type may allow component implementations to declare ***lifecycle methods*** that are called when an implementation is instantiated or the scope is expired. ***@Init*** denotes the method to be called upon first use of an instance during the lifetime of the scope (except for composite scoped

implementation marked to eagerly initialize, see Section XXX). ***@Destroy*** specifies the method to be called when the scope ends. Note that only public, no argument methods may be annotated as lifecycle methods.

The following snippet shows a fragment of a service implementation annotated with lifecycle methods.

```
@Init
public void start() {
      ...
}

@Destroy
public void stop() {
      ...
}
```

The following sections specify four standard scopes Java-based implementation types may support.

1.2.4.1. Stateless scope

For stateless components, there is no implied correlation between service requests.

1.2.4.2. Request scope

The lifecycle of request scope extends from the point a request on a remotable interface enters the SCA runtime and a thread processes that request until the thread completes synchronously processing the request. During that time, all service requests will be delegated to the same implementation instance of a request-scoped component.

There are times when a local request scoped service is called without there being a remotable service earlier in the call stack, such as when a local service is called from a non-SCA entity. In these cases, a remote request is always considered to be present, but the lifetime of the request is implementation dependent. For example, a timer event could be treated as a remote request.

1.2.4.3. Composite scope

All service requests are dispatched to the same implementation instance for the lifetime of the containing composite. The lifetime of the containing composite is defined as the time it becomes active in the runtime to the time it is deactivated, either normally or abnormally.

A composite scoped implementation may also specify eager initialization using the @EagerInit annotation. When marked for eager initialization, the composite scoped instance will be created when its containing component is started. If a method is marked with the @Init annotation, it will be called when the instance is created.

1.2.4.4. Conversation scope

A conversation is defined as a series of correlated interactions between a client and a target service. A conversational scope starts when the first service request is dispatched to an implementation instance offering a conversational service. A conversational scope completes after an end operation defined by the service contract is called and completes processing or the conversation expires. A conversation may be long-running and the SCA runtime may choose to passivate implementation instances. If this occurs, the runtime must guarantee implementation instance state is preserved.

Note that in the case where a conversational service is implemented by a Java class marked as conversation scoped, the SCA runtime will transparently handle implementation state. It is also possible for an implementation to manage its own state. For example, a Java class having a stateless (or other) scope could implement a conversational service.

1.3. Interface Metadata

This section describes SCA metadata for Java interfaces.

1.3.1. @Remotable

The @Remotable annotation on a Java interface indicates that the interface is designed to be used for remote communication. Remotable interfaces are intended to be used for **coarse grained** services. Operations parameters and return values are passed **by-value**.

1.3.2. @Conversational

Java service interfaces may be annotated to specify whether their contract is conversational as described in the Assembly Specification [1] by using the @Conversational annotation. A conversational service indicates that requests to the service are correlated in some way

When @Conversational is not specified on a service interface, the service contract is stateless.

1.4. Client API

This section describes how SCA services may be programmatically accessed from components and non-managed code, i.e. code not running as an SCA component. .

1.4.1. Accessing Services from an SCA Component

An SCA component may obtain a service reference through injection or programmatically through the component Context API. Using reference injection is the recommended way to access a service, since it results in code with minimal use of middleware APIs. The ComponentContext API should be used in cases where reference injection is not possible.

1.4.1.1. Using the Component Context API

When a component implementation needs access to a service where the reference to the service is not known at compile time, the reference can be located using the component's ComponentContext.

1.4.2. Accessing Services from non-SCA component implementations

This section describes how Java code not running as an SCA component that is part of an SCA composite accesses SCA services via references.

1.4.2.1. ComponentContext

Non-SCA client code can use the ComponentContext API to perform operations against a component in an SCA domain. How client code obtains a reference to a ComponentContext is runtime specific. The following example demonstrates the use of the component Context API by non-SCA code:

```
ComponentContext context = // obtained through host environment-specific means

HelloService helloService =  context.getService(HelloService.class,"HelloService");
String result = helloService.hello("Hello World!");
```

1.5. Error Handling

Clients calling service methods may experience business exceptions and SCA runtime exceptions.

Business exceptions are thrown by the implementation of the called service method, and are defined as checked exceptions on the interface that types the service.

SCA runtime exceptions are raised by the SCA runtime and signal problems in the management of component execution and in the interaction with remote services. The SCA runtime exceptions ServiceRuntimeException and ServiceUnavailableException, as defined in section 1.5, are used.

1.6. Asynchronous and Conversational Programming

Asynchronous programming of a service is where a client invokes a service and carries on executing without waiting for the service to execute. Typically, the invoked service executes at some later time. Output from the invoked service, if any, must be fed back to the client through a separate mechanism, since no output is available at the point where the service is invoked. This is in contrast to the call-and-return style of synchronous programming, where the invoked service executes and returns any output to the client before the client continues. The SCA asynchronous programming model consists of support for non-blocking method calls, conversational services, and callbacks. Each of these topics is discussed in the following sections.

Conversational services are services where there is an ongoing sequence of interactions between the client and the service provider, which involve some set of state data – in contrast to the simple case of stateless interactions between a client and a provider. Asynchronous services may often involve the use of a conversation, although this is not mandatory.

1.6.1. @OneWay

Nonblocking calls represent the simplest form of asynchronous programming, where the client of the service invokes the service and continues processing immediately, without waiting for the service to execute.

Any method that returns "void" and has no declared exceptions may be marked with an @OneWay annotation. This means that the method is non-blocking and communication with the service provider may use a binding that buffers the requests and sends it at some later time.

SCA does not currently define a mechanism for making non-blocking calls to methods that return values or are declared to throw exceptions. It is recommended that service designers define one-way methods as often as possible, in order to give the greatest degree of binding flexibility to deployers.

1.6.2. Conversational Services

A service may be declared as conversational by marking its Java interface with @Conversational. If a service interface is not marked with @Conversational, it is stateless.

1.6.2.1. ConversationAttributes

A Java-based implementation class may be decorated with ***@ConversationAttributes***, which can be used to specify the expiration rules for conversational implementation instances.

An example of @ ***ConversationAttributes*** is shown below:

```
package com.bigbank;
import org.osoa.sca.annotations.Conversation;
import org.osoa.sca.annotations.ConversationID;

@ConversationAttributes(maxAge="30 days");
public class LoanServiceImpl implements LoanService {

}
```

1.6.2.2. @EndsConversation

A method of a conversational interface may be marked with an @EndsConversation annotation. Once a method marked with @EndsConversation has been called, the conversation between client and service provider is at an end, which implies no further methods may be called on that service within the same conversation. This enables both the client and the service provider to free up resources that were associated with the conversation.

It is also possible to mark a method on a callback interface (described later) with @EndsConversation, in order for the service provider to be the party that chooses to end the conversation.

If a method on a conversational interface is called after the conversation has ended, the ConversationEndedException (which extends ServiceRuntimeException) is thrown. This may also occur if there is a race condition between the client and the service provider calling their respective @EndsConversation methods.

1.6.3. Passing Conversational Services as Parameters

The service reference which represents a single conversation can be passed as a parameter to another service, even if that other service is remote. This may be used in order to allow one component to continue a conversation that had been started by another.

A service provider may also create a service reference for itself that it can pass to other services. A service implementation does this with a call to

```
interface ComponentContext{
    …
    <B> ServiceReference<B> createSelfReference (Class businessInterface);
    <B> ServiceReference<B> createSelfReference (Class businessInterface,
                                                 String serviceName);
}
```

The second variant, which takes an additional ***serviceName*** parameter, must be used if the component implements multiple services.

This capability may be used to support complex callback patterns, such as when a callback is applicable only to a subset of a larger conversation. Simple callback patterns are handled by the built-in callback support described later.

1.6.4. Conversational Client

The client of a conversational service does not need to code in a special way. The client can take advantage of the conversational nature of the interface through the relationship of the different methods in the interface and the data they may share in common. If the service is asynchronous, the client may like to use a feature such as the conversationID to keep track of any state data relating to the conversation.

The developer of the client knows that the service is conversational by introspecting the service contract. The following shows how a client accesses the conversational service described above:

```
    @Reference
    LoanService loanService;
    // Known to be conversational because the interface is marked as
    // conversational

      public void applyForMortgage(Customer customer, HouseInfo houseInfo,
                                   int term)
      {
          LoanApplication loanApp;
          loanApp = createApplication(customer, houseInfo);
          loanService.apply(loanApp);
          loanService.lockCurrentRate(term);
```

```
        }

        public boolean isApproved() {
              return loanService.getLoanStatus().equals("approved");
         }
        public LoanApplication createApplication(Customer customer,
                                                 HouseInfo houseInfo) {
            return …;
         }
```

1.6.5. Conversation Lifetime Summary

Starting conversations

Conversations start on the client side when one of the following occur:

- A @Reference to a conversational service is injected
- A call is made to CompositeContext.getServiceReference

and then a method of the service is called.

Continuing conversations

The client can continue an existing conversation, by:

- Holding the service reference that was created when the conversation started
- Getting the service reference object passed as a parameter from another service, even remotely
- Loading a service reference that had been written to some form of persistent storage

Ending conversations

A conversation ends, and any state associated with the conversation is freed up, when:

- A server operation that has been annotated @EndConveration has been called
- The server calls an @EndsConversation method on the @Callback reference
- The server's conversation lifetime timeout occurs
- The client calls `Conversation.end()`
- Any non-business exception is thrown by a conversational operation

If a method is invoked on a service reference after an @EndsConversation method has been called then a new conversation will automatically be started. If `ServiceReference.getConversationID()` is called after the @EndsConversationmethod is called, but before the next conversation has been started, it will return null.

If a service reference is used after the service provider's conversation timeout has caused the conversation to be ended, then ConversationEndedException will be thrown. In order to use that service reference for a new conversation, its endConversation () method must be called.

1.6.6. Conversations ID

If a protected or public field or setter method is annotated with ***@ConversationID***, then the conversation ID for the conversation is injected onto the field The type of the field is not necessarily String. System

generated conversation IDs are always strings, but application generated conversation IDs may be other complex types.

1.6.6.1. Application Specified Conversation IDs

It is also possible to take advantage of the state management aspects of conversational services while using a client-provided conversation ID. To do this, the client would not use reference injection, but would use the of ServiceReference.setConversationID() API.

The conversation ID that is passed into this method should be an instance of either a String or an object that is serializable into XML. The ID must be unique to the client component over all time. If the client is not an SCA component, then the ID must be globally unique.

Not all conversational service bindings support application-specified conversation IDs or may only support application-specified conversation IDs that are Strings.

1.6.6.2. Accessing Conversation IDs from Clients

Whether the conversation ID is chosen by the client or is generated by the system, the client may access the conversation ID by calling `ServiceReference.getConversationID()`.

If the conversation ID is not application specified, then the `ServiceReference.getConversationID()` method is only guaranteed to return a valid value after the first operation has been invoked, otherwise it returns null.

1.6.7. Callbacks

A callback service is a service that is used for asynchronous communication from a service provider back to its client in contrast to the communication through return values from synchronous operations. Callbacks are used by ***bidirectional services***, which are services that have two interfaces:

- an interface for the provided service
- a callback interface that must be provided by the client

Callbacks may be used for both remotable and local services. Either both interfaces of a bidirectional service must be remotable, or both must be local. It is illegal to mix the two. There are two basic forms of callbacks: stateless callbacks and stateful callbacks.

A callback interface is declared by using the @Callback annotation on a remotable service interface, which takes the Java Class object of the interface as a parameter. The annotation may also be applied to a method or to a field of an implementation, which is used in order to have a callback injected, as explained in the next section.

1.6.7.1. Stateful Callbacks

A stateful callback represents a specific implementation instance of the component that is the client of the service. The interface of a stateful callback should be marked as ***conversational***.

The following example interfaces define an interaction over stateful callback.

```
package somepackage;
import org.osoa.sca.annotations.Callback;
import org.osoa.sca.annotations.Conversational;
import org.osoa.sca.annotations.Remotable;
@Remotable
@Conversational
@Callback(MyServiceCallback.class)
public interface MyService {
```

```
        public void someMethod(String arg);
    }

    @Remotable
    public interface MyServiceCallback {

        public void receiveResult(String result);
    }
```

An implementation of the service in this example could use the @Callback annotation to request that a stateful callback be injected. The following is a fragment of an implementation of the example service. In this example, the request is passed on to some other component, so that the example service acts essentially as an intermediary. Because the service is conversation scoped, the callback will still be available when the backend service sends back its asynchronous response.

```
  @Callback
  protected MyServiceCallback callback;

  @Reference
  protected MyService backendService;

  public void someMethod(String arg) {
        backendService.someMethod(arg);
    }

  public void receiveResult(String result) {
        callback.receiveResult(result);
    }
```

This fragment must come from an implementation that offers two services, one that it offers to it clients (MyService) and one that is used for receiving callbacks from the back end (MyServiceCallback). The client of this service would also implement the methods defined in MyServiceCallback.

```
   private MyService myService;

   @Reference
   public void setMyService(MyService service){
         myService = service;
   }

   public void aClientMethod() {
         ...
         myService.someMethod(arg);
   }
         public void receiveResult(String result) {
        // code to process the result
   }
```

Stateful callbacks support some of the same use cases as are supported by the ability to pass service references as parameters. The primary difference is that stateful callbacks do not require any additional parameters be passed with service operations. This can be a great convenience. If the service has many operations and any of those operations could be the first operation of the conversation, it would be unwieldy to have to take a callback parameter as part of every operation, just in case it is the first operation of the conversation. It is also more natural than requiring the application developers to invoke an explicit operation whose only purpose is to pass the callback object that should be used.

611

612 ***1.6.7.2. Stateless Callbacks***

613 A stateless callback interface is a callback whose interface is not marked as ***conversational***. Unlike
614 stateless services, the client of that uses stateless callbacks will not have callback methods routed to an
615 instance of the client that contains any state that is relevant to the conversation. As such, it is the
616 responsibility of such a client to perform any persistent state management itself. The only information that
617 the client has to work with (other than the parameters of the callback method) is a callback ID object that is
618 passed with requests to the service and is guaranteed to be returned with any callback.

619 The following is a repeat of the client code fragment above, but with the assumption that in this case the
620 MyServiceCallback is stateless. The client in this case needs to set the callback ID before invoking the
621 service and then needs to get the callback ID when the response is received.

622

623
624
```
private ServiceReference<MyService> myService;

    @Reference
    public void setMyService(ServiceReference<MyService> service){
             myService = service;
    }

    public void aClientMethod() {
         String someKey = "1234";
         ...

      myService.setCallbackID(someKey);
         myService.getService().someMethod(arg);
    }
    public void receiveResult(String result) {
         Object key = myService.getCallbackID();
         // Lookup any relevant state based on "key"
         // code to process the result
    }
```
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643

644 Just as with stateful callbacks, a service implementation gets access to the callback object by annotating a
645 field or setter method with the @Callback annotation, such as the following:

646
647
```
@Callback
protected MyServiceCallback callback;
```
648
649

650 The difference for stateless services is that the callback field would not be available if the component is
651 servicing a request for anything other than the original client. So, the technique used in the previous
652 section, where there was a response from the backendService which was forwarded as a callback from
653 MyService would not work because the callback field would be null when the message from the backend
654 system was received.

655

656 ***1.6.7.3. Implementing Multiple Bidirectional Interfaces***

657

658 Since it is possible for a single implementation class to implement multiple services, it is also possible for
659 callbacks to be defined for each of the services that it implements. The service implementation can include
660 an injected field for each of its callbacks. The runtime injects the callback onto the appropriate field based
661 on the type of the callback. The following shows the declaration of two fields, each of which corresponds to
662 a particular service offered by the implementation.

663

```
@Callback
protected MyService1Callback callback1;

@Callback
protected MyService2Callback callback2;
```

If a single callback has a type that is compatible with multiple declared callback fields, then all of them will be set.

1.6.7.4. Accessing Callbacks

In addition to injecting a reference to a callback service, it is also possible to obtain a reference to a Callback instance by annotating a field or method with the @Callback annotation.

A reference implementing the callback service interface may be obtained using `CallableReference.getService()`.

The following fragments come from a service implementation that uses the callback API:

```
@Callback;
protected CallableReference<MyCallback> callback;

 public void someMethod() {


     MyCallback myCallback = callback.getCallback();

     …

     callback.receiveResult(theResult);
 }
```

Alternatively a callback may be retrieved programmatically using the RequestContext API. The snippet below show how to retrieve a callback in a method programmatically:

```
 public void someMethod() {


    MyCallback myCallback = ComponentContext.getRequestContext().getCallback();

     …

    callback.receiveResult(theResult);
 }
```

On the client side, the service that implements the callback can access the callback ID (i.e. reference parameters) that was returned with the callback operation also by accessing the request context, as follows:

```
@Context;
 protected RequestContext requestContext;

 void receiveResult(Object theResult) {
```

```
        Object refParams = requestContext.getServiceReference().getCallbackID();
        …
    }
```

On the client side, the object returned by the `getServiceReference()` method represents the service reference that was used to send the original request. The object returned by `getCallbackID()` represents the identity associated with the callback, which may be a single String or may be an object (as described below in "Customizing the Callback Identity").

1.6.7.5. Customizing the Callback

By default, the client component of a service is assumed to be the callback service for the bidirectional service. However, it is possible to change the callback by using the `ServiceReference.setCallback()` method. The object passed as the callback should implement the interface defined for the callback, including any additional SCA semantics on that interface such as its scope and whether or not it is remotable.

Since a service other than the client can be used as the callback implementation, SCA does not generate a deployment-time error if a client does not implement the callback interface of one of its references. However, if a call is made on such a reference without the `setCallback()` method having been called, then a ***NoRegisteredCallbackException*** will be thrown on the client.

A callback object for a stateful callback interface has the additional requirement that it must be serializable. The SCA runtime may serialize a callback object and persistently store it.

A callback object may be a service reference to another service. In that case, the callback messages go directly to the service that has been set as the callback. If the callback object is not a service reference, then callback messages go to the client and are then routed to the specific instance that has been registered as the callback object. However, if the callback interface has a stateless scope, then the callback object **must** be a service reference.

1.6.7.6. Customizing the Callback Identity

The identity that is used to identify a callback request is, by default, generated by the system. However, it is possible to provide an application specified identity that should be used to identify the callback by calling the `ServiceReference.setCallbackID()` method. This can be used even either stateful or stateless callbacks. The identity will be sent to the service provider, and the binding must guarantee that the service provider will send the ID back when any callback method is invoked.

The callback identity has the same restrictions as the conversation ID. It should either be a string or an object that can be serialized into XML. Bindings determine the particular mechanisms to use for transmission of the identity and these may lead to further restrictions when using a given binding.

1.6.8. Bindings for Conversations and Callbacks

There are potentially many ways of representing the conversation ID for conversational services depending on the type of binding that is used. For example, it may be possible WS-RM sequence ids for the conversation ID if reliable messaging is used in a Web services binding. WS-Eventing uses a different technique (the wse:Identity header). There is also a WS-Context OASIS TC that is creating a general purpose mechanism for exactly this purpose.

SCA's programming model supports conversations, but it leaves up to the binding the means by which the conversation ID is represented on the wire.

1.7. Java API

This section provides a reference for the Java API offered by SCA.

1.7.1. Component Context

The following snippet defines ComponentContext:

```
package org.osoa.sca;

public interface ComponentContext {

    String getURI();

   <B> B getService(Class<B> businessInterface, String referenceName);

   <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
                                               String referenceName);

   <B> ServiceReference<B> createSelfReference(Class<B> businessInterface);

   <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
                                               String serviceName);

   <B> B getProperty(Class<B> type, String propertyName);

    <B, R extends CallableReference<B>> R cast(B target)
                  throws  IllegalArgumentException;

   RequestContext getRequestContext();
   <B> ServiceReference<B> cast(B target) throws IllegalArgumentException;
}
```

- **getURI**() - returns the absolute URI of the component within the SCA domain
- **getService*(Class<B> businessInterface, String referenceName)*** – Returns a proxy for the reference defined by the current component.
- **getServiceReference*(Class<B> businessInterface, String referenceName)*** – Returns a ServiceReference defined by the current component.
- **createSelfReference*(Class<B> businessInterface)*** – Returns a ServiceReference that can be used to invoke this component over the designated service.
- **createSelfReference*(Class<B> businessInterface, String serviceName)*** – Returns a ServiceReference that can be used to invoke this component over the designated service. Service name explicitly declares the service name to invoke

- **getProperty** (***Class<B> type, String propertyName***) - Returns the value of an SCA property defined by this component.
- **getRequestContext()** - Returns the context for the current SCA service request, or null if there is no current request or if the context is unavailable.
- **cast**(***B target***) - Casts a type-safe reference to a CallableReference

A component may access its component context by defining a protected or public field or protected or public setter method typed by org.osoa.sca.ComponentContext and annotated with @Context. To access the target service, the component uses ComponentContext.getService(..).

The following snippet defines the ComponentContext Java interface with its ***getService()*** method.

```
package org.osoa.sca;

public interface ComponentContext {
…

      T getService(Class<T> serviceType, String referenceName);
}
```

The getService() method takes as its input arguments the Java type used to represent the target service on the client and the name of the service reference .It returns an object providing access to the service. The returned object implements the Java interface the service is typed with.

The following shows a sample of a component context definition in a Java class using the @Context annotation.

```
private ComponentContext componentContext;

@Context
public void setContext(ComponentContext context){
      componentContext = context;
}

public void doSomething(){
   HelloWorld service =
      componentContext.getService(HelloWorld.class,”HelloWorldComponent”);
   service.hello(“hello”);
}
```

Similarly, non-SCA client code can use the ComponentContext API to perform operations against a component in an SCA domain. How the non-SCA client code obtains a reference to a ComponentContext is runtime specific.

1.7.2. Request Context

The following snippet shows the RequestContext Java interface:

```
package org.osoa.sca;

import javax.security.auth.Subject;

public interface RequestContext {

    Subject getSecuritySubject();

    String getServiceName();

    <CB> CallbackReference<CB> getCallbackReference();

    <CB> CB getCallback();

    <B> CallableReference<B> getServiceReference();

}
```

The RequestContext Java interface has the following methods:

- **getSecuritySubject()** – Returns the JAAS Subject of the current request
- **getServiceName()** – Returns the name of the service on the Java implementation the request came in on
- **getCallbackReference()** – Returns a callable reference to the callback as specified by the caller
- **getCallback()** – Returns a proxy for the callback as specified by the caller
- **getServiceReference()** – Returns the callable reference that represents the service or callback reference that the request was invoked on. It is illegal for the service implementation to try to call the setCallback() on a returned service reference.

1.7.3. CallableReference

The following snippet defines CallableReference:

```
package org.osoa.sca;

public interface CallableReference<B> {

   B getService();
   Class<B> getBusinessInterface();
   boolean isConversational();
   Conversation getConversation();
   Object getCallbackID();
}
```

The CallableReference Java interface has the following methods:

- **getService()** - Returns a type-safe reference to the target of this reference. The instance returned is guaranteed to implement the business interface for this reference. The value returned is a proxy to the target that implements the business interface associated with this reference.

- **getBusinessInterface()** – Returns the Java class for the business interface associated with this reference.
- **isConversational()** – Returns true if this reference is conversational.
- **getConversation()** – Returns the conversation associated with this reference. Returns null if no conversation is currently active.
- **getCallbackID()** – Returns the callback ID.

1.7.4. ServiceReference

ServiceReferences may be injected using the @Reference annotation on a protected or public field or public setter method taking the type ServiceReference. The detailed description of the usage of these methods is described in the section on Asynchronous Programming in this document.

The following snippet defines ServiceReference:

```
package org.osoa.sca;

public interface ServiceReference<B>  extends CallableReference<B>{

   Object getConversationID();
   void setConversationID(Object conversationId) throws IllegalStateException;
   void setCallbackID(Object callbackID);
   Object getCallback();
   void setCallback(Object callback);
}
```

The ServiceReference Java interface has the methods of CallableReference plus the following:

- **getConversationID()** - Returns the id supplied by the user that will be associated with conversations initiated through this reference.
- **setConversationID(*Object conversationId*)** – Set the id to associate with any conversation started through this reference. If the value supplied is null then the id will be generated by the implementation. Throws an IllegalStateException if a conversation is currently associated with this reference.
- **setCallbackID(*Object callbackID*)** – Sets the callback ID.
- **getCallback()** – Returns the callback object.
- **setCallback(*Object callaback*)** – Sets the callback object.

1.7.5. Conversation

The following snippet defines Conversation:

```
package org.osoa.sca;
```

```
public interface Conversation {
   Object getConversionID();
   void end();
}
```

The ServiceReference Java interface has the following methods:

- **getConversationID()** – Returns the identifier for this conversation. If a user-defined identity had been supplied for this reference then its value will be returned; otherwise the identity generated by the system when the conversation was initiated will be returned.
- **end()** – Ends this conversation.

1.7.6. No Registered Callback Exception

The following snippet shows the NoRegisteredCallbackException.

```
package org.osoa.sca;

public class NoRegsiteredCallbackException extends ServiceRuntimeException {
    ...
}
```

1.7.7. Service Runtime Exception

The following snippet shows the ServiceRuntimeException.

```
package org.osoa.sca;

public class ServiceRuntimeException extends RuntimeException {
    ...
}
```

This exception signals problems in the management of SCA component execution.

1.7.8. Service Unavailable Exception

The following snippet shows the ServiceRuntimeException.

```
package org.osoa.sca;

public class ServiceUnavailableException extends ServiceRuntimeException {
    ...
}
```

This exception signals problems in the interaction with remote services. This extends ServiceRuntimeException. These are exceptions that may be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException that is *not* a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since it most likely requires human intervention

1.7.9. Conversation Ended Exception

The following snippet shows the ConversationEndedException.

```
package org.osoa.sca;
```

```
    public class ConversationEndedException extends ServiceRuntimeException {
        ...
    }
```

1.8. Java Annotations

This section provides definitions of all the Java annotations which apply to SCA.

1.8.1. @AllowsPassByReference

The following snippet shows the @AllowsPassByReference annotation type definition.

```
    package org.osoa.sca.annotations;

    import static java.lang.annotation.ElementType.TYPE;
    import static java.lang.annotation.ElementType.METHOD;
    import static java.lang.annotation.RetentionPolicy.RUNTIME;
    import java.lang.annotation.Retention;
    import java.lang.annotation.Target;

    @Target({TYPE, METHOD})
    @Retention(RUNTIME)
    public @interface AllowsPassByReference {

    }
```

The ***@AllowsPassByReference*** annotation is used on implementations of remotable interfaces to indicate that interactions with the service within the same address space are allowed to use pass by reference data exchange semantics. The implementation promises that its by-value semantics will be maintained even if the parameters and return values are actually passed by-reference. This means that the service will not modify any operation input parameter or return value, even after returning from the operation. Either a whole class implementing a remotable service or an individual remotable service method implementation can be annotated using the @AllowsPassByReference annotation.

@AllowsPassByReference has no attributes

The following snippet shows a sample where @AllowsPassByReference is defined for the implementation of a service method on the Java component implementation class.

```
   @AllowsPassByReference
   public String hello(String message) {
   …
   }
```

1.8.2. @Callback

The following snippet shows the @Callback annotation type definition:

```
    package org.osoa.sca.annotations;
```

```
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE, METHOD, FIELD)
@Retention(RUNTIME)
public @interface  Callback {

    Class<?> value() default Void.class;
}
```

The @Callback annotation type is used to annotate a remotable service interface with a callback interface, which takes the Java Class object of the callback interface as a parameter.

The @Callback annotation has the following attribute:

- ***value*** – the name of a Java class file containing the callback interface

The @Callback annotation may also be used to annotate a method or a field of an SCA implementation class, in order to have a callback injected

The following snippet shows a callback annotation on an interface:

```
@Remotable
@Callback(MyServiceCallback.class)
public interface MyService {

    public void someAsyncMethod(String arg);
}
```

An example use of the @Callback annotation to declare a callback interface follows:

```
package somepackage;
import org.osoa.sca.annotations.Callback;
import org.osoa.sca.annotations.Remotable;
@Remotable
@Callback(MyServiceCallback.class)
public interface MyService {

    public void someMethod(String arg);
}

@Remotable
public interface MyServiceCallback {

    public void receiveResult(String result);
}
```

In this example, the implied component type is:

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0" >

<service name="MyService">
         <interface.java interface="somepackage.MyService"
               callbackInterface="somepackage.MyServiceCallback"/>
</service>
</componentType>
```

1.8.3. @ComponentName

The following snippet shows the @ComponentName annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface  ComponentName {

}
```

The @ComponentName annotation type is used to annotate a Java class field or setter method that is used to inject the component name.

The following snippet shows a component name field definition sample.

```
@ComponentName
private String componentName;


@ComponentName
public void setComponentName(String name){
 //…
}
```

1.8.4. @Conversation

The following snippet shows the @Conversation annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;
```

```
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface  Conversation {
}
```

1.8.5. @Constructor

The following snippet shows the @Constructor annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(CONSTRUCTOR)
@Retention(RUNTIME)
public @interface  Constructor {
   String[] value() default "";
}
```

The @Constructor annotation is used to mark a particular constructor to use when instantiating a Java component implementation.

The @Constructor annotation has the following attribute:

- ***value (optional)*** – identifies the property/reference names that correspond to each of the constructor arguments. The position in the array determines which of the arguments are being named.

1.8.6. @Context

The following snippet shows the @Context annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
```

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Context {

}
```

The @Context annotation type is used to annotate a Java class field or a setter method that is used to inject a composite context for the component. The type of context to be injected is defined by the type of the Java class field or type of the setter method input argument, the type is either ComponentContext or RequestContext.

The @Context annotation has no attributes.

The following snippet shows a ComponentContext field definition sample.

```
@Context
private ComponentContext context;
```

1.8.7. @Conversational

The following snippet shows the @Conversational annotation type definition:

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Conversational {
}
```

The @Conversational annotation is used on a Java interface to denote a conversational service contract.

The @Conversational annotation has no attributes.

1.8.8. @Destroy

The following snippet shows the @Destroy annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface Destroy {
```

```
}
```

The @Destroy annotation type is used to annotate a Java class method that will be called when the scope defined for the local service implemented by the class ends. The method must have a void return value and no arguments. The annotated method must be public.

The @Destroy annotation has no attributes.

The following snippet shows a sample for a destroy method definition.

```
@Destroy
void myDestroyMethod() {
    …
}
```

1.8.9. @EagerInit

The following snippet shows the @EagerInit annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface EagerInit {

}
```

1.8.10. @EndsConversation

The following snippet shows the @EndsConversation annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface EndsConversation {

```

```
}
```

The @EndsConversation annotation type is used to decorate a method on a Java interface that is called to end a conversation.

The @EndsConversation annotation has no attributes.

1.8.11.@Init

The following snippet shows the @Init annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface  Init {


}
```

The @Init annotation type is used to annotate a Java class method that is called when the scope defined for the local service implemented by the class starts. The method must have a void return value and no arguments. The annotated method must be public. The annotated method is called after all property and reference injection is complete.

The @Init annotation has no attributes.

The following snippet shows a sample for a init method definition.

```
@Init
void myInitMethod() {
   …
}
```

1.8.12.@OneWay

The following snippet shows the @OneWay annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
```

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface  OneWay {


}
```

The @OneWay annotation type is used to annotate a Java interface method to indicate that invocations will be dispatched in a non-blocking fashion as described in the section on Asynchronous Programming.

The @OneWay annotation has no attributes.

1.8.13.@Property

The following snippet shows the @Property annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface  Property {

    public String name() default "";
    public boolean required() default false;
}
```

The @Property annotation type is used to annotate a Java class field or a setter method that is used to inject an SCA property value. The type of the property injected, which can be a simple Java type or a complex Java type, is defined by the type of the Java class field or the type of the setter method input argument.

The @Property annotation may be used on protected or public fields and on setter methods or on a constructor method.

Properties may also be injected via public setter methods even when the @Property annotation is not present. However, the @Property annotation must be used in order to inject a property onto a non-public field. In the case where there is no @Property annotation, the name of the property is the same as the name of the field or setter.

Where there is both a setter method and a field for a property, the setter method is used.

The @Property annotation has the following attributes:

- ***name (optional)*** – the name of the property, defaults to the name of the field of the Java class
- ***required (optional)*** – specifies whether injection is required, defaults to false

The following snippet shows a property field definition sample.

```
@Property(name="currency", required=true)
protected String currency;
```

The following snippet shows a property setter sample

```
@Property(name="currency", required=true)
public void setCurrency( String theCurrency );
```

If the property is defined as an array or as a ***java.util.Collection***, then the implied component type has a property with a ***many*** attribute set to true.

The following snippet shows the definition of a configuration property using the @Property annotation for a collection.

```
    ...
    private List<String> helloConfigurationProperty;

    @Property(required=true)
    public void setHelloConfigurationProperty(List<String> property){
            helloConfigurationProperty = property;
    }
    ...
```

1.8.14.@Reference

The following snippet shows the @Reference annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
@Target({METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface  Reference {

    public String name() default "";
    public boolean required() default true;
}
```

The @Reference annotation type is used to annotate a Java class field or a setter method that is used to inject a service that resolves the reference. The interface of the service injected is defined by the type of the Java class field or the type of the setter method input argument.

References may also be injected via public setter methods even when the @Reference annotation is not present. However, the @Reference annotation must be used in order to inject a reference onto a non-public field. In the case where there is no @Reference annotation, the name of the reference is the same as the name of the field or setter.

Where there is both a setter method and a field for a reference, the setter method is used.

The @Reference annotation has the following attributes:

- ***name (optional)*** – the name of the reference, defaults to the name of the field of the Java class
- ***required (optional)*** – whether injection of service or services is required. Defaults to true.

The following snippet shows a reference field definition sample.

```
@Reference(name="stockQuote", required=true)
protected StockQuoteService stockQuote;
```

The following snippet shows a reference setter sample

```
@Reference(name="stockQuote", required=true)
public void setStockQuote( StockQuoteService theSQService );
```

The following fragment from a component implementation shows a sample of a service reference using the @Reference annotation. The name of the reference is “helloService” and its type is HelloService. The clientMethod() calls the “hello” operation of the service referenced by the helloService reference.

```
    private HelloService helloService;

    @Reference(name="helloService", required=true)
    public setHelloService(HelloService service){
            helloService = service;
    }

    public void clientMethod() {
            String result = helloService.hello("Hello World!");
            …
    }
```

The presence of a @Reference annotation is reflected in the componentType information that the runtime generates through reflection on the implementation class. The following snippet shows the component type for the above component implementation fragment.

```
    <?xml version="1.0" encoding="ASCII"?>
    <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">

          <!—Any services offered by the component would be listed here →
          <reference name="helloService" multiplicity=”1..1”>
                <interface.java interface="services.hello.HelloService"/>
          </reference>

    </componentType>
```

If the reference is not an array or collection, then the implied component type has a reference with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference ***required*** attribute – 1..1 applies if required=true.

If the reference is defined as an array or as a ***java.util.Collection***, then the implied component type has a reference with a ***multiplicity*** of either ***1..n or 0..n***, depending on whether the ***required*** attribute of the **@Reference** annotation is set to true or false – 1..n applies if required=true.

The following fragment from a component implementation shows a sample of a service reference definition using the @Reference annotation on a java.util.List. The name of the reference is "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the services referenced by the helloServices reference. In this case, at least one HelloService should be present, so ***required*** is true.

```
@Reference(name="helloService", required=true)
protected List<HelloService> helloServices;

public void clientMethod() {

      …
      HelloService helloService = (HelloService)helloServices.get(index);
      String result = helloService.hello("Hello World!");
      …
}
```

The following snippet shows the XML representation of the component type reflected from for the former component implementation fragment. There is no need to author this component type in this case since it can be reflected from the Java class.

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">

      <!—Any services offered by the component would be listed here →
      <reference name="helloService" multiplicity="1..n">
            <interface.java interface="services.hello.HelloService"/>
      </reference>

</componentType>
```

1.8.15.@Remotable

The following snippet shows the @Remotable annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;


@Target(TYPE)
@Retention(RUNTIME)
public @interface  Remotable {
```

```
}
```

The @Remotable annotation type is used to annotate a Java service interface as remotable. A remotable service can be published externally as a service and must be translatable into WSDL portTypes.

The @Remotable annotation has no attributes.

The following snippet shows the Java interface for a remotable service with its @Remotable annotation.

```
package services.hello;

import org.osoa.sca.annotations.*;

@Remotable
public interface HelloService {

    String hello(String message);
}
```

The style of remotable interfaces is typically ***coarse grained*** and intended for ***loosely coupled*** interactions. Remotable service Interfaces are not allowed to make use of method ***overloading***.

Complex data types exchanged via remotable service interfaces must be compatible with the marshalling technology used by the service binding. For example, if the service is going to be exposed using the standard web service binding, then the parameters must be Service Data Objects (SDOs) 2.0 [2] or JAXB [3] types.

Independent of whether the remotable service is called from outside of the composite that contains it or from another component in the same composite, the data exchange semantics are ***by-value***.

Implementations of remotable services may modify input data during or after an invocation and may modify return data after the invocation. If a remotable service is called locally or remotely, the SCA container is responsible for making sure that no modification of input data or post-invocation modifications to return data are seen by the caller.

The following snippets show a remotable Java service interface.

```
package services.hello;

import org.osoa.sca.annotations.*;

@Remotable
public interface HelloService {

    String hello(String message);
}
```

```
package services.hello;

import org.osoa.sca.annotations.*;

@Service(HelloService.class)
@AllowsPassByReference
public class HelloServiceImpl implements HelloService {

```

```
        public String hello(String message) {
                ...
        }
    }
```

1.8.16.@Scope

The following snippet shows the @Scope annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface  Scope {

    String value() default "STATELESS";
}
```

The @Scope annotation type is used on either a service's interface definition or on a service implementation class itself.

The @Scope annotation has the following attribute:

- ***value*** – the name of the scope.
  The default value is 'STATELESS'. For 'STATELESS' implementations, a different implementation instance may be used to service each request. Implementation instances may be newly created or be drawn from a pool of instances.

The following snippet shows a sample for a scoped service interface definition.

```
package services.shoppingcart;
import org.osoa.sca.annotations.Scope;


@Scope(“CONVERSATION”)
public interface ShoppingCartService {

    void addToCart(Item item);
}
```

1.8.17.@Service

The following snippet shows the @Service annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;

import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface  Service {

    Class<?>[] interfaces() default {};
    Class<?> value() default Void.class;
}
```

The @Service annotation type is used on a component implementation class to specify the SCA services offered by the implementation. The class need not be declared as implementing all of the interfaces implied by the services, but all methods of the service interfaces must be present. A class used as the implementation of a service is not required to have an @Service annotation. If a class has no @Service annotation, then the rules determining which services are offered and what interfaces those services have are determined by the specific implementation type.

The @Service annotation has the following attributes:

- ***interfaces*** – The value is an array of interface or class objects that should be exposed as services by this component.
- ***value*** – A shortcut for the case when the class provides only a single service interface.

  Only one of these attributes should be specified.

A @Service annotation with no attributes is meaningless, it is the same as not having the annotation there at all.

The ***service names*** of the defined services default to the names of the interfaces or class, without the package name.

If a Java implementation needs to realize two services with the same interface, then this is achieved through subclassing of the interface. The subinterface must not add any methods. Both interfaces are listed in the @Service annotation of the Java implementation class.

1.8.18.@ConversationAttributes

The following snippet shows the @ConversationAttributes annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface  ConversationAttributes {

    public String maxIdleTime() default "";
    public String maxAge() default "";
    public boolean singlePrincipal() default false;
}
```

The @ConversationAttributes annotation type is used to define a set of attributes which apply to conversational interfaces of services or references of a Java class. The annotation has the following attributes:

- ***maxIdleTime (optional)*** - The maximum time that can pass between operations within a single conversation. If more time than this passes, then the container may end the conversation.
- ***maxAge (optional)*** - The maximum time that the entire conversation can remain active. If more time than this passes, then the container may end the conversation.
- ***singlePrincipal (optional)*** – If true, only the principal (the user) that started the conversation has authority to continue the conversation. The default value is false.

The two attributes that take a time express the time as a string that starts with an integer, is followed by a space and then one of the following: "seconds", "minutes", "hours", "days" or "years".

Not specifying timeouts means that timeouts are defined by the implementation of the SCA run-time, however it chooses to do so.

The following snippet shows a component name field definition sample.

```
package service.shoppingcart;

import org.osoa.sca.annotations.*

@ConversationAttributes (maxAge="30 days");
public class ShoppingCartServiceImpl implements ShoppingCartService {
    ...
}
```

1.8.19.@ConversationID

The following snippet shows the @ConversationID annotation type definition.

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;

import static java.lang.annotation.ElementType.FIELD;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface  ConversationID {

}
```

The ConversationID annotation type is used to annotate a Java class field or setter method that is used to inject the conversation ID. System generated conversation IDs are always strings, but application generated conversation IDs may be other complex types.

The following snippet shows a conversation ID field definition sample.

```
    @ConversationID
    private String ConversationID;
```

The type of the field is not necessarily String.

1.9. WSDL to Java and Java to WSDL

The SCA Client and Implementation Model for Java applies the ***WSDL to Java*** and ***Java to WSDL*** mapping rules as defined by the JAX-WS specification [4] for generating remotable Java interfaces from WSDL portTypes and vice versa.

For the mapping from Java types to XML schema types SCA supports both the SDO 2.0 [2] mapping and the JAXB [3] mapping.

The JAX-WS mappings are applied with the following restrictions:

- No support for holders

Note: This specification needs more examples and discussion of how JAX-WS's client asynchronous model is used.

2. Policy Annotations for Java

SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which influence how implementations, services and references behave at runtime. The policy facilities are described in the SCA Policy Framework specification [5]. In particular, the facilities include Intents and Policy Sets, where intents express abstract, high-level policy requirements and policy sets express low-level detailed concrete policies.

Policy metadata can be added to SCA assemblies through the means of declarative statements placed into Composite documents and into Component Type documents. These annotations are completely independent of implementation code, allowing policy to be applied during the assembly and deployment phases of application development.

However, it can be useful and more natural to attach policy metadata directly to the code of implementations. This is particularly important where the policies concerned are relied on by the code itself. An example of this from the Security domain is where the implementation code expects to run under a specific security Role and where any service operations invoked on the implementation must be authorized to ensure that the client has the correct rights to use the operations concerned. By annotating the code with appropriate policy metadata, the developer can rest assured that this metadata is not lost or forgotten during the assembly and deployment phases.

The SCA Java Common Annotations specification provides a series of annotations which provide the capability for the developer to attach policy information to Java implementation code. The annotations concerned first provide general facilities for attaching SCA Intents and Policy Sets to Java code. Secondly, there are further specific annotations that deal with particular policy intents for certain policy domains such as Security.

The SCA Java Common Annotations specification supports using the Common Annotation for Java Platform specification (JSR-250) [6]. An implication of adopting the common annotation for Java platform specification is that the SCA Java specification support consistent annotation and Java class inheritance relationships.

2.1. General Intent Annotations

SCA provides the annotation ***@Requires*** for the attachment of any intent to a Java class, to a Java interface or to elements within classes and interfaces such as methods and fields.

The @Requires annotation can attach one or multiple intents in a single statement.

Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI followed by the name of the Intent. The precise form used follows the string representation used by the javax.xml.namespace.QName class, which is as follows:

"{" + Namespace URI + "}" + intentname

Intents may be qualified, in which case the string consists of the base intent name, followed by a ".", followed by the name of the qualifier. There may also be multiple levels of qualification.

This representation is quite verbose, so we expect that reusable String constants will be defined for the namespace part of this string, as well as for each intent that is used by Java code. SCA defines constants for intents such as the following:

```
public static final String SCA_PREFIX="{http://www.osoa.org/xmlns/sca/1.0}";

public static final String CONFIDENTIALITY = SCA_PREFIX + “confidentiality”;

public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + “.message”;
```

Notice that, by convention, qualified intents include the qualifier as part of the name of the constant, separated by an underscore. These intent constants are defined in the file that defines an annotation for the intent (annotations for intents, and the formal definition of these constants, are covered in a following section).

Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

An example of the @Requires annotation with 2 qualified intents (from the Security domain) follows:

```
@Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

This attaches the intents "confidentiality.message" and "integrity.message".

The following is an example of a reference requiring support for confidentiality:

```
package org.osoa.sca.annotation;

import static org.osoa.sca.annotation.Confidentiality.*;

public class Foo {
   @Requires(CONFIDENTIALITY)
   @Reference
   public void setBar(Bar bar)
   …
}
```

Users may also choose to only use constants for the namespace part of the QName, so that they may add new intents without having to define new constants. In that case, this definition would instead look like this:

```
package org.osoa.sca.annotation;

import static org.osoa.sca.Constants.*;

public class Foo {
   @Requires(SCA_PREFIX+”confidentiality”)
   @Reference
   public void setBar(Bar bar)
}
```

The formal syntax for the @Requires annotation follows:

@Requires(“qualifiedIntent” | {“qualifiedIntent” [, “qualifiedIntent”]}

where

qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2

The following shows the formal definition of the @Requires annotation:

```
package org.osoa.sca.annotation;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Inherited;
```

```
@Inherited
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})

public @interface Requires {
   String[] value() default "";
}
```

The SCA_NS constant is defined in the Constants interface:

```
package org.osoa.sca;

public interface Constants {
   public static final String SCA_NS=
       "http://www.osoa.org/xmlns/sca/1.0";
   public static final String SCA_PREFIX = "{"+SCA_NS+"}";
}
```

2.2. Specific Intent Annotations

In addition to the general intent annotation supplied by the @Requires annotation described above, it is also possible to have Java annotations that correspond to specific policy intents. SCA provides a number of these specific intent annotations and it is also possible to create new specific intent annotations for any intent.

The general form of these specific intent annotations is an annotation with a name derived from the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an attribute to the annotation in the form of a string or an array of strings.

For example, the SCA confidentiality intent described in the section on General Intent Annotations using the @Requires(CONFIDENTIALITY) intent can also be specified with the specific @Confidentiality intent annotation. The specific intent annotation for the "integrity" security intent is:

```
@Integrity
```

An example of a qualified specific intent for the "authentication" intent is:

```
@Authentication( {“message”, “transport”} )
```

This annotation attaches the pair of qualified intents: "authentication.message" and "authentication.transport" (the sca: namespace is assumed in this both of these cases – "http://www.osoa.org/xmlns/sca/1.0").

The general form of specific intent annotations is:

@<Intent>[(qualifiers)]

where Intent is an NCName that denotes a particular type of intent.

Intent ::= NCName

qualifiers ::= ”qualifier” | {“qualifier” [, “qualifier”] }

qualifier ::= NCName | NCName/qualifier

2.2.1. How to Create Specific Intent Annotations

SCA identifies annotations that correspond to intents by providing an @Intent annotation which must be used in the definition of an intent annotation.

The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the String form of the QName of the intent. As part of the intent definition, it is good practice (although not required) to also create String constants for the Namespace, the Intent and for Qualified versions of the Intent (if defined). These String constants are then available for use with the @Requires annotation and it should also be possible to use one or more of them as parameters to the @Intent annotation.

Alternatively, the QName of the intent may be specified using separate parameters for the targetNamespace and the localPart for example:

```
@Intent(targetNamespace=SCA_NS, localPart=”confidentiality”).
```

The definition of the @Intent annotation is the following:

```
package org.osoa.sca.annotation;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Inherited;

@Retention(RUNTIME)
@Target(ANNOTATION_TYPE)
public @interface  Intent {
    String value() default "";
    String targetNamespace() default "";
    String localPart() default "";
}
```

When an intent can be qualified, it is good practice for the first attribute of the annotation to be a string (or an array of strings) which holds one or more qualifiers.

In this case, the attribute’s definition should be marked with the @Qualifier annotation. The @Qualifier tells SCA that the value of the attribute should be treated as a qualifier for the intent represented by the whole annotation. If more than one qualifier value is specified in an annotation, it means that multiple qualified forms are required. For example:

```
@Confidentiality({“message”,”transport”})
```

implies that both of the qualified intents ”confidentiality.message" and "confidentiality.transport” are set for the element to which the confidentiality intent is attached.

The following is the definition of the @Qualifier annotation.

```
package org.osoa.sca.annotation;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Inherited;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface  Qualifier {
}
```

Examples of the use of the @Intent and @Qualifier annotations in the definition of specific intent annotations are shown in the section dealing with Security Interaction Policy.

2.3. Application of Intent Annotations

The SCA Intent annotations can be applied to the following Java elements:

- Java class
- Java interface
- Method
- Field

Where multiple intent annotations (general or specific) are applied to the same Java element, they are additive in effect. An example of multiple policy annotations being used together follows:

```
@Authentication
@Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

In this case, the effective intents are "authentication", “confidentiality.message” and “integrity.message”.

If an annotation is specified at both the class/interface level and the method or field level, then the method or field level annotation completely overrides the class level annotation of the same type.

The intent annotation can be applied either to classes or to class methods when adding annotated policy on SCA services. Applying an intent to the setter method in a reference injection approach allows intents to be defined at references.

2.3.1. Inheritance And Annotation

The inheritance rules for annotations are consistent with the common annotation specification, JSR 250.

The following example shows the inheritance relations of intents on classes, operations, and super classes.

```
package services.hello;
import org.osoa.sca.annotations.Remotable;
import org.osoa.sca.annotations.Integrity;
import org.osoa.sca.annotations.Authentication;

@Remotable
@Integrity(“transport”)
@Authentication
public class HelloService {
       @Integrity
       @Authentication(“message”)
       public String hello(String message) {...}

       @Integrity
       @Authentication(“transport”)
       public String helloThere() {...}
}
```

```
package services.hello;
import org.osoa.sca.annotations.Remotable;import
org.osoa.sca.annotations.Confidentiality;
import org.osoa.sca.annotations.Authentication;

@Remotable
@Confidentiality(“message”)
public class HelloChildService extends HelloService {
      @Confidentiality(“transport”)
      public String hello(String message) {...}
      @Authentication
      String helloWorld(){...}
}
```

Example 2a. Usage example of annotated policy and inheritance.

The effective intent annotation on the helloWorld method is Integrity(“transport”), @Authentication, and @Confidentiality(“message”).

The effective intent annotation on the hello method of the HelloChildService is @Integrity(“transport”), @Authentication, and @Confidentiality(“transport”),

The effective intent annotation on the helloThere method of the HelloChildService is @Integrity and @Authentication(“transport”), the same as in HelloService class.

The effective intent annotation on the hello method of the HelloService is @Integrity and @Authentication(“message”)

The listing below contains the equivalent declarative security interaction policy of the HelloService and HelloChildService implementation corresponding to the Java interfaces and classes shown in Example 2a.

```
<?xml version="1.0" encoding="ASCII"?>

<composite  xmlns="http://www.osoa.org/xmlns/sca/1.0"
                  name="HelloServiceComposite" >
      <service name=”HelloService” requires=”integrity/transport
            authentication”>
            …
      </service>
      <service name=”HelloChildService” requires=”integrity/transport
            authentication confidentiality/message”>
      …
      </service>
      ...

      <component name="HelloServiceComponent">*
            <implementation.java class="services.hello.HelloService"/>
                  <operation name=”hello” requires=”integrity
                        authentication/message”/>
                  <operation name=”helloThere” requires=”integrity
                        authentication/transport”/>
      </component>
      <component name="HelloChildServiceComponent">*
            <implementation.java class="services.hello.HelloChildService" />
```

```
                <operation name=”hello” requires=”confidentiality/transport”/>
                <operation name=”helloThere” requires=” integrity/transport
                      authentication”/>
                <operation name=helloWorld” requires=”authentication”/>
            </component>

            ...

        </composite>
```

Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.

2.4. Relationship of Declarative And Annotated Intents

Annotated intents on a Java class cannot be overridden by declarative intents either in a composite document which uses the class as an implementation or by statements in a component Type document associated with the class. This rule follows the general rule for intents that they represent fundamental requirements of an implementation.

An unqualified version of an intent expressed through an annotation in the Java class may be qualified by a declarative intent in a using composite document.

2.5. Policy Set Annotations

The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for example, a concrete policy is the specific encryption algorithm to use when encrypting messages when using a specific communication protocol to link a reference to a service).

Policy Sets can be applied directly to Java implementations using the ***@PolicySets*** annotation. The PolicySets annotation either takes the QName of a single policy set as a string or the name of two or more policy sets as an array of strings:

```
@PolicySets( “<policy set QName>” |
             { “<policy set QName>” [, “<policy set QName>”] })
```

As for intents, PolicySet names are QNames – in the form of “{Namespace-URI}localPart”.

An example of the @PolicySets annotation:

```
@Reference(name="helloService", required=true)
@PolicySets({ MY_NS + “WS_Encryption_Policy",
              MY_NS + "WS_Authentication_Policy" })
public setHelloService(HelloService service){
      . . .
}
```

In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both using the namespace defined for the constant MY_NS.

PolicySets must satisfy intents expressed for the implementation when both are present, according to the rules defined in the Policy Framework specification [5].

The SCA Policy Set annotation can be applied to the following Java elements:

- Java class
- Java interface

- Method
- Field

2.6. Security Policy Annotations

This section introduces annotations for SCA's security intents, as defined in the SCA Policy Framework specification [5].

2.6.1. Security Interaction Policy

The following interaction policy Intents and qualifiers are defined for Security Policy, which apply to the operation of services and references of an implementation:

- @Integrity
- @Confidentiality
- @Authentication

All three of these intents have the same pair of Qualifiers:

- message
- transport

The following snippets shows the @Integrity, @Confidentiality and @Authentication annotation type definitions:

```
package org.osoa.sca.annotation;

import java.lang.annotation.*;
import static org.osoa.sca.Constants.SCA_NS;

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,ElementType.METHOD,
                          ElementType.FIELD ,ElementType.PARAMETER})
@Intent(Integrity.INTEGRITY)
public @interface  Integrity {
    public static final String INTEGRITY = SCA_NS+"integrity";
    public static final String INTEGRITY_MESSAGE = INTEGRITY+".message";
    public static final String INTEGRITY_TRANSPORT = INTEGRITY+".transport";
    @Qualifier
   String[] value() default "";
}
```

```
package org.osoa.sca.annotation;

import java.lang.annotation.*;
import static org.osoa.sca.Constants.SCA_NS;

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,ElementType.METHOD,
                        ElementType.FIELD ,ElementType.PARAMETER})
@Intent(Confidentiality.CONFIDENTIALITY)
public @interface  Confidentiality {
   public static final String CONFIDENTIALITY = SCA_NS+"confidentiality";
   public static final String CONFIDENTIALITY_MESSAGE =
```

```
2102              CONFIDENTIALITY+".message";
2103           public static final String CONFIDENTIALITY_TRANSPORT =
2104              CONFIDENTIALITY+".transport";
2105           @Qualifier
2106           String[] value() default "";
2107        }
2108
2109
2110        package org.osoa.sca.annotation;
2111
2112        import java.lang.annotation.*;
2113        import static org.osoa.sca.Constants.SCA_NS;
2114
2115        @Inherited
2116        @Retention(RetentionPolicy.RUNTIME)
2117        @Target({ElementType.TYPE,ElementType.METHOD,
2118                                 ElementType.FIELD ,ElementType.PARAMETER})
2119        @Intent(Authentication.AUTHENTICATION)
2120        public @interface  Authentication {
2121            public static final String AUTHENTICATION = SCA_NS+"authentication";
2122            public static final String AUTHENTICATION_MESSAGE =
2123              AUTHENTICATION+".message";
2124            public static final String AUTHENTICATION_TRANSPORT =
2125              AUTHENTICATION+".transport";
2126            @Qualifier
2127           String[] value() default "";
2128        }
2129
2130
```

2131 The following example shows an example of applying an intent to the setter method used to inject a
2132 reference. Accessing the hello operation of the referenced HelloService requires both "integrity.message"
2133 and "authentication.message" intents to be honored.

2134

```
2135        //Interface for HelloService
2136        public interface service.hello.HelloService {
2137              String hello(String helloMsg);
2138        }
2139
2140        // Interface for ClientService
2141        public interface service.client.ClientService {
2142              public void clientMethod();
2143        }
2144
2145        // Implementation class for ClientService
2146        package services.client;
2147
2148        import services.hello.HelloService;
2149
2150        import org.osoa.sca.annotations.*;
2151
2152        @Service(ClientService.class)
2153        public class ClientServiceImpl implements ClientService {
2154
2155
2156              private HelloService helloService;
2157
2158              @Reference(name="helloService", required=true)
```

```
        @Integrity(“message”)
        @Authentication(“message”)
        public void setHelloService(HelloService service){
              helloService = service;
        }

        public void clientMethod() {
              String result = helloService.hello("Hello World!");
              …
        }
    }
```

Example 1. Usage of annotated intents on a reference.

2.6.2. Security Implementation Policy

SCA defines a number of security policy annotations that apply as policies to implementations themselves. These annotations mostly have to do with authorization and security identity. The following authorization and security identity annotations (as defined in JSR 250) are supported:

- RunAs

  Takes as a parameter a string which is the name of a Security role.
  eg. @RunAs("Manager")
  Code marked with this annotation will execute with the Security permissions of the identified role.

- RolesAllowed

  Takes as a parameter a single string or an array of strings which represent one or more role names. When present, the implementation can only be accessed by principals whose role corresponds to one of the role names listed in the @roles attribute. How role names are mapped to security principals is implementation dependent (SCA does not define this).
  eg. @RolesAllowed({"Manager", "Employee"})

- PermitAll

  No parameters. When present, grants access to all roles.

- DenyAll

  No parameters. When present, denies access to all roles.

- DeclareRoles

  Takes as a parameter a string or an array of strings which identify one or more role names that form the set of roles used by the implementation.
  eg. @DeclareRoles({"Manager", "Employee", "Customer"})

(all these are declared in the Java package javax.annotation.security)

For a full explanation of these intents, see the Policy Framework specification [5].

2.6.2.1. Annotated Implementation Policy Example

The following is an example showing annotated security implementation policy:

```
    package services.account;
```

```
2208        @Remotable
2209        public interface AccountService{
2210              public AccountReport getAccountReport(String customerID);
2211        }
2212
```

2213 The following is a full listing of the AccountServiceImpl class, showing the Service it implements, plus the
2214 service references it makes and the settable properties that it has, along with a set of implementation policy
2215 annotations:

2216

```
2217    package services.account;
2218    import java.util.List;
2219    import commonj.sdo.DataFactory;
2220    import org.osoa.sca.annotations.Property;
2221    import org.osoa.sca.annotations.Reference;
2222    import org.osoa.sca.annotations.RolesAllowed;
2223    import org.osoa.sca.annotations.RunAs;
2224    import org.osoa.sca.annotations.PermitAll;
2225    import services.accountdata.AccountDataService;
2226    import services.accountdata.CheckingAccount;
2227    import services.accountdata.SavingsAccount;
2228    import services.accountdata.StockAccount;
2229    import services.stockquote.StockQuoteService;
2230    @RolesAllowed(“customers”)
2231    @RunAs(“accountants” )
2232    public class AccountServiceImpl implements AccountService {
2233
2234        @Property
2235        protected String currency = "USD";
2236
2237        @Reference
2238        protected AccountDataService accountDataService;
2239        @Reference
2240        protected StockQuoteService stockQuoteService;
2241
2242        @RolesAllowed({“customers”, “accountants”})
2243        public AccountReport getAccountReport(String customerID) {
2244
2245         DataFactory dataFactory = DataFactory.INSTANCE;
2246         AccountReport accountReport =
2247              (AccountReport)dataFactory.create(AccountReport.class);
2248         List accountSummaries = accountReport.getAccountSummaries();
2249
2250         CheckingAccount checkingAccount =
2251              accountDataService.getCheckingAccount(customerID);
2252         AccountSummary checkingAccountSummary =
2253              (AccountSummary)dataFactory.create(AccountSummary.class);
2254         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
2255         checkingAccountSummary.setAccountType("checking");
2256         checkingAccountSummary.setBalance(fromUSDollarToCurrency
2257              (checkingAccount.getBalance()));
2258         accountSummaries.add(checkingAccountSummary);
2259
2260         SavingsAccount savingsAccount =
2261              accountDataService.getSavingsAccount(customerID);
2262         AccountSummary savingsAccountSummary =
2263              (AccountSummary)dataFactory.create(AccountSummary.class);
2264         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
```

```
        savingsAccountSummary.setAccountType("savings");
        savingsAccountSummary.setBalance(fromUSDollarToCurrency
             (savingsAccount.getBalance()));
        accountSummaries.add(savingsAccountSummary);

        StockAccount stockAccount = accountDataService.getStockAccount(customerID);
        AccountSummary stockAccountSummary =
             (AccountSummary)dataFactory.create(AccountSummary.class);
        stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
        stockAccountSummary.setAccountType("stock");
        float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
                        stockAccount.getQuantity();
        stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
        accountSummaries.add(stockAccountSummary);

        return accountReport;
       }

       @PermitAll
       public float fromUSDollarToCurrency(float value){

        if (currency.equals("USD")) return value; else
        if (currency.equals("EURO")) return value * 0.8f; else
        return 0.0f;
       }
    }
```

Example 3. Usage of annotated security implementation policy for the java language.

In this example, the implementation class as a whole is marked:

- @RolesAllowed(“customers”) - indicating that customers have access to the implementation as a whole
- @RunAs(“accountants”) – indicating that the code in the implementation runs with the permissions of accountants

The getAccountReport(..) method is marked with @RolesAllowed({“customers”, “accountants”}), which indicates that this method can be called by both customers and accountants.

The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method can be called by any role.

3. Appendix

3.1. References

[1] SCA Assembly Specification
http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf

[2] SDO 2.0 Specification
http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf

[3] JAXB Specification
http://www.jcp.org/en/jsr/detail?id=31

[4] WSDL Specification
WSDL 1.1: http://www.w3.org/TR/wsdl
WSDL 2.0: http://www.w3.org/TR/wsdl20/

[6] Common Annotation for Java Platform specification (JSR-250)
http://www.jcp.org/en/jsr/detail?id=250