



Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1

**Committee Specification Draft 06 /
Public Review Draft 04**

15 August 2011

Specification URIs

This version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd04.pdf> (Authoritative)
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd04.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd04.doc>

Previous version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd03.pdf> (Authoritative)
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd03.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd03.doc>

Latest version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.pdf> (Authoritative)
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.doc>

Technical Committee:

[OASIS Service Component Architecture / J \(SCA-J\) TC](#)

Chairs:

David Booz (booz@us.ibm.com), IBM
Anish Karmarkar (Anish.Karmarkar@oracle.com), Oracle

Editors:

David Booz (booz@us.ibm.com), IBM
Mike Edwards (mike_edwards@uk.ibm.com), IBM
Anish Karmarkar (Anish.Karmarkar@oracle.com), Oracle

Additional artifacts:

This prose specification is one component of a Work Product which also includes:

- Compiled Java API:
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd04/sca-j-caa-apis-1.1-csprd04.jar>
- Java artifacts:
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd04/sca-j-caa-sources-1.1-csprd04.zip>
- Downloadable Javadoc:
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd04/sca-j-caa-javadoc-1.1-csprd04.zip>
- Hosted Javadoc:
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd04/javadoc/index.html>

Related work:

This specification replaces or supersedes:

- Service Component Architecture Java Common Annotations and APIs Specification Version 1.00. March 21 2007.
http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf?version=1

This specification is related to:

- *Service Component Architecture Assembly Model Specification Version 1.1*. Latest version.
<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.html>
- *SCA Policy Framework Version 1.1*. Latest version.
<http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1.html>

Declared XML namespaces:

- <http://docs.oasis-open.org/ns/opencsa/sca/200912>

Abstract:

The SCA-J Common Annotations and APIs Specification defines a Java syntax for programming concepts defined in the SCA Assembly Model Specification. It specifies a set of APIs and annotations that can be used by Java-based artifacts described by other SCA specifications such as the POJO Component Implementation Specification [JAVA_CI].

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous services
4. Metadata for callbacks
5. Definitions of standard component implementation scopes
6. Java to WSDL and WSDL to Java mappings
7. Security policy annotations

Note that other Java-based SCA specifications can choose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

Status:

This document was last revised or approved by the OASIS Service Component Architecture / J (SCA-J) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at <http://www.oasis-open.org/committees/sca-j/>.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (<http://www.oasis-open.org/committees/sca-j/ipr.php>).

Citation format:

When referencing this specification the following citation format should be used:

[SCA-JavaCAA-v1.1]

Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1. 15 August 2011. OASIS Committee Specification Draft 06 / Public Review Draft 04.
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd04.html>

Notices

Copyright © OASIS Open 2011. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](#) may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of [OASIS](#), the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see <http://www.oasis-open.org/who/trademark.php> for above guidance.

Table of Contents

1	Introduction	7
1.1	Terminology	7
1.2	Normative References	7
1.3	Non-Normative References	8
1.4	Testcases.....	8
2	Implementation Metadata	9
2.1	Service Metadata	9
2.1.1	@Service	9
2.1.2	Java Semantics of a Remotable Service	9
2.1.3	Java Semantics of a Local Service	9
2.1.4	@Reference	10
2.1.5	@Property	10
2.2	Implementation Scopes: @Scope, @Init, @Destroy	10
2.2.1	Stateless Scope	10
2.2.2	Composite Scope	11
2.3	@AllowsPassByReference	11
2.3.1	Marking Services as “allows pass by reference”	12
2.3.2	Marking References as “allows pass by reference”	12
2.3.3	Applying “allows pass by reference” to Service Proxies	12
2.3.4	Using “allows pass by reference” to Optimize Remotable Calls	13
3	Interface.....	14
3.1	Java Interface Element – <interface.java>	14
3.2	@Remotable.....	15
3.3	@Callback	15
3.4	@AsyncInvocation	15
3.5	SCA Java Annotations for Interface Classes.....	16
3.6	Compatibility of Java Interfaces.....	16
4	SCA Component Implementation Lifecycle.....	17
4.1	Overview of SCA Component Implementation Lifecycle	17
4.2	SCA Component Implementation Lifecycle State Diagram	17
4.2.1	Constructing State	18
4.2.2	Injecting State	18
4.2.3	Initializing State	19
4.2.4	Running State.....	19
4.2.5	Destroying State	19
4.2.6	Terminated State	20
5	Client API.....	21
5.1	Accessing Services from an SCA Component	21
5.1.1	Using the Component Context API	21
5.2	Accessing Services from non-SCA Component Implementations	21
5.2.1	SCAClientFactory Interface and Related Classes	21
6	Error Handling	23
7	Asynchronous Programming	24

7.1	@OneWay	24
7.2	Callbacks	24
7.2.1	Using Callbacks.....	24
7.2.2	Callback Instance Management	26
7.2.3	Callback Injection	26
7.2.4	Implementing Multiple Bidirectional Interfaces	26
7.2.5	Accessing Callbacks	27
7.3	Asynchronous handling of Long Running Service Operations	28
7.4	SCA Asynchronous Service Interface	28
8	Policy Annotations for Java	31
8.1	General Intent Annotations	31
8.2	Specific Intent Annotations	33
8.2.1	How to Create Specific Intent Annotations.....	34
8.3	Application of Intent Annotations	34
8.3.1	Intent Annotation Examples	35
8.3.2	Inheritance and Annotation	37
8.4	Relationship of Declarative and Annotated Intents	38
8.5	Policy Set Annotations.....	38
8.6	Security Policy Annotations	39
8.7	Transaction Policy Annotations	40
9	Java API	42
9.1	Component Context.....	42
9.2	Request Context	47
9.3	ServiceReference Interface	49
9.4	ResponseDispatch interface.....	50
9.5	ServiceRuntimeException.....	51
9.6	ServiceUnavailableException	52
9.7	InvalidServiceException.....	52
9.8	Constants.....	52
9.9	SCAClientFactory Class	53
9.10	SCAClientFactoryFinder Interface	56
9.11	SCAClientFactoryFinderImpl Class	57
9.12	NoSuchDomainException.....	58
9.13	NoSuchServiceException	58
10	Java Annotations	59
10.1	@AllowsPassByReference.....	59
10.2	@AsyncFault	60
10.3	@AsyncInvocation	61
10.4	@Authentication	61
10.5	@Authorization	62
10.6	@Callback	62
10.7	@ComponentName	64
10.8	@Confidentiality.....	65
10.9	@Constructor.....	65
10.10	@Context.....	66

10.11	@Destroy	67
10.12	@EagerInit	67
10.13	@Init	68
10.14	@Integrity	68
10.15	@Intent	69
10.16	@ManagedSharedTransaction	70
10.17	@ManagedTransaction	70
10.18	@MutualAuthentication	71
10.19	@NoManagedTransaction	72
10.20	@OneWay	72
10.21	@PolicySets	73
10.22	@Property	74
10.23	@Qualifier	75
10.24	@Reference	76
10.24.1	Reinjection	78
10.25	@Remotable	80
10.26	@Requires	82
10.27	@Scope	82
10.28	@Service	83
11	WSDL to Java and Java to WSDL	85
11.1	JAX-WS Annotations and SCA Interfaces	85
11.2	JAX-WS Client Asynchronous API for a Synchronous Service	91
11.3	Treatment of SCA Asynchronous Service API	92
12	Conformance	93
12.1	SCA Java XML Document	93
12.2	SCA Java Class	93
12.3	SCA Runtime	93
Appendix A.	XML Schema: sca-interface-java-1.1.xsd	94
Appendix B.	Java Classes and Interfaces	95
B.1	SCAClient Classes and Interfaces	95
B.1.1	SCAClientFactory Class	95
B.1.2	SCAClientFactoryFinder interface	97
B.1.3	SCAClientFactoryFinderImpl class	98
B.1.4	SCAClient Classes and Interfaces - what does a vendor need to do?	103
Appendix C.	Conformance Items	104
Appendix D.	Acknowledgements	119
Appendix E.	Revision History	121

1 Introduction

The SCA-J Common Annotations and APIs specification defines a Java syntax for programming concepts defined in the SCA Assembly Model Specification [ASSEMBLY]. It specifies a set of APIs and annotations that can be used by SCA Java-based specifications.

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous services
4. Metadata for callbacks
5. Definitions of standard component implementation scopes
6. Java to WSDL and WSDL to Java mappings
7. Security policy annotations

The goal of defining the annotations and APIs in this specification is to promote consistency and reduce duplication across the various SCA Java-based specifications. The annotations and APIs defined in this specification are designed to be used by other SCA Java-based specifications in either a partial or complete fashion.

1.1 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

1.2 Normative References

- | | |
|-------------|--|
| [RFC2119] | S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> ,
http://www.ietf.org/rfc/rfc2119.txt , IETF RFC 2119, March 1997. |
| [ASSEMBLY] | OASIS Committee Specification Draft 08, <i>SCA Assembly Model Specification Version 1.1</i> , May 2011.
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec-v1.1-csd08.pdf |
| [SDO] | OASIS Committee Draft 02, <i>Service Data Objects Specification Version 3.0</i> , November 2009.
http://www.oasis-open.org/committees/download.php/35313/sdo-3.0-cd02.zip |
| [JAX-B] | JAXB 2.1 Specification, http://www.jcp.org/en/jsr/detail?id=222 |
| [WSDL] | WSDL Specification, WSDL 1.1: http://www.w3.org/TR/wsdl , |
| [POLICY] | OASIS Committee Specification Draft 05, <i>SCA Policy Framework Version 1.1</i> , July 2011.
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-spec-v1.1-csd05.pdf |
| [JSR-250] | Common Annotations for the Java Platform specification (JSR-250),
http://www.jcp.org/en/jsr/detail?id=250 |
| [JAX-WS] | JAX-WS 2.1 Specification (JSR-224), http://www.jcp.org/en/jsr/detail?id=224 |
| [JAVABEANS] | JavaBeans 1.01 Specification,
http://java.sun.com/javase/technologies/desktop/javabeans/api/ |
| [JAAS] | Java Authentication and Authorization Service Reference Guide
http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html |

1.3 Non-Normative References

- [EBNF-Syntax]** Extended BNF syntax format used for formal grammar of constructs
<http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation>
- [JAVA_CI]** OASIS Committee Specification Draft 04, *SCA POJO Component Implementation Specification Version 1.1*, August 2011.
<http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csd04.pdf>
- [CAA_Testcases]** OASIS Committee Specification Draft 02, TestCases for the SCA-J Common Annotations and APIs Version 1.1 Specification, August 2011.
<http://docs.oasis-open.org/opencsa/sca-j/sca-j-cao-testcases-v1.1-csd02.pdf>

1.4 Testcases

The TestCases for the SCA-J Common Annotations and APIs Version 1.1 Specification [CAA_Testcases] defines the TestCases for the SCA-J Common Annotations and API specification. The TestCases represent a series of tests that SCA runtimes are expected to pass in order to claim conformance to the requirements of the SCA-J Common Annotations and API specification.

2 Implementation Metadata

This section describes SCA Java-based metadata, which applies to Java-based implementation types.

2.1 Service Metadata

2.1.1 @Service

The **@Service annotation** is used on a Java class to specify the interfaces of the services provided by the implementation. Service interfaces are 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 [WSDL] (WSDL) portType (Java interfaces generated from WSDL portTypes are always **remotable**)

2.1.2 Java Semantics of a Remotable Service

A **remotable service** is defined using the @Remotable annotation on the Java interface or Java class that defines the service, or on a service reference. Remotable services are intended to be used for **coarse grained** services, and the parameters are passed **by-value**. **Remotable Services MUST NOT make use of method overloading.** [JCA20001]

Snippet 2-1 shows an example of a Java interface for a remotable service:

```
package services.hello;
@Remotable
public interface HelloService {
    String hello(String message);
}
```

Snippet 2-1: Remotable Java Interface

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 or a Java class with no @Remotable annotation.

Snippet 2-2 shows an example of a Java interface for a local service:

```
package services.hello;
public interface HelloService {
    String hello(String message);
}
```

Snippet 2-2: Local Java Interface

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

The data exchange semantic for calls to local services is **by-reference**. This means that implementation code which uses a local interface needs to 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.

2.1.4 @Reference

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

2.1.5 @Property

Implementations can be configured with data values through the use of properties, as defined in [the SCA Assembly Model specification \[ASSEMBLY\]](#). The **@Property** annotation is used to define an SCA property.

2.2 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 specification defines two scopes:

- STATELESS
- COMPOSITE

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

An implementation type can allow component implementations to declare **lifecycle methods** that are called when an implementation is instantiated or the scope is expired.

@Init denotes a method called upon first use of an instance during the lifetime of the scope (except for composite scoped implementation marked to eagerly initialize, see [section Composite Scope](#)).

@Destroy specifies a method called when the scope ends.

Note that only no-argument methods with a void return type can be annotated as lifecycle methods.

Snippet 2-3 is an example showing a fragment of a service implementation annotated with lifecycle methods:

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

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

Snippet 2-3: Java Component Implementation with Lifecycle Methods

The following sections specify the two standard scopes which a Java-based implementation type can support.

2.2.1 Stateless Scope

For stateless scope components, there is no implied correlation between implementation instances used to dispatch service requests.

The concurrency model for the stateless scope is single threaded. This means that **the SCA runtime MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one**

thread at any one time. [JCA20002] In addition, within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of one business method. [JCA20003] Note that the SCA lifecycle might not correspond to the Java object lifecycle due to runtime techniques such as pooling.

2.2.2 Composite Scope

The meaning of "composite scope" is defined in relation to the composite containing the component.

It is important to distinguish between different uses of a composite, where these uses affect the numbers of instances of components within the composite. There are 2 cases:

- a) Where the composite containing the component using the Java implementation is the SCA Domain (i.e. a deployment composite declares the component using the implementation)
- b) Where the composite containing the component using the Java implementation is itself used as the implementation of a higher level component (any level of nesting is possible, but the component is NOT at the Domain level)

Where an implementation is used by a "domain level component", and the implementation is marked "Composite" scope, the SCA runtime MUST ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation. [JCA20004]

Where an implementation is marked "Composite" scope and it is used by a component that is nested inside a composite that is used as the implementation of a higher level component, the SCA runtime MUST ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation. There can be multiple instances of the higher level component, each running on different nodes in a distributed SCA runtime. [JCA20008]

The SCA runtime can exploit shared state technology in combination with other well known high availability techniques to provide the appearance of a single runtime instance for consumers of composite scoped components.

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.

When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started. [JCA20005] If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created. [JCA20006]

The concurrency model for the composite scope is multi-threaded. This means that the SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and the SCA runtime MUST NOT perform any synchronization. [JCA20007]

2.3 @AllowsPassByReference

Calls to remotable services (see [section "Java Semantics of a Remotable Service"](#)) have by-value semantics. This means that input parameters passed to the service can be modified by the service without these modifications being visible to the client. Similarly, the return value or exception from the service can be modified by the client without these modifications being visible to the service implementation. For remote calls (either cross-machine or cross-process), these semantics are a consequence of marshalling input parameters, return values and exceptions "on the wire" and unmarshalling them "off the wire" which results in physical copies being made. For local method calls within the same JVM, Java language calling semantics are by-reference and therefore do not provide the correct by-value semantics for SCA remotable interfaces. To compensate for this, the SCA runtime can intervene in these calls to provide by-value semantics by making copies of any mutable objects passed.

The cost of such copying can be very high relative to the cost of making a local call, especially if the data being passed is large. Also, in many cases this copying is not needed if the implementation observes certain conventions for how input parameters, return values and exceptions are used. The @AllowsPassByReference annotation allows service method implementations and client references to be marked as "allows pass by reference" to indicate that they use input parameters, return values and

exceptions in a manner that allows the SCA runtime to avoid the cost of copying mutable objects when a remotable service is called locally within the same JVM.

2.3.1 Marking Services as “allows pass by reference”

Marking a service method implementation as “allows pass by reference” asserts that the method implementation observes the following restrictions:

- Method execution will not modify any input parameter before the method returns.
- The service implementation will not retain a reference to any mutable input parameter, mutable return value or mutable exception after the method returns.
- The method will observe “allows pass by reference” client semantics (see section 2.3.2) for any callbacks that it makes.

See [section “@AllowsPassByReference”](#) for details of how the @AllowsPassByReference annotation is used to mark a service method implementation as “allows pass by reference”.

2.3.2 Marking References as “allows pass by reference”

Marking a client reference as “allows pass by reference” asserts that method calls through the reference observe the following restrictions:

- The client implementation will not modify any of the method’s input parameters before the method returns. Such modifications might occur in callbacks or separate client threads.
- If the method is one-way, the client implementation will not modify any of the method’s input parameters at any time after calling the method. This is because one-way method calls return immediately without waiting for the service method to complete.

See [section “Applying “allows pass by reference” to Service Proxies”](#) for details of how the @AllowsPassByReference annotation is used to mark a client reference as “allows pass by reference”.

2.3.3 Applying “allows pass by reference” to Service Proxies

Service method calls are made by clients using service proxies, which can be obtained by injection into client references or by making API calls. A service proxy is marked as “allows pass by reference” if and only if any of the following applies:

- It is injected into a reference or callback reference that is marked “allows pass by reference”.
- It is obtained by calling `ComponentContext.getService()` or `ComponentContext.getServices()` with the name of a reference that is marked “allows pass by reference”.
- It is obtained by calling `RequestContext.getCallback()` from a service implementation that is marked “allows pass by reference”.
- It is obtained by calling `ServiceReference.getService()` on a service reference that is marked “allows pass by reference”.

A service reference for a remotable service call is marked “allows pass by reference” if and only if any of the following applies:

- It is injected into a reference or callback reference that is marked “allows pass by reference”.
- It is obtained by calling `ComponentContext.getServiceReference()` or `ComponentContext.getServiceReferences()` with the name of a reference that is marked “allows pass by reference”.
- It is obtained by calling `RequestContext.getCallbackReference()` from a service implementation that is marked “allows pass by reference”.
- It is obtained by calling `ComponentContext.cast()` on a proxy that is marked “allows pass by reference”.

2.3.4 Using “allows pass by reference” to Optimize Remotable Calls

The SCA runtime **MUST** use by-value semantics when passing input parameters, return values and exceptions on calls to remotable services within the same JVM if the service method implementation is not marked “allows pass by reference” or the service proxy used by the client is not marked “allows pass by reference”. [JCA20010]

The SCA runtime can use by-reference semantics when passing input parameters, return values or exceptions on calls to remotable services within the same JVM if both the service method implementation and the service proxy used by the client are marked “allows pass by reference”.

3 Interface

This section describes the SCA Java interface element and the SCA metadata for Java interfaces.

3.1 Java Interface Element – <interface.java>

The Java interface element is used in SCA Documents in places where an interface is declared in terms of a Java interface class. The Java interface element identifies the Java interface class and can also identify a callback interface, where the first Java interface represents the forward (service) call interface and the second interface represents the interface used to call back from the service to the client.

It is possible that the Java interface class referenced by the <interface.java/> element contains one or more annotations defined by the JAX-WS specification [JAX-WS]. These annotations can affect the interpretation of the <interface.java/> element. In the most extreme case, the annotations cause the replacement of the <interface.java/> element with an <interface.wsdl/> element. The relevant JAX-WS annotations and their effects on the <interface.java/> element are described in the section "JAX-WS Annotations and SCA Interfaces".

The interface.java element MUST conform to the schema defined in the sca-interface-java.xsd schema. [JCA30004]

Snippet 3-1 is the pseudo-schema for the interface.java element

```
<interface.java interface="NCName" callbackInterface="NCName"?  
    requires="list of xs:QName"?  
    policySets="list of xs:QName"?  
    remotable="boolean"?/>
```

Snippet 3-1: interface.java Pseudo-Schema

The interface.java element has the attributes:

- **interface : NCName (1..1)** – the Java interface class to use for the service interface. The value of the @interface attribute MUST be the fully qualified name of a Java class [JCA30001]
If the identified class is annotated with either the JAX-WS @WebService or @WebServiceProvider annotations and the annotation has a non-empty **wsdlLocation** property, then the SCA Runtime MUST act as if an <interface.wsdl/> element is present instead of the <interface.java/> element, with an @interface attribute identifying the portType mapped from the Java interface class and containing @requires and @policySets attribute values equal to the @requires and @policySets attribute values of the <interface.java/> element. [JCA30010]
- **callbackInterface : NCName (0..1)** – the Java interface class to use for the callback interface. The value of the @callbackInterface attribute MUST be the fully qualified name of a Java interface used for callbacks [JCA30002]
- **requires : QName (0..1)** – a list of policy intents. See the Policy Framework specification [POLICY] for a description of this attribute
- **policySets : QName (0..1)** – a list of policy sets. See the Policy Framework specification [POLICY] for a description of this attribute.
- **remotable : boolean (0..1)** – indicates whether or not the interface is remotable. A value of “true” means the interface is remotable and a value of “false” means it is not. This attribute does not have a default value. If it is not specified then the remotability is determined by the presence or absence of the @Remotable annotation on the interface class. The @remotable attribute applies to both the interface and any optional callbackInterface. The @remotable attribute is intended as an alternative to using the @Remotable annotation on the interface class. The value of the @remotable attribute

on the <interface.java/> element does not override the presence of a @Remotable annotation on the interface class and so if the interface class contains a @Remotable annotation and the @remotable attribute has a value of "false", then the SCA Runtime MUST raise an error and MUST NOT run the component concerned. [JCA30005]

Snippet 3-2 shows an example of the Java interface element:

```
<interface.java interface="services.stockquote.StockQuoteService"
  callbackInterface="services.stockquote.StockQuoteServiceCallback"/>
```

Snippet 3-2 Example interface.java Element

Here, the Java interface is defined in the Java class file **./services/stockquote/StockQuoteService.class**, where the root directory is defined by the contribution in which the interface exists. Similarly, the callback interface is defined in the Java class file **./services/stockquote/StockQuoteServiceCallback.class**.

Note that the Java interface class identified by the @interface attribute can contain a Java @Callback annotation which identifies a callback interface. If this is the case, then it is not necessary to provide the @callbackInterface attribute. However, **if the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same interface class.** [JCA30003]

For the Java interface type system, parameters and return types of the service methods are described using Java classes or simple Java types. It is recommended that the Java Classes used conform to the requirements of either JAXB [JAX-B] or of Service Data Objects [SDO] because of their integration with XML technologies.

3.2 @Remotable

The **@Remotable** annotation on a Java interface, a service implementation class, or a service reference denotes an interface or class that is designed to be used for remote communication. Remotable interfaces are intended to be used for **coarse grained** services. Operations' parameters, return values and exceptions are passed **by-value**. Remotable Services are not allowed to make use of method **overloading**.

3.3 @Callback

A callback interface is declared by using a @Callback annotation on a Java service interface, with the Java Class object of the callback interface as a parameter. There is another form of the @Callback annotation, without any parameters, that specifies callback injection for a setter method or a field of an implementation.

3.4 @AsyncInvocation

An interface can be annotated with @AsyncInvocation or with the equivalent @Requires("sca:asyncInvocation") annotation to indicate that request/response operations of that interface are **long running** and that response messages are likely to be sent an arbitrary length of time after the initial request message is sent to the target service. This is described in the [SCA Assembly Specification \[ASSEMBLY\]](#).

For a service client, it is strongly recommended that the client uses the asynchronous form of the client interface when using a reference to a service with an interface annotated with @AsyncInvocation, using either polling or callbacks to receive the response message. See the sections "[Asynchronous Programming](#)" and the section "[JAX-WS Client Asynchronous API for a Synchronous Service](#)" for more details about the asynchronous client API.

335 For a service implementation, SCA provides an **asynchronous service** mapping of the WSDL
336 request/response interface which enables the service implementation to send the response message at
337 an arbitrary time after the original service operation is invoked. This is described in the section
338 "[Asynchronous handling of Long Running Service Operations](#)".

339 3.5 SCA Java Annotations for Interface Classes

340 A Java interface referenced by the @interface attribute of an <interface.java/> element MUST NOT
341 contain any of the following SCA Java annotations:

342 @AllowsPassByReference, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit,
343 @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service. [JCA30006]

344 A Java interface referenced by the @callbackInterface attribute of an <interface.java/> element MUST
345 NOT contain any of the following SCA Java annotations:

346 @AllowsPassByReference, @Callback, @ComponentName, @Constructor, @Context, @Destroy,
347 @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service. [JCA30007]

348 3.6 Compatibility of Java Interfaces

349 The SCA Assembly Model specification [ASSEMBLY] defines a number of criteria that need to be
350 satisfied in order for two interfaces to be compatible or have a compatible superset or subset relationship.
351 If these interfaces are both Java interfaces, compatibility also means that every method that is present in
352 both interfaces is defined consistently in both interfaces with respect to the @OneWay annotation, that is,
353 the annotation is either present in both interfaces or absent in both interfaces. [JCA30009]

4 SCA Component Implementation Lifecycle

This section describes the lifecycle of an SCA component implementation.

4.1 Overview of SCA Component Implementation Lifecycle

At a high level, there are 3 main phases through which an SCA component implementation will transition when it is used by an SCA Runtime:

- **The Initialization phase.** This involves constructing an instance of the component implementation class and injecting any properties and references. Once injection is complete, the method annotated with `@Init` is called, if present, which provides the component implementation an opportunity to perform any internal initialization it requires.
- **The Running phase.** This is where the component implementation has been initialized and the SCA Runtime can dispatch service requests to it over its Service interfaces.
- **The Destroying phase.** This is where the component implementation's scope has ended and the SCA Runtime destroys the component implementation instance. The SCA Runtime calls the method annotated with `@Destroy`, if present, which provides the component implementation an opportunity to perform any internal clean up that is required.

4.2 SCA Component Implementation Lifecycle State Diagram

The state diagram in Figure 4-1 shows the lifecycle of an SCA component implementation. The sections that follow it describe each of the states that it contains.

It should be noted that some component implementation specifications might not implement all states of the lifecycle. In this case, that state of the lifecycle is skipped over.

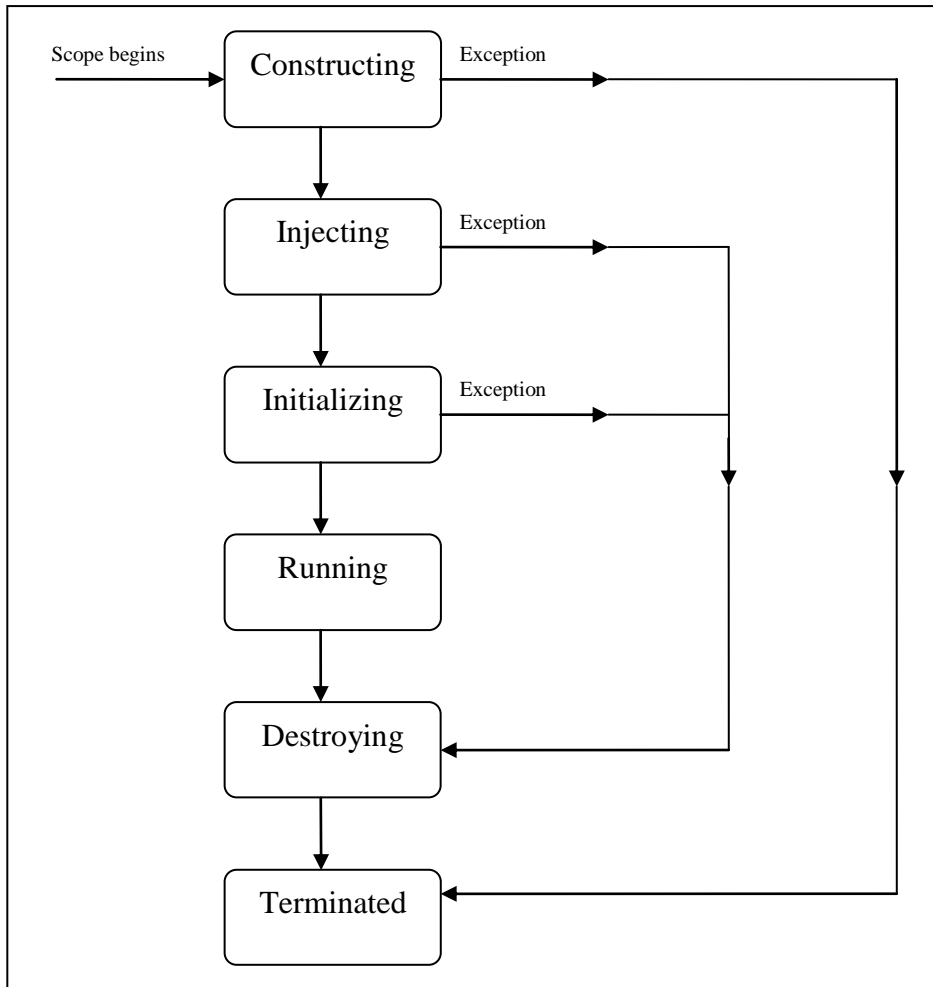


Figure 4-1: SCA - Component Implementation Lifecycle

4.2.1 Constructing State

The SCA Runtime MUST call a constructor of the component implementation at the start of the Constructing state. [JCA40001] The SCA Runtime MUST perform any constructor reference or property injection when it calls the constructor of a component implementation. [JCA40002]

The result of invoking operations on any injected references when the component implementation is in the Constructing state is undefined.

When the constructor completes successfully, the SCA Runtime MUST transition the component implementation to the Injecting state. [JCA40003] If an exception is thrown whilst in the Constructing state, the SCA Runtime MUST transition the component implementation to the Terminated state. [JCA40004]

4.2.2 Injecting State

When a component implementation instance is in the Injecting state, the SCA Runtime MUST first inject all field and setter properties that are present into the component implementation. [JCA40005] The order in which the properties are injected is unspecified.

When a component implementation instance is in the Injecting state, the SCA Runtime MUST inject all field and setter references that are present into the component implementation, after all the properties have been injected. [JCA40006] The order in which the references are injected is unspecified.

The SCA Runtime MUST ensure that the correct synchronization model is used so that all injected properties and references are made visible to the component implementation without requiring the component implementation developer to do any specific synchronization. [JCA40007]

The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation is in the Injecting state. [JCA40008]

The result of invoking operations on any injected references when the component implementation is in the Injecting state is undefined.

When the injection of properties and references completes successfully, the SCA Runtime MUST transition the component implementation to the Initializing state. [JCA40009] If an exception is thrown whilst injecting properties or references, the SCA Runtime MUST transition the component implementation to the Destroying state. [JCA40010] If a property or reference is unable to be injected, the SCA Runtime MUST transition the component implementation to the Destroying state. [JCA40024]

4.2.3 Initializing State

When the component implementation enters the Initializing State, the SCA Runtime MUST call the method annotated with @Init on the component implementation, if present. [JCA40011]

The component implementation can invoke operations on any injected references when it is in the Initializing state. However, depending on the order in which the component implementations are initialized, the target of the injected reference might not be available since it has not yet been initialized. If a component implementation invokes an operation on an injected reference that refers to a target that has not yet been initialized, the SCA Runtime MUST throw a ServiceUnavailableException. [JCA40012]

The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Initializing state. [JCA40013]

Once the method annotated with @Init completes successfully, the SCA Runtime MUST transition the component implementation to the Running state. [JCA40014] If an exception is thrown whilst initializing, the SCA Runtime MUST transition the component implementation to the Destroying state. [JCA40015]

4.2.4 Running State

The SCA Runtime MUST invoke Service methods on a component implementation instance when the component implementation is in the Running state and a client invokes operations on a service offered by the component. [JCA40016]

The component implementation can invoke operations on any injected references when the component implementation instance is in the Running state.

When the component implementation scope ends, the SCA Runtime MUST transition the component implementation to the Destroying state. [JCA40017]

4.2.5 Destroying State

When a component implementation enters the Destroying state, the SCA Runtime MUST call the method annotated with @Destroy on the component implementation, if present. [JCA40018]

The component implementation can invoke operations on any injected references when it is in the Destroying state. However, depending on the order in which the component implementations are destroyed, the target of the injected reference might no longer be available since it has been destroyed. If a component implementation invokes an operation on an injected reference that refers to a target that has been destroyed, the SCA Runtime MUST throw an InvalidServiceException. [JCA40019]

The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Destroying state. [JCA40020]

Once the method annotated with @Destroy completes successfully, the SCA Runtime MUST transition the component implementation to the Terminated state. [JCA40021] If an exception is thrown whilst destroying, the SCA Runtime MUST transition the component implementation to the Terminated state. [JCA40022]

440 **4.2.6 Terminated State**

441 The lifecycle of the SCA Component has ended.

442 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
443 component implementation instance is in the Terminated state. [\[JCA40023\]](#)

5 Client API

This section describes how SCA services can be programmatically accessed from components and also from non-managed code, that is, code not running as an SCA component.

5.1 Accessing Services from an SCA Component

An SCA component can obtain a service reference either through injection or programmatically through the **ComponentContext** 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 is provided for use in cases where reference injection is not possible.

5.1.1 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.

5.2 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.

5.2.1 SCAClientFactory Interface and Related Classes

Client code can use the **SCAClientFactory** class to obtain proxy reference objects for a service which is in an SCA Domain. The URI of the domain, the relative URI of the service and the business interface of the service must all be known in order to use the SCAClientFactory class.

Objects which implement the SCAClientFactory are obtained using the newInstance() methods of the SCAClientFactory class.

Snippet 5-1 is a sample of the code that a client would use:

```
package org.oasisopen.sca.client.example;

import java.net.URI;

import org.oasisopen.sca.client.SCAClientFactory;
import org.oasisopen.sca.client.example.HelloService;

/**
 * Example of use of Client API for a client application to obtain
 * an SCA reference proxy for a service in an SCA Domain.
 */
public class Client1 {

    public void someMethod() {

        try {

            String serviceURI = "SomeHelloServiceURI";
            URI domainURI = new URI("SomeDomainURI");

            SCAClientFactory scaClient =
                SCAClientFactory.newInstance( domainURI );
            HelloService helloService =
                scaClient.getService(HelloService.class,
                                    serviceURI);
```

```
492         String reply = helloService.sayHello("Mark");
493
494     } catch (Exception e) {
495         System.out.println("Received exception");
496     }
497 }
498 }
```

499 *Snippet 5-1: Using the SCAClientFactory Interface*

500

501 For details about the SCAClientFactory interface and its related classes see the section
502 ["SCAClientFactory Class"](#).

6 Error Handling

Clients calling service methods can 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 management of component execution or problems interacting with remote services. The SCA runtime exceptions are defined in [the Java API section](#).

7 Asynchronous 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, is 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
- callbacks

Each of these topics is discussed in the following sections.

7.1 @OneWay

Non-blocking 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.

A method with a void return type and which has no declared exceptions can be marked with a **@OneWay** annotation. This means that the method is non-blocking and communication with the service provider can use a binding that buffers the request and sends it at some later time.

For a Java client to make a non-blocking call to methods that either return values or throw exceptions, a Java client can use the JAX-WS asynchronous client API model that is described in [the section "JAX-WS Client Asynchronous API for a Synchronous Service"](#). It is considered to be a best practice that service designers define one-way methods as often as possible, in order to give the greatest degree of binding flexibility to deployers.

7.2 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 is provided by the client

Callbacks can be used for both remotable and local services. Either both interfaces of a bidirectional service are remotable, or both are local. It is illegal to mix the two, as defined in [the SCA Assembly Model specification \[ASSEMBLY\]](#).

A callback interface is declared by using a **@Callback** annotation on a service interface, with the Java Class object of the interface as a parameter. The annotation can 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.

7.2.1 Using Callbacks

Bidirectional interfaces and callbacks are used when a simple request/response pattern isn't sufficient to capture the business semantics of a service interaction. Callbacks are well suited for cases when a service request can result in multiple responses or new requests from the service back to the client, or where the service might respond to the client some time after the original request has completed.

Snippet 7-1 shows a scenario in which bidirectional interfaces and callbacks could be used. A client requests a quotation from a supplier. To process the enquiry and return the quotation, some suppliers might need additional information from the client. The client does not know which additional items of information will be needed by different suppliers. This interaction can be modeled as a bidirectional interface with callback requests to obtain the additional information.


```

554
555 package somepackage;
556 import org.oasisopen.sca.annotation.Callback;
557 import org.oasisopen.sca.annotation.Remotable;
558
559 @Remotable
560 @Callback(QuotationCallback.class)
561 public interface Quotation {
562     double requestQuotation(String productCode, int quantity);
563 }
564
565 @Remotable
566 public interface QuotationCallback {
567     String getState();
568     String getZipCode();
569     String getCreditRating();
570 }

```

571 *Snippet 7-1: Using a Bidirectional Interface*

572
573 In Snippet 7-1, the `requestQuotation` operation requests a quotation to supply a given quantity of a
574 specified product. The `QuotationCallback` interface provides a number of operations that the supplier can
575 use to obtain additional information about the client making the request. For example, some suppliers
576 might quote different prices based on the state or the ZIP code to which the order will be shipped, and
577 some suppliers might quote a lower price if the ordering company has a good credit rating. Other
578 suppliers might quote a standard price without requesting any additional information from the client.

579 Snippet 7-2 illustrates a possible implementation of the example service, using the `@Callback` annotation
580 to request that a callback proxy be injected.

```

581
582 @Callback
583 protected QuotationCallback callback;
584
585 public double requestQuotation(String productCode, int quantity) {
586     double price = getPrice(productCode, quantity);
587     double discount = 0;
588     if (quantity > 1000 && callback.getState().equals("FL")) {
589         discount = 0.05;
590     }
591     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
592         discount += 0.05;
593     }
594     return price * (1-discount);
595 }

```

596 *Snippet 7-2: Example Implementation of a Service with a Bidirectional Interface*

597
598 Snippet 7-3 is taken from the client of this example service. The client's service implementation class
599 implements the methods of the `QuotationCallback` interface as well as those of its own service interface
600 `ClientService`.

```

601
602 public class ClientImpl implements ClientService, QuotationCallback {
603
604     private QuotationService myService;
605
606     @Reference
607     public void setMyService(QuotationService service) {
608         myService = service;
609     }
610 }

```

```

611     public void aClientMethod() {
612         ...
613         double quote = myService.requestQuotation("AB123", 2000);
614         ...
615     }
616
617     public String getState() {
618         return "TX";
619     }
620     public String getZipCode() {
621         return "78746";
622     }
623     public String getCreditRating() {
624         return "AA";
625     }
626 }

```

627 *Snippet 7-3: Example Client Using a Bidirectional Interface*

628

629 Snippet 7-3 the callback is **stateless**, i.e., the callback requests do not need any information relating to
630 the original service request. For a callback that needs information relating to the original service request
631 (a **stateful** callback), this information can be passed to the client by the service provider as parameters
632 on the callback request.

633 7.2.2 Callback Instance Management

634 Instance management for callback requests received by the client of the bidirectional service is handled in
635 the same way as instance management for regular service requests. If the client implementation has
636 STATELESS scope, the callback is dispatched using a newly initialized instance. If the client
637 implementation has COMPOSITE scope, the callback is dispatched using the same shared instance that
638 is used to dispatch regular service requests.

639 As described in [the section "Using Callbacks"](#), a stateful callback can obtain information relating to the
640 original service request from parameters on the callback request. Alternatively, a composite-scoped
641 client could store information relating to the original request as instance data and retrieve it when the
642 callback request is received. These approaches could be combined by using a key passed on the
643 callback request (e.g., an order ID) to retrieve information that was stored in a composite-scoped instance
644 by the client code that made the original request.

645 7.2.3 Callback Injection

646 When a bidirectional service is invoked, the SCA runtime MUST inject a callback reference for the
647 invoking service into all fields and setter methods of the service implementation class that are marked
648 with a @Callback annotation and typed by the callback interface of the bidirectional service, and the SCA
649 runtime MUST inject null into all other fields and setter methods of the service implementation class that
650 are marked with a @Callback annotation. [\[JCA60001\]](#) When a non-bidirectional service is invoked, the
651 SCA runtime MUST inject null into all fields and setter methods of the service implementation class that
652 are marked with a @Callback annotation. [\[JCA60002\]](#)

653 7.2.4 Implementing Multiple Bidirectional Interfaces

654 Since it is possible for a single implementation class to implement multiple services, it is also possible for
655 callbacks to be defined for each of the services that it implements. The service implementation can
656 include an injected field for each of its callbacks. The runtime injects the callback onto the appropriate
657 field based on the type of the callback. Snippet 7-4 shows the declaration of two fields, each of which
658 corresponds to a particular service offered by the implementation.

```

659
660     @Callback
661     protected MyService1Callback callback1;

```

```
@Callback
protected MyService2Callback callback2;
```

Snippet 7-4: Multiple Bidirectional Interfaces in an Implementation

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

7.2.5 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 of type **ServiceReference** with the **@Callback** annotation.

A reference implementing the callback service interface can be obtained using `ServiceReference.getService()`.

Snippet 7-5 comes from a service implementation that uses the callback API:

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

public void someMethod() {
    MyCallback myCallback = callback.getService();    ...
    myCallback.receiveResult(theResult);
}
```

Snippet 7-5: Using the Callback API

Because `ServiceReference` objects are serializable, they can be stored persistently and retrieved at a later time to make a callback invocation after the associated service request has completed. `ServiceReference` objects can also be passed as parameters on service invocations, enabling the responsibility for making the callback to be delegated to another service.

Alternatively, a callback can be retrieved programmatically using the **RequestContext** API. Snippet 7-6 shows how to retrieve a callback in a method programmatically:

```
@Context
ComponentContext context;

public void someMethod() {
    MyCallback myCallback = context.getRequestContext().getCallback();
    ...
    myCallback.receiveResult(theResult);
}
```

Snippet 7-6: Using RequestContext to get a Callback

This is necessary if the service implementation has COMPOSITE scope, because callback injection is not performed for composite-scoped implementations.

7.3 Asynchronous handling of Long Running Service Operations

Long-running request-response operations are described in the SCA Assembly Specification [ASSEMBLY]. These operations are characterized by following the WSDL request-response message exchange pattern, but where the timing of the sending of the response message is arbitrarily later than the receipt of the request message, with an impact on the client component, on the service component and also on the transport binding used to communicate between them.

In SCA, such operations are marked with an intent "asyncInvocation" and is expected that the client component, the service component and the binding are all affected by the presence of this intent. This specification does not describe the effects of the intent on the binding, other than to note that in general, there is an implication that the sending of the response message is typically separate from the sending of the request message, typically requiring a separate response endpoint on the client to which the response can be sent.

For components that are clients of a long-running request-response operation, it is strongly recommended that the client makes use of the JAX-WS Client Asynchronous API, either using the polling interface or the callback mechanism described in the section "[JAX-WS Client Asynchronous API for a Synchronous Service](#)". The principle is that the client should not synchronously wait for a response from the long running operation since this could take a long time and it is preferable not to tie up resources while waiting.

For the service implementation component, the JAX-WS client asynchronous API is not suitable, so the SCA Java Common Annotations and APIs specification defines the SCA Asynchronous Service interface, which, like the JAX-WS client asynchronous API, is an alternative mapping of a WSDL request-response operation into a Java interface.

7.4 SCA Asynchronous Service Interface

The SCA Asynchronous Service interface follows some of the patterns defined by the JAX-WS client asynchronous API, but it is a simpler interface aligned with the needs of a service implementation class.

As an example, for a WSDL portType with a single operation "getPrice" with a String request parameter and a float response, the synchronous Java interface mapping appears in Snippet 7-7.

```
// synchronous mapping
public interface StockQuote {
    float getPrice(String ticker);
}
```

Snippet 7-7: Example Synchronous Java Interface Mapping

The JAX-WS client asynchronous API for the same portType adds two asynchronous forms for each synchronous method, as shown in Snippet 7-8.

```
// asynchronous mapping
public interface StockQuote {
    float getPrice(String ticker);
    Response<Float> getPriceAsync(String ticker);
    Future<?> getPriceAsync(String ticker, AsyncHandler<Float> handler);
}
```

Snippet 7-8: Example JAX-WS Client Asynchronous Java interface Mapping

The SCA Asynchronous Service interface has a single method similar to the final one in the asynchronous client interface, as shown in Snippet 7-8.

```
// asynchronous mapping
```

```

759 @Requires("sca:asyncInvocation")
760 public interface StockQuote {
761     void getPriceAsync(String ticker, ResponseDispatch<Float> dispatch);
762 }

```

763 *Snippet 7-9: Example SCA Asynchronous Service Java interface Mapping*

764

765 The main characteristics of the SCA asynchronous mapping are:

- 766 • there is a single method, with a name with the string "Async" appended to the operation name
- 767 • it has a void return type
- 768 • it has two input parameters, the first is the request message of the operation and the second is a
769 ResponseDispatch object typed by the response message of the operation (following the rules
770 expressed in the JAX-WS specification for the typing of the AsyncHandler object in the client
771 asynchronous API)
- 772 • it is annotated with the asyncInvocation intent
- 773 • if the synchronous method has any business faults/exceptions, it is annotated with @AsyncFault,
774 containing a list of the exception classes

775 Unlike the JAX-WS asynchronous client interface, there is only a single operation for the service
776 implementation to provide (it would be inconvenient for the service implementation to be required to
777 implement multiple methods for each operation in the WSDL interface).

778 The ResponseDispatch parameter is the mechanism by which the service implementation sends back the
779 response message resulting from the invocation of the service method. The ResponseDispatch is
780 serializable and it can be invoked once at any time after the invocation of the service method, either
781 before or after the service method returns. This enables the service implementation to store the
782 ResponseDispatch in serialized form and release resources while waiting for the completion of whatever
783 activities result from the processing of the initial invocation.

784 The ResponseDispatch object is allocated by the SCA runtime/binding implementation and it is expected
785 to contain whatever metadata is required to deliver the response message back to the client that invoked
786 the service operation.

787 The SCA asynchronous service Java interface mapping of a WSDL request-response operation
788 MUST appear as follows:

789 The interface is annotated with the "asyncInvocation" intent.

- 790 – For each service operation in the WSDL, the Java interface contains an operation with
- 791 – a name which is the JAX-WS mapping of the WSDL operation name, with the suffix "Async"
792 added
- 793 – a void return type
- 794 – a set of input parameter(s) which match the JAX-WS mapping of the input parameter(s) of the
795 WSDL operation plus an additional last parameter which is a ResponseDispatch object typed by
796 the JAX-WS Response Bean mapping of the output parameter(s) of the WSDL operation, where
797 ResponseDispatch is the type defined in the SCA Java Common Annotations and APIs
798 specification. [JCA60003]

799 An SCA Runtime MUST support the use of the SCA asynchronous service interface for the interface of an
800 SCA service. [JCA60004]

801 The ResponseDispatch object passed in as a parameter to a method of a service implementation using
802 the SCA asynchronous service Java interface can be invoked once only through either its sendResponse
803 method or through its sendFault method to return the response resulting from the service method
804 invocation. If the SCA asynchronous service interface ResponseDispatch handleResponse method is
805 invoked more than once through either its sendResponse or its sendFault method, the SCA runtime
806 MUST throw an IllegalStateException. [JCA60005]

807

808 For the purposes of matching interfaces (when wiring between a reference and a service, or when using
809 an implementation class by a component), an interface which has one or more methods which follow the
810 SCA asynchronous service pattern MUST be treated as if those methods are mapped as the equivalent
811 synchronous methods, as follows:

812 Asynchronous service methods are characterized by:

- 813 – void return type
- 814 – a method name with the suffix "Async"
- 815 – a last input parameter with a type of ResponseDispatch<X>
- 816 – annotation with the asyncInvocation intent
- 817 – possible annotation with the @AsyncFault annotation

818 The mapping of each such method is as if the method had the return type "X", the method name without
819 the suffix "Async" and all the input parameters except the last parameter of the type
820 ResponseDispatch<X>, plus the list of exceptions contained in the @AsyncFault annotation. [JCA60006]

8 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 \[POLICY\]](#). 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 have to 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.

This specification has 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 and Transactions.

This specification supports using [the Common Annotations for the Java Platform specification \(JSR-250\) \[JSR-250\]](#). An implication of adopting the common annotation for Java platform specification is that the SCA Java specification supports consistent annotation and Java class inheritance relationships. SCA policy annotation semantics follow the General Guidelines for Inheritance of Annotations in [the Common Annotations for the Java Platform specification \[JSR-250\]](#), except that member-level annotations in a class or interface do not have any effect on how class-level annotations are applied to other members of the class or interface.

8.1 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 shown in Snippet 8-1.

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

Snippet 8-1: Intent Format

Intents can be qualified, in which case the string consists of the base intent name, followed by a ".", followed by the name of the qualifier. There can 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 those in Snippet 8-2.

867
868
869
870
871
872

```
public static final String SCA_PREFIX =  
    "{http://docs.oasis-open.org/ns/opencsa/sca/200912}";  
public static final String CONFIDENTIALITY =  
    SCA_PREFIX + "confidentiality";  
public static final String CONFIDENTIALITY_MESSAGE =  
    CONFIDENTIALITY + ".message";
```

873 *Snippet 8-2: Example Intent Constants*

874

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

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

880 An example of the @Requires annotation with 2 qualified intents (from the Security domain) is shown in
881 Snippet 8-3:

882

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

884 *Snippet 8-3: Multiple Intents in One Annotation*

885

886 The annotation in Snippet 8-3 attaches the intents "confidentiality.message" and "integrity.message".

887 Snippet 8-4 is an example of a reference requiring support for confidentiality:

888

```
package com.foo;  
  
import static org.oasisopen.sca.annotation.Confidentiality.*;  
import static org.oasisopen.sca.annotation.Reference;  
import static org.oasisopen.sca.annotation.Requires;  
  
public class Foo {  
    @Requires(CONFIDENTIALITY)  
    @Reference  
    public void setBar(Bar bar) {  
        ...  
    }  
}
```

902 *Snippet 8-4: Annotation a Reference*

903

904 Users can also choose to only use constants for the namespace part of the QName, so that they can add
905 new intents without having to define new constants. In that case, the definition of Snippet 8-4 would
906 instead look like Snippet 8-5.

907

```
package com.foo;  
  
import static org.oasisopen.sca.Constants.*;  
import static org.oasisopen.sca.annotation.Reference;  
import static org.oasisopen.sca.annotation.Requires;  
  
public class Foo {  
    @Requires(SCA_PREFIX+"confidentiality")  
    @Reference  
    public void setBar(Bar bar) {  
        ...  
    }  
}
```


Snippet 8-5: Using Intent Constants and strings

The formal syntax [EBNF-Syntax] for the @Requires annotation follows:

```
'@Requires('' QualifiedIntent '' ('','' QualifiedIntent '')* '')
```

where

```
QualifiedIntent ::= QName('.' Qualifier)*  
Qualifier ::= NCName
```

See [section @Requires](#) for the formal definition of the @Requires annotation.

8.2 Specific Intent Annotations

In addition to the general intent annotation supplied by the @Requires annotation described in section 8.2, 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) annotation can also be specified with the @Confidentiality specific intent annotation. The specific intent annotation for the "integrity" security intent is shown in Snippet 8-6.

```
@Integrity
```

Snippet 8-6: Example Specific Intent Annotation

An example of a qualified specific intent for the "authentication" intent is shown in Snippet 8-7.

```
@Authentication( {"message", "transport"} )
```

Snippet 8-7: Example Qualified Specific Intent Annotation

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://docs.oasis-open.org/ns/opencsa/sca/200912").

The general form of specific intent annotations is shown in Snippet 8-8

```
'@' Intent '('(' qualifiers '')
```

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

```
Intent          ::= NCName  
qualifiers      ::= '' qualifier '' ('','' qualifier '')*  
qualifier ::= NCName ('.' qualifier)?
```

Snippet 8-8: Specific Intent Annotation Format

8.2.1 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 a specific intent annotation. [JCA70001]

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, for 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 is also possible to use one or more of them as parameters to the specific intent annotation.

Alternatively, the QName of the intent can be specified using separate parameters for the `targetNamespace` and the `localPart`, as shown in Snippet 8-9:

```
@Intent(targetNamespace=SCA_NS, localPart="confidentiality")
```

Snippet 8-9: Defining a Specific Intent Annotation

See [section @Intent](#) for the formal definition of the `@Intent` annotation.

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 needs to be marked with the `@Qualifier` annotation. The `@Qualifier` tells SCA that the value of the attribute is 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 exist. For example the annotation in Snippet 8-10

```
@Confidentiality({"message", "transport"})
```

Snippet 8-10: Multiple Qualifiers in an Annotation'

implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport" are set for the element to which the `@Confidentiality` annotation is attached.

See [section @Qualifier](#) for the formal definition of the `@Qualifier` annotation.

Examples of the use of the `@Intent` and the `@Qualifier` annotations in the definition of specific intent annotations are shown in [the section dealing with Security Interaction Policy](#).

8.3 Application of Intent Annotations

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

- Java class
- Java interface
- Method
- Field
- Constructor parameter

Intent annotations MUST NOT be applied to the following:

- A method of a service implementation class, except for a setter method that is either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class field that is not either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class constructor parameter that is not annotated with `@Reference`

[JCA70002]

Intent annotations can be applied to classes, interfaces, and interface methods. Applying an intent annotation to a field, setter method, or constructor parameter allows intents to be defined at references. Intent annotations can also be applied to reference interfaces and their methods.

Where multiple intent annotations (general or specific) are applied to the same Java element, the SCA runtime MUST compute the combined intents for the Java element by merging the intents from all intent annotations on the Java element according to the SCA Policy Framework [POLICY] rules for merging intents at the same hierarchy level. [JCA70003]

An example of multiple policy annotations being used together is shown in Snippet 8-11:

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

Snippet 8-11: Multiple Policy Annotations

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

If intent annotations are specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective intents for the method by merging the combined intents from the method with the combined intents for the interface according to the SCA Policy Framework [POLICY] rules for merging intents within a structural hierarchy, with the method at the lower level and the interface at the higher level. [JCA70004] This merging process does not remove or change any intents that are applied to the interface.

8.3.1 Intent Annotation Examples

The following examples show how the rules defined in section 8.3 are applied.

Snippet 8-12 shows how intents on references are merged. In this example, the intents for `myRef` are "authentication" and "confidentiality.message".

```
@Authentication
@Requires(CONFIDENTIALITY)
@Confidentiality("message")
@Reference
protected MyService myRef;
```

Snippet 8-12: Merging Intents on References

Snippet 8-13 shows that mutually exclusive intents cannot be applied to the same Java element. In this example, the Java code is in error because of contradictory mutually exclusive intents "managedTransaction" and "noManagedTransaction".

```
@Requires({SCA_PREFIX+"managedTransaction",
           SCA_PREFIX+"noManagedTransaction"})
@Reference
protected MyService myRef;
```

Snippet 8-13: Mutually Exclusive Intents

Snippet 8-14 shows that intents can be applied to Java service interfaces and their methods. In this example, the effective intents for `MyService.mymethod()` are "authentication" and "confidentiality".

```
@Authentication
public interface MyService {
```

```

1053     @Confidentiality
1054     public void mymethod();
1055 }
1056 @Service(MyService.class)
1057 public class MyServiceImpl {
1058     public void mymethod() {...}
1059 }

```

1060 *Snippet 8-14: Intents on Java Interfaces, Interface Methods, and Java Classes*

1061
1062 Snippet 8-15 shows that intents can be applied to Java service implementation classes. In this example,
1063 the effective intents for `MyService.mymethod()` are "authentication", "confidentiality", and
1064 "managedTransaction".

```

1065
1066 @Authentication
1067 public interface MyService {
1068     @Confidentiality
1069     public void mymethod();
1070 }
1071 @Service(MyService.class)
1072 @Requires(SCA_PREFIX+"managedTransaction")
1073 public class MyServiceImpl {
1074     public void mymethod() {...}
1075 }

```

1076 *Snippet 8-15: Intents on Java Service Implementation Classes*

1077
1078 Snippet 8-16 shows that intents can be applied to Java reference interfaces and their methods, and also
1079 to Java references. In this example, the effective intents for the method `mymethod()` of the reference
1080 `myRef` are "authentication", "integrity", and "confidentiality".

```

1081
1082 @Authentication
1083 public interface MyRefInt {
1084     @Integrity
1085     public void mymethod();
1086 }
1087 @Service(MyService.class)
1088 public class MyServiceImpl {
1089     @Confidentiality
1090     @Reference
1091     protected MyRefInt myRef;
1092 }

```

1093 *Snippet 8-16: Intents on Java References and their Interfaces and Methods*

1094
1095 Snippet 8-17 shows that intents cannot be applied to methods of Java implementation classes. In this
1096 example, the Java code is in error because of the `@Authentication` intent annotation on the
1097 implementation method `MyServiceImpl.mymethod()`.

```

1098
1099 public interface MyService {
1100     public void mymethod();
1101 }
1102 @Service(MyService.class)
1103 public class MyServiceImpl {
1104     @Authentication
1105     public void mymethod() {...}
1106 }

```

Snippet 8-17: Intent on Implementation Method

Snippet 8-18 shows one effect of applying the SCA Policy Framework rules for merging intents within a structural hierarchy to Java service interfaces and their methods. In this example a qualified intent overrides an unqualified intent, so the effective intent for `MyService.mymethod()` is "confidentiality.message".

```
@Confidentiality("message")
public interface MyService {
    @Confidentiality
    public void mymethod();
}
```

Snippet 8-18: Merging Qualified and Unqualified Intents on Java Interfaces and Methods

Snippet 8-19 shows another effect of applying the SCA Policy Framework rules for merging intents within a structural hierarchy to Java service interfaces and their methods. In this example a lower-level intent causes a mutually exclusive higher-level intent to be ignored, so the effective intent for `mymethod1()` is "managedTransaction" and the effective intent for `mymethod2()` is "noManagedTransaction".

```
@Requires(SCA_PREFIX+"managedTransaction")
public interface MyService {
    public void mymethod1();
    @Requires(SCA_PREFIX+"noManagedTransaction")
    public void mymethod2();
}
```

Snippet 8-19: Merging Mutually Exclusive Intents on Java Interfaces and Methods

8.3.2 Inheritance and Annotation

Snippet 8-20 shows the inheritance relations of intents on classes, operations, and super classes.

```
package services.hello;
import org.oasisopen.sca.annotation.Authentication;
import org.oasisopen.sca.annotation.Integrity;

@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.oasisopen.sca.annotation.Authentication;
import org.oasisopen.sca.annotation.Confidentiality;

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

Snippet 8-20: Usage example of Annotated Policy and Inheritance

The effective intent annotation on the **helloWorld** method of **HelloChildService** is **@Authentication** and **@Confidentiality("message")**.
The effective intent annotation on the **hello** method of **HelloChildService** is **@Confidentiality("transport")**,
The effective intent annotation on the **helloThere** method of **HelloChildService** is **@Integrity** and **@Authentication("transport")**, the same as for this method in the **HelloService** class.
The effective intent annotation on the **hello** method of **HelloService** is **@Integrity** and **@Authentication("message")**

Table 8-1 shows the equivalent declarative security interaction policy of the methods of the **HelloService** and **HelloChildService** implementations corresponding to the Java classes shown in Snippet 8-20.

Class	Method		
	hello()	helloThere()	helloWorld()
HelloService	integrity authentication.message	integrity authentication.transport	N/A
HelloChildService	confidentiality.transport	integrity authentication.transport	authentication confidentiality.message

Table 8-1: Declarative Intents Equivalent to Annotated Intents in Snippet 8-20

8.4 Relationship of Declarative and Annotated Intents

Annotated intents on a Java class cannot be overridden by declarative intents in a composite document which uses the class as an implementation. This rule follows the general rule for intents that they represent requirements of an implementation in the form of a restriction that cannot be relaxed.
However, a restriction can be made more restrictive so that an unqualified version of an intent expressed through an annotation in the Java class can be qualified by a declarative intent in a using composite document.

8.5 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({' policySetQName ', ' policySetQName ' * '})'
```

Snippet 8-21: PolicySet Annotation Format

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

An example of the **@PolicySets** annotation is shown in Snippet 8-22:

```
@Reference(name="helloService", required=true)
```

```

1198 @PolicySets({ MY_NS + "WS_Encryption_Policy",
1199               MY_NS + "WS_Authentication_Policy" })
1200 public setHelloService(HelloService service) {
1201     . . .
1202 }

```

1203 *Snippet 8-22: Use of @PolicySets*

1204

1205 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
 1206 using the namespace defined for the constant MY_NS.

1207 PolicySets need to satisfy intents expressed for the implementation when both are present, according to
 1208 the rules defined in [the Policy Framework specification \[POLICY\]](#).

1209 The SCA Policy Set annotation can be applied to the following Java elements:

- 1210 • Java class
- 1211 • Java interface
- 1212 • Method
- 1213 • Field
- 1214 • Constructor parameter

1215 The @PolicySets annotation MUST NOT be applied to the following:

- 1216 • A method of a service implementation class, except for a setter method that is either annotated with
 1217 @Reference or introspected as an SCA reference according to the rules in the appropriate
 1218 Component Implementation specification
- 1219 • A service implementation class field that is not either annotated with @Reference or introspected as
 1220 an SCA reference according to the rules in the appropriate Component Implementation specification
- 1221 • A service implementation class constructor parameter that is not annotated with @Reference

1222 [\[JCA70005\]](#)

1223 The @PolicySets annotation can be applied to classes, interfaces, and interface methods. Applying a
 1224 @PolicySets annotation to a field, setter method, or constructor parameter allows policy sets to be
 1225 defined at references. The @PolicySets annotation can also be applied to reference interfaces and their
 1226 methods.

1227 If the @PolicySets annotation is specified on both an interface method and the method's declaring
 1228 interface, the SCA runtime MUST compute the effective policy sets for the method by merging the policy
 1229 sets from the method with the policy sets from the interface. [\[JCA70006\]](#) This merging process does not
 1230 remove or change any policy sets that are applied to the interface.

1231 8.6 Security Policy Annotations

1232 This section introduces annotations for commonly used SCA security intents, as defined in [the SCA](#)
 1233 [Policy Framework Specification \[POLICY\]](#). Also see the SCA Policy Framework Specification for
 1234 additional security policy intents that can be used with the @Requires annotation. The following
 1235 annotations for security policy intents and qualifiers are defined:

- 1236 • @Authentication
- 1237 • @Authorization
- 1238 • @Confidentiality
- 1239 • @Integrity
- 1240 • @MutualAuthentication

1241 The @Authentication, @Confidentiality, and @Integrity intents have the same pair of Qualifiers:

- 1242 • message
- 1243 • transport

The formal definitions of the security intent annotations are found in the section “Java Annotations”. Snippet 8-23 shows an example of applying security intents to the setter method used to inject a reference. Accessing the hello operation of the referenced HelloService requires both "integrity.message" and "authentication.message" intents to be honored.

```
package services.hello;
// Interface for HelloService
public interface HelloService {
    String hello(String helloMsg);
}

package services.client;
// Interface for ClientService
public interface ClientService {
    public void clientMethod();
}

// Implementation class for ClientService
package services.client;

import services.hello.HelloService;
import org.oasisopen.sca.annotation.*;

@Service(ClientService.class)
public class ClientServiceImpl implements ClientService {

    private HelloService helloService;

    @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!");
        ...
    }
}
```

Snippet 8-23: Usage of Security Intents on a Reference

8.7 Transaction Policy Annotations

This section introduces annotations for commonly used SCA transaction intents, as defined in [the SCA Policy Framework specification \[POLICY\]](#). Also see the SCA Policy Framework Specification for additional transaction policy intents that can be used with the @Requires annotation. The following annotations for transaction policy intents and qualifiers are defined:

- @ManagedTransaction
- @NoManagedTransaction
- @SharedManagedTransaction

The @ManagedTransaction intent has the following Qualifiers:

- global
- local

The formal definitions of the transaction intent annotations are found in the section “Java Annotations”.

Snippet 8-24 shows an example of applying a transaction intent to a component implementation, where the component implementation requires a global transaction.

```
package services.hello;
// Interface for HelloService
public interface HelloService {
    String hello(String helloMsg);
}

// Implementation class for HelloService
package services.hello.impl;

import services.hello.HelloService;
import org.oasisopen.sca.annotation.*;

@Service(HelloService.class)
@ManagedTransaction("global")
public class HelloServiceImpl implements HelloService {

    public void someMethod() {
        ...
    }
}
```

Snippet 8-24: Usage of Transaction Intents in an Implementation

9 Java API

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

9.1 Component Context

Figure 9-1 defines the **ComponentContext** interface:

```
package org.oasisopen.sca;
import java.util.Collection;
public interface ComponentContext {

    String getURI();

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

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

    <B> Collection<B> getServices( Class<B> businessInterface,
                               String referenceName);

    <B> Collection<ServiceReference<B>> getServiceReferences(
        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);

    RequestContext getRequestContext();

    <B> ServiceReference<B> cast(B target) throws IllegalArgumentException;

}
```

Figure 9-1: ComponentContext Interface

getURI () method:

Returns the structural URI **[ASSEMBLY]** of the component within the SCA Domain.

Returns:

- **String** which contains the absolute URI of the component in the SCA Domain
The ComponentContext.getURI method MUST return the structural URI of the component in the SCA Domain. [JCA80008]

Parameters:

- **none**

Exceptions:

- **none**

getService (Class businessInterface, String referenceName) method:

1369 Returns a typed service proxy object for a reference defined by the current component, where the
1370 reference has multiplicity 0..1 or 1..1.

1371 Returns:

- 1372 • **B** which is a proxy object for the reference, which implements the interface B contained in the
1373 businessInterface parameter.

1374 The ComponentContext.getService method MUST return the proxy object implementing the interface
1375 provided by the businessInterface parameter, for the reference named by the referenceName
1376 parameter with the interface defined by the businessInterface parameter when that reference has a
1377 target service configured. [JCA80009]

1378 The ComponentContext.getService method MUST return null if the multiplicity of the reference
1379 named by the referenceName parameter is 0..1 and the reference has no target service configured.
1380 [JCA80010]

1381 Parameters:

- 1382 • **Class businessInterface** - the Java interface for the service reference
- 1383 • **String referenceName** - the name of the service reference

1384 Exceptions:

- 1385 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the
1386 reference identified by the referenceName parameter has multiplicity of 0..n or 1..n. [JCA80001]
- 1387 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the
1388 component does not have a reference with the name supplied in the referenceName parameter.
1389 [JCA80011]
- 1390 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the service
1391 reference with the name supplied in the referenceName does not have an interface compatible with
1392 the interface supplied in the businessInterface parameter. [JCA80012]

1393

1394 **getServiceReference (Class businessInterface, String referenceName) method:**

1395 Returns a ServiceReference object for a reference defined by the current component, where the
1396 reference has multiplicity 0..1 or 1..1.

1397 Returns:

- 1398 • **ServiceReference** which is a ServiceReference proxy object for the reference, which implements
1399 the interface contained in the businessInterface parameter.

1400 The ComponentContext.getServiceReference method MUST return a ServiceReference object typed
1401 by the interface provided by the businessInterface parameter, for the reference named by the
1402 referenceName parameter with the interface defined by the businessInterface parameter when that
1403 reference has a target service configured. [JCA80013]

1404 The ComponentContext.getServiceReference method MUST return null if the multiplicity of the
1405 reference named by the referenceName parameter is 0..1 and the reference has no target service
1406 configured. [JCA80007]

1407 Parameters:

- 1408 • **Class businessInterface** - the Java interface for the service reference
- 1409 • **String referenceName** - the name of the service reference

1410 Exceptions:

- 1411 • The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if
1412 the reference named by the referenceName parameter has multiplicity greater than one. [JCA80004]
- 1413 • The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if
1414 the reference named by the referenceName parameter does not have an interface of the type defined
1415 by the businessInterface parameter. [JCA80005]

- The `ComponentContext.getServiceReference` method MUST throw an `IllegalArgumentException` if the component does not have a reference with the name provided in the `referenceName` parameter. [JCA80006]

`getServices(Class businessInterface, String referenceName)` method:

Returns a list of typed service proxies for a reference defined by the current component, where the reference has multiplicity 0..n or 1..n.

Returns:

- **`Collection`** which is a collection of proxy objects for the reference, one for each target service to which the reference is wired, where each proxy object implements the interface B contained in the `businessInterface` parameter.

The `ComponentContext.getServices` method MUST return a collection containing one proxy object implementing the interface provided by the `businessInterface` parameter for each of the target services configured on the reference identified by the `referenceName` parameter. [JCA80014]

The `ComponentContext.getServices` method MUST return an empty collection if the service reference with the name supplied in the `referenceName` parameter is not wired to any target services.

[JCA80015]

Parameters:

- **`Class businessInterface`** - the Java interface for the service reference
- **`String referenceName`** - the name of the service reference

Exceptions:

- The `ComponentContext.getServices` method MUST throw an `IllegalArgumentException` if the reference identified by the `referenceName` parameter has multiplicity of 0..1 or 1..1. [JCA80016]
- The `ComponentContext.getServices` method MUST throw an `IllegalArgumentException` if the component does not have a reference with the name supplied in the `referenceName` parameter. [JCA80017]
- The `ComponentContext.getServices` method MUST throw an `IllegalArgumentException` if the service reference with the name supplied in the `referenceName` does not have an interface compatible with the interface supplied in the `businessInterface` parameter. [JCA80018]

`getServiceReferences(Class businessInterface, String referenceName)` method:

Returns a list of typed `ServiceReference` objects for a reference defined by the current component, where the reference has multiplicity 0..n or 1..n.

Returns:

- **`Collection<ServiceReference>`** which is a collection of `ServiceReference` objects for the reference, one for each target service to which the reference is wired, where each proxy object implements the interface B contained in the `businessInterface` parameter. The collection is empty if the reference is not wired to any target services.

The `ComponentContext.getServiceReferences` method MUST return a collection containing one `ServiceReference` object typed by the interface provided by the `businessInterface` parameter for each of the target services configured on the reference identified by the `referenceName` parameter. [JCA80019]

The `ComponentContext.getServiceReferences` method MUST return an empty collection if the service reference with the name supplied in the `referenceName` parameter is not wired to any target services. [JCA80020]

Parameters:

- **`Class businessInterface`** - the Java interface for the service reference

1463 • **String referenceName** - the name of the service reference

1464 Exceptions:

- 1465 • The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if
1466 the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1. [JCA80021]
- 1467 • The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if
1468 the component does not have a reference with the name supplied in the referenceName parameter.
1469 [JCA80022]
- 1470 • The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if
1471 the service reference with the name supplied in the referenceName does not have an interface
1472 compatible with the interface supplied in the businessInterface parameter. [JCA80023]

1473

1474 **createSelfReference(Class businessInterface) method:**

1475 Returns a ServiceReference object that can be used to invoke this component over the designated
1476 service.

1477 Returns:

- 1478 • **ServiceReference** which is a ServiceReference object for the service of this component which
1479 has the supplied business interface. If the component has multiple services with the same business
1480 interface the SCA runtime can return a ServiceReference for any one of them.
- 1481 The ComponentContext.createSelfReference method MUST return a ServiceReference object typed
1482 by the interface defined by the businessInterface parameter for one of the services of the invoking
1483 component which has the interface defined by the businessInterface parameter. [JCA80024]

1484 Parameters:

- 1485 • **Class businessInterface** - the Java interface for the service

1486 Exceptions:

- 1487 • The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if
1488 the component does not have a service which implements the interface identified by the
1489 businessInterface parameter. [JCA80025]

1490

1491 **createSelfReference(Class businessInterface, String serviceName) method:**

1492 Returns a ServiceReference that can be used to invoke this component over the designated service. The
1493 serviceName parameter explicitly declares the service name to invoke

1494 Returns:

- 1495 • **ServiceReference** which is a ServiceReference proxy object for the reference, which implements
1496 the interface contained in the businessInterface parameter.
- 1497 The ComponentContext.createSelfReference method MUST return a ServiceReference object typed
1498 by the interface defined by the businessInterface parameter for the service identified by the
1499 serviceName of the invoking component and which has the interface defined by the businessInterface
1500 parameter. [JCA80026]

1501 Parameters:

- 1502 • **Class businessInterface** - the Java interface for the service reference
- 1503 • **String serviceName** - the name of the service reference

1504 Exceptions:

- 1505 • The ComponentContext.createSelfReference method MUST throw an IllegalArgumentException if the
1506 component does not have a service with the name identified by the serviceName parameter.
1507 [JCA80027]

- 1508 • The `ComponentContext.createSelfReference` method MUST throw an `IllegalArgumentException` if the
1509 component service with the name identified by the `serviceName` parameter does not implement a
1510 business interface which is compatible with the supplied `businessInterface` parameter. [JCA80028]

1511

1512 ***getProperty(Class type, String propertyName) method:***

1513 Returns the value of an SCA property defined by this component.

1514 Returns:

- 1515 • **** which is an object of the type identified by the type parameter containing the value specified for
1516 the property in the SCA configuration of the component. **null** if the SCA configuration of the
1517 component does not specify any value for the property.

1518 The `ComponentContext.getProperty` method MUST return an object of the type identified by the type
1519 parameter containing the value specified in the component configuration for the property named by
1520 the `propertyName` parameter or null if no value is specified in the configuration. [JCA80029]

1521 Parameters:

- 1522 • **Class type** - the Java class of the property (Object mapped type for primitive Java types - e.g.
1523 Integer if the type is int)
- 1524 • **String propertyName** - the name of the property

1525 Exceptions:

- 1526 • The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the
1527 component does not have a property with the name identified by the `propertyName` parameter.
1528 [JCA80030]
- 1529 • The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the
1530 component property with the name identified by the `propertyName` parameter does not have a type
1531 which is compatible with the supplied type parameter. [JCA80031]

1532

1533 ***getRequestContext() method:***

1534 Returns the `RequestContext` for the current SCA service request.

1535 Returns:

- 1536 • **RequestContext** which is the `RequestContext` object for the current SCA service invocation. **null** if
1537 there is no current request or if the context is unavailable.

1538 The `ComponentContext.getRequestContext` method MUST return non-null when invoked during the
1539 execution of a Java business method for a service operation or a callback operation, on the same
1540 thread that the SCA runtime provided, and MUST return null in all other cases. [JCA80002]

1541 Parameters:

- 1542 • **none**

1543 Exceptions:

- 1544 • **none**

1545

1546 ***cast(B target) method:***

1547 Casts a type-safe reference to a `ServiceReference`

1548 Returns:

- 1549 • **ServiceReference** which is a `ServiceReference` object which implements the same business
1550 interface B as a reference proxy object

1551 The `ComponentContext.cast` method MUST return a `ServiceReference` object which is typed by the
1552 same business interface as specified by the reference proxy object supplied in the target parameter.
1553 [JCA80032]

Parameters:

- **B target** - a type safe reference proxy object which implements the business interface B

Exceptions:

- The `ComponentContext.cast` method MUST throw an `IllegalArgumentException` if the supplied target parameter is not an SCA reference proxy object. [JCA80033]

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

Snippet 9-1 shows an example of component context usage 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");
}
```

Snippet 9-1: ComponentContext Injection Example

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.

9.2 Request Context

Figure 9-2 shows the **RequestContext** interface:

```
package org.oasisopen.sca;

import javax.security.auth.Subject;

public interface RequestContext {

    Subject getSecuritySubject();

    String getServiceName();
    <CB> ServiceReference<CB> getCallbackReference();
    <CB> CB getCallback();
    <B> ServiceReference<B> getServiceReference();
}
```

Figure 9-2: RequestContext Interface

getSecuritySubject () method:

Returns the JAAS Subject of the current request (see [the JAAS Reference Guide \[JAAS\]](#) for details of JAAS).

Returns:

- **javax.security.auth.Subject** object which is the JAAS subject for the request.
- **null** if there is no subject for the request.

1604 The RequestContext.getSecuritySubject method MUST return the JAAS subject of the current
1605 request, or null if there is no subject or null if the method is invoked from code not processing a
1606 service request or callback request. [JCA80034]

1607 Parameters:

- 1608 • **none**

1609 Exceptions:

- 1610 • **none**

1611

1612 **getServiceName () method:**

1613 Returns the name of the service on the Java implementation the request came in on.

1614 Returns:

- 1615 • **String** containing the name of the service. **null** if the method is invoked from a thread that is not
1616 processing a service operation or a callback operation.

1617 The RequestContext.getServiceName method MUST return the name of the service for which an
1618 operation is being processed, or null if invoked from a thread that is not processing a service
1619 operation or a callback operation. [JCA80035]

1620 Parameters:

- 1621 • **none**

1622 Exceptions:

- 1623 • **none**

1624

1625 **getCallbackReference () method:**

1626 Returns a service reference proxy for the callback for the invoked service operation, as specified by the
1627 service client.

1628 Returns:

- 1629 • **ServiceReference<CB>** which is a service reference for the callback for the invoked service, as
1630 supplied by the service client. It is typed with the callback interface.
1631 **null** if the invoked service has an interface which is not bidirectional or if the getCallbackReference()
1632 method is called during the processing of a callback operation.

1633 **null** if the method is invoked from a thread that is not processing a service operation.

1634 The RequestContext.getCallbackReference method MUST return a ServiceReference object typed by
1635 the interface of the callback supplied by the client of the invoked service, or null if either the invoked
1636 service is not bidirectional or if the method is invoked from a thread that is not processing a service
1637 operation. [JCA80036]

1638 Parameters:

- 1639 • **none**

1640 Exceptions:

- 1641 • **none**

1642

1643 **getCallback () method:**

1644 Returns a proxy for the callback for the invoked service as specified by the service client.

1645 Returns:

- 1646 • **CB** proxy object for the callback for the invoked service as supplied by the service client. It is typed
1647 with the callback interface.

null if the invoked service has an interface which is not bidirectional or if the `getCallback()` method is called during the processing of a callback operation.

null if the method is invoked from a thread that is not processing a service operation.

The `RequestContext.getCallback` method **MUST** return a reference proxy object typed by the interface of the callback supplied by the client of the invoked service, or **null** if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation. [JCA80037]

Parameters:

- **none**

Exceptions:

- **none**

`getServiceReference ()` method:

Returns a `ServiceReference` object for the service that was invoked.

Returns:

- **`ServiceReference`** which is a service reference for the invoked service. It is typed with the interface of the service.

null if the method is invoked from a thread that is not processing a service operation or a callback operation.

When invoked during the execution of a service operation, the `RequestContext.getServiceReference` method **MUST** return a `ServiceReference` that represents the service that was invoked. [JCA80003]

When invoked during the execution of a callback operation, the `RequestContext.getServiceReference` method **MUST** return a `ServiceReference` that represents the callback that was invoked. [JCA80038]

When invoked from a thread not involved in the execution of either a service operation or of a callback operation, the `RequestContext.getServiceReference` method **MUST** return **null**. [JCA80039]

Parameters:

- **none**

Exceptions:

- **none**

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

9.3 ServiceReference Interface

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

Figure 9-3 defines the **`ServiceReference`** interface:

```
package org.oasisopen.sca;

public interface ServiceReference<B> extends java.io.Serializable {

    B getService();
    Class<B> getBusinessInterface();
}
```

Figure 9-3: ServiceReference Interface

getService () method:

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.

Returns:

- **** which is type-safe reference proxy object to the target of this reference. It is typed with the interface of the target service.

The ServiceReference.getService method MUST return a reference proxy object which can be used to invoke operations on the target service of the reference and which is typed with the business interface of the reference. [JCA80040]

Parameters:

- **none**

Exceptions:

- **none**

getBusinessInterface () method:

Returns the Java class for the business interface associated with this ServiceReference.

Returns:

- **Class** which is a Class object of the business interface associated with the reference.

The ServiceReference.getBusinessInterface method MUST return a Class object representing the business interface of the reference. [JCA80041]

Parameters:

- **none**

Exceptions:

- **none**

9.4 ResponseDispatch interface

The **ResponseDispatch** interface is shown in Figure 9-4:

```
package org.oasisopen.sca;

public interface ResponseDispatch<T> {
    void sendResponse(T res);
    void sendFault(Throwable e);
    Map<String, Object> getContext();
}
```

Figure 9-4: ResponseDispatch Interface

sendResponse (T response) method:

Sends the response message from an asynchronous service method. This method can only be invoked once for a given ResponseDispatch object and cannot be invoked if sendFault has previously been invoked for the same ResponseDispatch object.

Returns:

- **void**

The `ResponseDispatch.sendResponse()` method MUST send the response message to the client of an asynchronous service. [JCA50057]

Parameters:

- **T** - an instance of the response message returned by the service operation

Exceptions:

- The `ResponseDispatch.sendResponse()` method MUST throw an `InvalidStateException` if either the `sendResponse` method or the `sendFault` method has already been called once. [JCA80058]

sendFault (Throwable e) method:

Sends an exception as a fault from an asynchronous service method. This method can only be invoked once for a given `ResponseDispatch` object and cannot be invoked if `sendResponse` has previously been invoked for the same `ResponseDispatch` object.

Returns:

- **void**

The `ResponseDispatch.sendFault()` method MUST send the supplied fault to the client of an asynchronous service. [JCA80059]

Parameters:

- **e** - an instance of an exception returned by the service operation

Exceptions:

- The `ResponseDispatch.sendFault()` method MUST throw an `InvalidStateException` if either the `sendResponse` method or the `sendFault` method has already been called once. [JCA80060]

getContext () method:

Obtains the context object for the `ResponseDispatch` method

Returns:

- **Map<String, object>** which is the context object for the `ResponseDispatch` object. The invoker can update the context object with appropriate context information, prior to invoking either the `sendResponse` method or the `sendFault` method

Parameters:

- **none**

Exceptions:

- **none**

9.5 ServiceRuntimeException

Figure 9-5 shows the ***ServiceRuntimeException***.

```
package org.oasisopen.sca;

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

Figure 9-5: *ServiceRuntimeException*

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

9.6 ServiceUnavailableException

Figure 9-6 shows the **ServiceUnavailableException**.

```
package org.oasisopen.sca;

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

Figure 9-6: ServiceUnavailableException

This exception signals problems in the interaction with remote services. These are exceptions that can 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

9.7 InvalidServiceException

Figure 9-7 shows the **InvalidServiceException**.

```
package org.oasisopen.sca;

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

Figure 9-7: InvalidServiceException

This exception signals that the ServiceReference is no longer valid. This can happen when the target of the reference is undeployed. This exception is not transient and therefore is unlikely to be resolved by retrying the operation and will most likely require human intervention.

9.8 Constants

The SCA **Constants** interface defines a number of constant values that are used in the SCA Java APIs and Annotations. Figure 9-8 shows the Constants interface:

```
package org.oasisopen.sca;

public interface Constants {

    String SCA_NS = "http://docs.oasis-open.org/ns/opencsa/sca/200912";

    String SCA_PREFIX = "{" + SCA_NS + "}";

    String SERVERAUTHENTICATION = SCA_PREFIX + "serverAuthentication";
    String CLIENTAUTHENTICATION = SCA_PREFIX + "clientAuthentication";
    String ATLEASTONCE = SCA_PREFIX + "atLeastOnce";
    String ATMOSTONCE = SCA_PREFIX + "atMostOnce";
    String EXACTLYONCE = SCA_PREFIX + "exactlyOnce";
    String ORDERED = SCA_PREFIX + "ordered";
    String TRANSACTEDONEWAY = SCA_PREFIX + "transactedOneWay";
    String IMMEDIATEONEWAY = SCA_PREFIX + "immediateOneWay";
    String PROPAGATESTransaction = SCA_PREFIX + "propagatesTransaction";
    String SUSPENDSTransaction = SCA_PREFIX + "suspendsTransaction";
    String ASYNCINVOCATION = SCA_PREFIX + "asyncInvocation";
    String SOAP = SCA_PREFIX + "SOAP";
}
```

```

String JMS = SCA_PREFIX + "JMS";
String NOLISTENER = SCA_PREFIX + "noListener";
String EJB = SCA_PREFIX + "EJB";
}

```

Figure 9-8: Constants Interface

9.9 SCAClientFactory Class

The SCAClientFactory class provides the means for client code to obtain a proxy reference object for a service within an SCA Domain, through which the client code can invoke operations of that service. This is particularly useful for client code that is running outside the SCA Domain containing the target service, for example where the code is "unmanaged" and is not running under an SCA runtime.

The SCAClientFactory is an abstract class which provides a set of static newInstance(...) methods which the client can invoke in order to obtain a concrete object implementing the SCAClientFactory interface for a particular SCA Domain. The returned SCAClientFactory object provides a getService() method which provides the client with the means to obtain a reference proxy object for a service running in the SCA Domain.

The SCAClientFactory class is shown in Figure 9-9:

```

package org.oasisopen.sca.client;

import java.net.URI;
import java.util.Properties;

import org.oasisopen.sca.NoSuchDomainException;
import org.oasisopen.sca.NoSuchServiceException;
import org.oasisopen.sca.client.SCAClientFactoryFinder;
import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;

public abstract class SCAClientFactory {

    protected static SCAClientFactoryFinder factoryFinder;

    private URI domainURI;

    private SCAClientFactory() {
    }

    protected SCAClientFactory(URI domainURI)
        throws NoSuchDomainException {
        this.domainURI = domainURI;
    }

    protected URI getDomainURI() {
        return domainURI;
    }

    public static SCAClientFactory newInstance( URI domainURI )
        throws NoSuchDomainException {
        return newInstance(null, null, domainURI);
    }

    public static SCAClientFactory newInstance(Properties properties,
                                                URI domainURI)
        throws NoSuchDomainException {
        return newInstance(properties, null, domainURI);
    }

    public static SCAClientFactory newInstance(ClassLoader classLoader,

```

```

1890                                     URI domainURI)
1891     throws NoSuchDomainException {
1892     return newInstance(null, classLoader, domainURI);
1893 }
1894
1895 public static SCAClientFactory newInstance(Properties properties,
1896                                     ClassLoader classLoader,
1897                                     URI domainURI)
1898     throws NoSuchDomainException {
1899     final SCAClientFactoryFinder finder =
1900         factoryFinder != null ? factoryFinder :
1901             new SCAClientFactoryFinderImpl();
1902     final SCAClientFactory factory
1903         = finder.find(properties, classLoader, domainURI);
1904     return factory;
1905 }
1906
1907 public abstract <T> T getService(Class<T> interfaze, String serviceURI)
1908     throws NoSuchServiceException, NoSuchDomainException;
1909 }

```

Figure 9-9: SCAClientFactory Class

newInstance (URI domainURI) method:

Obtains a object implementing the SCAClientFactory class.

Returns:

- **object** which implements the SCAClientFactory class

The SCAClientFactory.newInstance(URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter. [JCA80042]

Parameters:

- **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

Exceptions:

- The SCAClientFactory.newInstance(URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain. [JCA80043]

newInstance(Properties properties, URI domainURI) method:

Obtains a object implementing the SCAClientFactory class, using a specified set of properties.

Returns:

- **object** which implements the SCAClientFactory class

The SCAClientFactory.newInstance(Properties, URI) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter. [JCA80044]

Parameters:

- **properties** - a set of Properties that can be used when creating the object which implements the SCAClientFactory class.
- **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

Exceptions:

- The SCAClientFactory.newInstance(Properties, URI) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain. [JCA80045]

1940 ***newInstance(Classloader classLoader, URI domainURI) method:***

1941 Obtains a object implementing the SCAClientFactory class using a specified classloader.

1942 Returns:

1943 • **object** which implements the SCAClientFactory class

1944 The SCAClientFactory.newInstance(Classloader, URI) method MUST return an object which

1945 implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.

1946 [JCA80046]

1947 Parameters:

1948 • **classLoader** - a ClassLoader to use when creating the object which implements the

1949 SCAClientFactory class.

1950 • **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

1951 Exceptions:

1952 • The SCAClientFactory.newInstance(Classloader, URI) method MUST throw a

1953 NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.

1954 [JCA80047]

1955

1956 ***newInstance(Properties properties, Classloader classLoader, URI domainURI) method:***

1957 Obtains a object implementing the SCAClientFactory class using a specified set of properties and a

1958 specified classloader.

1959 Returns:

1960 • **object** which implements the SCAClientFactory class

1961 The SCAClientFactory.newInstance(Properties, Classloader, URI) method MUST return an object

1962 which implements the SCAClientFactory class for the SCA Domain identified by the domainURI

1963 parameter. [JCA80048]

1964 Parameters:

1965 • **properties** - a set of Properties that can be used when creating the object which implements the

1966 SCAClientFactory class.

1967 • **classLoader** - a ClassLoader to use when creating the object which implements the

1968 SCAClientFactory class.

1969 • **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

1970 Exceptions:

1971 • The SCAClientFactory.newInstance(Properties, Classloader, URI) MUST throw a

1972 NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.

1973 [JCA80049]

1974

1975 ***getService(Class<T> interfaze, String serviceURI) method:***

1976 Obtains a proxy reference object for a specified target service in a specified SCA Domain.

1977 Returns:

1978 • **<T>** a proxy object which implements the business interface T

1979 Invocations of a business method of the proxy causes the invocation of the corresponding operation

1980 of the target service.

1981 The SCAClientFactory.getService method MUST return a proxy object which implements the

1982 business interface defined by the interfaze parameter and which can be used to invoke operations on

1983 the service identified by the serviceURI parameter. [JCA80050]

1984 Parameters:

1985 • **interfaze** - a Java interface class which is the business interface of the target service

- **serviceURI** - a String containing the relative URI of the target service within its SCA Domain.
Takes the form componentName/serviceName or can also take the extended form
componentName/serviceName/bindingName to use a specific binding of the target service

Exceptions:

- The `SCAClientFactory.getService` method MUST throw a `NoSuchServiceException` if a service with the relative URI `serviceURI` and a business interface which matches interface cannot be found in the SCA Domain targeted by the `SCAClient` object. [JCA80051]

SCAClientFactory (URI) method: a single argument constructor that must be available on all concrete subclasses of `SCAClientFactory`. The URI required is the URI of the Domain targeted by the `SCAClientFactory`

getDomainURI() method:

Obtains the Domain URI value for this `SCAClientFactory`

Returns:

- **URI** of the target SCA Domain for this `SCAClientFactory`

The `SCAClientFactory.getDomainURI` method MUST return the SCA Domain URI of the Domain associated with the `SCAClientFactory` object. [JCA80053]

Parameters:

- **none**

Exceptions:

- **none**

private SCAClientFactory() method:

This private no-argument constructor prevents instantiation of an `SCAClientFactory` instance without the use of the constructor with an argument, even by subclasses of the abstract `SCAClientFactory` class.

factoryFinder protected field:

Provides a means by which a provider of an `SCAClientFactory` implementation can inject a factory finder implementation into the abstract `SCAClientFactory` class - once this is done, future invocations of the `SCAClientFactory` use the injected factory finder to locate and return an instance of a subclass of `SCAClientFactory`.

9.10 SCAClientFactoryFinder Interface

The `SCAClientFactoryFinder` interface is a Service Provider Interface representing a `SCAClientFactory` finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can create alternative implementations of this interface that use different class loading or lookup mechanisms:

```
package org.oasisopen.sca.client;

public interface SCAClientFactoryFinder {

    SCAClientFactory find(Properties properties,
                          ClassLoader classLoader,
                          URI domainURI )
        throws NoSuchDomainException ;

}
```


Figure 9-10: SCAClientFactoryFinder Interface

find (Properties properties, ClassLoader classloader, URI domainURI) method:

Obtains an implementation of the SCAClientFactory interface.

Returns:

- **SCAClientFactory** implementation object

The implementation of the SCAClientFactoryFinder.find method MUST return an object which is an implementation of the SCAClientFactory interface, for the SCA Domain represented by the domainURI parameter, using the supplied properties and classloader. [JCA80055]

Parameters:

- **properties** - a set of Properties that can be used when creating the object which implements the SCAClientFactory interface.
- **classLoader** - a ClassLoader to use when creating the object which implements the SCAClientFactory interface.
- **domainURI** - a URI for the SCA Domain targeted by the SCAClientFactory

Exceptions:

- The implementation of the SCAClientFactoryFinder.find method MUST throw a ServiceRuntimeException if the SCAClientFactory implementation could not be found. [JCA80056]

9.11 SCAClientFactoryFinderImpl Class

This class is a default implementation of an SCAClientFactoryFinder, which is used to find an implementation of an SCAClientFactory subclass, as used to obtain an SCAClient object for use by a client. SCA runtime providers can replace this implementation with their own version.

```
package org.oasisopen.sca.client.impl;

public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {
    ...
    public SCAClientFactoryFinderImpl() {...}

    public SCAClientFactory find(Properties properties,
                                ClassLoader classLoader
                                URI domainURI)
    throws NoSuchDomainException, ServiceRuntimeException {...}
    ...
}
```

Snippet 9-2: SCAClientFactoryFinderImpl Class

SCAClientFactoryFinderImpl () method:

Public constructor for the SCAClientFactoryFinderImpl.

Returns:

- **SCAClientFactoryFinderImpl** which implements the SCAClientFactoryFinder interface

Parameters:

- **none**

Exceptions:

- **none**

find (Properties, ClassLoader, URI) method:

Obtains an implementation of the SCAClientFactory interface. It discovers a provider's SCAClientFactory implementation by referring to the following information in this order:

1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the newInstance() method call if specified
2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

Returns:

- **SCAClientFactory** implementation object

Parameters:

- **properties** - a set of Properties that can be used when creating the object which implements the SCAClientFactory interface.
- **classLoader** - a ClassLoader to use when creating the object which implements the SCAClientFactory interface.
- **domainURI** - a URI for the SCA Domain targeted by the SCAClientFactory

Exceptions:

- **ServiceRuntimeException** - if the SCAClientFactory implementation could not be found

9.12 NoSuchDomainException

Figure 9-11 shows the **NoSuchDomainException**:

```
package org.oasisopen.sca;

public class NoSuchDomainException extends Exception {
    ...
}
```

Figure 9-11: NoSuchDomainException Class

This exception indicates that the Domain specified could not be found.

9.13 NoSuchServiceException

Figure 9-12 shows the **NoSuchServiceException**:

```
package org.oasisopen.sca;

public class NoSuchServiceException extends Exception {
    ...
}
```

Figure 9-12: NoSuchServiceException Class

This exception indicates that the service specified could not be found.

10 Java Annotations

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

This specification places constraints on some annotations that are not detectable by a Java compiler. For example, the definition of the `@Property` and `@Reference` annotations indicate that they are allowed on parameters, but the sections "`@Property`" and "`@Reference`" constrain those definitions to constructor parameters. An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code. [JCA90001]

SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an SCA annotation on a static method or a static field of an implementation class and the SCA runtime MUST NOT instantiate such an implementation class. [JCA90002]

10.1 @AllowsPassByReference

Figure 10-1 defines the `@AllowsPassByReference` annotation:

```
package org.oasisopen.sca.annotation;

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

    boolean value() default true;
}
```

Figure 10-1: `AllowsPassByReference` Annotation

The `@AllowsPassByReference` annotation allows service method implementations and client references to be marked as “allows pass by reference” to indicate that they use input parameters, return values and exceptions in a manner that allows the SCA runtime to avoid the cost of copying mutable objects when a remotable service is called locally within the same JVM.

The `@AllowsPassByReference` annotation has the attribute:

- **value** – specifies whether the “allows pass by reference” marker applies to the service implementation class, service implementation method, or client reference to which this annotation applies; if not specified, defaults to true.

The `@AllowsPassByReference` annotation MUST only annotate the following locations:

- a service implementation class
- an individual method of a remotable service implementation
- an individual reference which uses a remotable interface, where the reference is a field, a setter method, or a constructor parameter [JCA90052]

The “allows pass by reference” marking of a method implementation of a remotable service is determined as follows:

1. If the method has an `@AllowsPassByReference` annotation, the method is marked “allows pass by reference” if and only if the value of the method’s annotation is true.
 2. Otherwise, if the class has an `@AllowsPassByReference` annotation, the method is marked “allows pass by reference” if and only if the value of the class’s annotation is true.
 3. Otherwise, the method is not marked “allows pass by reference”.
- The “allows pass by reference” marking of a reference for a remotable service is determined as follows:
1. If the reference has an `@AllowsPassByReference` annotation, the reference is marked “allows pass by reference” if and only if the value of the reference’s annotation is true.
 2. Otherwise, if the service implementation class containing the reference has an `@AllowsPassByReference` annotation, the reference is marked “allows pass by reference” if and only if the value of the class’s annotation is true.
 3. Otherwise, the reference is not marked “allows pass by reference”.
- Snippet 10-1 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) {
    ...
}
```

Snippet 10-1: Use of @AllowsPassByReference on a Method

Snippet 10-2 shows a sample where `@AllowsPassByReference` is defined for a client reference of a Java component implementation class.

```
@AllowsPassByReference
@Reference
private StockQuoteService stockQuote;
```

Snippet 10-2: Use of @AllowsPassByReference on a Reference

10.2 @AsyncFault

Figure 10-2 defines the **@AsyncFault** annotation:

```
package org.oasisopen.sca.annotation;

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

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

@Inherited
@Target({METHOD})
@Retention(RUNTIME)
public @interface AsyncFault {

    Class<?>[] value() default {};

}
```

Figure 10-2: AsyncFault Annotation

2214 The **@AsyncFault** annotation is used to indicate the faults/exceptions which are returned by the
2215 asynchronous service method which it annotates.

2216 10.3 @AsyncInvocation

2217 Figure 10-3 defines the **@AsyncInvocation** annotation, which is used to attach the "asyncInvocation"
2218 policy intent to an interface or to a method:

2219

```
2220 package org.oasisopen.sca.annotation;  
2221  
2222 import static java.lang.annotation.ElementType.METHOD;  
2223 import static java.lang.annotation.ElementType.TYPE;  
2224 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2225 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2226  
2227 import java.lang.annotation.Inherited;  
2228 import java.lang.annotation.Retention;  
2229 import java.lang.annotation.Target;  
2230  
2231 @Inherited  
2232 @Target({TYPE, METHOD})  
2233 @Retention(RUNTIME)  
2234 @Intent(AsyncInvocation.ASYNCINVOCATION)  
2235 public @interface AsyncInvocation {  
2236     String ASYNCINVOCATION = SCA_PREFIX + "asyncInvocation";  
2237  
2238     boolean value() default true;  
2239 }
```

2240 *Figure 10-3: AsyncInvocation Annotation*

2241

2242 The **@AsyncInvocation** annotation is used to indicate that the operations of a Java interface uses the
2243 long-running request-response pattern as described in the SCA Assembly specification.

2244 10.4 @Authentication

2245 The following Java code defines the **@Authentication** annotation:

2246

```
2247 package org.oasisopen.sca.annotation;  
2248  
2249 import static java.lang.annotation.ElementType.FIELD;  
2250 import static java.lang.annotation.ElementType.METHOD;  
2251 import static java.lang.annotation.ElementType.PARAMETER;  
2252 import static java.lang.annotation.ElementType.TYPE;  
2253 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2254 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2255  
2256 import java.lang.annotation.Inherited;  
2257 import java.lang.annotation.Retention;  
2258 import java.lang.annotation.Target;  
2259  
2260 @Inherited  
2261 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2262 @Retention(RUNTIME)  
2263 @Intent(Authentication.AUTHENTICATION)  
2264 public @interface Authentication {  
2265     String AUTHENTICATION = SCA_PREFIX + "authentication";  
2266     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";  
2267     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";  
2268 }
```

```

2268
2269     /**
2270      * List of authentication qualifiers (such as "message"
2271      * or "transport").
2272      *
2273      * @return authentication qualifiers
2274      */
2275     @Qualifier
2276     String[] value() default "";
2277 }

```

2278 *Figure 10-4: Authentication Annotation*

2279

2280 The **@Authentication** annotation is used to indicate the need for authentication. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

2283 10.5 @Authorization

2284 Figure 10-5 defines the @Authorization annotation:

```

2285
2286 package org.oasisopen.sca.annotation;
2287
2288 import static java.lang.annotation.ElementType.FIELD;
2289 import static java.lang.annotation.ElementType.METHOD;
2290 import static java.lang.annotation.ElementType.PARAMETER;
2291 import static java.lang.annotation.ElementType.TYPE;
2292 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2293 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2294
2295 import java.lang.annotation.Inherited;
2296 import java.lang.annotation.Retention;
2297 import java.lang.annotation.Target;
2298
2299 /**
2300  * The @Authorization annotation is used to indicate that
2301  * an authorization policy is required.
2302  */
2303 @Inherited
2304 @Target({TYPE, FIELD, METHOD, PARAMETER})
2305 @Retention(RUNTIME)
2306 @Intent(Authorization.AUTHORIZATION)
2307 public @interface Authorization {
2308     String AUTHORIZATION = SCA_PREFIX + "authorization";
2309 }

```

2310 *Figure 10-5: Authorization Annotation*

2311

2312 The **@Authorization** annotation is used to indicate the need for an authorization policy. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

2315 10.6 @Callback

2316 Figure 10-6 defines the **@Callback** annotation:

```

2317
2318 package org.oasisopen.sca.annotation;
2319

```

```

2320 import static java.lang.annotation.ElementType.FIELD;
2321 import static java.lang.annotation.ElementType.METHOD;
2322 import static java.lang.annotation.ElementType.TYPE;
2323 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2324 import java.lang.annotation.Retention;
2325 import java.lang.annotation.Target;
2326
2327 @Target({TYPE, METHOD, FIELD})
2328 @Retention(RUNTIME)
2329 public @interface Callback {
2330
2331     Class<?> value() default Void.class;
2332 }

```

Figure 10-6: Callback Annotation

The @Callback annotation is used to annotate a service interface or to annotate a Java class (used to define an interface) with a callback interface by specifying the Java class object of the callback interface as an attribute.

The @Callback annotation has the attribute:

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

The @Callback annotation can also be used to annotate a method or a field of an SCA implementation class, in order to have a callback object injected. When used to annotate a method or a field of an implementation class for injection of a callback object, the @Callback annotation MUST NOT specify any attributes. [JCA90046] When used to annotate a method or a field of an implementation class for injection of a callback object, the type of the method or field MUST be the callback interface of at least one bidirectional service offered by the implementation class. [JCA90054] When used to annotate a setter method or a field of an implementation class for injection of a callback object, the SCA runtime MUST inject a callback reference proxy into that method or field when the Java class is initialized, if the component is invoked via a service which has a callback interface and where the type of the setter method or field corresponds to the type of the callback interface. [JCA90058]

The @Callback annotation MUST NOT appear on a setter method or a field of a Java implementation class that has COMPOSITE scope. [JCA90057]

Snippet 10-3 shows an example use of the @Callback annotation to declare a callback interface.

```

2354 package somepackage;
2355 import org.oasisopen.sca.annotation.Callback;
2356 import org.oasisopen.sca.annotation.Remotable;
2357 @Remotable
2358 @Callback(MyServiceCallback.class)
2359 public interface MyService {
2360
2361     void someMethod(String arg);
2362 }
2363
2364 @Remotable
2365 public interface MyServiceCallback {
2366
2367     void receiveResult(String result);
2368 }

```

Snippet 10-3: Use of @Callback

The implied component type is for Snippet 10-3 is shown in Snippet 10-4.

```
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" >
  <service name="MyService">
    <interface.java interface="somepackage.MyService"
      callbackInterface="somepackage.MyServiceCallback"/>
  </service>
</componentType>
```

Snippet 10-4: Implied componentType for Snippet 10-3

10.7 @ComponentName

Figure 10-7 defines the **@ComponentName** annotation:

```
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.FIELD;
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, FIELD})
@Retention(RUNTIME)
public @interface ComponentName {
}
```

Figure 10-7: ComponentName Annotation

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

Snippet 10-5 shows a component name field definition sample.

```
@ComponentName
private String componentName;
```

Snippet 10-5: Use of @ComponentName on a Field

Snippet 10-6 shows a component name setter method sample.

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

Snippet 10-6: Use of @ComponentName on a Setter

10.8 @Confidentiality

Figure 10-8 defines the **@Confidentiality** annotation:

```
package org.oasisopen.sca.annotation;
```



```

2418
2419 import static java.lang.annotation.ElementType.FIELD;
2420 import static java.lang.annotation.ElementType.METHOD;
2421 import static java.lang.annotation.ElementType.PARAMETER;
2422 import static java.lang.annotation.ElementType.TYPE;
2423 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2424 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2425
2426 import java.lang.annotation.Inherited;
2427 import java.lang.annotation.Retention;
2428 import java.lang.annotation.Target;
2429
2430 @Inherited
2431 @Target({TYPE, FIELD, METHOD, PARAMETER})
2432 @Retention(RUNTIME)
2433 @Intent(Confidentiality.CONFIDENTIALITY)
2434 public @interface Confidentiality {
2435     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
2436     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
2437     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
2438
2439     /**
2440      * List of confidentiality qualifiers such as "message" or
2441      * "transport".
2442      *
2443      * @return confidentiality qualifiers
2444      */
2445     @Qualifier
2446     String[] value() default "";
2447 }

```

Figure 10-8: Confidentiality Annotation

The **@Confidentiality** annotation is used to indicate the need for confidentiality. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

10.9 @Constructor

Figure 10-9 defines the **@Constructor** annotation:

```

2456 package org.oasisopen.sca.annotation;
2457
2458 import static java.lang.annotation.ElementType.CONSTRUCTOR;
2459 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2460 import java.lang.annotation.Retention;
2461 import java.lang.annotation.Target;
2462
2463 @Target(CONSTRUCTOR)
2464 @Retention(RUNTIME)
2465 public @interface Constructor { }

```

Figure 10-9: Constructor Annotation

The **@Constructor** annotation is used to mark a particular constructor to use when instantiating a Java component implementation. If a constructor of an implementation class is annotated with **@Constructor** and the constructor has parameters, each of these parameters MUST have either a **@Property** annotation or a **@Reference** annotation. [JCA90003]

Snippet 10-7 shows a sample for the `@Constructor` annotation.

```
public class HelloServiceImpl implements HelloService {  
    public HelloServiceImpl() {  
        ...  
    }  
  
    @Constructor  
    public HelloServiceImpl(@Property(name="someProperty")  
        String someProperty ) {  
        ...  
    }  
  
    public String hello(String message) {  
        ...  
    }  
}
```

Snippet 10-7: Use of `@Constructor`

10.10 @Context

Figure 10-10 defines the `@Context` annotation:

```
package org.oasisopen.sca.annotation;  
  
import static java.lang.annotation.ElementType.FIELD;  
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, FIELD})  
@Retention(RUNTIME)  
public @interface Context {  
}
```

Figure 10-10: Context Annotation

The `@Context` annotation is used to denote 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.

Snippet 10-8 shows a `ComponentContext` field definition sample.

```
@Context  
protected ComponentContext context;
```

Snippet 10-8: Use of `@Context` for a `ComponentContext`

Snippet 10-9 shows a `RequestContext` field definition sample.

```
@Context
```

2523 protected RequestContext context;

2524 *Snippet 10-9: Use of @Context for a RequestContext*

2525 10.11 @Destroy

2526 Figure 10-11 defines the **@Destroy** annotation:

2527

```
2528   package org.oasisopen.sca.annotation;  
2529  
2530   import static java.lang.annotation.ElementType.METHOD;  
2531   import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2532   import java.lang.annotation.Retention;  
2533   import java.lang.annotation.Target;  
2534  
2535   @Target (METHOD)  
2536   @Retention (RUNTIME)  
2537   public @interface Destroy {  
2538  
2539   }
```

2540 *Figure 10-11: Destroy Annotation*

2541

2542 The @Destroy annotation is used to denote a single Java class method that will be called when the scope
2543 defined for the implementation class ends. A method annotated with @Destroy can have any access
2544 modifier and MUST have a void return type and no arguments. [JCA90004]

2545 If there is a method annotated with @Destroy that matches the criteria for the annotation, the SCA
2546 runtime MUST call the annotated method when the scope defined for the implementation class ends.
2547 [JCA90005]

2548 Snippet 10-10 shows a sample for a destroy method definition.

2549

```
2550   @Destroy  
2551   public void myDestroyMethod() {  
2552       ...  
2553   }
```

2554 *Snippet 10-10: Use of @Destroy*

2555 10.12 @EagerInit

2556 Figure 10-12: EagerInit Annotation defines the **@EagerInit** annotation:

2557

```
2558   package org.oasisopen.sca.annotation;  
2559  
2560   import static java.lang.annotation.ElementType.TYPE;  
2561   import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2562   import java.lang.annotation.Retention;  
2563   import java.lang.annotation.Target;  
2564  
2565   @Target (TYPE)  
2566   @Retention (RUNTIME)  
2567   public @interface EagerInit {  
2568  
2569   }
```

2570 *Figure 10-12: EagerInit Annotation*

The **@EagerInit** annotation is used to mark the Java class of a COMPOSITE scoped implementation for eager initialization. When marked for eager initialization with an @EagerInit annotation, the composite scoped instance MUST be created when its containing component is started. [JCA90007]

10.13 @Init

Figure 10-13: Init Annotation defines the **@Init** annotation:

```
package org.oasisopen.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;

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

Figure 10-13: Init Annotation

The @Init annotation is used to denote a single Java class method that is called when the scope defined for the implementation class starts. A method marked with the @Init annotation can have any access modifier and MUST have a void return type and no arguments. [JCA90008]

If there is a method annotated with @Init that matches the criteria for the annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete. [JCA90009]

Snippet 10-11 shows an example of an init method definition.

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

Snippet 10-11: Use of @Init

10.14 @Integrity

Figure 10-14 defines the **@Integrity** annotation:

```
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import static org.oasisopen.sca.Constants.SCA_PREFIX;

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

```

2621 @Inherited
2622 @Target({TYPE, FIELD, METHOD, PARAMETER})
2623 @Retention(RUNTIME)
2624 @Intent(Integrity.INTEGRITY)
2625 public @interface Integrity {
2626     String INTEGRITY = SCA_PREFIX + "integrity";
2627     String INTEGRITY_MESSAGE = INTEGRITY + ".message";
2628     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
2629
2630     /**
2631      * List of integrity qualifiers (such as "message" or "transport").
2632      *
2633      * @return integrity qualifiers
2634      */
2635     @Qualifier
2636     String[] value() default "";
2637 }

```

Figure 10-14: Integrity Annotation

The **@Integrity** annotation is used to indicate that the invocation requires integrity (i.e. no tampering of the messages between client and service). See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

10.15 @Intent

Figure 10-15 defines the **@Intent** annotation:

```

2647 package org.oasisopen.sca.annotation;
2648
2649 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
2650 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2651 import java.lang.annotation.Retention;
2652 import java.lang.annotation.Target;
2653
2654 @Target({ANNOTATION_TYPE})
2655 @Retention(RUNTIME)
2656 public @interface Intent {
2657     /**
2658      * The qualified name of the intent, in the form defined by
2659      * {@link javax.xml.namespace.QName#toString}.
2660      * @return the qualified name of the intent
2661      */
2662     String value() default "";
2663
2664     /**
2665      * The XML namespace for the intent.
2666      * @return the XML namespace for the intent
2667      */
2668     String targetNamespace() default "";
2669
2670     /**
2671      * The name of the intent within its namespace.
2672      * @return name of the intent within its namespace
2673      */
2674     String localPart() default "";
2675 }

```

Figure 10-15: Intent Annotation

2677
2678 The @Intent annotation is used for the creation of new annotations for specific intents. It is not expected
2679 that the @Intent annotation will be used in application code.
2680 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to define new
2681 intent annotations.

2682 10.16 @ManagedSharedTransaction

2683 Figure 10-16 defines the @ManagedSharedTransaction annotation:
2684

```
2685 package org.oasisopen.sca.annotation;  
2686  
2687 import static java.lang.annotation.ElementType.FIELD;  
2688 import static java.lang.annotation.ElementType.METHOD;  
2689 import static java.lang.annotation.ElementType.PARAMETER;  
2690 import static java.lang.annotation.ElementType.TYPE;  
2691 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2692 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2693  
2694 import java.lang.annotation.Inherited;  
2695 import java.lang.annotation.Retention;  
2696 import java.lang.annotation.Target;  
2697  
2698 /**  
2699  * The @ManagedSharedTransaction annotation is used to indicate that  
2700  * a distributed ACID transaction is required.  
2701  */  
2702 @Inherited  
2703 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2704 @Retention(RUNTIME)  
2705 @Intent(ManagedSharedTransaction.MANAGEDSHAREDTRANSACTION)  
2706 public @interface ManagedSharedTransaction {  
2707     String MANAGEDSHAREDTRANSACTION = SCA_PREFIX + "managedSharedTransaction";  
2708 }
```

2709 *Figure 10-16: ManagedSharedTransaction Annotation*

2710
2711 The **@ManagedSharedTransaction** annotation is used to indicate the need for a distributed and globally
2712 coordinated ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on the
2713 meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent
2714 annotations are used in Java.

2715 10.17 @ManagedTransaction

2716 Figure 10-17 defines the @ManagedTransaction annotation:
2717

```
2718 import static java.lang.annotation.ElementType.FIELD;  
2719 import static java.lang.annotation.ElementType.METHOD;  
2720 import static java.lang.annotation.ElementType.PARAMETER;  
2721 import static java.lang.annotation.ElementType.TYPE;  
2722 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2723 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2724  
2725 import java.lang.annotation.Inherited;  
2726 import java.lang.annotation.Retention;  
2727 import java.lang.annotation.Target;  
2728  
2729 /**
```

```

2730 * The @ManagedTransaction annotation is used to indicate the
2731 * need for an ACID transaction environment.
2732 */
2733 @Inherited
2734 @Target({TYPE, FIELD, METHOD, PARAMETER})
2735 @Retention(RUNTIME)
2736 @Intent(ManagedTransaction.MANAGEDTRANSACTION)
2737 public @interface ManagedTransaction {
2738     String MANAGEDTRANSACTION = SCA_PREFIX + "managedTransaction";
2739     String MANAGEDTRANSACTION_LOCAL = MANAGEDTRANSACTION + ".local";
2740     String MANAGEDTRANSACTION_GLOBAL = MANAGEDTRANSACTION + ".global";
2741
2742     /**
2743      * List of managedTransaction qualifiers (such as "global" or "local").
2744      *
2745      * @return managedTransaction qualifiers
2746      */
2747     @Qualifier
2748     String[] value() default "";
2749 }

```

Figure 10-17: ManagedTransaction Annotation

The **@ManagedTransaction** annotation is used to indicate the need for an ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

10.18 @MutualAuthentication

Figure 10-18 defines the @MutualAuthentication annotation:

```

2758 package org.oasisopen.sca.annotation;
2759
2760 import static java.lang.annotation.ElementType.FIELD;
2761 import static java.lang.annotation.ElementType.METHOD;
2762 import static java.lang.annotation.ElementType.PARAMETER;
2763 import static java.lang.annotation.ElementType.TYPE;
2764 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2765 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2766
2767 import java.lang.annotation.Inherited;
2768 import java.lang.annotation.Retention;
2769 import java.lang.annotation.Target;
2770
2771 /**
2772  * The @MutualAuthentication annotation is used to indicate that
2773  * a mutual authentication policy is needed.
2774  */
2775 @Inherited
2776 @Target({TYPE, FIELD, METHOD, PARAMETER})
2777 @Retention(RUNTIME)
2778 @Intent(MutualAuthentication.MUTUALAUTHENTICATION)
2779 public @interface MutualAuthentication {
2780     String MUTUALAUTHENTICATION = SCA_PREFIX + "mutualAuthentication";
2781 }

```

Figure 10-18: MutualAuthentication Annotation

The **@MutualAuthentication** annotation is used to indicate the need for mutual authentication between a service consumer and a service provider. See the SCA Policy Framework Specification [POLICY] for

2786 details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of
2787 how intent annotations are used in Java.

2788 10.19 @NoManagedTransaction

2789 Figure 10-19 defines the @NoManagedTransaction annotation:

2790

```
2791 package org.oasisopen.sca.annotation;  
2792  
2793 import static java.lang.annotation.ElementType.FIELD;  
2794 import static java.lang.annotation.ElementType.METHOD;  
2795 import static java.lang.annotation.ElementType.PARAMETER;  
2796 import static java.lang.annotation.ElementType.TYPE;  
2797 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2798 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2799  
2800 import java.lang.annotation.Inherited;  
2801 import java.lang.annotation.Retention;  
2802 import java.lang.annotation.Target;  
2803  
2804 /**  
2805  * The @NoManagedTransaction annotation is used to indicate that  
2806  * a non-transactional environment is needed.  
2807  */  
2808 @Inherited  
2809 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2810 @Retention(RUNTIME)  
2811 @Intent(NoManagedTransaction.NOMANAGEDTRANSACTION)  
2812 public @interface NoManagedTransaction {  
2813     String NOMANAGEDTRANSACTION = SCA_PREFIX + "noManagedTransaction";  
2814 }
```

2815 *Figure 10-19: NoManagedTransaction Annotation*

2816

2817 The **@NoManagedTransaction** annotation is used to indicate that the component does not want to run in
2818 an ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on the meaning
2819 of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations
2820 are used in Java.

2821 10.20 @OneWay

2822 Figure 10-20 defines the **@OneWay** annotation:

2823

```
2824 package org.oasisopen.sca.annotation;  
2825  
2826 import static java.lang.annotation.ElementType.METHOD;  
2827 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2828 import java.lang.annotation.Retention;  
2829 import java.lang.annotation.Target;  
2830  
2831 @Target(METHOD)  
2832 @Retention(RUNTIME)  
2833 public @interface OneWay {  
2834  
2835  
2836 }
```

2837 *Figure 10-20: OneWay Annotation*

A method annotated with `@OneWay` MUST have a void return type and MUST NOT have declared checked exceptions. [JCA90055]

When a method of a Java interface is annotated with `@OneWay`, the SCA runtime MUST ensure that all invocations of that method are executed in a non-blocking fashion, as described in the section on Asynchronous Programming. [JCA90056]

The `@OneWay` annotation has no attributes.

Snippet 10-12 shows the use of the `@OneWay` annotation on an interface.

```
package services.hello;

import org.oasisopen.sca.annotation.OneWay;

public interface HelloService {
    @OneWay
    void hello(String name);
}
```

Snippet 10-12: Use of `@OneWay`

10.21 @PolicySets

Figure 10-21 defines the `@PolicySets` annotation:

```
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
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, FIELD, METHOD, PARAMETER})
@Retention(RUNTIME)
public @interface PolicySets {
    /**
     * Returns the policy sets to be applied.
     *
     * @return the policy sets to be applied
     */
    String[] value() default "";
}
```

Figure 10-21: PolicySets Annotation

The `@PolicySets` annotation is used to attach one or more SCA Policy Sets to a Java implementation class or to one of its subelements.

See the [section "Policy Set Annotations"](#) for details and samples.

10.22 @Property

Figure 10-22 defines the `@Property` annotation:

```

2888 package org.oasisopen.sca.annotation;
2889
2890 import static java.lang.annotation.ElementType.FIELD;
2891 import static java.lang.annotation.ElementType.METHOD;
2892 import static java.lang.annotation.ElementType.PARAMETER;
2893 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2894 import java.lang.annotation.Retention;
2895 import java.lang.annotation.Target;
2896
2897 @Target({METHOD, FIELD, PARAMETER})
2898 @Retention(RUNTIME)
2899 public @interface Property {
2900
2901     String name() default "";
2902     boolean required() default true;
2903 }

```

Figure 10-22: Property Annotation

The @Property annotation is used to denote a Java class field, a setter method, or a constructor parameter 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 input parameter of the setter method or constructor.

When the Java type of a field, setter method or constructor parameter with the @Property annotation is a primitive type or a JAXB annotated class, the SCA runtime MUST convert a property value specified by an SCA component definition into an instance of the Java type as defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation enabled. [JCA90061]

When the Java type of a field, setter method or constructor parameter with the @Property annotation is not a JAXB annotated class, the SCA runtime can use any XML to Java mapping when converting property values into instances of the Java type.

The @Property annotation MUST NOT be used on a class field that is declared as final. [JCA90011]

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

The @Property annotation has the attributes:

- **name (0..1)** – the name of the property. For a field annotation, the default is the name of the field of the Java class. For a setter method annotation, the default is the JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a @Property annotation applied to a constructor parameter, there is no default value for the name attribute and the name attribute MUST be present. [JCA90013]
- **required (0..1)** – a boolean value which specifies whether injection of the property value is required or not, where true means injection is required and false means injection is not required. Defaults to true. For a @Property annotation applied to a constructor parameter, the required attribute MUST NOT have the value false. [JCA90014]

Snippet 10-13 shows a property field definition sample.

```

2932 @Property(name="currency", required=true)
2933 protected String currency;
2934
2935 The following snippet shows a property setter sample
2936
2937 @Property(name="currency", required=true)
2938 public void setCurrency( String theCurrency ) {
2939     ....
2940 }

```

Snippet 10-13: Use of @Property on a Field

For a @Property annotation, if the type of the Java class field or the type of the input parameter of the setter method or constructor is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <property/> element with a @many attribute set to true, otherwise @many MUST be set to false. [JCA90047]

Snippet 10-14 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;
}
...
```

Snippet 10-14: Use of @Property with a Collection

10.23 @Qualifier

Figure 10-23 defines the @Qualifier annotation:

```
package org.oasisopen.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;

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

Figure 10-23: Qualifier Annotation

The @Qualifier annotation is applied to an attribute of a specific intent annotation definition, defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the intent has qualifiers. [JCA90015]

See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to define new intent annotations.

10.24 @Reference

Figure 10-24 defines the @Reference annotation:

```
package org.oasisopen.sca.annotation;

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

```

2991 import java.lang.annotation.Retention;
2992 import java.lang.annotation.Target;
2993 @Target({METHOD, FIELD, PARAMETER})
2994 @Retention(RUNTIME)
2995 public @interface Reference {
2996
2997     String name() default "";
2998     boolean required() default true;
2999 }

```

Figure 10-24: Reference Annotation

The @Reference annotation type is used to annotate a Java class field, a setter method, or a constructor parameter 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 input parameter of the setter method or constructor.

The @Reference annotation MUST NOT be used on a class field that is declared as final. [JCA90016]

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

The @Reference annotation has the attributes:

- **name : String (0..1)** – the name of the reference. For a field annotation, the default is the name of the field of the Java class. For a setter method annotation, the default is the JavaBeans property name corresponding to the setter method name. **For a @Reference annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present. [JCA90018]**
- **required (0..1)** – a boolean value which specifies whether injection of the service reference is required or not, where true means injection is required and false means injection is not required. Defaults to true. **For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true. [JCA90019]**

Snippet 10-15 shows a reference field definition sample.

```

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

```

Snippet 10-15: Use of @Reference on a Field

Snippet 10-16 shows a reference setter sample

```

@Reference(name="stockQuote", required=true)
public void setStockQuote( StockQuoteService theSQService ) {
    ...
}

```

Snippet 10-16: Use of @Reference on a Setter

Snippet 10-17 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.

```

package services.hello;

private HelloService helloService;

@Reference(name="helloService", required=true)

```

```

3041 public setHelloService(HelloService service) {
3042     helloService = service;
3043 }
3044
3045 public void clientMethod() {
3046     String result = helloService.hello("Hello World!");
3047     ...
3048 }

```

3049 *Snippet 10-17: Use of @Reference and a ServiceReference*

3050

3051 The presence of a @Reference annotation is reflected in the componentType information that the runtime
3052 generates through reflection on the implementation class. Snippet 10-18 shows the component type for
3053 the component implementation fragment in Snippet 10-17.

3054

```

3055 <?xml version="1.0" encoding="ASCII"?>
3056 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
3057
3058     <!-- Any services offered by the component would be listed here -->
3059     <reference name="helloService" multiplicity="1..1">
3060         <interface.java interface="services.hello.HelloService"/>
3061     </reference>
3062
3063 </componentType>

```

3064 *Snippet 10-18: Implied componentType for Implementation in Snippet 10-17*

3065

3066 If the type of a reference is not an array or any type that extends or implements java.util.Collection, then
3067 the SCA runtime MUST introspect the component type of the implementation with a <reference/> element
3068 with @multiplicity= 0..1 if the @Reference annotation required attribute is false and with
3069 @multiplicity=1..1 if the @Reference annotation required attribute is true. [JCA90020]

3070 If the type of a reference is defined as an array or as any type that extends or implements
3071 java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation
3072 with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is
3073 false and with @multiplicity=1..n if the @Reference annotation required attribute is true. [JCA90021]

3074 Snippet 10-19 shows a sample of a service reference definition using the @Reference annotation on a
3075 java.util.List. The name of the reference is “helloServices” and its type is HelloService. The clientMethod()
3076 calls the “hello” operation of all the services referenced by the helloServices reference. In this case, at
3077 least one HelloService needs to be present, so **required** is true.

3078

```

3079 @Reference(name="helloServices", required=true)
3080 protected List<HelloService> helloServices;
3081
3082 public void clientMethod() {
3083
3084     ...
3085     for (int index = 0; index < helloServices.size(); index++) {
3086         HelloService helloService =
3087         (HelloService)helloServices.get(index);
3088         String result = helloService.hello("Hello World!");
3089     }
3090     ...
3091 }

```

3092 *Snippet 10-19: Use of @Reference with a List of ServiceReferences*

3093

Snippet 10-20 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://docs.oasis-open.org/ns/opencsa/sca/200912">

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

</componentType>
```

Snippet 10-20: Implied componentType for Implementation in Snippet 10-19

An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null [JCA90022] An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection [JCA90023]

10.24.1 Reinjection

References can be reinjected by an SCA runtime after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized.

In order for reinjection to occur, the following need to be true:

1. The component is not STATELESS scoped.
2. The reference needs to use either field-based injection or setter injection. References that are injected through constructor injection cannot be changed.

Setter injection allows for code in the setter method to perform processing in reaction to a change.

If a reference target changes and the reference is not reinjected, the reference needs to continue to work as if the reference target was not changed.

If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime is advised throw an `InvalidServiceException`. Likewise, if an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime is advised throw a `ServiceUnavailableException`. In general, if the target service of the reference is changed, the reference either continues to work or throws an `InvalidServiceException` when it is invoked.

A `ServiceReference` that has been obtained from a reference by `ComponentContext.cast()` corresponds to the reference that is passed as a parameter to `cast()`. If the reference is subsequently reinjected, it is expected that the `ServiceReference` obtained from the original reference continues to work as if the reference target was not changed. If the target of a `ServiceReference` has been undeployed, the SCA runtime is advised to throw a `InvalidServiceException` when an operation is invoked on the `ServiceReference`. If the target of a `ServiceReference` has become unavailable, the SCA runtime is advised to throw a `ServiceUnavailableException` when an operation is invoked on the `ServiceReference`. If the target service of a `ServiceReference` is changed, the reference either continues to work or throws an `InvalidServiceException` when it is invoked.

A reference or `ServiceReference` accessed through the component context by calling `getService()` or `getServiceReference()` is expected to correspond to the current configuration of the domain. This applies whether or not reinjection has taken place. If the target of a reference or `ServiceReference` accessed through the component context by calling `getService()` or `getServiceReference()` has been undeployed or has become unavailable, the result is expected to be a reference to the undeployed or unavailable service, and attempts to call business methods throw an `InvalidServiceException` or a `ServiceUnavailableException`. If the target service of a reference or `ServiceReference` accessed through the component context by calling `getService()` or `getServiceReference()` has changed, the returned value is expected to be a reference to the changed service.

The expected behaviour for reference reinjection also applies to references with a multiplicity of 0..n or 1..n. This means that in the cases where reference reinjection is not allowed, the array or Collection for a reference of multiplicity 0..n or multiplicity 1..n does change its contents when changes occur to the reference wiring or to the targets of the wiring. In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the SCA runtime is expected to call the setter method for any change to the contents. A reinjected array or Collection for a reference is expected to be a different array or Collection object from that previously injected to the component.

<u>Change event</u>	<u>Effect on</u>		
	Injected Reference or ServiceReference	Existing ServiceReference Object**	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	can be reinjected (if other conditions* apply). If not reinjected, then it continues to work as if the reference target was not changed.	continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.
Target service undeployed	Business methods throw InvalidServiceException.	Business methods throw InvalidServiceException.	Result is a reference to the undeployed service. Business methods throw InvalidServiceException.
Target service becomes unavailable	Business methods throw ServiceUnavailableException	Business methods throw ServiceUnavailableException	Result is be a reference to the unavailable service. Business methods throw ServiceUnavailableException.
Target service changed	might continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	might continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	Result is a reference to the changed service.
<p>* Other conditions:</p> <p>The component cannot be STATELESS scoped.</p> <p>The reference has to use either field-based injection or setter injection. References that are injected through constructor injection cannot be changed.</p> <p>** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().</p>			

Table 10-1Reinjection Effects

10.25 @Remotable

Figure 10-25 defines the **@Remotable** annotation:


```

3159 package org.oasisopen.sca.annotation;
3160
3161 import static java.lang.annotation.ElementType.TYPE;
3162 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3163 import java.lang.annotation.Retention;
3164 import java.lang.annotation.Target;
3165
3166
3167 @Target (TYPE,METHOD,FIELD,PARAMETER)
3168 @Retention(RUNTIME)
3169 public @interface Remotable {
3170
3171 }

```

Figure 10-25: Remotable Annotation

The @Remotable annotation is used to indicate that an SCA service interface is remotable. The @Remotable annotation is valid only on a Java interface, a Java class, a field, a setter method, or a constructor parameter. It MUST NOT appear anywhere else. [JCA90053] A remotable service can be published externally as a service and MUST be translatable into a WSDL portType. [JCA90040]

The @Remotable annotation has no attributes. When placed on a Java service interface, it indicates that the interface is remotable. When placed on a Java service implementation class, it indicates that all SCA service interfaces provided by the class (including the class itself, if the class defines an SCA service interface) are remotable. When placed on a service reference, it indicates that the interface for the reference is remotable.

Snippet 10-21 shows the Java interface for a remotable service with its @Remotable annotation.

```

3185 package services.hello;
3186
3187 import org.oasisopen.sca.annotation.*;
3188
3189 @Remotable
3190 public interface HelloService {
3191
3192     String hello(String message);
3193 }

```

Snippet 10-21: Use of @Remotable on an Interface

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 need to 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 can be JAXB [JAX-B] types or they can be Service Data Objects (SDOs) [SDO].

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 can modify input data during or after an invocation and can 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.

Snippet 10-22 shows how a Java service implementation class can use the @Remotable annotation to define a remotable SCA service interface using a Java service interface that is not marked as remotable.


```

3211 package services.hello;
3212
3213 import org.oasisopen.sca.annotation.*;
3214
3215 public interface HelloService {
3216     String hello(String message);
3217 }
3218
3219 package services.hello;
3220
3221 import org.oasisopen.sca.annotation.*;
3222
3223 @Remotable
3224 @Service(HelloService.class)
3225 public class HelloServiceImpl implements HelloService {
3226     public String hello(String message) {
3227         ...
3228     }
3229 }
3230
3231 }

```

3232 *Snippet 10-22: Use of @Remotable on a Class*

3233

3234 Snippet 10-23 shows how a reference can use the @Remotable annotation to define a remotable SCA
3235 service interface using a Java service interface that is not marked as remotable.

3236

```

3237 package services.hello;
3238
3239 import org.oasisopen.sca.annotation.*;
3240
3241 public interface HelloService {
3242     String hello(String message);
3243 }
3244
3245 package services.hello;
3246
3247 import org.oasisopen.sca.annotation.*;
3248
3249 public class HelloClient {
3250
3251     @Remotable
3252     @Reference
3253     protected HelloService myHello;
3254
3255     public String greeting(String message) {
3256         return myHello.hello(message);
3257     }
3258 }
3259

```

3260 *Snippet 10-23: Use of @Remotable on a Reference*

3261 10.26 @Requires

3262 Figure 10-26 defines the **@Requires** annotation:

3263

```

3264 package org.oasisopen.sca.annotation;
3265
3266 import static java.lang.annotation.ElementType.FIELD;
3267 import static java.lang.annotation.ElementType.METHOD;

```

```

3268 import static java.lang.annotation.ElementType.PARAMETER;
3269 import static java.lang.annotation.ElementType.TYPE;
3270 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3271
3272 import java.lang.annotation.Inherited;
3273 import java.lang.annotation.Retention;
3274 import java.lang.annotation.Target;
3275
3276 @Inherited
3277 @Retention(RUNTIME)
3278 @Target({TYPE, METHOD, FIELD, PARAMETER})
3279 public @interface Requires {
3280     /**
3281      * Returns the attached intents.
3282      *
3283      * @return the attached intents
3284      */
3285     String[] value() default "";
3286 }

```

Figure 10-26: Requires Annotation

The **@Requires** annotation supports general purpose intents specified as strings. Users can also define specific intent annotations using the @Intent annotation.

See the [section "General Intent Annotations"](#) for details and samples.

10.27 @Scope

Figure 10-27 defines the **@Scope** annotation:

```

3295 package org.oasisopen.sca.annotation;
3296
3297 import static java.lang.annotation.ElementType.TYPE;
3298 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3299 import java.lang.annotation.Retention;
3300 import java.lang.annotation.Target;
3301
3302 @Target(TYPE)
3303 @Retention(RUNTIME)
3304 public @interface Scope {
3305
3306     String value() default "STATELESS";
3307 }

```

Figure 10-27: Scope Annotation

The **@Scope** annotation **MUST** only be used on a service's implementation class. It is an error to use this annotation on an interface. [JCA90041]

The **@Scope** annotation has the attribute:

- **value** – the name of the scope.

SCA defines the following scope names, but others can be defined by particular Java-based implementation types

STATELESS

COMPOSITE

The default value is STATELESS.

Snippet 10-24 shows a sample for a COMPOSITE scoped service implementation:

```
package services.hello;

import org.oasisopen.sca.annotation.*;

@Service(HelloService.class)
@Scope("COMPOSITE")
public class HelloServiceImpl implements HelloService {

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

Snippet 10-24: Use of @Scope

10.28 @Service

Figure 10-28 defines the @Service annotation:

```
package org.oasisopen.sca.annotation;

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<?>[] value();
    String[] names() default {};
}
```

Figure 10-28: Service Annotation

The @Service annotation is used on a component implementation class to specify the SCA services offered by the implementation. An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present. [JCA90042] A class used as the implementation of a service is not required to have a @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 attributes:

- **value (1..1)** – An array of interface or class objects that are exposed as services by this implementation. If the array is empty, no services are exposed.
- **names (0..1)** - An array of Strings which are used as the service names for each of the interfaces declared in the **value** array. The number of Strings in the names attribute array of the @Service annotation MUST match the number of elements in the value attribute array. [JCA90050] The value of each element in the @Service names array MUST be unique amongst all the other element values in the array. [JCA90060]

The **service name** of an exposed service defaults to the name of its interface or class, without the package name. If the names attribute is specified, the service name for each interface or class in the value attribute array is the String declared in the corresponding position in the names attribute array.

If a component implementation has two services with the same Java simple name, the names attribute of the @Service annotation MUST be specified. [JCA90045] If a Java implementation needs to realize two services with the same Java simple name then this can be achieved through subclassing of the interface.

Snippet 10-25 shows an implementation of the HelloService marked with the @Service annotation.

```
package services.hello;

import org.oasisopen.sca.annotation.Service;

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

    public void hello(String name) {
        System.out.println("Hello " + name);
    }
}
```

Snippet 10-25: Use of @Service

11 WSDL to Java and Java to WSDL

This specification applies the WSDL to Java and Java to WSDL mapping rules as defined by the JAX-WS 2.1 specification [JAX-WS] for generating remotable Java interfaces from WSDL portTypes and vice versa.

SCA runtimes MUST support the JAX-WS 2.1 mappings from WSDL to Java and from Java to WSDL. [JCA100022] For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't. [JCA100001] The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation. [JCA100002] For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable. [JCA100003]

For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B] mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MUST support the JAXB 2.1 mapping from XML Schema to Java and from Java to XML Schema. [JCA100004] SCA runtimes MAY support the SDO 2.1 mapping from XML schema types to Java and from Java to XML Schema. [JCA100005] Having a choice of binding technologies is allowed, as noted in the first paragraph of section 5 of the JSR 181 (version 2) specification, which is referenced by the JAX-WS specification.

11.1 JAX-WS Annotations and SCA Interfaces

A Java class or interface used to define an SCA interface can contain JAX-WS annotations. In addition to affecting the Java to WSDL mapping defined by the JAX-WS specification [JAX-WS] these annotations can impact the SCA interface. An SCA runtime MUST apply the JAX-WS annotations as described in Table 11-1 and Table 11-2 when introspecting a Java class or interface class. [JCA100011] This could mean that the interface of a Java implementation is defined by a WSDL interface declaration. If the services provided by an implementation class are explicitly identified by an @Service annotation, only the identified classes define services of the implementation even if implemented interfaces that are not listed in the @Service annotation contain @JAX-WS annotations.

Annotation	Property	Impact to SCA Interface
@WebService		A Java interface or class annotated with @WebService MUST be treated as if annotated with the SCA @Remotable annotation [JCA100012]
	name	The value of the name attribute of the @WebService annotation, if present, MUST be used to define the name of an SCA service when there is no @Service annotation present in the SCA component implementation. [JCA100023] The value of the name attribute of the @WebService annotation, if present, MUST be used to define the

name of an SCA service when the `@Service` annotation is present without the `names` attribute and indicates that the Java interface or class annotated with the `@WebService` annotation defines an SCA service interface. [JCA100028]

`targetNamespace` None

`serviceName` None

`wsdlLocation` A Java class annotated with the `@WebService` annotation with its `wsdlLocation` attribute set **MUST** have its interface defined by the referenced WSDL definition instead of the annotated Java class. [JCA100013]

`endpointInterface` A Java class annotated with the `@WebService` annotation with its `endpointInterface` attribute set **MUST** have its interface defined by the referenced interface instead of annotated Java class. [JCA100014]

`portName` None

`@WebMethod`

`operationName` For a Java method annotated with the `@WebMethod` annotation with the `operationName` set, an SCA runtime **MUST** use the value of the `operationName` attribute as the SCA operation name. [JCA100024]

`action` None

`exclude` An SCA runtime **MUST NOT** include a Java method annotated with the `@WebMethod` annotation with the

exclude attribute set to true in an SCA interface.

[JCA100025]

@OneWay

The SCA runtime MUST treat an

@org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.

[JCA100002]

@WebParam

name	Sets parameter name
targetNamespace	None
mode	For a Java parameter annotated with the @WebParam annotation with the mode attribute set, an SCA runtime MUST apply the value of the mode attribute when comparing interfaces. [JCA100026]
header	A Java class or interface containing an @WebParam annotation with its header attribute set to “true” MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100015]
partName	Overrides name

@WebResult

name	Sets parameter name
targetNamespace	None
header	A Java class or interface containing an @WebResult annotation with its header attribute set to “true” MUST be treated as if the SOAP intent is applied to the Java

		class or interface. [JCA100016]
	partName	Overrides name
@SOAPBinding		A Java class or interface containing an @SOAPBinding annotation MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100021]
	style	
	use	
	parameterStyle	
@HandlerChain		None
	file	
	name	

3414 Table 11-1: JSR 181 Annotations and SCA Interfaces

3415

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
@ServiceMode		A Java class containing an @ServiceMode annotation MUST be treated as if the SOAP intent is applied to the Java class. [JCA100017]
	value	
@WebFault		
	name	None
	targetNamespace	None
	faultBean	None
@RequestWrapper		None

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
	localName	
	targetNamespace	
	className	
@ResponseWrapper		None
	localName	
	targetNamespace	
	className	
@WebServiceClient		An interface or class annotated with @WebServiceClient MUST NOT be used to define an SCA interface. [JCA100018]
	name	
	targetNamespace	
	wsdlLocation	
@WebEndpoint		None
	name	
@WebServiceProvider		A class annotated with @WebServiceProvider MUST be treated as if annotated with the SCA @Remotable annotation. [JCA100019]
	wsdlLocation	A Java class annotated with the @WebServiceProvider annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition is used instead of the annotated Java class.

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
		[JCA100020]
	serviceName	None
	portName	None
	targetNamespace	None
@BindingType		None
	value	
@WebServiceRef		See JEE specification
	name	
	wsdlLocation	
	type	
	value	
	mappedName	
@WebServiceRefs		See JEE specification
	value	
@Action		None
	fault	
	input	
	output	
@FaultAction		None
	value	

output

Table 11-2: JSR 224 Annotations and SCA Interfaces

11.2 JAX-WS Client Asynchronous API for a Synchronous Service

The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client application with a means of invoking that service asynchronously, so that the client can invoke a service operation and proceed to do other work without waiting for the service operation to complete its processing. The client application can retrieve the results of the service either through a polling mechanism or via a callback method which is invoked when the operation completes.

For SCA service interfaces defined using `interface.java`, the Java interface MUST NOT contain the additional client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100006] For SCA reference interfaces defined using `interface.java`, the SCA runtime MUST support a Java interface which contains the additional client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100007] If the additional client-side asynchronous polling and callback methods defined by JAX-WS are present in the interface which declares the type of a reference in the implementation, SCA Runtimes MUST NOT include these methods in the SCA reference interface in the component type of the implementation. [JCA100008]

The additional client-side asynchronous polling and callback methods defined by JAX-WS are recognized in a Java interface according to the steps:

For each method M in the interface, if another method P in the interface has

- a method name that is M's method name with the characters "Async" appended, and
- the same parameter signature as M, and
- a return type of `Response<R>` where R is the return type of M

then P is a JAX-WS polling method that isn't part of the SCA interface contract.

For each method M in the interface, if another method C in the interface has

- a method name that is M's method name with the characters "Async" appended, and
- a parameter signature that is M's parameter signature with an additional final parameter of type `AsyncHandler<R>` where R is the return type of M, and
- a return type of `Future<?>`

then C is a JAX-WS callback method that isn't part of the SCA interface contract.

As an example, an interface can be defined in WSDL as shown in Snippet 11-1:

```
<!-- WSDL extract -->
<message name="getPrice">
  <part name="ticker" type="xsd:string"/>
</message>

<message name="getPriceResponse">
  <part name="price" type="xsd:float"/>
</message>

<portType name="StockQuote">
  <operation name="getPrice">
    <input message="tns:getPrice"/>
    <output message="tns:getPriceResponse"/>
  </operation>
</portType>
```

Snippet 11-1: Example WSDL Interface

3462

3463 The JAX-WS asynchronous mapping will produce the Java interface in Snippet 11-2:

3464

```
3465 // asynchronous mapping
3466 @WebService
3467 public interface StockQuote {
3468     float getPrice(String ticker);
3469     Response<Float> getPriceAsync(String ticker);
3470     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
3471 }
```

3472 *Snippet 11-2: JAX-WS Asynchronous Interface for WSDL Interface in Snippet 11-1*

3473

3474 For SCA interface definition purposes, this is treated as equivalent to the interface in Snippet 11-3:

3475

```
3476 // synchronous mapping
3477 @WebService
3478 public interface StockQuote {
3479     float getPrice(String ticker);
3480 }
```

3481 *Snippet 11-3: Equivalent SCA Interface Corresponding to Java Interface in Snippet 11-2*

3482

3483 **SCA runtimes MUST support the use of the JAX-WS client asynchronous model.** [JCA100009] If the
3484 client implementation uses the asynchronous form of the interface, the two additional getPriceAsync()
3485 methods can be used for polling and callbacks as defined by the JAX-WS specification.

3486 11.3 Treatment of SCA Asynchronous Service API

3487 **For SCA service interfaces defined using interface.java, the SCA runtime MUST support a Java interface**
3488 **which contains the server-side asynchronous methods defined by SCA.** [JCA100010]

3489 Asynchronous service methods are identified as described in the section "Asynchronous handling of Long
3490 Running Service Operations" and are mapped to WSDL in the same way as the equivalent synchronous
3491 method described in that section.

3492 Generating an asynchronous service method from a WSDL request/response operation follows the
3493 algorithm described in the same section.

12 Conformance

The XML schema pointed to by the RDDDL document at the namespace URI, defined by this specification, are considered to be authoritative and take precedence over the XML schema defined in the appendix of this document.

Normative code artifacts related to this specification are considered to be authoritative and take precedence over specification text.

There are three categories of artifacts for which this specification defines conformance:

- a) SCA Java XML Document,
- b) SCA Java Class
- c) SCA Runtime.

12.1 SCA Java XML Document

An SCA Java XML document is an SCA Composite Document, or an SCA ComponentType Document, as defined by the [SCA Assembly Model specification \[ASSEMBLY\]](#), that uses the <interface.java> element. Such an SCA Java XML document MUST be a conformant SCA Composite Document or SCA ComponentType Document, as defined by the [SCA Assembly Model specification \[ASSEMBLY\]](#), and MUST comply with the requirements specified in [the Interface section](#) of this specification.

12.2 SCA Java Class

An SCA Java Class is a Java class or interface that complies with Java Standard Edition version 5.0 and MAY include annotations and APIs defined in this specification. An SCA Java Class that uses annotations and APIs defined in this specification MUST comply with the requirements specified in this specification for those annotations and APIs.

12.3 SCA Runtime

The APIs and annotations defined in this specification are meant to be used by Java-based component implementation models in either partial or complete fashion. A Java-based component implementation specification that uses this specification specifies which of the APIs and annotations defined here are used. The APIs and annotations an SCA Runtime has to support depends on which Java-based component implementation specification the runtime supports. For example, see the [SCA POJO Component Implementation Specification \[JAVA_CI\]](#).

An implementation that claims to conform to this specification MUST meet the following conditions:

1. The implementation MUST meet all the conformance requirements defined by the SCA Assembly Model Specification [ASSEMBLY].
2. The implementation MUST support <interface.java> and MUST comply with all the normative statements in Section 3.
3. The implementation MUST reject an SCA Java XML Document that does not conform to the sca-interface-java.xsd schema.
4. The implementation MUST support and comply with all the normative statements in Section 10.

Appendix A. XML Schema: sca-interface-java-1.1.xsd

```
3531 <?xml version="1.0" encoding="UTF-8"?>
3532 <!-- Copyright(C) OASIS(R) 2005,2010. All Rights Reserved.
3533 OASIS trademark, IPR and other policies apply. -->
3534 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3535 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3536 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3537 elementFormDefault="qualified">
3538
3539 <include schemaLocation="sca-core-1.1-cd06.xsd"/>
3540
3541 <!-- Java Interface -->
3542 <element name="interface.java" type="sca:JavaInterface"
3543 substitutionGroup="sca:interface"/>
3544 <complexType name="JavaInterface">
3545 <complexContent>
3546 <extension base="sca:Interface">
3547 <sequence>
3548 <any namespace="##other" processContents="lax" minOccurs="0"
3549 maxOccurs="unbounded"/>
3550 </sequence>
3551 <attribute name="interface" type="NCName" use="required"/>
3552 <attribute name="callbackInterface" type="NCName"
3553 use="optional"/>
3554 </extension>
3555 </complexContent>
3556 </complexType>
3557
3558 </schema>
```

Appendix B. Java Classes and Interfaces

B.1 SCAClient Classes and Interfaces

B.1.1 SCAClientFactory Class

SCA provides an abstract base class SCAClientFactory. Vendors can provide subclasses of this class which create objects that implement the SCAClientFactory class suitable for linking to services in their SCA runtime.

```
/*
 * Copyright(C) OASIS(R) 2005,2010. All Rights Reserved.
 * OASIS trademark, IPR and other policies apply.
 */
package org.oasisopen.sca.client;

import java.net.URI;
import java.util.Properties;

import org.oasisopen.sca.NoSuchDomainException;
import org.oasisopen.sca.NoSuchServiceException;
import org.oasisopen.sca.client.SCAClientFactoryFinder;
import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;

/**
 * The SCAClientFactory can be used by non-SCA managed code to
 * lookup services that exist in a SCADomain.
 *
 * @see SCAClientFactoryFinderImpl
 *
 * @author OASIS Open
 */
public abstract class SCAClientFactory {

    /**
     * The SCAClientFactoryFinder.
     * Provides a means by which a provider of an SCAClientFactory
     * implementation can inject a factory finder implementation into
     * the abstract SCAClientFactory class - once this is done, future
     * invocations of the SCAClientFactory use the injected factory
     * finder to locate and return an instance of a subclass of
     * SCAClientFactory.
     */
    protected static SCAClientFactoryFinder factoryFinder;

    /**
     * The Domain URI of the SCA Domain which is accessed by this
     * SCAClientFactory
     */
    private URI domainURI;

    /**
     * Prevent concrete subclasses from using the no-arg constructor
     */
    private SCAClientFactory() {
    }

    /**
     * Constructor used by concrete subclasses
     */
}
```

```

3615     * @param domainURI - The Domain URI of the Domain accessed via this
3616     * SCAClientFactory
3617     */
3618     protected SCAClientFactory(Uri domainURI) throws NoSuchDomainException {
3619         this.domainURI = domainURI;
3620     }
3621
3622     /**
3623     * Gets the Domain URI of the Domain accessed via this SCAClientFactory
3624     * @return - the URI for the Domain
3625     */
3626     protected Uri getDomainURI() {
3627         return domainURI;
3628     }
3629
3630
3631     /**
3632     * Creates a new instance of the SCAClientFactory that can be
3633     * used to lookup SCA Services.
3634     *
3635     * @param domainURI      Uri of the target domain for the SCAClientFactory
3636     * @return A new SCAClientFactory
3637     */
3638     public static SCAClientFactory newInstance( Uri domainURI )
3639         throws NoSuchDomainException {
3640         return newInstance(null, null, domainURI);
3641     }
3642
3643     /**
3644     * Creates a new instance of the SCAClientFactory that can be
3645     * used to lookup SCA Services.
3646     *
3647     * @param properties      Properties that may be used when
3648     * creating a new instance of the SCAClientFactory
3649     * @param domainURI      Uri of the target domain for the SCAClientFactory
3650     * @return A new SCAClientFactory instance
3651     */
3652     public static SCAClientFactory newInstance(Properties properties,
3653                                                Uri domainURI)
3654         throws NoSuchDomainException {
3655         return newInstance(properties, null, domainURI);
3656     }
3657
3658     /**
3659     * Creates a new instance of the SCAClientFactory that can be
3660     * used to lookup SCA Services.
3661     *
3662     * @param classLoader      ClassLoader that may be used when
3663     * creating a new instance of the SCAClientFactory
3664     * @param domainURI      Uri of the target domain for the SCAClientFactory
3665     * @return A new SCAClientFactory instance
3666     */
3667     public static SCAClientFactory newInstance(ClassLoader classLoader,
3668                                                Uri domainURI)
3669         throws NoSuchDomainException {
3670         return newInstance(null, classLoader, domainURI);
3671     }
3672
3673     /**
3674     * Creates a new instance of the SCAClientFactory that can be
3675     * used to lookup SCA Services.
3676     *
3677     * @param properties      Properties that may be used when
3678     * creating a new instance of the SCAClientFactory

```



```

3679      * @param classLoader    ClassLoader that may be used when
3680      * creating a new instance of the SCAClientFactory
3681      * @param domainURI      URI of the target domain for the SCAClientFactory
3682      * @return A new SCAClientFactory instance
3683      */
3684      public static SCAClientFactory newInstance(Properties properties,
3685                                              ClassLoader classLoader,
3686                                              URI domainURI)
3687          throws NoSuchDomainException {
3688          final SCAClientFactoryFinder finder =
3689              factoryFinder != null ? factoryFinder :
3690                  new SCAClientFactoryFinderImpl();
3691          final SCAClientFactory factory
3692              = finder.find(properties, classLoader, domainURI);
3693          return factory;
3694      }
3695
3696      /**
3697      * Returns a reference proxy that implements the business interface <T>
3698      * of a service in the SCA Domain handled by this SCAClientFactory
3699      *
3700      * @param serviceURI the relative URI of the target service. Takes the
3701      * form componentName/serviceName.
3702      * Can also take the extended form componentName/serviceName/bindingName
3703      * to use a specific binding of the target service
3704      *
3705      * @param interfaze The business interface class of the service in the
3706      * domain
3707      * @param <T> The business interface class of the service in the domain
3708      *
3709      * @return a proxy to the target service, in the specified SCA Domain
3710      * that implements the business interface <B>.
3711      * @throws NoSuchServiceException Service requested was not found
3712      * @throws NoSuchDomainException Domain requested was not found
3713      */
3714      public abstract <T> T getService(Class<T> interfaze, String serviceURI)
3715          throws NoSuchServiceException, NoSuchDomainException;
3716      }

```

3717 B.1.2 SCAClientFactoryFinder interface

3718 The SCAClientFactoryFinder interface is a Service Provider Interface representing a SCAClientFactory
3719 finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can
3720 create alternative implementations of this interface that use different class loading or lookup mechanisms.

```

3721      /*
3722      * Copyright(C) OASIS(R) 2005,2010. All Rights Reserved.
3723      * OASIS trademark, IPR and other policies apply.
3724      */
3725
3726      package org.oasisopen.sca.client;
3727
3728      import java.net.URI;
3729      import java.util.Properties;
3730
3731      import org.oasisopen.sca.NoSuchDomainException;
3732
3733      /* A Service Provider Interface representing a SCAClientFactory finder.
3734      * SCA provides a default reference implementation of this interface.
3735      * SCA runtime vendors can create alternative implementations of this
3736      * interface that use different class loading or lookup mechanisms.
3737      */
3738      public interface SCAClientFactoryFinder {
3739

```

```

3740
3741 /**
3742  * Method for finding the SCAClientFactory for a given Domain URI using
3743  * a specified set of properties and a specified ClassLoader
3744  * @param properties - properties to use - may be null
3745  * @param classLoader - ClassLoader to use - may be null
3746  * @param domainURI - the Domain URI - must be a valid SCA Domain URI
3747  * @return - the SCAClientFactory or null if the factory could not be
3748  * @throws - NoSuchDomainException if the domainURI does not reference
3749  * a valid SCA Domain
3750  * found
3751  */
3752 SCAClientFactory find(Properties properties,
3753                      ClassLoader classLoader,
3754                      URI domainURI )
3755     throws NoSuchDomainException ;
3756 }

```

3757 **B.1.3 SCAClientFactoryFinderImpl class**

3758 This class provides a default implementation for finding a provider's SCAClientFactory implementation
3759 class. It is used if the provider does not inject its SCAClientFactoryFinder implementation class into the
3760 base SCAClientFactory class.

3761 It discovers a provider's SCAClientFactory implementation by referring to the following information in this
3762 order:

- 3763 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the
3764 newInstance() method call if specified
- 3765 2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
- 3766 3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

```

3767
3768 /**
3769  * Copyright(C) OASIS(R) 2005,2010. All Rights Reserved.
3770  * OASIS trademark, IPR and other policies apply.
3771  */
3772 package org.oasisopen.sca.client.impl;
3773
3774 import org.oasisopen.sca.client.SCAClientFactoryFinder;
3775
3776 import java.io.BufferedReader;
3777 import java.io.Closeable;
3778 import java.io.IOException;
3779 import java.io.InputStream;
3780 import java.io.InputStreamReader;
3781 import java.lang.reflect.Constructor;
3782 import java.net.URI;
3783 import java.net.URL;
3784 import java.util.Properties;
3785
3786 import org.oasisopen.sca.NoSuchDomainException;
3787 import org.oasisopen.sca.ServiceRuntimeException;
3788 import org.oasisopen.sca.client.SCAClientFactory;
3789
3790 /**
3791  * This is a default implementation of an SCAClientFactoryFinder which is
3792  * used to find an implementation of the SCAClientFactory interface.
3793  *
3794  * @see SCAClientFactoryFinder
3795  * @see SCAClientFactory
3796  *
3797  * @author OASIS Open
3798  */

```

```

3799 public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {
3800
3801     /**
3802      * The name of the System Property used to determine the SPI
3803      * implementation to use for the SCAClientFactory.
3804      */
3805     private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
3806         SCAClientFactory.class.getName();
3807
3808     /**
3809      * The name of the file loaded from the ClassPath to determine
3810      * the SPI implementation to use for the SCAClientFactory.
3811      */
3812     private static final String SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
3813         = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
3814
3815     /**
3816      * Public Constructor
3817      */
3818     public SCAClientFactoryFinderImpl() {
3819     }
3820
3821     /**
3822      * Creates an instance of the SCAClientFactorySPI implementation.
3823      * This discovers the SCAClientFactorySPI Implementation and instantiates
3824      * the provider's implementation.
3825      *
3826      * @param properties    Properties that may be used when creating a new
3827      * instance of the SCAClient
3828      * @param classLoader    ClassLoader that may be used when creating a new
3829      * instance of the SCAClient
3830      * @return new instance of the SCAClientFactory
3831      * @throws ServiceRuntimeException Failed to create SCAClientFactory
3832      * Implementation.
3833      */
3834     public SCAClientFactory find(Properties properties,
3835                                 ClassLoader classLoader,
3836                                 URI domainURI )
3837         throws NoSuchDomainException, ServiceRuntimeException {
3838         if (classLoader == null) {
3839             classLoader = getThreadContextClassLoader ();
3840         }
3841         final String factoryImplClassName =
3842             discoverProviderFactoryImplClass(properties, classLoader);
3843         final Class<? extends SCAClientFactory> factoryImplClass
3844             = loadProviderFactoryClass(factoryImplClassName,
3845                                       classLoader);
3846         final SCAClientFactory factory =
3847             instantiateSCAClientFactoryClass(factoryImplClass,
3848                                             domainURI, properties );
3849         return factory;
3850     }
3851
3852     /**
3853      * Gets the Context ClassLoader for the current Thread.
3854      *
3855      * @return The Context ClassLoader for the current Thread.
3856      */
3857     private static ClassLoader getThreadContextClassLoader () {
3858         return AccessController.doPrivileged(
3859             new PrivilegedAction<ClassLoader>() {
3860                 public ClassLoader run() {
3861                     return Thread.currentThread().getContextClassLoader();
3862                 }
3863             }
3864         );
3865     }
3866 }

```

```

3863     });
3864 }
3865
3866 /**
3867  * Attempts to discover the class name for the SCAClientFactorySPI
3868  * implementation from the specified Properties, the System Properties
3869  * or the specified ClassLoader.
3870  *
3871  * @return The class name of the SCAClientFactorySPI implementation
3872  * @throw ServiceRuntimeException Failed to find implementation for
3873  * SCAClientFactorySPI.
3874  */
3875 private static String
3876     discoverProviderFactoryImplClass(Properties properties,
3877                                     ClassLoader classLoader)
3878     throws ServiceRuntimeException {
3879     String providerClassName =
3880         checkPropertiesForSPIClassName(properties);
3881     if (providerClassName != null) {
3882         return providerClassName;
3883     }
3884
3885     providerClassName =
3886         checkPropertiesForSPIClassName(System.getProperties());
3887     if (providerClassName != null) {
3888         return providerClassName;
3889     }
3890
3891     providerClassName = checkMETAINFServicesForSPIClassName(classLoader);
3892     if (providerClassName == null) {
3893         throw new ServiceRuntimeException(
3894             "Failed to find implementation for SCAClientFactory");
3895     }
3896
3897     return providerClassName;
3898 }
3899
3900 /**
3901  * Attempts to find the class name for the SCAClientFactorySPI
3902  * implementation from the specified Properties.
3903  *
3904  * @return The class name for the SCAClientFactorySPI implementation
3905  * or <code>null</code> if not found.
3906  */
3907 private static String
3908     checkPropertiesForSPIClassName(Properties properties) {
3909     if (properties == null) {
3910         return null;
3911     }
3912
3913     final String providerClassName =
3914         properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
3915     if (providerClassName != null && providerClassName.length() > 0) {
3916         return providerClassName;
3917     }
3918
3919     return null;
3920 }
3921
3922 /**
3923  * Attempts to find the class name for the SCAClientFactorySPI
3924  * implementation from the META-INF/services directory
3925  *
3926  * @return The class name for the SCAClientFactorySPI implementation or

```

```

3927     * <code>null</code> if not found.
3928     */
3929     private static String checkMETAINFServicesForSPIClassName(ClassLoader cl)
3930     {
3931         final URL url =
3932             cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
3933         if (url == null) {
3934             return null;
3935         }
3936
3937         InputStream in = null;
3938         try {
3939             in = url.openStream();
3940             BufferedReader reader = null;
3941             try {
3942                 reader =
3943                     new BufferedReader(new InputStreamReader(in, "UTF-8"));
3944
3945                 String line;
3946                 while ((line = readNextLine(reader)) != null) {
3947                     if (!line.startsWith("#") && line.length() > 0) {
3948                         return line;
3949                     }
3950                 }
3951
3952                 return null;
3953             } finally {
3954                 closeStream(reader);
3955             }
3956         } catch (IOException ex) {
3957             throw new ServiceRuntimeException(
3958                 "Failed to discover SCAClientFactory provider", ex);
3959         } finally {
3960             closeStream(in);
3961         }
3962     }
3963
3964     /**
3965     * Reads the next line from the reader and returns the trimmed version
3966     * of that line
3967     *
3968     * @param reader The reader from which to read the next line
3969     * @return The trimmed next line or <code>null</code> if the end of the
3970     * stream has been reached
3971     * @throws IOException I/O error occurred while reading from Reader
3972     */
3973     private static String readNextLine(BufferedReader reader)
3974         throws IOException {
3975
3976         String line = reader.readLine();
3977         if (line != null) {
3978             line = line.trim();
3979         }
3980         return line;
3981     }
3982
3983     /**
3984     * Loads the specified SCAClientFactory Implementation class.
3985     *
3986     * @param factoryImplClassName The name of the SCAClientFactory
3987     * Implementation class to load
3988     * @return The specified SCAClientFactory Implementation class
3989     * @throws ServiceRuntimeException Failed to load the SCAClientFactory
3990     * Implementation class

```

```

3991     */
3992     private static Class<? extends SCAClientFactory>
3993         loadProviderFactoryClass(String factoryImplClassName,
3994                                 ClassLoader classLoader)
3995         throws ServiceRuntimeException {
3996
3997         try {
3998             final Class<?> providerClass =
3999                 classLoader.loadClass(factoryImplClassName);
4000             final Class<? extends SCAClientFactory> providerFactoryClass =
4001                 providerClass.asSubclass(SCAClientFactory.class);
4002             return providerFactoryClass;
4003         } catch (ClassNotFoundException ex) {
4004             throw new ServiceRuntimeException(
4005                 "Failed to load SCAClientFactory implementation class "
4006                 + factoryImplClassName, ex);
4007         } catch (ClassCastException ex) {
4008             throw new ServiceRuntimeException(
4009                 "Loaded SCAClientFactory implementation class "
4010                 + factoryImplClassName
4011                 + " is not a subclass of "
4012                 + SCAClientFactory.class.getName() , ex);
4013         }
4014     }
4015
4016     /**
4017     * Instantiate an instance of the specified SCAClientFactorySPI
4018     * Implementation class.
4019     *
4020     * @param factoryImplClass The SCAClientFactorySPI Implementation
4021     * class to instantiate.
4022     * @return An instance of the SCAClientFactorySPI Implementation class
4023     * @throws ServiceRuntimeException Failed to instantiate the specified
4024     * specified SCAClientFactorySPI Implementation class
4025     */
4026     private static SCAClientFactory instantiateSCAClientFactoryClass(
4027         Class<? extends SCAClientFactory> factoryImplClass,
4028         URI domainURI, Properties properties)
4029         throws NoSuchDomainException, ServiceRuntimeException {
4030
4031         try {
4032             Constructor<? extends SCAClientFactory> URIConstructor =
4033                 factoryImplClass.getConstructor(URI.class, Properties.class);
4034             SCAClientFactory provider =
4035                 URIConstructor.newInstance( domainURI, properties );
4036             return provider;
4037         } catch (Throwable ex) {
4038             throw new ServiceRuntimeException(
4039                 "Failed to instantiate SCAClientFactory implementation class "
4040                 + factoryImplClass, ex);
4041         }
4042     }
4043
4044     /**
4045     * Utility method for closing Closeable Object.
4046     *
4047     * @param closeable The Object to close.
4048     */
4049     private static void closeStream(Closeable closeable) {
4050         if (closeable != null) {
4051             try{
4052                 closeable.close();
4053             } catch (IOException ex) {
4054                 throw new ServiceRuntimeException("Failed to close stream",

```

4055
4056
4057
4058
4059

```
ex);  
    }  
    }  
}
```

4060

B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?

4061
4062
4063

The SCAClient classes and interfaces are designed so that vendors can provide their own implementation suited to the needs of their SCA runtime. This section describes the tasks that a vendor needs to consider in relation to the SCAClient classes and interfaces.

4064

- Implement their SCAClientFactory implementation class

4065
4066
4067
4068

Vendors need to provide a subclass of SCAClientFactory that is capable of looking up Services in their SCA Runtime. Vendors need to subclass SCAClientFactory and implement the getService() method so that it creates reference proxies to services in SCA Domains handled by their SCA runtime(s).

4069

- Configure the Vendor SCAClientFactory implementation class so that it gets used

4070

Vendors have several options:

4071

Option 1: Set System Property to point to the Vendor's implementation

4072
4073

Vendors set the org.oasisopen.sca.client.SCAClientFactory System Property to point to their implementation class and use the reference implementation of SCAClientFactoryFinder

4074

Option 2: Provide a META-INF/services file

4075
4076

Vendors provide a META-INF/services/org.oasisopen.sca.client.SCAClientFactory file that points to their implementation class and use the reference implementation of SCAClientFactoryFinder

4077
4078

Option 3: Inject a vendor implementation of the SCAClientFactoryFinder interface into SCAClientFactory

4079
4080
4081
4082
4083

Vendors inject an instance of the vendor implementation of SCAClientFactoryFinder into the factoryFinder field of the SCAClientFactory abstract class. The reference implementation of SCAClientFactoryFinder is not used in this scenario. The vendor implementation of SCAClientFactoryFinder can find the vendor implementation(s) of SCAClientFactory by any means.

4084 Appendix C. Conformance Items

4085 This section contains a list of conformance items for the SCA-J Common Annotations and APIs
4086 specification.
4087

Conformance ID	Description
[JCA20001]	Remotable Services MUST NOT make use of <i>method overloading</i> .
[JCA20002]	the SCA runtime MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one thread at any one time.
[JCA20003]	within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of one business method.
[JCA20004]	Where an implementation is used by a "domain level component", and the implementation is marked "Composite" scope, the SCA runtime MUST ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation.
[JCA20005]	When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.
[JCA20006]	If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.
[JCA20007]	the SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and the SCA runtime MUST NOT perform any synchronization.
[JCA20008]	Where an implementation is marked "Composite" scope and it is used by a component that is nested inside a composite that is used as the implementation of a higher level component, the SCA runtime MUST ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation. There can be multiple instances of the higher level component, each running on different nodes in a distributed SCA runtime.
[JCA20010]	The SCA runtime MUST use by-value semantics when passing input parameters, return values and exceptions on calls to remotable services within the same JVM if the service method implementation is not marked "allows pass by reference" or the service proxy used by the client is not marked "allows pass by reference".
[JCA30001]	The value of the @interface attribute MUST be the fully qualified name of a Java class
[JCA30002]	The value of the @callbackInterface attribute MUST be the fully qualified name of a Java interface used for callbacks
[JCA30003]	if the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same

interface class.

- [JCA30004] The `<interface.java>` element MUST conform to the schema defined in the `sca-interface-java.xsd` schema.
- [JCA30005] The value of the `@remotable` attribute on the `<interface.java>` element does not override the presence of a `@Remotable` annotation on the interface class and so if the interface class contains a `@Remotable` annotation and the `@remotable` attribute has a value of "false", then the SCA Runtime MUST raise an error and MUST NOT run the component concerned.
- [JCA30006] A Java interface referenced by the `@interface` attribute of an `<interface.java>` element MUST NOT contain any of the following SCA Java annotations:
`@AllowsPassByReference`, `@ComponentName`, `@Constructor`, `@Context`, `@Destroy`, `@EagerInit`, `@Init`, `@Intent`, `@Property`, `@Qualifier`, `@Reference`, `@Scope`, `@Service`.
- [JCA30007] A Java interface referenced by the `@callbackInterface` attribute of an `<interface.java>` element MUST NOT contain any of the following SCA Java annotations:
`@AllowsPassByReference`, `@Callback`, `@ComponentName`, `@Constructor`, `@Context`, `@Destroy`, `@EagerInit`, `@Init`, `@Intent`, `@Property`, `@Qualifier`, `@Reference`, `@Scope`, `@Service`.
- [JCA30009] The SCA Assembly Model specification [ASSEMBLY] defines a number of criteria that need to be satisfied in order for two interfaces to be compatible or have a compatible superset or subset relationship. If these interfaces are both Java interfaces, compatibility also means that every method that is present in both interfaces is defined consistently in both interfaces with respect to the `@OneWay` annotation, that is, the annotation is either present in both interfaces or absent in both interfaces.
- [JCA30010] If the identified class is annotated with either the JAX-WS `@WebService` or `@WebServiceProvider` annotations and the annotation has a non-empty ***wsdlLocation*** property, then the SCA Runtime MUST act as if an `<interface.wsdl>` element is present instead of the `<interface.java>` element, with an `@interface` attribute identifying the portType mapped from the Java interface class and containing `@requires` and `@policySets` attribute values equal to the `@requires` and `@policySets` attribute values of the `<interface.java>` element.
- [JCA40001] The SCA Runtime MUST call a constructor of the component implementation at the start of the Constructing state.
- [JCA40002] The SCA Runtime MUST perform any constructor reference or property injection when it calls the constructor of a component implementation.
- [JCA40003] When the constructor completes successfully, the SCA Runtime MUST transition the component implementation to the Injecting state.
- [JCA40004] If an exception is thrown whilst in the Constructing state, the SCA Runtime MUST transition the component implementation to the Terminated state.

- [JCA40005] When a component implementation instance is in the Injecting state, the SCA Runtime MUST first inject all field and setter properties that are present into the component implementation.
- [JCA40006] When a component implementation instance is in the Injecting state, the SCA Runtime MUST inject all field and setter references that are present into the component implementation, after all the properties have been injected.
- [JCA40007] The SCA Runtime MUST ensure that the correct synchronization model is used so that all injected properties and references are made visible to the component implementation without requiring the component implementation developer to do any specific synchronization.
- [JCA40008] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation is in the Injecting state.
- [JCA40009] When the injection of properties and references completes successfully, the SCA Runtime MUST transition the component implementation to the Initializing state.
- [JCA40010] If an exception is thrown whilst injecting properties or references, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA40011] When the component implementation enters the Initializing State, the SCA Runtime MUST call the method annotated with @Init on the component implementation, if present.
- [JCA40012] If a component implementation invokes an operation on an injected reference that refers to a target that has not yet been initialized, the SCA Runtime MUST throw a ServiceUnavailableException.
- [JCA40013] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Initializing state.
- [JCA40014] Once the method annotated with @Init completes successfully, the SCA Runtime MUST transition the component implementation to the Running state.
- [JCA40015] If an exception is thrown whilst initializing, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA40016] The SCA Runtime MUST invoke Service methods on a component implementation instance when the component implementation is in the Running state and a client invokes operations on a service offered by the component.
- [JCA40017] When the component implementation scope ends, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA40018] When a component implementation enters the Destroying state, the SCA Runtime MUST call the method annotated with @Destroy on the component implementation, if present.
- [JCA40019] If a component implementation invokes an operation on an injected reference that refers to a target that has been destroyed, the SCA

Runtime MUST throw an `InvalidServiceException`.

- [JCA40020] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Destroying state.
- [JCA40021] Once the method annotated with `@Destroy` completes successfully, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40022] If an exception is thrown whilst destroying, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40023] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Terminated state.
- [JCA40024] If a property or reference is unable to be injected, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA60001] When a bidirectional service is invoked, the SCA runtime MUST inject a callback reference for the invoking service into all fields and setter methods of the service implementation class that are marked with a `@Callback` annotation and typed by the callback interface of the bidirectional service, and the SCA runtime MUST inject null into all other fields and setter methods of the service implementation class that are marked with a `@Callback` annotation.
- [JCA60002] When a non-bidirectional service is invoked, the SCA runtime MUST inject null into all fields and setter methods of the service implementation class that are marked with a `@Callback` annotation.
- [JCA60003] The SCA asynchronous service Java interface mapping of a WSDL request-response operation MUST appear as follows:
- The interface is annotated with the "asyncInvocation" intent.
 - For each service operation in the WSDL, the Java interface contains an operation with
 - a name which is the JAX-WS mapping of the WSDL operation name, with the suffix "Async" added
 - a void return type
 - a set of input parameter(s) which match the JAX-WS mapping of the input parameter(s) of the WSDL operation plus an additional last parameter which is a `ResponseDispatch` object typed by the JAX-WS Response Bean mapping of the output parameter(s) of the WSDL operation, where `ResponseDispatch` is the type defined in the SCA Java Common Annotations and APIs specification.
- [JCA60004] An SCA Runtime MUST support the use of the SCA asynchronous service interface for the interface of an SCA service.
- [JCA60005] If the SCA asynchronous service interface `ResponseDispatch` `handleResponse` method is invoked more than once through either its `sendResponse` or its `sendFault` method, the SCA runtime MUST throw an `IllegalStateException`.
- [JCA60006] For the purposes of matching interfaces (when wiring between a

reference and a service, or when using an implementation class by a component), an interface which has one or more methods which follow the SCA asynchronous service pattern MUST be treated as if those methods are mapped as the equivalent synchronous methods, as follows:

Asynchronous service methods are characterized by:

- void return type
- a method name with the suffix "Async"
- a last input parameter with a type of `ResponseDispatch<X>`
- annotation with the `asyncInvocation` intent
- possible annotation with the `@AsyncFault` annotation

The mapping of each such method is as if the method had the return type "X", the method name without the suffix "Async" and all the input parameters except the last parameter of the type `ResponseDispatch<X>`, plus the list of exceptions contained in the `@AsyncFault` annotation.

[JCA70001]

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

[JCA70002]

Intent annotations MUST NOT be applied to the following:

- A method of a service implementation class, except for a setter method that is either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class field that is not either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class constructor parameter that is not annotated with `@Reference`

[JCA70003]

Where multiple intent annotations (general or specific) are applied to the same Java element, the SCA runtime MUST compute the combined intents for the Java element by merging the intents from all intent annotations on the Java element according to the SCA Policy Framework [POLICY] rules for merging intents at the same hierarchy level.

[JCA70004]

If intent annotations are specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective intents for the method by merging the combined intents from the method with the combined intents for the interface according to the SCA Policy Framework [POLICY] rules for merging intents within a structural hierarchy, with the method at the lower level and the interface at the higher level.

[JCA70005]

The `@PolicySets` annotation MUST NOT be applied to the following:

- A method of a service implementation class, except for a setter method that is either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class field that is not either annotated

with @Reference or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification

- A service implementation class constructor parameter that is not annotated with @Reference

[JCA70006]	If the @PolicySets annotation is specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective policy sets for the method by merging the policy sets from the method with the policy sets from the interface.
[JCA80001]	The ComponentContext.getService method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..n or 1..n.
[JCA80002]	The ComponentContext.getRequestContext method MUST return non-null when invoked during the execution of a Java business method for a service operation or a callback operation, on the same thread that the SCA runtime provided, and MUST return null in all other cases.
[JCA80003]	When invoked during the execution of a service operation, the RequestContext.getServiceReference method MUST return a ServiceReference that represents the service that was invoked.
[JCA80004]	The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if the reference named by the referenceName parameter has multiplicity greater than one.
[JCA80005]	The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if the reference named by the referenceName parameter does not have an interface of the type defined by the businessInterface parameter.
[JCA80006]	The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if the component does not have a reference with the name provided in the referenceName parameter.
[JCA80007][JCA80007]	The ComponentContext.getServiceReference method MUST return null if the multiplicity of the reference named by the referenceName parameter is 0..1 and the reference has no target service configured.
[JCA80008]	The ComponentContext.getURI method MUST return the structural URI of the component in the SCA Domain.
[JCA80009]	The ComponentContext.getService method MUST return the proxy object implementing the interface provided by the businessInterface parameter, for the reference named by the referenceName parameter with the interface defined by the businessInterface parameter when that reference has a target service configured.
[JCA80010]	The ComponentContext.getService method MUST return null if the multiplicity of the reference named by the referenceName parameter is 0..1 and the reference has no target service configured.
[JCA80011]	The ComponentContext.getService method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
[JCA80012]	The ComponentContext.getService method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible

- with the interface supplied in the businessInterface parameter.
- [JCA80013] The ComponentContext.getServiceReference method MUST return a ServiceReference object typed by the interface provided by the businessInterface parameter, for the reference named by the referenceName parameter with the interface defined by the businessInterface parameter when that reference has a target service configured.
- [JCA80014] The ComponentContext.getServices method MUST return a collection containing one proxy object implementing the interface provided by the businessInterface parameter for each of the target services configured on the reference identified by the referenceName parameter.
- [JCA80015] The ComponentContext.getServices method MUST return an empty collection if the service reference with the name supplied in the referenceName parameter is not wired to any target services.
- [JCA80016] The ComponentContext.getServices method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1.
- [JCA80017] The ComponentContext.getServices method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
- [JCA80018] The ComponentContext.getServices method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.
- [JCA80019] The ComponentContext.getServiceReferences method MUST return a collection containing one ServiceReference object typed by the interface provided by the businessInterface parameter for each of the target services configured on the reference identified by the referenceName parameter.
- [JCA80020] The ComponentContext.getServiceReferences method MUST return an empty collection if the service reference with the name supplied in the referenceName parameter is not wired to any target services.
- [JCA80021] The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1.
- [JCA80022] The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
- [JCA80023] The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.
- [JCA80024] The ComponentContext.createSelfReference method MUST return a ServiceReference object typed by the interface defined by the businessInterface parameter for one of the services of the invoking component which has the interface defined by the businessInterface parameter.
- [JCA80025] The ComponentContext.getServiceReferences method MUST throw an

	IllegalArgumentException if the component does not have a service which implements the interface identified by the businessInterface parameter.
[JCA80026]	The ComponentContext.createSelfReference method MUST return a ServiceReference object typed by the interface defined by the businessInterface parameter for the service identified by the serviceName of the invoking component and which has the interface defined by the businessInterface parameter.
[JCA80027]	The ComponentContext.createSelfReference method MUST throw an IllegalArgumentException if the component does not have a service with the name identified by the serviceName parameter.
[JCA80028]	The ComponentContext.createSelfReference method MUST throw an IllegalArgumentException if the component service with the name identified by the serviceName parameter does not implement a business interface which is compatible with the supplied businessInterface parameter.
[JCA80029]	The ComponentContext.getProperty method MUST return an object of the type identified by the type parameter containing the value specified in the component configuration for the property named by the propertyName parameter or null if no value is specified in the configuration.
[JCA80030]	The ComponentContext.getProperty method MUST throw an IllegalArgumentException if the component does not have a property with the name identified by the propertyName parameter.
[JCA80031]	The ComponentContext.getProperty method MUST throw an IllegalArgumentException if the component property with the name identified by the propertyName parameter does not have a type which is compatible with the supplied type parameter.
[JCA80032]	The ComponentContext.cast method MUST return a ServiceReference object which is typed by the same business interface as specified by the reference proxy object supplied in the target parameter.
[JCA80033]	The ComponentContext.cast method MUST throw an IllegalArgumentException if the supplied target parameter is not an SCA reference proxy object.
[JCA80034]	The RequestContext.getSecuritySubject method MUST return the JAAS subject of the current request, or null if there is no subject or null if the method is invoked from code not processing a service request or callback request.
[JCA80035]	The RequestContext.getServiceName method MUST return the name of the service for which an operation is being processed, or null if invoked from a thread that is not processing a service operation or a callback operation.
[JCA80036]	The RequestContext.getCallbackReference method MUST return a ServiceReference object typed by the interface of the callback supplied by the client of the invoked service, or null if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation.
[JCA80037]	The RequestContext.getCallback method MUST return a reference proxy object typed by the interface of the callback supplied by the client

- of the invoked service, or null if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation.
- [JCA80038] When invoked during the execution of a callback operation, the `RequestContext.getServiceReference` method MUST return a `ServiceReference` that represents the callback that was invoked.
- [JCA80039] When invoked from a thread not involved in the execution of either a service operation or of a callback operation, the `RequestContext.getServiceReference` method MUST return null.
- [JCA80040] The `ServiceReference.getService` method MUST return a reference proxy object which can be used to invoke operations on the target service of the reference and which is typed with the business interface of the reference.
- [JCA80041] The `ServiceReference.getBusinessInterface` method MUST return a `Class` object representing the business interface of the reference.
- [JCA80042] The `SCAClientFactory.newInstance(URI)` method MUST return an object which implements the `SCAClientFactory` class for the SCA Domain identified by the `domainURI` parameter.
- [JCA80043] The `SCAClientFactory.newInstance(URI)` method MUST throw a `NoSuchDomainException` if the `domainURI` parameter does not identify a valid SCA Domain.
- [JCA80044] The `SCAClientFactory.newInstance(Properties, URI)` method MUST return an object which implements the `SCAClientFactory` class for the SCA Domain identified by the `domainURI` parameter.
- [JCA80045] The `SCAClientFactory.newInstance(Properties, URI)` method MUST throw a `NoSuchDomainException` if the `domainURI` parameter does not identify a valid SCA Domain.
- [JCA80046] The `SCAClientFactory.newInstance(Classloader, URI)` method MUST return an object which implements the `SCAClientFactory` class for the SCA Domain identified by the `domainURI` parameter.
- [JCA80047] The `SCAClientFactory.newInstance(Classloader, URI)` method MUST throw a `NoSuchDomainException` if the `domainURI` parameter does not identify a valid SCA Domain.
- [JCA80048] The `SCAClientFactory.newInstance(Properties, Classloader, URI)` method MUST return an object which implements the `SCAClientFactory` class for the SCA Domain identified by the `domainURI` parameter.
- [JCA80049] The `SCAClientFactory.newInstance(Properties, Classloader, URI)` MUST throw a `NoSuchDomainException` if the `domainURI` parameter does not identify a valid SCA Domain.
- [JCA80050] The `SCAClientFactory.getService` method MUST return a proxy object which implements the business interface defined by the `interfaze` parameter and which can be used to invoke operations on the service identified by the `serviceURI` parameter.
- [JCA80051] The `SCAClientFactory.getService` method MUST throw a `NoSuchServiceException` if a service with the relative URI `serviceURI` and a business interface which matches `interfaze` cannot be found in

the SCA Domain targeted by the SCAClient object.

- [JCA80053] The SCAClientFactory.getDomainURI method MUST return the SCA Domain URI of the Domain associated with the SCAClientFactory object.
- [JCA80055] The implementation of the SCAClientFactoryFinder.find method MUST return an object which is an implementation of the SCAClientFactory interface, for the SCA Domain represented by the domainURI parameter, using the supplied properties and classloader.
- [JCA80056] The implementation of the SCAClientFactoryFinder.find method MUST throw a ServiceRuntimeException if the SCAClientFactory implementation could not be found.
- [JCA50057] The ResponseDispatch.sendResponse() method MUST send the response message to the client of an asynchronous service.
- [JCA80058] The ResponseDispatch.sendResponse() method MUST throw an InvalidStateException if either the sendResponse method or the sendFault method has already been called once.
- [JCA80059] The ResponseDispatch.sendFault() method MUST send the supplied fault to the client of an asynchronous service.
- [JCA80060] The ResponseDispatch.sendFault() method MUST throw an InvalidStateException if either the sendResponse method or the sendFault method has already been called once.
- [JCA90001] An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code.
- [JCA90001] SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an SCA annotation on a static method or a static field of an implementation class and the SCA runtime MUST NOT instantiate such an implementation class.
- [JCA90003] If a constructor of an implementation class is annotated with @Constructor and the constructor has parameters, each of these parameters MUST have either a @Property annotation or a @Reference annotation.
- [JCA90004] A method annotated with @Destroy can have any access modifier and MUST have a void return type and no arguments.
- [JCA90005] If there is a method annotated with @Destroy that matches the criteria for the annotation, the SCA runtime MUST call the annotated method when the scope defined for the implementation class ends.
- [JCA90007] When marked for eager initialization with an @EagerInit annotation, the composite scoped instance MUST be created when its containing component is started.
- [JCA90008] A method marked with the @Init annotation can have any access modifier and MUST have a void return type and no arguments.
- [JCA90009] If there is a method annotated with @Init that matches the criteria for the annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete.

[JCA90011]	The @Property annotation MUST NOT be used on a class field that is declared as final.
[JCA90013]	For a @Property annotation applied to a constructor parameter, there is no default value for the name attribute and the name attribute MUST be present.
[JCA90014]	For a @Property annotation applied to a constructor parameter, the required attribute MUST NOT have the value false.
[JCA90015]	The @Qualifier annotation MUST be used in a specific intent annotation definition where the intent has qualifiers.
[JCA90016]	The @Reference annotation MUST NOT be used on a class field that is declared as final.
[JCA90018]	For a @Reference annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present.
[JCA90019]	For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true.
[JCA90020]	If the type of a reference is not an array or any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference annotation required attribute is false and with @multiplicity=1..1 if the @Reference annotation required attribute is true.
[JCA90021]	If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation required attribute is true.
[JCA90022]	An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null (either via injection or via API call).
[JCA90023]	An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection (either via injection or via API call).

[JCA90040]	A remotable service can be published externally as a service and MUST be translatable into a WSDL portType.
[JCA90041]	The @Scope annotation MUST only be used on a service's implementation class. It is an error to use this annotation on an interface.
[JCA90042]	An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present.
[JCA90045]	If a component implementation has two services with the same Java simple name, the names attribute of the @Service annotation MUST be specified.
[JCA90046]	When used to annotate a method or a field of an implementation class for injection of a callback object, the @Callback annotation MUST NOT specify any attributes.
[JCA90047]	For a @Property annotation, if the type of the Java class field or the type of the input parameter of the setter method or constructor is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <property/> element with a @many attribute set to true, otherwise @many MUST be set to false.
[JCA90050]	The number of Strings in the names attribute array of the @Service annotation MUST match the number of elements in the value attribute array.
[JCA90052]	<p>The @AllowsPassByReference annotation MUST only annotate the following locations:</p> <ul style="list-style-type: none"> • a service implementation class • an individual method of a remotable service implementation • an individual reference which uses a remotable interface, where the reference is a field, a setter method, or a constructor parameter
[JCA90053]	The @Remotable annotation is valid only on a Java interface, a Java class, a field, a setter method, or a constructor parameter. It MUST NOT appear anywhere else.
[JCA90054]	When used to annotate a method or a field of an implementation class for injection of a callback object, the type of the method or field MUST be the callback interface of at least one bidirectional service offered by the implementation class.

- [JCA90055] A method annotated with `@OneWay` MUST have a void return type and MUST NOT have declared checked exceptions.
- [JCA90056] When a method of a Java interface is annotated with `@OneWay`, the SCA runtime MUST ensure that all invocations of that method are executed in a non-blocking fashion, as described in the section on Asynchronous Programming.
- [JCA90057] The `@Callback` annotation MUST NOT appear on a setter method or a field of a Java implementation class that has COMPOSITE scope.
- [JCA90058] When used to annotate a setter method or a field of an implementation class for injection of a callback object, the SCA runtime MUST inject a callback reference proxy into that method or field when the Java class is initialized, if the component is invoked via a service which has a callback interface and where the type of the setter method or field corresponds to the type of the callback interface.
- [JCA90060] The value of each element in the `@Service` names array MUST be unique amongst all the other element values in the array.
- [JCA90061] When the Java type of a field, setter method or constructor parameter with the `@Property` annotation is a primitive type or a JAXB annotated class, the SCA runtime MUST convert a property value specified by an SCA component definition into an instance of the Java type as defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation enabled.
- [JCA100001] For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a `@WebService` annotation on the class, even if it doesn't.
- [JCA100002] The SCA runtime MUST treat an `@org.oasisopen.sca.annotation.OneWay` annotation as a synonym for the `@javax.jws.OneWay` annotation.
- [JCA100003] For the WSDL-to-Java mapping, the SCA runtime MUST take the generated `@WebService` annotation to imply that the Java interface is `@Remotable`.
- [JCA100004] SCA runtimes MUST support the JAXB 2.1 mapping from XML Schema to Java and from Java to XML Schema.
- [JCA100005] SCA runtimes MAY support the SDO 2.1 mapping from XML schema types to Java and from Java to XML Schema.
- [JCA100006] For SCA service interfaces defined using `interface.java`, the Java interface MUST NOT contain the additional client-side asynchronous polling and callback methods defined by JAX-WS.
- [JCA100007] For SCA reference interfaces defined using `interface.java`, the SCA runtime MUST support a Java interface which contains the additional client-side asynchronous polling and callback methods defined by JAX-WS.
- [JCA100008] If the additional client-side asynchronous polling and callback methods defined by JAX-WS are present in the interface which declares the type of a reference in the implementation, SCA Runtimes MUST NOT

	include these methods in the SCA reference interface in the component type of the implementation.
[JCA100009]	SCA runtimes MUST support the use of the JAX-WS client asynchronous model.
[JCA100010]	For SCA service interfaces defined using interface.java, the SCA runtime MUST support a Java interface which contains the server-side asynchronous methods defined by SCA.
[JCA100011]	An SCA runtime MUST apply the JAX-WS annotations as described in Table 11-1 and Table 11-2 when introspecting a Java class or interface class.
[JCA100012]	A Java interface or class annotated with @WebService MUST be treated as if annotated with the SCA @Remotable annotation
[JCA100013]	A Java class annotated with the @WebService annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition instead of the annotated Java class.
[JCA100014]	A Java class annotated with the @WebService annotation with its endpointInterface attribute set MUST have its interface defined by the referenced interface instead of annotated Java class.
[JCA100015]	A Java class or interface containing an @WebParam annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface.
[JCA100016]	A Java class or interface containing an @WebResult annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface.
[JCA100017]	A Java class containing an @ServiceMode annotation MUST be treated as if the SOAP intent is applied to the Java class.
[JCA100018]	An interface or class annotated with @WebServiceClient MUST NOT be used to define an SCA interface.
[JCA100019]	A class annotated with @WebServiceProvider MUST be treated as if annotated with the SCA @Remotable annotation.
[JCA100020]	A Java class annotated with the @WebServiceProvider annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition is used instead of the annotated Java class.
[JCA100021]	A Java class or interface containing an @SOAPBinding annotation MUST be treated as if the SOAP intent is applied to the Java class or interface.
[JCA100022]	SCA runtimes MUST support the JAX-WS 2.1 mappings from WSDL to Java and from Java to WSDL.
[JCA100023]	The value of the name attribute of the @WebService annotation, if present, MUST be used to define the name of an SCA service when there is no @Service annotation present in the SCA component implementation.
[JCA100024]	For a Java method annotated with the @WebMethod annotation with the operationName set, an SCA runtime MUST use the value of the operationName attribute as the SCA operation name.

[JCA100025]

An SCA runtime MUST NOT include a Java method annotated with the `@WebMethod` annotation with the `exclude` attribute set to true in an SCA interface.

[JCA100026]

For a Java parameter annotated with the `@WebParam` annotation with the `mode` attribute set, an SCA runtime MUST apply the value of the `mode` attribute when comparing interfaces.

The value of the `name` attribute of the `@WebService` annotation, if present, MUST be used to define the name of an SCA service when the `@Service` annotation is present without the `name` attribute and indicates that the Java interface or class annotated with the `@WebService` annotation defines an SCA service interface. [JCA100028]

The value of the `name` attribute of the `@WebService` annotation, if present, MUST be used to define the name of an SCA service when the `@Service` annotation is present without the `name` attribute and indicates that the Java interface or class annotated with the `@WebService` annotation defines an SCA service interface.

4088

Appendix D. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

Participant Name	Affiliation
Bryan Aupperle	IBM
Ron Barack	SAP AG*
Mirza Begg	Individual
Michael Beisiegel	IBM
Henning Blohm	SAP AG*
David Booz	IBM
Martin Chapman	Oracle Corporation
Graham Charters	IBM
Shih-Chang Chen	Oracle Corporation
Chris Cheng	Primeton Technologies, Inc.
Vamsavardhana Reddy Chillakuru	IBM
Roberto Chinnici	Sun Microsystems
Pyounguk Cho	Oracle Corporation
Eric Clairambault	IBM
Mark Combella	Avaya, Inc.
Jean-Sebastien Delfino	IBM
Derek Dougans	Individual
Mike Edwards	IBM
Ant Elder	IBM
Raymond Feng	IBM
Bo Ji	Primeton Technologies, Inc.
Uday Joshi	Oracle Corporation
Anish Karmarkar	Oracle Corporation
Khanderao Kand	Oracle Corporation
Michael Keith	Oracle Corporation
Rainer Kerth	SAP AG*
Meeraj Kunnumpurath	Individual
Simon Laws	IBM
Yang Lei	IBM
Mark Little	Red Hat
Ashok Malhotra	Oracle Corporation
Jim Marino	Individual
Jeff Mischkinsky	Oracle Corporation
Sriram Narasimhan	TIBCO Software Inc.
Simon Nash	Individual
Sanjay Patil	SAP AG*
Plamen Pavlov	SAP AG*
Peter Peshev	SAP AG*
Ramkumar Ramalingam	IBM
Luciano Resende	IBM

Michael Rowley
Vladimir Savchenko
Pradeep Simha
Raghav Srinivasan
Scott Vorthmann
Feng Wang

Paul Yang

Active Endpoints, Inc.
SAP AG*
TIBCO Software Inc.
Oracle Corporation
TIBCO Software Inc.
Primeton Technologies, Inc.
Changfeng Open Standards
Platform Software

Appendix E. Revision History

Revision	Date	Editor	Changes Made
1	2007-09-26	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-02-28	Anish Karmarkar	Applied resolution of issues: 4, 11, and 26
3	2008-04-17	Mike Edwards	Ed changes
4	2008-05-27	Anish Karmarkar David Booz Mark Combella	Added InvalidServiceException in Section 7 Various editorial updates
WD04	2008-08-15	Anish Karmarkar	<ul style="list-style-type: none"> * Applied resolution of issue 9 (it was applied before, not sure by whom, but it was applied incorrectly) * Applied resolution of issue 12, 22, 23, 29, 31, 35, 36, 37, 44, 45 * Note that issue 33 was applied, but not noted, in a previous version * Replaced the osoa.org NS with the oasis-open.org NS
WD05	2008-10-03	Anish Karmarkar	<ul style="list-style-type: none"> * Fixed the resolution of issue 37 but re-adding the sentence: "However, the @... annotation must be used in order to inject a property onto a non-public field. -- in the @Property and @Reference section * resolution of issue 9 was applied incorrectly. Fixed that -- removed the requirement for throwing an exception on ComponentContext.getServiceReferences() when multiplicity of references > 1 * minor ed changes
cd01-rev1	2008-12-11	Anish Karmarkar	<ul style="list-style-type: none"> * Fixed reference style to [RFC2119] instead of [1]. * Applied resolutions of issues 20, 21, 41, 42, 43, 47, 48, 49.
cd01-rev2	2008-12-12	Anish Karmarkar	* Applied resolutions of issues 61, 71, 72, 73, 79, 81, 82, 84, 112
cd01-rev3	2008-12-16	David Booz	* Applied resolution of issues 56, 75, 111
cd01-rev4	2009-01-18	Anish Karmarkar	* Applied resolutions of issues 28, 52, 94, 96, 99, 101
cd02	2009-01-26	Mike Edwards	Minor editorial cleanup. All changes accepted.

			All comments removed.
cd02-rev1	2009-02-03	Mike Edwards	Issues 25+95 Issue 120
cd02-rev2	2009-02-08	Mike Edwards	Merge annotation definitions contained in section 10 into section 8 Move remaining parts of section 10 to section 7. Accept all changes.
cd02-rev3	2009-03-16	Mike Edwards	Issue 104 - RFC2119 work and formal marking of all normative statements - all sections - Completion of Appendix B (list of all normative statements) Accept all changes
cd02-rev4	2009-03-20	Mike Edwards	Editorially removed sentence about componentType side files in Section1 Editorially changed package name to org.oasisopen from org.osoa in lines 291, 292 Issue 6 - add Section 2.3, modify section 9.1 Issue 30 - Section 2.2.2 Issue 76 - Section 6.2.4 Issue 27 - Section 7.6.2, 7.6.2.1 Issue 77 - Section 1.2 Issue 102 - Section 9.21 Issue 123 - conersations removed Issue 65 - Added a new Section 4 ** Causes renumbering of later sections ** ** NB new numbering is used below ** Issue 119 - Added a new section 12 Issue 125 - Section 3.1 Issue 130 - (new number) Section 8.6.2.1 Issue 132 - Section 1 Issue 133 - Section 10.15, Section 10.17 Issue 134 - Section 10.3, Section 10.18 Issue 135 - Section 10.21 Issue 138 - Section 11 Issue 141 - Section 9.1 Issue 142 - Section 10.17.1
cd02-rev5	2009-04-20	Mike Edwards	Issue 154 - Appendix A Issue 129 - Section 8.3.1.1
cd02-rev6	2009-04-28	Mike Edwards	Issue 148 - Section 3 Issue 98 - Section 8
cd02-rev7	2009-04-30	Mike Edwards	Editorial cleanup throughout the spec

cd02-rev8	2009-05-01	Mike Edwards	Further extensive editorial cleanup throughout the spec Issue 160 - Section 8.6.2 & 8.6.2.1 removed
cd02-rev8a	2009-05-03	Simon Nash	Minor editorial cleanup
cd03	2009-05-04	Anish Karmarkar	Updated references and front page clean up
cd03-rev1	2009-09-15	David Booz	Applied Issues: 1,13,125,131,156,157,158,159,161,165,172,177
cd03-rev2	2010-01-19	David Booz	Updated to current Assembly namespace Applied issues: 127,155,168,181,184,185,187,189,190,194
cd03-rev3	2010-02-01	Mike Edwards	Applied issue 54. Editorial updates to code samples.
cd03-rev4	2010-02-05	Bryan Aupperle, Dave Booz	Editorial update for OASIS formatting
CD04	2010-02-06	Dave Booz	Editorial updates for Committee Draft 04 All changes accepted
CD04-rev1	2010-07-13	Dave Booz	Applied issues 199, 200
CD04-rev2	2010-10-19	Dave Booz	Applied issues 201,212,213
CSD04-rev3	2010-11-05	Dave Booz	Applied issue 216, ed. updates for CSD vote
CSD05	2010-11-08	OASIS TC Admin	Cleaned and published.
WD051	2011-06-20	Mike Edwards	Issues 240, 241, 242: 1) Made non-normative JCA90024 thru JCA90039 inclusive. Reword section 10.24.1 2) Made JCA20009 non-normative. Section 2.3.4 reworded. 3) Removed JCA80052 Issues 233 - updated frontmatter, added section 1.4
WD052	2011-07-18	Mike Edwards	Issue 243: Changes to the Java Client API - all affect the SCAClientFactoryFinderImpl class in section B.1.3 Removed JCA80054 as part of JAVA-240
WD053	2011-08-08	Mike Edwards	All changes accepted
WD054	2011-08-15	Mike Edwards	Issue 244 - reword [JCA30001] in Section 3.1 All changes accepted.